Ruffle: Rapid 3-party shuffle protocols

Pranav Shriram A*
JP Morgan Chase
Bangalore, India
pranavshriram99@gmail.com

Nishat Koti
Indian Institute of Science
Bangalore, India
kotis@iisc.ac.in

Varsha Bhat Kukkala
Indian Institute of Science
Bangalore, India
varshak@iisc.ac.in

Arpita Patra
Indian Institute of Science
Bangalore, India
arpita@iisc.ac.in

Bhavish Raj Gopal
Indian Institute of Science
Bangalore, India
bhavishraj@iisc.ac.in

Somya Sangal
Indian Institute of Science
Bangalore, India
somyasangal@iisc.ac.in

ABSTRACT

Secure shuffle is an important primitive that finds use in several applications such as secure electronic voting, oblivious RAMs, secure sorting, to name a few. For time-sensitive shuffle-based applications that demand a fast response time, it is essential to design a fast and efficient shuffle protocol. In this work, we design secure and fast shuffle protocols relying on the techniques of secure multiparty computation. We make several design choices that aid in achieving highly efficient protocols. Specifically, we consider malicious 3-party computation setting with an honest majority and design robust ring-based protocols. Our shuffle protocols provide a fast online (i.e., input-dependent) phase compared to the state-of-the-art for the considered setting.

To showcase the efficiency improvements brought in by our shuffle protocols, we consider two distinct applications of anonymous broadcast and secure graph computation via the GraphSC paradigm. In both cases, multiple shuffle invocations are required. Hence, going beyond standalone shuffle invocation, we identify two distinct scenarios of multiple invocations and provide customised protocols for the same. Further, we showcase that our customized protocols not only provide a fast response time, but also provide improved *overall* run time for multiple shuffle invocations. With respect to the applications, we not only improve in terms of efficiency, but also work towards providing improved security guarantees, thereby outperforming the respective state-of-the-art works. We benchmark our shuffle protocols and the considered applications to analyze the efficiency improvements with respect to various parameters.

KEYWORDS

secure shuffle, anonymous broadcast, secure graph computation, secure multiparty computation

1 INTRODUCTION

Shuffle is a technique of rearranging the elements of an ordered set. Performing a shuffle in a privacy-preserving manner entails randomly permuting the elements of the ordered set while ensuring that the permutation, as well as the elements in the ordered set, are not known on clear. Secure shuffle finds wide-spread use as a primitive in various applications such as electronic voting [24, 44], secure sorting [25, 26], oblivious RAM [7, 12], GraphSC paradigm [6, 43], anonymous broadcast [20], to name a few. Several works in the literature [6, 13, 20, 34, 35] provide a secure shuffle protocol using the cryptographic technique of secure multiparty computation (MPC). This technique enables a set of n parties to jointly compute a function on their private inputs while guaranteeing that no subset of at most $t < n$ parties, controlled by an adversary, learns anything other than the function output.

An essential factor to be considered when designing a secure shuffle protocol is its response time (which accounts for the time taken from submission of the input, it's processing, to delivery of the output). To minimize the response time, we focus on designing secure shuffle protocols in the preprocessing paradigm, which allow offloading heavy input-independent computations to a preprocessing phase, thereby obtaining a very fast input-dependent online phase. Although secure shuffle is used in various applications, we use the representative example of anonymous broadcast to motivate the need for a fast online phase (quick response time). An anonymous broadcast system allows a set of N clients to broadcast their messages such that none learns about the association between a message and the identity of its sender. Such a system finds application in use cases like live anonymous polling/feedback. Further, note that the output is required in real-time due to the live nature of the event. For such time-sensitive applications, it is important to have a system which provides a fast response time. Since secure shuffle forms an integral part of anonymous broadcast, a fast protocol for shuffle is essential to facilitate a fast anonymous broadcast system. A similar argument applies to other real-time applications as well where secure shuffle is used, such as the GraphSC paradigm [6, 43] to securely evaluate breadth first search (BFS) for contact tracing, PageRank for fraud detection, etc. To further enhance the overall efficiency, we consider working in the small-party setting that is known to provide efficient, customized solutions [15, 19, 29, 30, 39, 46, 47], following the footsteps of prior works on shuffle such as [6, 20]. We make several other design choices to enhance the efficiency of the designed secure shuffle protocols. Our contributions are detailed next which elaborate on these choices.

*This work was done during the author's affiliation at Indian Institute of Science.

Proceedings on Privacy Enhancing Technologies YYYY(X), 1–19

https://doi.org/XXXXXXX.XXXXXXX

1.1 Our contributions

Keeping efficiency in mind, we design ring-based maliciously secure shuffle protocols in the threshold-optimal setting of 3-party computation (3PC), assuming an honest majority (i.e., $t < n/2$). We design our protocols in the preprocessing model and focus on attaining a fast online phase. Our protocols are robust and provide the security of guaranteed output delivery (GOD, also known as robustness) where honest parties are guaranteed to receive the correct output regardless of any adversarial behaviour. We note that this security guarantee is achieved at no additional (amortized) cost in comparison to weaker security notions such as security with fairness[1] or abort security[2].

We showcase the efficiency improvements brought in by our shuffle protocols for the specific applications of anonymous broadcast and GraphSC paradigm. In the process, we identify the need to handle multiple invocations of shuffle and hence design optimizations that are tailor-made to cater to this. We also work towards improving the security guarantees offered in the considered applications.

Secure shuffle. The shuffle protocol takes as input the secret shares of the elements of the ordered set that require to be shuffled. The output comprises secret shares of the shuffled elements. The random secret permutation used for shuffling is defined during the run of the protocol. The works of [6, 20] provide a secure shuffle protocol that offers the weaker notion of abort security[3]. We design a new shuffle protocol, Ruffle, whose highlight is the improved online efficiency in comparison both [20] and [6]. At a high level, Ruffle leverages the secret sharing semantics to offload the input-independent computations to a preprocessing phase. This allows achieving a super fast online phase by restricting computations to be performed only on the input-dependent shares. In fact, Ruffle also has a better overall run time than the shuffle protocol of [20] owing to our better round as well as communication complexity. Finally, we note that Ruffle is designed to offer the improved security guarantee of robustness in comparison to prior works.

Anonymous broadcast. Recall that the application allows N clients to securely shuffle their input messages. When realizing this via MPC, the clients rely on a set of servers to perform the secure shuffle on their behalf. Note that an anonymous broadcast system can run perpetually, i.e., client messages are received continuously, and the system is responsible for shuffling every consecutive set of N well-formed messages. Thus, an anonymous broadcast system in fact requires multiple sequential invocations of shuffle which can be captured by the following generic scenario.

Let $\mathsf{T}_1, \mathsf{T}_2, \ldots, \mathsf{T}_m$ be m ordered sets that are required to be shuffled under random secret permutations, say $\pi_1, \pi_2, \ldots, \pi_m$, respectively. Consider the scenario where these shuffles are performed sequentially such that $\pi_{i+1}(\mathsf{T}_{i+1})$ is invoked after $\pi_i(\mathsf{T}_i)$, and T_{i+1} is *independent* of $\pi_i(\mathsf{T}_i)$[4]. We refer to this as Independent-Shuffles scenario where multiple independent shuffles are required with the constraint that they are invoked sequentially.

While Ruffle is designed to handle the case of a single shuffle invocation, we extend it and design Ruffle-1 to handle the scenario of Independent-Shuffles. Ruffle-1 is designed to leverage the independence of the m shuffles in Independent-Shuffles, to facilitate performing the necessary preprocessing steps in parallel. Our use of Ruffle-1, allows us to design a more efficient shuffle-based anonymous broadcast system in comparison to the state-of-the-art system of Clarion [20]. Further, the drawback of Clarion is that it does not offer the property of censorship resistance. This property guarantees that a malicious server should not be allowed to discard an honest client's message by claiming it to be malformed. Hence, apart from improving efficiency, our system also guarantees censorship resistance. Our system also aims to minimize communication/computation overhead at client.

GraphSC paradigm. This paradigm [6, 43] provides a highly efficient and scalable solution for securely evaluating graph algorithms. Unlike the case of Independent-Shuffles, this paradigm requires performing a composition of shuffles which can be captured by the following generic scenario.

Unlike the previous scenario of m independent shuffles, in this case we are interested in determining the composition of m shuffles such that $\mathsf{T}_m = \pi_m(\pi_{m-1}(\ldots \pi_1(\mathsf{T})))$, where T is the input to be shuffled. Such a composition of m shuffles generates a sequence of intermediate shuffled sets, where the i^{th} ordered set is denoted as $\mathsf{T}_i = \pi_i(\ldots \pi_1(\mathsf{T}))$. In this way, the composition of permutations induces a sequential nature to the shuffle invocations with $\pi_{i+1}(\mathsf{T}_i)$ being invoked after $\pi_i(\mathsf{T}_{i-1})$ since $\mathsf{T}_i = \pi_i(\mathsf{T}_{i-1})$. We refer to this as the Composed-Shuffles scenario where the permutations are required to be composed such that the output of one shuffle invocation is fed as the input to the next[5].

Recall that Ruffle-1 is designed to leverage the independence of the m shuffles to facilitate parallel preprocessing. This is in contrast to Composed-Shuffles where the shuffles are no longer independent. Thus, Ruffle-1 is not apt for Composed-Shuffles, and hence we design Ruffle-2 to specifically cater to this scenario. Ruffle-2 strategically breaks the sequential dependence on shuffles in the preprocessing. This enables performing the preprocessing phase in parallel for the m shuffles. Although Ruffle-2 can be used in the scenario of Independent-Shuffles, we note that 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. Hence, the use of Ruffle-1 is apt for Independent-Shuffles and Ruffle-2 for Composed-Shuffles. We showcase the improvements that can be attained in the GraphSC paradigm of [6] via the Ruffle-2. Note that our shuffle protocols enable reusing and inverting the underlying secret permutation, which is an essential property needed in applications such as GraphSC paradigm.

Comparison. In summary, the current work provides efficient solutions for secure shuffle while accounting for different scenarios. A comparison of our shuffle protocols with that of [6, 20] is given in Table 1. Since all protocols have a common structure for the online

[1] This ensures that either all parties receive the output or none do.
[2] This allows the corrupt parties alone to get the output.
[3] Honest parties in [6] either receive the correct shuffled result, or they are informed of misbehaviour, if any. See §B for a discussion on security guarantees of [6].
[4] Note that $\pi_j(\mathsf{T}_j)$ denotes the operation of permuting the elements in T_j according to the permutation π_j.
[5] The scenario can indeed be generalized such that π_{i+1} can be invoked on some function f of $\pi_i(\mathsf{T}_i)$, rather than on $\pi_i(\mathsf{T}_i)$ itself.

Scenario	Protocol	Online				Preprocessing		Security
		Semi-honest shuffle		Verification				
		Rounds	Comm. (bits)	Rounds	Comm. (bits)	Rounds	Comm. (bits)	
Independent-Shuffles/ Composed-Shuffles	[20]‡	$2m$	$2N(2\ell+3p)m$	$4m$	$2N(\ell+2p)m+4pm$	2	$N(2\ell+9p)m+4pm$	Abort
Independent-Shuffles/ Composed-Shuffles	[6]	$3m$	$6N(\ell+\kappa)m$	$3(2+\log_2\kappa)m$	$(6N\kappa+3\kappa)m$	*		Abort ‡‡
Independent-Shuffles	Ruffle-1	$2m$	$3N\ell m$	2	3080†	$5+\log_2\kappa$	$(6N\ell+12N\kappa+3\kappa)m$**	GOD
Composed-Shuffles	Ruffle-2					$6+\log_2\kappa$***	$(9N\ell+12N\kappa+3\kappa)m$	

N: number of elements to be shuffled, where each element is an ℓ-bit string; $\kappa(=48)$: statistical security parameter; p: order of field. [20] uses a 128-bit field
‡: Although [20] does not have an explicit preprocessing phase, we observe that the shuffle correlation and other randomness can be preprocessed. Hence, we explicitly distinguish between preprocessing and online to provide a fair comparison.
*: The preprocessing for [6] only involves the generation of randomness, non-interactively. ‡‡: See §B for a discussion on security guarantees of [6].
†: The communication for verification comprises broadcasting 2 hashes and 2 bits, the cost of which gets amortized over multiple shuffle instances.
: Ruffle-2 for Independent-Shuffles additionally requires communicating $3N\ell m$ bits. *: Ruffle-1 for Composed-Shuffles instead requires $(5+\log_2\kappa)m$ rounds.

Table 1: Round complexity and communication (amortized) of various shuffle protocols for m invocations.

phase that comprises steps for semi-honest shuffle followed by its verification, the cost of these is reported in Table 1. The improved online phase of both, Ruffle-1 and Ruffle-2, supplemented by their parallel preprocessing in their respective scenarios, results in both of them improving in terms of overall run time in comparison to shuffle protocols in [6, 20] for multiple shuffles (i.e., $m \geq 2$). Further, as shown in the table, Ruffle-1 becomes prohibitively expensive for the Composed-Shuffles case, because it incurs an m factor inflation in the preprocessing round complexity (see the highlighted entry). Similarly, Ruffle-2 is inapt for Independent-Shuffles, due to the inflation of $3N\ell m$ bits in its preprocessing communication complexity (see the highlighted entry). The complexity of Ruffle is captured by the complexity of Ruffle-1 when $m = 1$.

Benchmarks. We benchmark the performance of our shuffle protocols, Ruffle, Ruffle-1 and Ruffle-2. We establish how the protocols, Ruffle-1, Ruffle-2 are apt for their respective scenarios of Independent-Shuffles, Composed-Shuffles. Further, we showcase the improvements brought in by our shuffle protocols in the applications of anonymous broadcast and securely evaluating BFS in the GraphSC paradigm. The summary of the improvements is:

- *Solitary shuffle.* When considering a single invocation, Ruffle improves over [20] and [6] in the online run time. Further, with respect to [20], Ruffle is also better in terms of overall communication as well as overall run time.
- *Multiple-sequential shuffles.* By considering multiple sequential shuffles, we establish the improvements of our shuffle protocols with respect to overall run time in comparison to [6][6]. Beginning with as low as *two* sequential shuffle invocations, both Ruffle-1 (for Independent-Shuffles) and Ruffle-2 (for Composed-Shuffles) outperform [6].
- *Anonymous broadcast.* The server-side complexity of our anonymous broadcast system outperforms [20] in every aspect. The client-side computation also sees improvements. The improvements we observe is not only attributed to our shuffle protocol but also to the improvements we bring in to the other components of the system.
- *BFS via GraphSC.* Our implementation of secure BFS evaluation in the GraphSC paradigm outperforms that of [6]. Further, we also showcase how the performance of our BFS varies with the number of processors in the multiprocessor setting, described in [43]. Unlike in anonymous broadcast, here the reported gain is only due to the improved shuffle protocol.

[6]We stick to comparing with [6] since the shuffle in [6] outperforms that in [20].

Organization. The rest of the paper is organized as follows. §2 describes the related work and §3 provides the preliminaries. §4 describes our shuffle protocol, Ruffle. This is followed by the applications of anonymous broadcast and GraphSC paradigm in §5, which also describe the protocols Ruffle-1 and Ruffle-2. The benchmark results appear in §6, followed by conclusion in §7. Additional preliminaries appear in §A. Details of the shuffle protocol of [6] are recollected in §B. This is followed by additional details of anonymous broadcast and BFS in GraphSC paradigm in §C, §D, respectively. Additional benchmark details and security proofs are provided in §E and §F, respectively.

2 RELATED WORK

One of the first techniques proposed for shuffling is that of mix networks (or mix-nets) [17, 22, 49, 52]. It comprises a sequence of mixes, where each mix receives a set of messages, shuffles them, and forwards them to the next mix. Unlinkability of a message to its sender is guaranteed if at least one mix is honest. To guarantee security when a mix is malicious, it must be ensured that a verifiable shuffle is performed by each mix [1, 3, 8, 27, 44]. This further adds to the expense of a mix-net. Not only are mix-net-based solutions computationally expensive, but they are also vulnerable to traffic analysis attacks [48, 50]. Hence, several works in the literature explore MPC-based techniques for secure shuffling [23, 28, 35, 38, 41, 42]. Some of these solutions rely on securely performing sort [35, 42], while some others consider securely evaluating a permutation network [23, 28, 38, 41]. These techniques require at least $O(\log n)$ rounds for shuffling n elements, which proves to be expensive for time-sensitive applications. The works of [6, 20] which appeared concurrent to each other consider performing a 3PC shuffle protocol in the honest majority setting. In the semi-honest 3PC honest-majority setting, [6] presents a shuffle protocol which is an adaption of the shuffle protocol of [35] to the 3-party setting. This semi-honest protocol requires three rounds of interaction. Note that, [6] contributes to making this semi-honest

protocol secure in the presence of a malicious adversary by augmenting with a verification phase to ensure the correctness of the semi-honest shuffle, which additionally requires $2 + \log_2 \kappa$ rounds. Further, [6] also provides a 2 round semi-honest protocol but leaves open the question of attaining malicious security for the same. Clarion [20] also gives a 2-round 3PC honest-majority shuffle protocol which builds on the semi-honest 2-party protocol of [13]. To guarantee malicious security, they add integrity checks by having MACs appended to the elements to be shuffled. The resulting maliciously secure protocol requires 6 rounds overall. Clarion also extends its shuffle protocol to the n-party dishonest majority setting that guarantees malicious security, which additionally requires maliciously secure OTs (oblivious transfer) in the preprocessing phase. It improves over the protocol in [37] in terms of efficiency, however, it lacks in terms of security guarantees where the latter provides fairness in the preprocessing phase and GOD only in the online phase, for the setting of $t < n/3$.

3 PRELIMINARIES

We work in the 3-party setting. Let $\mathcal{P} = \{P_0, P_1, P_2\}$ denote the set of three parties that are connected via a pairwise private and authentic channel. Let $\mathcal{A}$ be a static, probabilistic, polynomial-time adversary which corrupts at most one party maliciously. Our protocols are proven secure against a computationally bounded $\mathcal{A}$ in the standalone simulation-based security model of MPC, using the real-world/ideal-world simulation paradigm [36]. Parties use a one-time key setup [9, 14, 29, 39, 47] to establish common random keys for a pseudo-random function (PRF) between them. This is modelled as a functionality $\mathcal{F}_{\mathsf{setup}}$ (Fig. 7). This enables each subset of parties to non-interactively sample a common random ℓ-bit string $\mathsf{v} \in \mathbb{Z}_{2^\ell}$. Parties also have access to a collision-resistant hash function, $\mathsf{H}(\cdot)$, and a non-interactive commitment scheme, $\mathsf{Com}(\cdot)$. Formal details appear in §A.

Secret sharing semantics. Inspired from [29, 51], we use the following secret sharing semantics.

- $[\cdot]$-*sharing:* A value $\mathsf{v} \in \mathbb{Z}_2$ is said to be (3, 1) replicated secret shared (RSS) or $[\cdot]$-shared, if there exists $[\mathsf{v}]_1, [\mathsf{v}]_2, [\mathsf{v}]_3 \in \mathbb{Z}_2$ such that $\mathsf{v} = [\mathsf{v}]_1 \oplus [\mathsf{v}]_2 \oplus [\mathsf{v}]_3$, and each $[\mathsf{v}]_{ij} \in \{[\mathsf{v}]_1, [\mathsf{v}]_2, [\mathsf{v}]_3\}$ is held by $P_i, P_j \in \mathcal{P}$.
- $[\![\cdot]\!]$-*sharing:* A value $\mathsf{v} \in \mathbb{Z}_2$ is $[\![\cdot]\!]$-shared among $\mathcal{P}$, if there exists $\alpha_\mathsf{v} \in \mathbb{Z}_2$ that is $[\cdot]$-shared, and there exists $\beta_\mathsf{v} \in \mathbb{Z}_2$ such that $\beta_\mathsf{v} = \mathsf{v} \oplus \alpha_\mathsf{v}$ which is held by all parties in $\mathcal{P}$.

An ℓ-bit value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ (or equivalently $\mathbb{Z}_2^\ell$) is said to be $[\![\cdot]\!]$-shared ($[\cdot]$-shared) if each bit in v is $[\![\cdot]\!]$-shared ($[\cdot]$-shared). Henceforth, we use shares and secret-shares interchangeably.

Non-interactively generating $[\cdot]$-shares of a common $\mathsf{v} \in \mathbb{Z}_{2^\ell}$ held by P_l, P_m. To generate $[\mathsf{v}]$, parties need to define three shares $[\mathsf{v}]_1, [\mathsf{v}]_2, [\mathsf{v}]_3 \in \mathbb{Z}_{2^\ell}$ such that $\mathsf{v} = [\mathsf{v}]_1 \oplus [\mathsf{v}]_2 \oplus [\mathsf{v}]_3$, where each $[\mathsf{v}]_{ij}$ is held by parties $P_i, P_j \in \mathcal{P}$. Observe that this can be done non-interactively by setting the share $[\mathsf{v}]_{lm} = \mathsf{v}$ and the other two $[\cdot]$-shares of v as 0.

Random permutation. Let $\mathbb{N}$ denote the set of integers $\{1, 2, \ldots, N\}$. A permutation as any bijective function $\pi : \mathbb{N} \to \mathbb{N}$. That is, a permutation π denotes a mapping of a rearrangement of the elements in $\mathbb{N}$. The set denoted as S_N consists of all possible (bijective functions) rearrangements of elements in $\mathbb{N}$ and hence comprises $N!$ permutations. Note that permutations can be composed similar to composition of functions, and thus S_N forms a group with respect to composition ($\circ$) operation. S_N satisfies group properties of closure, associativity, and presence of identity. However, permutations are not commutative under composition but are invertible.

Sampling a random permutation denotes choosing a random $\pi \in S_N$. We next describe how parties P_i, P_j can do this non-interactively using the shared key established via $\mathcal{F}_{\mathsf{setup}}$. P_i, P_j non-interactively generate N common random values say $\mathsf{v}_1, \mathsf{v}_2, \ldots, \mathsf{v}_N \in \mathbb{Z}_{2^\ell}$ where $\ell >> \log_2 N$. The parties tag each of the values v_i with its index to obtain a list $S = \{(\mathsf{v}_i, \mathsf{x}_i)\}_{i=1}^N$, where $\mathsf{x}_i = i$. Each party then locally sorts this list of tuples based on the first entry v_i of each tuple to obtain a sorted list $S' = \{(\mathsf{v}'_j, \mathsf{x}'_j)\}_{j=1}^N$. The second element in each tuple of S' defines a random permutation where $\pi(\mathsf{x}_i) = \pi(i) = \mathsf{x}'_i$ for $i \in \{1, 2, \ldots, N\}$.

Joint message passing (jmp) primitive [29]. This primitive allows two parties to deliver a common message to a third party where one sender sends the message while the other sends its hash to the receiver. In the process, either the recipient receives the correct message or, if there is an inconsistency in the received messages, parties instead proceed to identify a trusted third party (TTP)[7]. The TTP is then responsible for performing the required computation on clear and guarantee delivery of output. Several works [19, 29, 30] rely on this primitive or its variation to ensure GOD. We let "P_i, P_j jmp v to P_k" denote invocation of jmp with P_i, P_j as senders, P_k as receiver, and v being the message to be sent. Formal protocol for jmp appears in Fig. 8.

Output reconstruction [29]. To enable reconstruction of a $[\![\cdot]\!]$-shared value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$, parties proceed as follows. During the preprocessing phase, in addition to generating $[\alpha_\mathsf{v}]$, parties also generate commitments on each of the $[\cdot]$-shares of α_v. Looking ahead, these commitments aid in guaranteeing the correct reconstruction of v in the online phase. To generate the commitments, each pair $P_i, P_j \in \mathcal{P}$ computes $\mathsf{Com}([\alpha_\mathsf{v}]_{ij})$ on the value $[\alpha_\mathsf{v}]_{ij}$ using the common randomness. P_i, P_j jmp $\mathsf{Com}([\alpha_\mathsf{v}]_{ij})$ to P_k. After the jmp invocations, it is guaranteed that each party in $\mathcal{P}$ either possess the correct commitment on each $[\cdot]$-share of α_v, or a TTP is identified and subsequent computation proceeds via the TTP[8]. Next, in the online phase to reconstruct v, observe that each party P_k misses the share $[\alpha_\mathsf{v}]_{ij}$ which is held by the other two parties $P_i, P_j \in \mathcal{P} \setminus P_k$. Hence, P_i, P_j send the opening of $\mathsf{Com}([\alpha_\mathsf{v}]_{ij})$ to P_k. Since at most one party among P_i, P_j can be malicious, even if the malicious party sends an incorrect opening, P_k is guaranteed to receive the correct opening from the honest party (the correct opening can be identified owing to the property of the commitment scheme which outputs a $\perp$ for incorrect ones). Party P_k uses the correct opening to obtain the missing share $[\alpha_\mathsf{v}]_{ij}$ and reconstructs v as $\mathsf{v} = \beta_\mathsf{v} \oplus [\alpha_\mathsf{v}]_{ij} \oplus [\alpha_\mathsf{v}]_{ik} \oplus [\alpha_\mathsf{v}]_{jk}$. Thus, reconstruction will not fail if a malicious party tries to disrupt it by sending an incorrect message, resulting in robust reconstruction.

[7] In the case of fair protocol, parties abort.
[8] In case of a fair protocol, the fair variant of jmp is used, which aborts in case of an inconsistency.

On the security of our protocols. While our protocol provides the strongest security of robustness, we note that depending on the application scenario, one may choose the desired level of security. Specifically, robustness is attained by relying on a TTP to carry out the computation on the honest party's inputs (in the clear) if misbehaviour is detected. Hence, if the application under consideration cannot tolerate revealing the inputs to a TTP even though the TTP is known to be honest, the application can settle for the weaker security notion of fairness (which is stronger than abort security achieved in prior systems). The fair version of our protocols can be derived from the robust version by making the following changes– (i) use of the fair version of jmp instead of the robust version, (ii) terminating the protocol when a party aborts instead of proceeding with TTP identification and, (iii) relying on a fair reconstruction protocol. We remark that even for this weaker security notion of fairness, our protocols are on par with the robust protocols in terms of efficiency, and hence, are more efficient than prior works. Finally, we would like to note that an alternative to the TTP-based approach of achieving robustness, is the recent notion of security with friends and foes proposed in [5]. This notion allows attaining robustness without relying on the TTP. We refer an interested reader to [5] for further details pertaining to the same.

4 3PC SHUFFLE

We begin with defining the ideal functionality for shuffle in Fig. 1. Let a table T denote a set of ordered rows where each row consists of an ℓ-bit string. Let N denote the size of T or the number of rows in T. Secure shuffle operation takes as input $[\![\cdot]\!]$-shares of table T, i.e., $[\![\cdot]\!]$-shares of each of the ℓ-bit string that constitutes a row in T. The output is random $[\![\cdot]\!]$-shares of a table T_o, which consists of rows of T in a randomly permuted order.

Functionality $\mathcal{F}_{\mathsf{Shuffle}}$

Without loss of generality, let $P_c \in \mathcal{P}$ denote the party corrupted by adversary $\mathcal{S}$. $\mathcal{F}_{\mathsf{Shuffle}}$ interacts with parties in $\mathcal{P}$ and $\mathcal{S}$. It receives as input $[\![\cdot]\!]$-shares of the input table T from all parties. Let T_o denote the randomly shuffled input table. $\mathcal{F}_{\mathsf{Shuffle}}$ also receives from $\mathcal{S}$ its $[\![\cdot]\!]$-shares of T_o, i.e. it receives $\beta_{\mathsf{T}_o}, \left[\alpha_{\mathsf{T}_o}\right]_{ic}, \left[\alpha_{\mathsf{T}_o}\right]_{jc}$ where P_i, P_j, P_c denote parties in $\mathcal{P}$.
$\mathcal{F}_{\mathsf{Shuffle}}$ proceeds as follows.

- 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 $\mathsf{T}_o = \pi(\mathsf{T})$.
- Set $\left[\alpha_{\mathsf{T}_o}\right]_{ij} = \mathsf{T}_o \oplus \beta_{\mathsf{T}_o} \oplus \left[\alpha_{\mathsf{T}_o}\right]_{ic} \oplus \left[\alpha_{\mathsf{T}_o}\right]_{jc}$. Let $[\![\mathsf{T}_o]\!]_x$ denote the $[\![\cdot]\!]$-share of T_o for $P_x \in \mathcal{P}$.
- Send (Output, $[\![\mathsf{T}_o]\!]_x$) to P_x.

Figure 1: Ideal functionality for shuffle

4.1 Ruffle

Given that the input table T is $[\![\cdot]\!]$-shared, there exists a $\beta_{\mathsf{T}}, \alpha_{\mathsf{T}} \in \mathbb{Z}_{2^\ell}^N$ such that $\beta_{\mathsf{T}} = \mathsf{T} \oplus \alpha_{\mathsf{T}}$ is held by all parties in $\mathcal{P}$, and α_{T} is $[\cdot]$-shared, i.e. $\alpha_{\mathsf{T}} = [\alpha_{\mathsf{T}}]_{01} \oplus [\alpha_{\mathsf{T}}]_{02} \oplus [\alpha_{\mathsf{T}}]_{12}$ where $P_i, P_j \in \mathcal{P}$ hold $[\alpha_{\mathsf{T}}]_{ij} \in \mathbb{Z}_{2^\ell}^N$. Let π be the random permutation used to shuffle the rows of T. Observe that, $\mathsf{T}_o = \pi(\mathsf{T}) = \pi(\beta_{\mathsf{T}} \oplus \alpha_{\mathsf{T}}) = \pi(\beta_{\mathsf{T}}) \oplus \pi(\alpha_{\mathsf{T}})$. To respect the $[\![\cdot]\!]$-sharing semantics for T_o, we require $\mathsf{T}_o = \beta_{\mathsf{T}_o} \oplus \alpha_{\mathsf{T}_o}$. A naive approach is to thus set $\beta_{\mathsf{T}_o} = \pi(\beta_{\mathsf{T}})$ and $\alpha_{\mathsf{T}_o} = \pi(\alpha_{\mathsf{T}})$ since this would satisfy $\mathsf{T}_o = \beta_{\mathsf{T}_o} \oplus \alpha_{\mathsf{T}_o} = \pi(\mathsf{T})$. Observe, however, that this approach leaks the secret permutation π to all the parties, since they all hold β_{T} and will now also hold $\pi(\beta_{\mathsf{T}})$ on clear, from which one can recover π. To keep π private, we observe that it suffices to mask $\pi(\beta_{\mathsf{T}})$ with some randomness $\mathsf{R} \in \mathbb{Z}_{2^\ell}^N$, and hence, define this masked value as β_{T_o}, i.e., $\beta_{\mathsf{T}_o} = \pi(\beta_{\mathsf{T}}) \oplus \mathsf{R}$. Further, to ensure that the relation $\mathsf{T}_o = \beta_{\mathsf{T}_o} \oplus \alpha_{\mathsf{T}_o}$ holds, we redefine $\alpha_{\mathsf{T}_o} = \pi(\alpha_{\mathsf{T}}) \oplus \mathsf{R}$. Thus, given $[\![\mathsf{T}]\!]$, our goal is to generate $[\cdot]$-shares of α_{T_o}, and ensure that all parties hold β_{T_o}. Observe that $\left[\alpha_{\mathsf{T}_o}\right] = [\pi(\alpha_{\mathsf{T}})] \oplus [\mathsf{R}]$. Looking ahead R gets defined during the generation of β_{T_o}. Thus, in what follows, we first describe steps to generate $[\pi(\alpha_{\mathsf{T}})]$, followed by steps to generate β_{T_o} and then $[\mathsf{R}]$.

Generation of $[\pi(\alpha_{\mathsf{T}})]$. Since α_{T} is independent of the input T, it is generated during a preprocessing phase. Hence, $[\cdot]$-shares of $\alpha' = \pi(\alpha_{\mathsf{T}})$ where π is a random secret permutation (independent of T), can be generated during preprocessing. For this, we employ the protocol of [6]. The protocol takes as input $[\cdot]$-shares of a table, and outputs $[\cdot]$-shares of the table shuffled using a random secret permutation π. It also outputs a flag that indicates correctness of $[\cdot]$-shares of shuffled table[9].

At a high level, the protocol of [6] relies on the semi-honest 3PC shuffle protocol from [35] which guarantees privacy against a malicious adversary. [6] then augments this with a robust Set-Equality protocol to verify the correctness of the semi-honest shuffle. The semi-honest shuffle comprises three invocations of Shuffle-Pair protocol. In each instance of Shuffle-Pair, a random permutation is applied to the input (of the Shuffle-Pair), where the permutation is known to a distinct pair of parties and is hidden from the third. The output of the current Shuffle-Pair is fed as input to the next Shuffle-Pair. The composition of all three permutations, thus, makes up the random secret permutation used to shuffle the input table. Since each party is aware of only two permutations, the final permutation remains private. Formal details of Shuffle-Pair protocol are recollected in §B. Each invocation of Shuffle-Pair is followed by a Set-Equality protocol which outputs a flag $\in \{0, 1\}$ indicating whether the table output by the Shuffle-Pair is indeed a random permutation of the input to this Shuffle-Pair. In this way, the output of the shuffle protocol is guaranteed to be correct if all instances of Shuffle-Pair are verified to be correct.

Let $\pi_{12}, \pi_{01}, \pi_{02}$ denote the three permutations used in the three Shuffle-Pair instances, where π_{ij} is held by $P_i, P_j \in \mathcal{P}$. Applying the protocol of [6] on $[\alpha_{\mathsf{T}}]$ outputs $[\pi(\alpha_{\mathsf{T}})]$ and a flag. Here, we let $\pi = \pi_{12} \circ \pi_{01} \circ \pi_{02}$ where $\circ$ denotes composition operation, and flag indicates correctness of $[\pi(\alpha_{\mathsf{T}})]$.

Generation of $\beta_{\mathsf{T}_o} = \pi(\beta_{\mathsf{T}}) \oplus \mathsf{R}$. As part of the Shuffle-Pair instances performed during the preprocessing, parties generate $\pi_{12}, \pi_{01}, \pi_{02}$. The goal now is to generate $\beta_{\mathsf{T}_o} = \pi(\beta_{\mathsf{T}}) \oplus \mathsf{R}$ where $\pi = \pi_{12} \circ \pi_{01} \circ \pi_{02}$. Observe that, unlike during preprocessing, the table to be shuffled is now held by all three parties on clear, while the permutation π is still private. Further, each party misses exactly one permutation that is held by the other two parties. We

[9] We choose [6] over [20] since the protocol in [6] follows the same $[\cdot]$-sharing semantics as required for α, which is not the case in the protocol of [20]. Hence, the use of [6] yields an efficient preprocessing phase. Towards the end of §4.1, we also showcase how to get GOD for the protocol of [6].

leverage these observations in designing our shuffle protocol to attain a highly efficient online phase. We explain case-by-case how each $P_i \in \mathcal{P}$ obtains β_{T_o}.

Generating β_{T_o} towards P_1. Recall that P_1 misses π_{02}. If P_1 is given $\pi_{02}(\beta_{\mathsf{T}})$, it can locally compute $\pi(\beta_{\mathsf{T}}) = \pi_{12} \circ \pi_{01}(\pi_{02}(\beta_{\mathsf{T}}))$ using its knowledge of $\pi_{12} \circ \pi_{01}$. However, as mentioned earlier, since P_1 holds β_{T} on clear, knowledge of $\pi_{02}(\beta_{\mathsf{T}})$ leaks the permutation π_{02} to it. Hence, we instead provide it with $\pi_{02}(\beta_{\mathsf{T}}) \oplus \mathsf{R}'$, where the randomness R' masks $\pi_{02}(\beta_{\mathsf{T}})$ and prevents leakage of π_{02}. For this, observe that π_{02} is held by both P_0, P_2. We let P_0, P_2 sample a random $\mathsf{R}_{02} \in \mathbb{Z}_{2^\ell}^N$, and compute and send $\pi_{02}(\beta_{\mathsf{T}} \oplus \mathsf{R}_{02})$ to P_1. Here, $\pi_{02}(\mathsf{R}_{02})$ serves as the random mask R'. Further, note that since at most one among P_0, P_2 can be malicious, making both send the value to P_1 enables the latter to check the consistency of the received messages and detect misbehaviour, if any. Since the message from the second sender only aids in verifying the consistency of the received messages, to save on communicating an entire table, it suffices for one sender to send the value and the other to send the hash of it[10] . On receiving a consistent $\pi_{02}(\beta_{\mathsf{T}} \oplus \mathsf{R}_{02})$ (which also guarantees its correctness, as otherwise the received messages would have been inconsistent), P_1 can compute β_{T_o} using the received value and the knowledge of permutations π_{12}, π_{01}. Note that $\pi_{02}(\mathsf{R}_{02})$ serves as a mask to hide π from P_1. Looking ahead, similar masks are required in β_{T_o} to keep π hidden from P_2 and P_0. This results in additionally introducing the random masks $\pi_{12}(\pi_{01}(\mathsf{R}_{01}))$ and $\pi_{12}(\mathsf{R}_{12})$, respectively. To ensure that all parties have the same β_{T_o} and use the same randomness for masking $\pi(\beta_{\mathsf{T}})$, β_{T_o} is defined as

$$\begin{aligned}\beta_{\mathsf{T}_o} &= \pi_{12}(\pi_{01}(\pi_{02}(\beta_{\mathsf{T}} \oplus \mathsf{R}_{02}) \oplus \mathsf{R}_{01}) \oplus \mathsf{R}_{12}) \\ &= \pi(\beta_{\mathsf{T}}) \oplus \pi(\mathsf{R}_{02}) \oplus \pi_{12}(\pi_{01}(\mathsf{R}_{01})) \oplus \pi_{12}(\mathsf{R}_{12})\end{aligned} \tag{1}$$

where $\mathsf{R}_{12}, \mathsf{R}_{01} \in \mathbb{Z}_{2^\ell}^N$, and R_{ij} is jointly sampled by $P_i, P_j \in \mathcal{P}$. At the end of first round, since P_1 holds $\mathsf{R}_{12}, \mathsf{R}_{01}, \pi_{12}, \pi_{01}$, and $\pi_{02}(\beta_{\mathsf{T}} \oplus \mathsf{R}_{02})$, it can compute β_{T_o} using Eq. (1).

Generating β_{T_o} towards P_2. Observe that P_2 lacks π_{01} that prevents it from computing $\pi_{12}(\pi_{01}(\pi_{02}(\beta_{\mathsf{T}} \oplus \mathsf{R}_{02})))$. On the other hand, if provided with the value $\pi_{01}(\pi_{02}(\beta_{\mathsf{T}} \oplus \mathsf{R}_{02}))$, then P_2 can obtain $\pi_{12}(\pi_{01}(\pi_{02}(\beta_{\mathsf{T}} \oplus \mathsf{R}_{02})))$ by applying π_{12} on it. However, similar to the case described earlier, this leaks the permutation π_{01} to P_2. To fix this leakage, we first mask $\pi_{02}(\beta_{\mathsf{T}} \oplus \mathsf{R}_{02})$ with the random value R_{01} and then apply π_{01} on this masked value and communicate it to P_2. This justifies the need for the term $\pi_{12}(\pi_{01}(\mathsf{R}_{01}))$ in Eq. (1). The value to be communicated, $\delta_{12} = \pi_{01}(\pi_{02}(\beta_{\mathsf{T}} \oplus \mathsf{R}_{02}) \oplus \mathsf{R}_{01})$ can be computed by P_0 and sent to P_2, since P_0 possesses the required values. Since we want to maintain the invariant that each message is communicated by two senders to aid in verification of correctness at the receiver, we require P_1 to also send hash of this message to P_2. Although P_1 does not possess R_{02} and π_{02} required to compute δ_{12}, observe that it receives $\delta_{02} = \pi_{02}(\beta_{\mathsf{T}} \oplus \mathsf{R}_{02})$ in the first round, and can compute and send the hash of $\delta_{12} = \pi_{01}(\delta_{02} \oplus \mathsf{R}_{01})$ in the next round to P_2. On receiving these values, P_2 can thus verify its correctness and then use Eq. (1) to compute β_{T_o}.

Generating β_{T_o} towards P_0. Given that β_{T_o} is made available to both P_1, P_2, they can send it to P_0 (one sends the value, the other sends the hash). This completes the generation of β_{T_o} towards all the parties. Observe the need for using R_{12} as a mask while computing β_{T_o}. Analogous to the cases for P_1, P_2, absence of R_{12} leaks the permutation π_{12} to P_0. Further, note that although P_2 can compute the correct β_{T_o} only after the second round, it receives δ_{12} required for computing β_{T_o} in the first round itself. Hence, communication of β_{T_o} from P_1, P_2 towards P_0 can happen in the second round. A pictorial view of the messages exchanged is given in Fig. 2.

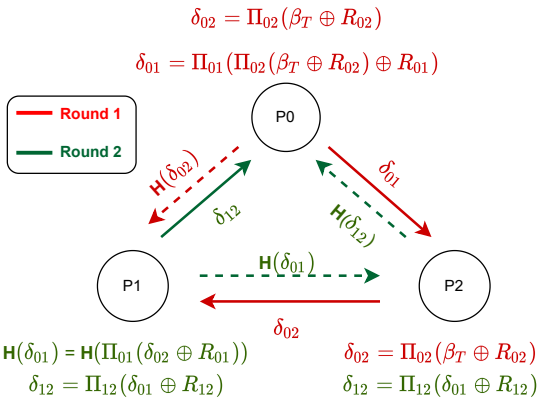


Figure 2: Online phase of Ruffle

Generation of $[\alpha_{\mathsf{T}_o}] = [\pi(\alpha_{\mathsf{T}})] \oplus [\mathsf{R}]$. Subsequent to the above discussion and as evident from Eq. (1), R is defined as $\mathsf{R} = \pi(\mathsf{R}_{02}) \oplus \pi_{12}(\pi_{01}(\mathsf{R}_{01})) \oplus \pi_{12}(\mathsf{R}_{12})$, whose $[\cdot]$-shares are required to be generated in the preprocessing phase. Observe that

$$\begin{aligned}[\alpha_{\mathsf{T}_o}] &= [\pi(\alpha_{\mathsf{T}})] \oplus [\mathsf{R}] \\ &= [\pi_{12}(\pi_{01}(\pi_{02}(\alpha_{\mathsf{T}})))] \oplus [\pi_{12}(\pi_{01}(\pi_{02}(\mathsf{R}_{02})))] \\ &\quad \oplus [\pi_{12}(\pi_{01}(\mathsf{R}_{01}))] \oplus [\pi_{12}(\mathsf{R}_{12})]\end{aligned} \tag{2}$$

P_1, P_2 hold $\pi_{12}(\mathsf{R}_{12})$ on clear. Hence, as described in §3, $[\pi_{12}(\mathsf{R}_{12})]$ can be generated non-interactively. A naive approach of generating $[\cdot]$-shares of remainder terms requires three invocations of Shuffle-Pair for generating $[\pi_{12}(\pi_{01}(\pi_{02}(\alpha_{\mathsf{T}})))]$, three for generating $[\pi_{12}(\pi_{01}(\pi_{02}(\mathsf{R}_{02})))]$, and two for $[\pi_{12}(\pi_{01}(\mathsf{R}_{01}))]$. However, all these remainder terms in Eq. (2), need an application of π_{01} followed by an application of π_{12}. Hence, instead of separately computing these terms via multiple Shuffle-Pair instances, we club these terms together in such a way that we require only three Shuffle-Pair instances to compute $[\alpha_{\mathsf{T}_o}]$. Elaborately, given that R_{02} is held by P_0, P_2 on clear, parties can non-interactively generate its $[\cdot]$-shares. Further, given $[\alpha_{\mathsf{T}} \oplus \mathsf{R}_{02}] = [\alpha_{\mathsf{T}}] \oplus [\mathsf{R}_{02}]$, parties invoke Shuffle-Pair with π_{02} as the secret permutation to generate $[\pi_{02}(\alpha_{\mathsf{T}} \oplus \mathsf{R}_{02})]$. Since $[\mathsf{R}_{01}]$ can also be generated non-interactively, the remainder terms in (2) can be alternatively expressed as,

$$\begin{aligned}&\pi_{12}(\pi_{01}(\pi_{02}(\alpha_{\mathsf{T}}))) \oplus \pi_{12}(\pi_{01}(\pi_{02}(\mathsf{R}_{02}))) \oplus \pi_{12}(\pi_{01}(\mathsf{R}_{01})) \\ &= \pi_{12}(\pi_{01}(\pi_{02}(\alpha_{\mathsf{T}} \oplus \mathsf{R}_{02}) \oplus \mathsf{R}_{01})) = \gamma\end{aligned}$$

Hence, given $[\pi_{02}(\alpha_{\mathsf{T}} \oplus \mathsf{R}_{02}) \oplus \mathsf{R}_{01}] = [\pi_{02}(\alpha_{\mathsf{T}} \oplus \mathsf{R}_{02})] \oplus [\mathsf{R}_{01}]$, one can apply two invocations of Shuffle-Pair with π_{01}, π_{12} to generate $[\gamma]$, as required for generating $[\alpha_{\mathsf{T}_o}]$.

[10] When performing multiple shuffle instances, the cost of sending a hash can be amortized by sending a single hash for messages corresponding to multiple shuffles.

Let $[\rho] = \text{Shuffle-Pair}([\mathsf{T}], \pi_{ij})$ denote the application of π_{ij} on $[\mathsf{T}]$ to obtain $[\rho]$ where $\rho = \pi_{ij}(\mathsf{T})$ and the parties P_i, P_j are the pair who knows π_{ij} on clear. If $[\mathsf{T}_1], [\mathsf{T}_2]$ denote the input and output of a Shuffle-Pair instance, let $\text{Set-Equality}([\mathsf{T}_1], [\mathsf{T}_2])$ output $\text{flag} = 0$ if Shuffle-Pair was performed correctly, and $\text{flag} = 1$ otherwise. The steps for generating α_{T_o} is summarised in Fig. 3.

1. $[\rho_2] = \text{Shuffle-Pair}([\rho_1], \pi_{02})$ where $\rho_1 = \alpha_{\mathsf{T}} \oplus \mathsf{R}_{02}$ and $\text{flag}_{02} = \text{Set-Equality}([\rho_1], [\rho_2])$

2. $[\rho_4] = \text{Shuffle-Pair}([\rho_3], \pi_{01})$ where $\rho_3 = \rho_2 \oplus \mathsf{R}_{01}$ and $\text{flag}_{01} = \text{Set-Equality}([\rho_3], [\rho_4])$

3. $[\rho_5] = \text{Shuffle-Pair}([\rho_4], \pi_{12})$, $\text{flag}_{12} = \text{Set-Equality}([\rho_4], [\rho_5])$

4. Set $\left\lfloor \alpha_{\mathsf{T}_o} \right\rfloor = [\rho_5] \oplus [\pi_{12}(\mathsf{R}_{12})]$

Figure 3: Generation of $\left\lfloor \alpha_{\mathsf{T}_o} \right\rfloor$ by parties in $\mathcal{P}$

Guaranteeing output delivery. Note that in the solution described above, an adversary can misbehave, resulting in an `abort` (i.e., failure of shuffle). However, to attain GOD and obtain as output the randomly shuffled input table irrespective of the adversarial behaviour, one can proceed as follows. Inspired by the techniques of [9, 11, 29, 30], we rely on a trusted third party (TTP) based approach. Elaborately, if shuffle fails, we work towards identifying an honest party in $\mathcal{P}$ that is designated as a TTP. Parties robustly reconstruct the input table to TTP, who performs the shuffle operation on the clear table, and sends the output (shares of the randomly shuffled input table) to all. We next describe how a TTP can be identified in preprocessing and online phase whenever shuffle fails.

Identifying a TTP *if shuffle fails during preprocessing phase.* The preprocessing phase involves three sequential invocations of the semi-honest Shuffle-Pair protocol where in each invocation, only two parties communicate a message (see §B for details). Each invocation of Shuffle-Pair is followed by a robust Set-Equality protocol to verify the correctness of the Shuffle-Pair, which outputs a flag indicating that shuffle failed if some misbehaviour was detected in this Shuffle-Pair instance. We make the following observation that aids in identifying a TTP: If any invocation of Set-Equality outputs a flag indicating that Shuffle-Pair fails, it must be due to a misbehaviour by one of the two (communicating) parties in the corresponding Shuffle-Pair instance. This is because Set-Equality protocol is robust against any misbehaviour (owing to the use of a robust 3PC for the same), and hence, shuffle can fail only due to a misbehaviour in Shuffle-Pair. Further, since at most one among the three parties is malicious, this guarantees that the (non-communicating) residual party is honest and can be designated as the TTP.

Identifying a TTP *if shuffle fails during online phase.* Each of the three messages that are exchanged in the online phase have the following communication pattern. There exist two senders who possess the message to be sent to the receiver, where one sender sends the message while the other sends the hash of it. Since this resembles the communication pattern of [19, 29], we use the techniques therein to identify a TTP, if any party receives an inconsistent (message, hash) pair. At a high level, if the received message and hash do not match at the receiver, it broadcasts a complaint accusing the senders. It also broadcasts the received messages. This is followed by the senders broadcasting a complaint against the receiver if the latter's broadcast message was inconsistent with the senders sent message. Depending on the publicly available complaints, parties can unanimously determine a pair of parties that are in conflict with each other, one of which is guaranteed to be corrupt. Due to at most one malicious corruption among the three parties, the third party that is not a part of this conflict is guaranteed to be honest and can be designated as the TTP. The formal steps are provided in Fig. 4, and correctness follows from [19][11].

Protocol $\Pi_{\text{Ruffle}}(\llbracket \mathsf{T} \rrbracket)$

Preprocessing:

– Each pair of parties $P_i, P_j \in \mathcal{P}$ non-interactively sample $\mathsf{R}_{ij} \in \mathbb{Z}_{2^\ell}^N$ and random permutations π_{ij}.

– P_1, P_2 compute $\pi_{12}(\mathsf{R}_{12})$, and parties generate its $[\cdot]$-shares, non-interactively.

– Parties in $\mathcal{P}$ generate $[\cdot]$-shares of $\mathsf{R}_{01}, \mathsf{R}_{02}$, non-interactively.

– Parties in $\mathcal{P}$ follow the steps in Fig. 3 to generate $\left\lfloor \alpha_{\mathsf{T}_o} \right\rfloor$.

– *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):
  - P_0, P_2 compute $\delta_{02} = \pi_{02}(\beta_{\mathsf{T}} \oplus \mathsf{R}_{02})$. P_2 sends δ_{02} to P_1. P_0 sends $\mathsf{H}(\delta_{02})$ to P_1, where H is a collision-resistant hash function.
  - P_0 computes and sends $\delta_{01} = \pi_{01}(\pi_{02}(\beta_{\mathsf{T}} \oplus \mathsf{R}_{02}) \oplus \mathsf{R}_{01})$ to P_2.
- Shuffle (Round 2):
  - P_1 computes and sends $\mathsf{H}(\delta_{01}) = \mathsf{H}(\pi_{01}(\delta_{02} \oplus \mathsf{R}_{01}))$ to P_2.
  - P_1, P_2 compute $\delta_{12} = \pi_{12}(\delta_{01} \oplus \mathsf{R}_{12})$.
  - P_1 sends δ_{12} and P_2 sends $\mathsf{H}(\delta_{12})$ to P_0.
- Verification (Round 3)[a]: 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 $c_j = \mathsf{H}(x)$, 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 $\text{TTP} = P_j$.
  - 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 $\text{TTP} = P_k$. 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 $\text{TTP} = P_i$.
- 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.

[a] Note that these can be performed as soon as the messages required for detecting inconsistency are available.

Figure 4: Secure shuffle protocol

Proof of security. The security of our shuffle protocol follows the fact that the secret permutation remains hidden since each party

[11] These steps can be optimized by using the technique described in [29].

knows only two of the three permutations. Moreover, throughout the protocol, parties only receive random messages. Hence, the security follows easily. A detailed proof of security appears in §F.

5 APPLICATIONS

Here, we focus on two applications of shuffle– (i) anonymous broadcast and (ii) GraphSC paradigm. Recall that anonymous broadcast being a perpetually running system, requires a shuffle protocol that is designed to handle Independent-Shuffles. On the other hand, the GraphSC paradigm requires a shuffle protocol that can cater to Composed-Shuffles. While Ruffle was described with respect to a solitary shuffle invocation in §4.1, it can easily be extended to handle the case of Independent-Shuffles. The resulting protocol is termed as Ruffle-1. However, due to the sequential dependence present in the preprocessing phase of Ruffle-1, it does not render itself efficient for the case of Composed-Shuffles. Hence, we enhance Ruffle and design an alternative protocol Ruffle-2 that is tailor-made to handle Composed-Shuffles. We describe the applications and the respective shuffle protocols they use, next.

5.1 Anonymous broadcast

Anonymous broadcast, as the name suggests, enables a set of N clients to anonymously broadcast their messages while guaranteeing that none learns about the association between a message and the identity of its sender. Instead of requiring the clients to send their messages to a centralized server, which can output the randomly shuffled messages back to the clients, we rely on a distributed solution to guarantee client privacy. At a high level, to achieve anonymous broadcast, the clients secret-share their messages to a set of three servers (the three parties in $\mathcal{P}$, henceforth interchangeably called as servers), who invoke a secure shuffle protocol on the same and reconstruct the shuffled output. The solution must guarantee the following desirable properties when at most one server and any number of clients are corrupt.

1. *Confidentiality:* A coalition of malicious clients and server should not learn the permutation used to shuffle the messages.
2. *Integrity:* Client's message should remain intact.
3. *Security against malicious client:* System should discard malformed messages sent by malicious clients.
4. *Robustness against malicious server:* Malicious server should not abort the computation, and halt the system.
5. *Censorship resistance:* A malicious server should not be enabled to discard an honest client's message from the system.

Many works in the literature consider mix-net [17] based approach to achieve anonymous broadcast [4, 20, 31–33, 37], while others [2, 18, 21, 45] are based on DC-networks proposed in [16]. The recent work of Clarion [20] improves in terms of efficiency over these and provides an alternative shuffle-based anonymous broadcast system. Hence, [20] provides the most efficient shuffle-based anonymous broadcast system in the 3-server setting. Our protocol offers the following improvements over Clarion: (i) censorship resistance which was missing in [20], (ii) usage of a more efficient shuffle, (iii) more efficient steps for verifying consistency of client's input message, and (iv) improved security guarantee of robustness, whereas that of [20] only provides security with abort. We now describe our anonymous broadcast in the 3-server setting.

5.1.1 Our anonymous broadcast system. The protocol can be described in the following steps.

1. *Input sharing and consistency check:* Each client wanting to broadcast a message receives randomness, using which it generates $[\![\cdot]\!]$-shares of its message. On receiving shares of a client's message, servers verify if these are malformed. If so, they discard the message.
2. *Shuffle:* Assuming N messages pass the verification, servers securely shuffle the N-sized table using Ruffle-1 described in §5.1.3.
3. *Output reconstruction:* On receiving the output shares after executing Ruffle-1, servers reconstruct the shuffled table using the steps described in §3. The shuffled table is then broadcast to clients.

Since steps for output reconstruction (step 3) were already described, we next elaborate on input sharing and consistency check (step 1), and the shuffle protocol (step 2).

5.1.2 Input sharing and consistency check. This comprises a preprocessing phase and an online phase, as elaborated below.

Preprocessing phase: Let the ℓ-bit client message be denoted as m. Consider the sharing semantics of our shuffle protocol. A message m which is $[\![\cdot]\!]$-shared comprises $\beta_{\mathsf{m}} = \mathsf{m} \oplus \alpha_{\mathsf{m}}$ held by all servers in $\mathcal{P}$, and $\alpha_{\mathsf{m}} = [\alpha_{\mathsf{m}}]_{01} \oplus [\alpha_{\mathsf{m}}]_{02} \oplus [\alpha_{\mathsf{m}}]_{12}$, where $[\alpha_{\mathsf{m}}]_{ij}$ is held by $P_i, P_j \in \mathcal{P}$. To enable the client to generate $[\![\mathsf{m}]\!]$ towards the servers while minimizing the computation as well as communication at the client, we proceed on similar lines as in [29]. We let the servers generate $[\cdot]$-shares for a random α_{m}, non-interactively. Observe that each of $[\alpha_{\mathsf{m}}]_{01}, [\alpha_{\mathsf{m}}]_{02}, [\alpha_{\mathsf{m}}]_{12}$ is held by exactly two servers, at most one of which can be maliciously corrupt. Making P_i, P_j send $[\alpha_{\mathsf{m}}]_{ij}$ to the client, may lead to uncertainity at the client if P_i, P_j send different versions of $[\alpha_{\mathsf{m}}]_{ij}$ to it. Hence, to ensure correct delivery of each $[\alpha_{\mathsf{m}}]_{ij}$ towards the client, servers rely on a commitment scheme. Elaborately, each pair of servers P_i, P_j generates a commitment $\mathsf{Com}([\alpha_{\mathsf{m}}]_{ij})$ of $[\alpha_{\mathsf{m}}]_{ij}$ and jmps it to P_k (see §3 for details of jmp). This ensures that all servers are in agreement with commitments generated for each $[\cdot]$-share of α_{m}.

Online phase: For a client who wishes to send a message, each server sends $\mathsf{Com}([\alpha_{\mathsf{m}}]_{01})$, $\mathsf{Com}([\alpha_{\mathsf{m}}]_{02})$ and $\mathsf{Com}([\alpha_{\mathsf{m}}]_{12})$ to the client. The client retains the values in majority for each $\mathsf{Com}([\alpha_{\mathsf{m}}]_{ij})$. At the same time, each P_i, P_j send the opening to $\mathsf{Com}([\alpha_{\mathsf{m}}]_{ij})$ towards the client. The client accepts the opening which is consistent with the commitment that was in majority. In this way, the client receives correct $[\alpha_{\mathsf{m}}]_{01}, [\alpha_{\mathsf{m}}]_{02}, [\alpha_{\mathsf{m}}]_{12}$. Upon receiving these values, the client generates and sends $\beta_{\mathsf{m}} = \mathsf{m} \oplus [\alpha_{\mathsf{m}}]_{01} \oplus [\alpha_{\mathsf{m}}]_{02} \oplus [\alpha_{\mathsf{m}}]_{12}$ to the servers. To ensure that each server receives the same β_{m} and guarantee that a client has not misbehaved, each server broadcasts the β_{m} received from the client. If there is a majority among the broadcast values, the client's message is accepted, and each server sets its β_{m} to be the majority value. Else, the client's message is deemed as malformed and discarded from the instance of anonymous broadcast protocol.

5.1.3 Ruffle-1. Ruffle protocol described in §4.1, was for a single invocation of shuffle. For the scenario of Independent-Shuffles, where m independent shuffles are required to be performed sequentially, we design Ruffle-1 as follows. In its preprocessing phase, Ruffle-1 performs m instances of the preprocessing of Ruffle in parallel, whereas for its online phase, it sequentially executes the

online phase of Ruffle m times. As seen in Table 1, Ruffle-1 has a better complexity than that of [6, 20].

A comparison of the concrete cost of our anonymous broadcast with Clarion, together with a justification of how our system attains the desirable properties, appears in §C.

5.2 Secure graph computation

The GraphSC paradigm [43] expresses any graph algorithm as a message-passing algorithm. The graph is stored as list of nodes and edges, where each entry in the list is associated with data or state. One round of the message passing algorithm involves updating the state of the nodes in the graph via the primary operations of Scatter and Gather, which realise sending and receiving messages across the edges, respectively. To ensure data obliviousness, before each Scatter/Gather operation can be invoked, the list representing the graph must be securely sorted. [6] takes a step towards improving the performance by replacing every secure sort operation with a secure shuffle. Thus, the entire graph algorithm reduces to performing shuffles and invocations of the Scatter and Gather operations across multiple rounds ($O(|\mathsf{V}|+|\mathsf{E}|)$). Note that shuffling in each round takes as input the result obtained in the previous iteration to update the state of the nodes. Hence, these sequential shuffles performed in different rounds indicate the scenario of Composed-Shuffles. As will be explained next, since Ruffle-1 results in having a prohibitively high complexity when employed for the case of Composed-Shuffles, we design Ruffle-2 that is tailor-made for this scenario. A detailed explanation of the GraphSC paradigm along with the representative use case of BFS algorithm is given in §D. We next describle Ruffle-2.

5.2.1 Ruffle-2. For scenarios that demand the composition of, say m, shuffles (i.e., Composed-Shuffles), observe that the preprocessing phase of Ruffle-1, which comprises m instances of the preprocessing of Ruffle, can no longer execute in parallel, but will have to be performed sequentially. This is because in Composed-Shuffles, the output of one shuffle operation, say T_1, constitutes the input to a subsequent shuffle operation, which say outputs $\mathsf{T}_2 = \pi(\mathsf{T}_1)$. Hence, once α_{T_1} is generated as output from the first (preprocessing phase) instance of Ruffle, only then can $\alpha_{\mathsf{T}_2} = \pi\left(\alpha_{\mathsf{T}_1}\right) \oplus \mathsf{R}$ be generated (see §4.1 for definition of α_{T}). This sequential dependency present in the preprocessing phase of Ruffle-1 when deployed in the case of Composed-Shuffles, makes its run time proportional to the number of sequential shuffles. However, it is desirable to facilitate generation of necessary preprocessing data in parallel and hence, decouple the dependency between generation of preprocessing data and pattern of shuffle invocations. This can aid in significantly reducing preprocessing phase's cost. Hence, in the following, we design an alternative protocol Ruffle-2, that breaks this dependence and is tailor-made to handle Composed-Shuffles.

Let T be the input table which has to be shuffled to obtain $\mathsf{T}_o = \pi(\mathsf{T})$. In Ruffle (Fig. 4), $\mathsf{T}_o = \beta_{\mathsf{T}_o} \oplus \alpha_{\mathsf{T}_o}$ where $\beta_{\mathsf{T}_o} = \pi(\beta_{\mathsf{T}}) \oplus \mathsf{R}$, $\alpha_{\mathsf{T}_o} = \pi(\alpha_{\mathsf{T}}) \oplus \mathsf{R}$, and $\pi = \pi_{12} \circ \pi_{01} \circ \pi_{02}$, $\mathsf{R} = \pi(\mathsf{R}_{02}) \oplus \pi_{12}(\pi_{01}(\mathsf{R}_{01})) \oplus \pi_{12}(\mathsf{R}_{12})$. To break the dependency and ensure that α_{T_o} can be generated independently of α_{T} we proceed as follows. Let $\alpha'_{\mathsf{T}_o}, \beta'_{\mathsf{T}_o}$ be the newly defined values such that $\mathsf{T}_o = \alpha'_{\mathsf{T}_o} \oplus \beta'_{\mathsf{T}_o}$, where α'_{T_o} is the decoupled equivalent of α_{T_o}. We let parties non-interactively sample $[\cdot]$-shares of a random $\alpha'_{\mathsf{T}_o} \in \mathbb{Z}^N_{2^\ell}$ during preprocessing. Having generated α'_{T_o} this way, we need to define β'_{T_o} to ensure that $\mathsf{T}_o = \beta'_{\mathsf{T}_o} \oplus \alpha'_{\mathsf{T}_o}$ holds. Hence, we define $\beta'_{\mathsf{T}_o} = \pi(\beta_{\mathsf{T}}) \oplus \mathsf{R} \oplus \pi(\alpha_{\mathsf{T}}) \oplus \mathsf{R} \oplus \alpha'_{\mathsf{T}_o}$. This is because recall that $\mathsf{T}_o = \pi(\beta_{\mathsf{T}}) \oplus \mathsf{R} \oplus \pi(\alpha_{\mathsf{T}}) \oplus \mathsf{R}$ (where R serves as a random mask for $\pi(\beta_{\mathsf{T}})$). We next describe how to generate this β'_{T_o}.

Let $\beta'_{\mathsf{T}_o} = \mathsf{B}_1 \oplus \mathsf{B}_2$, where $\mathsf{B}_1 = \pi(\beta_{\mathsf{T}}) \oplus \mathsf{R}$ and $\mathsf{B}_2 = \pi(\alpha_{\mathsf{T}}) \oplus \mathsf{R} \oplus \alpha'_{\mathsf{T}_o}$. In the preprocessing phase, observe that $[\cdot]$-shares of $\pi(\alpha_{\mathsf{T}}) \oplus \mathsf{R}$ can be generated as described in §4.1. Thus, parties can compute $[\mathsf{B}_2] = [\pi(\alpha_{\mathsf{T}}) \oplus \mathsf{R}] \oplus \left[\alpha'_{\mathsf{T}_o}\right]$ and reconstruct B_2 towards all parties. In the online phase, to generate β'_{T_o}, observe that parties can generate B_1, as described in §4.1. Given B_2 generated during preprocessing, parties set $\beta'_{\mathsf{T}_o} = \mathsf{B}_1 \oplus \mathsf{B}_2$. This completes the generation of $[\![\mathsf{T}_o]\!]$. In comparison to Ruffle-1 this protocol, Ruffle-2, only requires an additional reconstruction of B_2 (for each shuffle instance) towards all the parties during the preprocessing phase.

In summary, when dealing with Composed-Shuffles, the α and β values need to be redefined, and the computation proceeds as follows. Assuming that we are interested in computing the output shuffled table T_n defined as $\mathsf{T}_n = \pi_n(\pi_{n-1}(\ldots \pi_1(\mathsf{T}_0)))$ and T_0 is $[\![\cdot]\!]$-shared, let each intermediate shuffled table $\mathsf{T}_i = \pi_i(\ldots \pi_1(\mathsf{T}_0))$. As per the newly defined values, $\mathsf{T}_i = \alpha'_{\mathsf{T}_i} \oplus \beta'_{\mathsf{T}_i}$, for each $i \in \{1, \ldots, n\}$. Each of the α'_{T_i} are randomly sampled and hence $[\cdot]$-shares of each $\pi_i\left(\alpha'_{\mathsf{T}_{i-1}}\right) \oplus \mathsf{R}_i$ can be generated in parallel. Thus, the B_2 component of each β'_{T_i} can be computed as $[\mathsf{B}_2] = \left[\pi_i\left(\alpha'_{\mathsf{T}_{i-1}}\right) \oplus \mathsf{R}\right] \oplus \left[\alpha'_{\mathsf{T}_i}\right]$ in parallel during the preprocessing phase and reconstructed towards all the parties. Each β'_{T_i} is generated sequentially in the online phase by computing $\mathsf{B}_1 = \pi_i\left(\beta'_{\mathsf{T}_{i-1}}\right) \oplus \mathsf{R}_i$ and $\beta'_{\mathsf{T}_i} = \mathsf{B}_1 \oplus \mathsf{B}_2$.

6 BENCHMARKS

We empirically evaluate performance of our shuffle protocols under various parameters and application scenarios, and compare them against their state-of-the-art counterparts.

Benchmark environment. Benchmarking is performed over LAN using n1-standard instances of Google Cloud with 2.3 GHz Intel Xeon E5 v3 (Haswell) processors, and 240 GB of RAM. The machines have a bandwidth of 16Gbps. For a fair comparison, we implement all the protocols in python, including that of [20] and [6]. Thus, costs reported for prior works are higher than that reported in the original works (see §E). Hence, the concrete improvements over [20] and [6] reported next capture the relative improvements with respect to the underlying protocols. That is, we do not account for the system level optimizations that may have been included as a part of the implementations in the original works of [20] and [6]. However, we note that the reported communication costs are invariant of the implementation. Our code accounts for multi-threading with 64 threads. We instantiate the communication layer between the parties using PyTorch library. We use Crypto library for AES and hashlib for generating SHA256 hash. We note that our code[12] is developed for benchmarking, is not optimized for industry-grade use, and a C++ based implementation can give better performance.

[12] https://github.com/Bhavishrg/Ruffle

Benchmark parameters. We follow the standard practice and benchmark honest executions (with verification) as done in [10, 29, 39, 47]. We consider run time and communication of protocols as the parameters for comparison. We account for online as well as total (preprocessing + online) cost when doing so. To capture the combined effect of both these parameters, we additionally report online throughput (TP).

6.1 Shuffle

We begin by comparing Ruffle to the shuffle protocol of [20] and [6] for the case of a single invocation of shuffle. Table 2 reports the online phase comparisons to capture the fast response time and the communication involved. Observe that Ruffle clearly outperforms both [6, 20]. Concretely, we observe improvements up to 15× in run time and 2.5× in communication over [20]. When compared to [6], Ruffle has an improvement of up to 11.2× in run time and 2.5× in communication. The improvements in the run time and communication are reflected in a high throughput, which captures the number of such single invocations that can be performed in parallel. The improvements in throughput range up to 5.5× and 2.2× with respect to [20] and [6], respectively. When considering the overall cost, we note that Ruffle fares better than [20] but is slightly higher, yet comparable, to that of [6]. We report this in Table 3 for completeness. We remark that Ruffle-1 has same complexity as Ruffle for the case of a single shuffle, while Ruffle-2 is not apt for single shuffle invocation due to its higher preprocessing cost.

|T|	Protocol	Time (s)	Comm. (MB)	TP (per min)
10^3	Ruffle	0.005	0.092	12000.000
	[6]	0.056	0.231	5415.162
	[20]	0.075	0.228	2181.421
10^4	Ruffle	0.046	0.915	1200.000
	[6]	0.434	2.318	593.589
	[20]	0.718	2.289	218.177
10^5	Ruffle	0.457	9.155	120.000
	[6]	3.959	22.918	59.935
	[20]	7.692	22.888	21.818
10^6	Ruffle	7.033	91.553	12.000
	[6]	49.577	228.912	5.999
	[20]	95.089	228.881	2.181

Table 2: Online complexity of shuffle for varying table sizes for a single shuffle invocation.

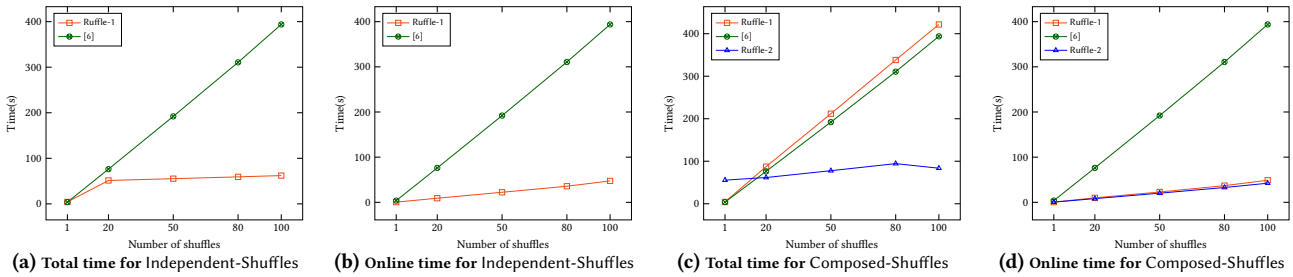
To capture the improvements of Ruffle-1 and Ruffle-2, we benchmark their performance for multiple sequential shuffle invocations, i.e., scenarios of Independent-Shuffles and Composed-Shuffles. Recall that Ruffle-1 is apt for Independent-Shuffles while Ruffle-2 for Composed-Shuffles. Since [6] outperforms [20] (as evident from Table 2 and Table 3), we restrict to comparing Ruffle-1 and Ruffle-2 in their respective settings against [6]. Further, to capture improvements Ruffle-2 protocol brings over Ruffle-1, we also report the cost for performing Ruffle-1 in the scenario of Composed-Shuffles. The

|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

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

comparison for varying number of shuffle invocations is reported in Fig. 5 (and Table 4). We make the following observations:

- The cost of [6] remains the same for Independent-Shuffles and Composed-Shuffles since it is indifferent to both.
- We infer the following with respect to the online complexity. Irrespective of the scenario and the number of shuffle invocations, recall from Table 1 that Ruffle-1 and Ruffle-2 are comparable since their online phase is same, except for the extra computation required in Ruffle-2. Hence, as expected, Ruffle-1 (and thereby Ruffle-2) outperforms [6] by up to 10×.
- We infer the following with respect to the overall run time. For a single shuffle invocation, both Ruffle-1 (i.e. Ruffle for $m = 1$) and Ruffle-2 have a slightly higher run time than [6]. However, starting from as low as two invocations, Ruffle-1 begins to outperform [6] for Independent-Shuffles. This is justified as follows– since Ruffle-1's online phase is faster than that of [6], performing the preprocessing for m shuffles in parallel results in improving the overall complexity. This improvement is not seen in [6] since the m shuffles that can be performed in parallel during our preprocessing are required to be performed sequentially in case of [6] which adds to the overhead. We see improvements of up to 6.4× in this case. On the other hand, for Composed-Shuffles, [6] continues to outperform Ruffle-1 for the following reason. The composition of shuffles induces a sequential nature in the preprocessing phase of Ruffle-1 (which indeed is the complete protocol of [6]). The computations performed additionally in the online phase of Ruffle-1 renders its overall complexity slightly higher than that of [6]. To break this chain of sequential shuffles in the preprocessing phase, Ruffle-2 was designed to outperform Ruffle-1 (and thereby [6]), where we see improvements of up to 4.7× with respect to [6].
- To capture the effect of both total run time and total communication, we additionally report the monetary cost in Table 4, which is the price paid for performing the secure shuffle computation. This



(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 5: 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** 10^5.

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

Table 4: 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** 10^5. **Note that the cost of [6] remains the same for both the scenarios.**

is calculated using the pricing of Google Cloud Platform[13], where for 1GB and 1 hour of usage, the costs are USD 0.12 and USD 3.3025 respectively. With respect to the monetary cost, we note that for a small number of shuffle invocation($\leq$ 25), our protocols have a slightly higher monetary cost in comparison to [6]. However, as the number of shuffle invocation increases($>$ 25), the savings in run time see in our shuffle protocols compensates for the increased communication. Thus, both Ruffle-1, Ruffle-2 outperform [6] in terms of monetary cost.

[13]We refer https://cloud.google.com/vpc/network-pricing for network cost and https://cloud.google.com/compute/vm-instance-pricing for computation cost.

6.2 Anonymous broadcast

We empirically compare our anonymous broadcast system presented in §5.1 (instantiated with Ruffle-1) with the most efficient shuffle-based 3-server system in [20]. Since complexity of anonymous broadcast varies based on number of clients (N) as well as their message size, we compare with respect to these parameters.

When varying the number of clients, we analyze the server-side complexity and report the performance in Table 5 which accounts for–checking the consistency of clients' message (32 bytes in size) where the check is performed in parallel for N clients, shuffling the

N-sized table, and reconstruction of the shuffled result. Observe from Table 5 that our anonymous broadcast system outperforms [20] in every aspect. This can be attributed not only to the use of our efficient shuffle protocol but also due to the simplicity of the input sharing and consistency check, and output reconstruction. On the other hand, [20] relies on several MAC verifications and encryption operations which render the system of [20] less efficient. The improvements we observe with respect to online and total time is up to 29× and 13× respectively, whereas that of online and total communication is up to 2×.

The effect of varying the client message size on run time and communication with respect to the server is reported in Table 6. Our system outperforms [20] in terms of both. Concretely, with respect to online and total time, we see improvements up to 39× and 9×, respectively. With respect to online and total communication, we see improvements up to 1.2× and 1.3×.

Table 5 and Table 6 do not account for the time/communication required to share a client's input. Hence, to showcase the overhead of input sharing, on both the client and the server, we report the costs in Table 7. Since this overhead is dependent on the client message size, Table 7 also account for the same. Recall that our system additionally requires the client to wait to receive the preprocessing data from the server. Despite this, the time for which a client has to remain online in our system is 18× lesser in comparison to [20]. The higher cost of [20] can be attributed to the need for PRG (pseudorandom generator) invocations, encryption of message followed by MAC tag computation at the client. This is unlike our system, which relies on simple operations such as XOR. On the other hand, since [20] requires the clients to communicate to only two servers instead of the three servers as required in our case, they have lesser communication. The reduced time a client has to remain online comes at the cost of server-to-client communication, which is absent in [20]. This we note is a small price paid. Thus, we note that our realization of the anonymous broadcast system not only provides improved efficiency but also offers censorship resistance and allows attaining the improved security of GOD.

N	Anonymous broadcast	Online		Total	
		Time (s)	Comm. (MB)	Time (s)	Comm. (MB)
10^3	Ours	0.01	0.36	0.09	0.62
	[20]	0.20	0.76	1.11	1.23
10^4	Ours	0.06	3.66	0.69	5.97
	[20]	1.88	7.63	10.19	12.34
10^5	Ours	0.61	36.62	6.73	59.53
	[20]	20.59	76.29	105.88	123.59
10^6	Ours	8.64	366.21	107.52	595.12
	[20]	248.99	762.94	1082.53	1235.96

Table 5: Comparison of online run time and communication of servers for varying number of clients and message size of 32 bytes.

6.3 Secure graph computation

We benchmark the application of BFS as described in §D via the GraphSC paradigm of [6]. We rely on the robust MPC framework

Message Size	Anonymous broadcast	Online		Total	
		Time (s)	Comm. (MB)	Time (s)	Comm. (MB)
32B	Ours	0.64	36.62	6.75	59.53
	[20]	20.59	76.29	104.55	123.59
160B	Ours	1.40	183.12	11.47	279.25
	[20]	49.97	247.96	135.96	395.51
1KB	Ours	6.97	1145.14	66.10	1721.21
	[20]	269.26	1373.29	553.28	2224.16

Table 6: Comparison of online run time and communication of servers for varying message size and clients of $N = 10^5$.

Message Size	Anonymous broadcast	Client time (ms)	Client-server Communication (KB)	Server-client Communication (KB)
32B	Ours	0.13	0.09	0.47
	[20]	2.34	0.16	-
160B	Ours	0.12	0.47	1.22
	[20]	6.05	0.41	-
1KB	Ours	1.74	3.00	6.97
	[20]	30.55	2.06	-

Table 7: Comparison of client-side and server-side complexity for input sharing by one client. %vspace-10mm

of SWIFT [29] wherever needed. To overcome the linear dependence on the size of the input graph, we cast our secure evaluation of BFS in the multiprocessor setting as described in [43]. This allows us to obtain a solution which has a round complexity of $O\left(\frac{|V|+|E|}{P} + \log(|V| + |E|)\right)$, where P is the number of processors, using the parallel variants of Scatter and Gather primitives as described in [43]. The formal details of the Scatter and Gather primitives required for the specific case of BFS is described in §D. Since GraphSC paradigm requires the composition of shuffles (Composed-Shuffles), we implement BFS using Ruffle-2. The improvements that Ruffle-2 brings over the shuffle of [6] has already been established in Table 4. We now compare our implementation of BFS that uses Ruffle-2 with the BFS implementation of [6] and we report it in Table 8. We see improvements up to 11.5× in the online run time in comparison to [6], while also outperforming in terms of the total run time. Note that these improvements were observed in the 32-processor setting. Since the number of processors affects the run time, we estimate the cost for varying number of processors, and the same appears in Fig. 6.

7 CONCLUSION

We design secure shuffle protocols which not only provide a fast online phase but also improve on the overall run time when considering two or more sequential shuffle invocations. We showcase the significant improvements that arise when using our secure shuffle protocols in the application of anonymous broadcast and secure BFS computation via GraphSC paradigm, and thereby provide a solution that improves over the respective state-of-the-art works.

# message passing rounds	Protocol	Online		Total	
		Time (min)	Comm. (MB)	Time (min)	Comm. (MB)
1	Ours	8.50	24.32	73.66	1394.17
	[6]	26.81	79.25	98.88	1394.17
2	Ours	8.70	31.65	93.33	2627.86
	[6]	50.60	141.50	126.81	2627.86
3	Ours	8.78	38.97	105.37	3861.56
	[6]	73.37	203.77	172.01	3861.56
4	Ours	8.91	46.30	115.94	5095.25
	[6]	93.51	266.03	194.38	5095.25
5	Ours	8.98	53.62	135.36	6328.95
	[6]	103.64	328.28	230.39	6328.95

Table 8: Comparison of BFS using our shuffle and the shuffle of [6] for a graph of size 10^4 (= $|\mathcal{L}|$) in 32 processors setting.

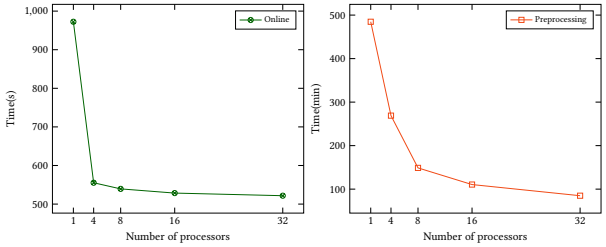


Figure 6: Online(left) and preprocessing(right) time of BFS on a graph of size 10^4, when varying the number of processors for 1 message passing round.

With secure shuffle being an integral part of various other applications, it would be interesting to see how our shuffle protocols can be used to bring about improvements therein. Since applications may demand a varying number of parties, going ahead, we believe it is an important question to design secure shuffle protocols for the arbitrary n-party setting. A naive extension of our shuffle protocols to the n-party setting would result exponential blow-up in the number of permutations held at every party. This would in turn affect the communication and round complexity adversely. Hence, the challenge would be to circumvent the above issues and design efficient solutions for n-party setting.

ACKNOWLEDGMENTS

Arpita Patra, Varsha Bhat Kukkala, Nishat Koti and Bhavish Raj Gopal would like to acknowledge financial support from National Security Council, India. The authors also acknowledge the support from Google Cloud for benchmarking.

REFERENCES

[1] Masayuki Abe. 1998. Universally Verifiable Mix-net with Verification Work Independent of the Number of Mix-servers. In *EUROCRYPT*.
[2] Ittai Abraham, Benny Pinkas, and Avishay Yanai. 2020. Blinder–Scalable, Robust Anonymous Committed Broadcast. In *ACM CCS*.
[3] Ben Adida and Douglas Wikström. 2007. How to shuffle in public. In *TCC*.
[4] Nikolaos Alexopoulos, Aggelos Kiayias, Riivo Talviste, and Thomas Zacharias. 2017. MCMix: Anonymous Messaging via Secure Multiparty Computation. In *USENIX Security*.
[5] Bar Alon, Eran Omri, and Anat Paskin-Cherniavsky. 2020. MPC with Friends and Foes. In *CRYPTO*.
[6] Toshinori Araki, Jun Furukawa, Kazuma Ohara, Benny Pinkas, Hanan Rosemarin, and Hikaru Tsuchida. 2021. Secure Graph Analysis at Scale. In *ACM CCS*.
[7] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. 2020. Optorama: Optimal oblivious ram. In *ASIACRYPT*.
[8] Stephanie Bayer and Jens Groth. 2012. Efficient Zero-Knowledge Argument for Correctness of a Shuffle. In *EUROCRYPT*.
[9] Elette Boyle, Niv Gilboa, Yuval Ishai, and Ariel Nof. 2019. Practical Fully Secure Three-Party Computation via Sublinear Distributed Zero-Knowledge Proofs. In *ACM CCS*.
[10] Megha Byali, Harsh Chaudhari, Arpita Patra, and Ajith Suresh. 2020. FLASH: Fast and Robust Framework for Privacy-preserving Machine Learning. *PETS* (2020).
[11] Megha Byali, Arun Joseph, Arpita Patra, and Divya Ravi. 2018. Fast Secure Computation for Small Population over the Internet. In *ACM CCS*.
[12] T-H Hubert Chan, Jonathan Katz, Kartik Nayak, Antigoni Polychroniadou, and Elaine Shi. 2018. More is less: Perfectly secure oblivious algorithms in the multi-server setting. In *ASIACRYPT*.
[13] Melissa Chase, Esha Ghosh, and Oxana Poburinnaya. 2020. Secret-shared shuffle. In *ASIACRYPT*.
[14] Harsh Chaudhari, Ashish Choudhury, Arpita Patra, and Ajith Suresh. 2019. ASTRA: High Throughput 3PC over Rings with Application to Secure Prediction. In *ACM CCSW@CCS*.
[15] Harsh Chaudhari, Rahul Rachuri, and Ajith Suresh. 2020. Trident: Efficient 4PC Framework for Privacy Preserving Machine Learning. In *NDSS*.
[16] David Chaum. 1988. The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of Cryptology* (1988).
[17] David L. Chaum. 1981. Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms. *Communications of ACM* (1981).
[18] Henry Corrigan-Gibbs, Dan Boneh, and David Mazières. 2015. Riposte: An anonymous messaging system handling millions of users. In *IEEE SP*.
[19] Anders Dalskov, Daniel Escudero, and Marcel Keller. 2020. Fantastic Four: Honest-Majority Four-Party Secure Computation With Malicious Security. In *USENIX Security*.
[20] Saba Eskandarian and Dan Boneh. 2022. Clarion: Anonymous Communication from Multiparty Shuffling Protocols. In *NDSS*.
[21] Saba Eskandarian, Henry Corrigan-Gibbs, Matei Zaharia, and Dan Boneh. 2021. Express: Lowering the cost of metadata-hiding communication with cryptographic privacy. In *USENIX Security*.
[22] Philippe Golle and Ari Juels. 2004. Parallel Mixing. In *ACM CCS*.
[23] Michael T. Goodrich and Michael Mitzenmacher. 2011. Privacy-Preserving Access of Outsourced Data via Oblivious RAM Simulation. In *ICALP*.
[24] Thomas Haines, Rajeev Goré, and Bhavesh Sharma. 2021. Did you mix me? formally verifying verifiable mix nets in electronic voting. In *IEEE SP*.
[25] Koki Hamada, Dai Ikarashi, Koji Chida, and Katsumi Takahashi. 2014. Oblivious radix sort: An efficient sorting algorithm for practical secure multi-party computation. *Cryptology ePrint Archive* (2014).
[26] Koki Hamada, Ryo Kikuchi, Dai Ikarashi, Koji Chida, and Katsumi Takahashi. 2012. Practically Efficient Multi-party Sorting Protocols from Comparison Sort Algorithms. In *ICISC*.
[27] Markus Jakobsson, Ari Juels, and Ronald L Rivest. 2002. Making mix nets robust for electronic voting by randomized partial checking. In *USENIX Security*.
[28] Marcel Keller and Peter Scholl. 2014. Efficient, Oblivious Data Structures for MPC. In *ASIACRYPT*.
[29] Nishat Koti, Mahak Pancholi, Arpita Patra, and Ajith Suresh. 2021. SWIFT: Superfast and Robust Privacy-Preserving Machine Learning. In *USENIX Security*.
[30] Nishat Koti, Arpita Patra, Rahul Rachuri, and Ajith Suresh. 2022. Tetrad: Actively Secure 4PC for Secure Training and Inference. In *NDSS*.
[31] Albert Kwon, Henry Corrigan-Gibbs, Srinivas Devadas, and Bryan Ford. 2016. Atom: Scalable Anonymity Resistant to Traffic Analysis. In *SOSP*.
[32] Albert Kwon, David Lazar, Srinivas Devadas, and Bryan Ford. 2016. Riffle: An Efficient Communication System With Strong Anonymity. *PoPETs* (2016).
[33] Albert Kwon, David Lu, and Srinivas Devadas. 2020. XRD: Scalable Messaging System with Cryptographic Privacy. In *USENIX NSDI*.
[34] Peeter Laud. 2021. Linear-Time Oblivious Permutations for SPDZ. In *CANS*.
[35] Sven Laur, Jan Willemson, and Bingsheng Zhang. 2011. Round-efficient oblivious database manipulation. In *ISC*.
[36] Yehuda Lindell. 2017. How to Simulate It - A Tutorial on the Simulation Proof Technique. In *Tutorials on the Foundations of Cryptography*.
[37] Donghang Lu, Thomas Yurek, Samarth Kulshreshtha, Rahul Govind, Aniket Kate, and Andrew Miller. 2019. HoneyBadgerMPC and AsynchroMix: Practical Asynchronous MPC and Its Application to Anonymous Communication. In *ACM CCS*.
[38] Dhaneshwar Mardi, Surbhi Tanwar, and Jaydeep Howlader. 2021. Multiparty protocol that usually shuffles. *Security and Privacy* (2021).
[39] Payman Mohassel and Peter Rindal. 2018. ABY^3: A Mixed Protocol Framework for Machine Learning. In *ACM CCS*.

[40] Payman Mohassel, Mike Rosulek, and Ye Zhang. 2015. Fast and Secure Three-party Computation: The Garbled Circuit Approach. In *ACM CCS*.
[41] Ben Morris. 2008. The mixing time of the Thorp shuffle. *SIAM J. Comput.* (2008).
[42] Mahnush Movahedi, Jared Saia, and Mahdi Zamani. 2015. Secure Multi-Party Shuffling. In *SIROCCO*.
[43] Kartik Nayak, Xiao Shaun Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine Shi. 2015. GraphSC: Parallel secure computation made easy. In *IEEE S&P*.
[44] C Andrew Neff. 2001. A verifiable secret shuffle and its application to e-voting. In *ACM CCS*.
[45] Zachary Newman, Sacha Servan-Schreiber, and Srinivas Devadas. 2022. Spectrum: High-bandwidth Anonymous Broadcast. In *USENIX NSDI*.
[46] Arpita Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. 2021. ABY2.0: Improved Mixed-Protocol Secure Two-Party Computation. In *USENIX Security*.
[47] Arpita Patra and Ajith Suresh. 2020. BLAZE: Blazing Fast Privacy-Preserving Machine Learning. In *NDSS*.
[48] Jean-François Raymond. 2001. Traffic analysis: Protocols, attacks, design issues, and open problems. In *Designing privacy enhancing technologies*.
[49] Fatemeh Shirazi, Elena Andreeva, Markulf Kohlweiss, and Claudia Díaz. 2017. Multiparty Routing: Secure Routing for Mixnets. *ArXiv* abs/1708.03387 (2017).
[50] Fatemeh Shirazi, Milivoj Simeonovski, Muhammad Rizwan Asghar, Michael Backes, and Claudia Diaz. 2018. A survey on routing in anonymous communication protocols. *Comput. Surveys* (2018).
[51] Ajith Suresh. 2021. MPCLeague: Robust MPC Platform for Privacy-Preserving Machine Learning. *arXiv preprint arXiv:2112.13338* (2021).
[52] Douglas Wikström. 2004. A Universally Composable Mix-Net. In *TCC*.

A PRELIMINARIES

Shared key setup. Let $F : \{0,1\}^\kappa \times \{0,1\}^\kappa \rightarrow X$ be a pseudo-random function (PRF), with $X = \mathbb{Z}_{2^\ell}$. To enable parties to sample common random values non-interactively, the following keys (for the PRF F) are established between the parties: each pair of parties $P_i, P_j \in \mathcal{P}$ know a common k_{ij}, and all parties in $\mathcal{P}$ know $k_{\mathcal{P}}$. P_i, P_j can now sample a common value $r \in \mathbb{Z}_{2^\ell}$, non-interactively, by computing $F_{k_{ij}}(id_{ij})$. Here, id_{ij} denotes a counter maintained by P_i, P_j, which is updated after every PRF invocation. The ideal functionality for robustly establishing these common keys among parties, as described in [29], appears in Fig. 7.

Functionality $\mathcal{F}_{\mathsf{setup}}$

$\mathcal{F}_{\mathsf{setup}}$ interacts with the parties in $\mathcal{P}$ and the adversary $\mathcal{S}$. $\mathcal{F}_{\mathsf{setup}}$ 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

- $\mathsf{y}_x = (k_{01}, k_{02}$ and $k_{\mathcal{P}})$ when $P_x = P_0$.
- $\mathsf{y}_x = (k_{01}, k_{12}$ and $k_{\mathcal{P}})$ when $P_x = P_1$.
- $\mathsf{y}_x = (k_{02}, k_{12}$ and $k_{\mathcal{P}})$ when $P_x = P_2$.

Output: Send $(\text{Output}, \mathsf{y}_x)$ to every $P_x \in \mathcal{P}$.

Figure 7: Ideal functionality for shared-key setup

Collision resistant hash function. Let $\mathsf{H} : \mathcal{K} \times \mathcal{L} \rightarrow \mathcal{Y}$ be a hash function family. 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 $\mathsf{negl}(\cdot)$ such that $\Pr[(x_1, x_2) \leftarrow \mathcal{A}(k) : (x_1 \neq x_2) \wedge \mathsf{H}_k(x_1) = \mathsf{H}_k(x_2)] \leq \mathsf{negl}(\kappa)$, where $x_1, x_2 \in \mathcal{L}$.

Commitment scheme. Let $\mathsf{Com}(x)$ denote the commitment of a value x. The commitment scheme $\mathsf{Com}(x)$ possesses two properties; *hiding* and *binding*. The former ensures privacy of the value x given its commitment $\mathsf{Com}(x)$, while the latter prevents a corrupt party from opening the commitment to a different value $x' \neq x$. Note that providing an incorrect opening for a commitment results in outputting a $\perp$. We instantiate a commitment scheme (as described in [40]) is via a hash function $\mathsf{H}(\cdot)$ given below, whose security can be proved in the random-oracle model– the commitment is defined as $\mathsf{Com}(x; r) = \mathsf{H}(x||r)$, and its opening is defined as $(x||r)$.

Joint message passing. The robust protocol for Π_{jmp} appears in Fig. 8. We refer the reader to [19] for further details.

We note that at any point during the run of the protocol if invocation of Π_{jmp} results in identifying a TTP, parties do not execute the rest of the protocol steps, and the remainder computation is carried out by the TTP who guarantees the delivery of the correct function output to all parties. Elaborately, on identifying a TTP, all parties send their inputs on clear to the TTP, who evaluates the necessary function on these clear inputs. It then sends the computed output to all the parties. In this way, reliance on Π_{jmp} to send protocol messages ensures that if any malicious party misbehaves to prevent the parties from obtaining the output, a TTP is identified and the output is now obtained via the TTP. We remark that most protocols in the 3-party setting that guarantee output delivery follow the TTP-based approach [9, 11, 29].

Protocol $\Pi_{\mathsf{jmp}}(\mathsf{v}, P_i, P_j, P_k)$

Send Phase: P_i sends $\mathsf{msg}_i = \mathsf{v}$ to P_k.
Verify Phase: P_j sends $\mathsf{msg}_j = \mathsf{H}(\mathsf{v})$ 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.

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

Figure 8: Joint Message Passing

B SHUFFLE PROTOCOL OF [6]

The shuffle protocol of [6] requires three invocations of Shuffle-Pair, each of which is followed by a Set-Equality protocol to verify the correctness of Shuffle-Pair. In the following, we first provide the Shuffle-Pair protocol followed by the Set-Equality protocol.

Shuffle-Pair. Let table T be $[\cdot]$-shared. Shuffle-Pair enables parties to generate $[\mathsf{T}']$ where $\mathsf{T}' = \pi_{ij}(\mathsf{T})$ and π_{ij} is a random permutation held by $P_i, P_j \in \mathcal{P}$. We describe the protocol with respect to P_0, P_1 who hold the permutation π_{01} in Fig. 9. The protocol for the other pair of parties can be worked out analogously.

Set-Equality. The input for the Set-Equality protocol is a pair of tables $(\mathsf{T}, \mathsf{T}')$, each comprising of n rows and an m-bit string in each row. Thus, the table comprises n rows and m columns. Here, T' is obtained by randomly shuffling T. The protocol returns as output a 1, if the two tables are different and a 0 otherwise. The protocol chooses random subsets of rows and columns, and verifies that the bits in the intersection of the chosen rows and columns are the same for both the underlying tables. This verification is performed κ times to ensure a low probability of cheating. In order

to choose the subset of rows randomly, each row of T is extended by κ random and secret bits before shuffle. Consequently, after the shuffle, every row in T′ has the same $\kappa-$sized suffix which it had in T. The rows for the ℓth test is picked based on the bit in the ℓth column of the κ-sized extension, with the row being chosen if the corresponding bit is a 1. Let R_ℓ be the n bit vector that denotes this selection of rows (i.e., $m+\ell$th column of the table). Similarly, let C_ℓ be the $m' = m + \kappa$ bit public vector that denotes the choice of columns picked for the ℓth test. The verification test compares the XOR of the values in the intersection of the chosen rows and columns in T and T′ to check for the correctness of the shuffle. This is captured by the operations performed as described in Eq. (3), where $C_{j,\ell}$ denotes the jth component of the vector C_ℓ. The output of the check is a bit V_ℓ such that if the tables are different, some V_ℓ, for $\ell \in \{1, \ldots, \kappa\}$, is non-zero with high probability. To detect if any V_ℓ is non-zero, a flag is computed which is the OR of all these V_ℓ's, followed by reconstructing flag. A non-zero flag indicates a misbehaviour in the Shuffle-Pair instance for which Set-Equality is performed.

$$[V_\ell] = \sum_{i=1}^{n} [\mathsf{T}'_{i,m+\ell}] \cdot \sum_{j=1}^{m'} C_{j,\ell} \cdot [\mathsf{T}'_{i,j}] - \sum_{i=1}^{n} [\mathsf{T}_{i,m+\ell}] \cdot \sum_{j=1}^{m'} C_{j,\ell} \cdot [\mathsf{T}_{i,j}] \quad (3)$$

We note that performing the steps of Set-Equality by relying on a robust MPC yields a robust Set-Equality protocol, as done in the work of [6].

Protocol Shuffle-Pair $([\mathsf{T}], P_0, P_1, \pi_{01})$

Preprocessing:

1. P_0, P_2 randomly sample $[\mathsf{T}']_{02} \in \mathbb{Z}_{2^\ell}^N$, and P_1, P_2 randomly sample $[\mathsf{T}']_{12} \in \mathbb{Z}_{2^\ell}^N$, non-interactively using the common keys established via $\mathcal{F}_{\text{setup}}$.

Online:

1. P_0 computes $\rho_0 = \pi_{01}([\mathsf{T}]_{02}) \oplus [\mathsf{T}']_{02}$ and sends it to P_1. Similarly, P_1 computes $\rho_1 = \pi_{01}([\mathsf{T}]_{12}) \oplus [\mathsf{T}']_{12}$ and sends it to P_0.
2. P_0 and P_1 compute $[\mathsf{T}']_{01} = \pi_{01}([\mathsf{T}]_{01}) \oplus \rho_0 \oplus \rho_1$.

Figure 9: Shuffle-Pair [6, 35]

Regarding the security of the shuffle protocol of [6]. As per the discussion above, observe that their protocol correctly determines the output shares of the shuffled table for each party, in case no misbehaviour occurs during any of the three Shuffle-Pairs. However, since Set-Equality is performed via robust MPC, it guarantees that any malicious act in the corresponding Shuffle-Pair is detected since some V_ℓ, for $\ell \in \{1, \ldots, \kappa\}$, will be non-zero with high probability. Such a misbehaviour in the Shuffle-Pair is indicated by a flag bit being set to a 1. In such an event, observe that parties will not possess the correct output shares and the protocol cannot proceed further. Thus, the protocol only provides security with abort (considering the robust ideal functionality of shuffle, defined in Fig. 1). Moreover, observe that a malicious party can also misbehave such that it learns the output shares but deprives the honest parties of the correct shares. This is possible if a malicious party aborts in the last Shuffle-Pair invocation. Elaborately, consider the last Shuffle-Pair among the three invocations used for getting a random shuffle. Without loss of generality, the protocol proceeds exactly as given in Fig. 9 with P_0 and P_1 being the communicating parties. In case, P_1 is corrupt, it may receive ρ_0 from P_0, obtain the output shares of the shuffled table, and then abort. Since the honest party P_0 does not receive ρ_1 from P_1, it does not learn its output shares. Hence, we believe [6]'s protocol gives only security with abort.

Complexity of shuffle protocol of [6]. Observe that the protocol of [6] requires three invocations of Shuffle-Pair, each of which is followed by a Set-Equality protocol that verifies the correctness of the semi-honest Shuffle-Pair. While Shuffle-Pair requires only one round of communication, the Set-Equality protocol requires $2 + \log_2 \kappa$ rounds[14], where κ is the security parameter. Thus, the overall round complexity is $3 + 2 + \log_2 \kappa$. With respect to the communication complexity, it requires communicating $6N\ell$ bits for the three Shuffle-Pair instances, and additional communication for evaluating the Set-Equality protocol that involves computing $2N\kappa$ Boolean multiplications, computing OR of κ bits and a robust reconstruction.

C ANONYMOUS BROADCAST

Regarding the threat model. An anonymous broadcast system with a powerful adversary corrupting arbitrary number of clients, faces *inherent issues*. In any set of messages submitted to the clients, a significant subset will belong to the adversary. As a result, the honest clients' messages can always be linked to their sender with some probability. This probability depends on the size of the anonymity set for a message, which is the number of honest users who submit their messages in the same round. In the worst case instance, if an adversary is powerful enough to corrupt $(N-1)$ clients in a system, then we can no longer protect the honest client from deanonymization. For our system, we assume that the anonymity set is large enough in each instance of the protocol and that complete deanonymization attacks cannot happen.

Attaining the desirable properties. Confidentiality: Observe that this property is attained since the messages to be shuffled as well as the permutation remain hidden from the servers and clients owing to the secure shuffle protocol.
Censorship resistance: The only way a malicious server can cheat to discard an honest client's message m is by broadcasting an incorrect β_{m} during the input consistency check. Since at most one server among the three is malicious, there will always be agreement among the honest servers with respect to the correct β_{m}. Thus, our protocol does not allow discarding an honest client's messages, thereby attaining censorship resistance.
Integrity and robustness against malicious adversary: Observe that due to the robust input sharing, shuffle, and output reconstruction protocols, our system is robust against any malicious behaviour, and guarantees the integrity of honest clients' messages.

Comparison of our anonymous broadcast system with Clarion [20]. We look at each of the steps in an anonymous broadcast system and draw out a comparison in terms of the communication cost, round complexity and the level of security provided.

[14] One round is required for performing multiplications as per (3), followed by $\log_2 \kappa$ rounds for computing OR of κ bits, followed by one round of reconstruction.

System guarantees: As described earlier, Clarion provides security with abort while our protocol allows attaining the strongest security notion of GOD. Clarion provides no way of distinguishing between a malicious act by a server and client, and the system rejects the request from a client whenever the verification for input consistency fails. Specifically, since Clarion only provides security with abort, it allows a malicious server to make the input consistency check (with respect to an honest client's input) fail by aborting the computation. Thus, an honest client may be dropped due to misbehaviour by a malicious server, and hence Clarion does not guarantee censorship resistance. On the other hand, as described earlier, our scheme guarantees the same.

Input sharing: Our scheme requires 3 commitments be communicated among the servers in the preprocessing. In the online phase, our scheme requires 9 commitments be communicated from servers to the client. Additionally, each server sends the opening for two commitments to the client. Our next step requires the client to communicate an ℓ-bit string β_{m} to each server. Finally, to verify that client has sent consistent β_{m}, each server engages in a broadcast of a message of length ℓ bits[15]. Clarion's preprocessing involves PRG (pseudo-random generator) invocations, encryption of message, and tag computation at the client side. At the servers, one of the servers is responsible for transferring $N(\ell+1)$ Beaver's triples to the other two shuffling parties. Each message of length ℓ is divided into B blocks where $B = \frac{\ell}{|B|}$. For Clarion, $|B| = 128$ bits. To send a message in the online, a client communicates $(B+3)$ blocks to each of the shuffling servers. Moreover, since a client can launch an in-protocol denial-of-service attack by sending incorrect MACs (message authentication code), the servers proceed to verify correctness of client's message by performing $(\ell+1)$ multiplications using Beaver's.

Shuffle: The preprocessing phase of our shuffle protocol requires one invocation of the 3PC shuffle protocol of [6] and requires $6N\ell$ bits of communication for the Shuffle-Pairs. Additionally, it requires computation of $2N\kappa$ Boolean multiplications, OR of κ bits (κ denotes the security parameter), and one robust reconstruction. The online phase of our shuffle requires communication $3N\ell$ bits and at the end of 2 rounds guarantees that misbehaviour, if any, will be detected by a party. If a misbehaviour is detected, it requires 2 additional rounds to broadcast complaints that enable determining a TTP, via which GOD is attained. The 3-server honest-majority shuffle protocol of Clarion is also cast in the preprocessing paradigm. Preprocessing requires communication of $N(2B+3)$ blocks along with N Beaver's triples to each of the 2 shuffling servers. The online phase occurs in 2 rounds, and $2N(2B+3)$ blocks of communication. Clarion additionally needs a verification phase to ensure that the servers haven't misbehaved during shuffling. This includes verifying the MACs associated with the client messages. Hence, 4 additional rounds are performed as no proof of correctness of the shuffle is generated. These 4 rounds involve revealing the ciphertext shares N multiplications, revealing hashes for the shares of difference between computed and actual tag and then giving the actual shares to check that the difference turns out to be 0.

Output reconstruction: In our protocol, output reconstruction requires a single round where openings with respect to the commitments generated on the missing $[\cdot]$-share of α are communicated. On the other hand, Clarion takes 2 rounds of communication and guarantees only security with abort.

D GRAPHSC PARADIGM AND BREADTH FIRST SEARCH

We describe the GraphSC paradigm proposed in [43] and improved in [6], which provides a highly efficient and scalable solution for securely evaluating graph algorithms. The paradigm relies on expressing graph algorithms as *message-passing* algorithms where each node updates its state by repeatedly sending/receiving messages on its edges over multiple rounds. The sending and receiving of messages are realized via the primitive operations of Scatter and Gather. The paradigm provides secure realizations of these primitives such that a message-passing algorithm expressed as a composition of these primitives can be evaluated securely.

The GraphSC paradigm takes as input a directed graph $\mathsf{G}(\mathsf{V},\mathsf{E})$ such that each node $\mathsf{u} \in \mathsf{V}$ is represented as a tuple $(\mathsf{u},\mathsf{u},\mathsf{d})$ and an edge $(\mathsf{u},\mathsf{v}) \in \mathsf{E}$ as $(\mathsf{u},\mathsf{v},\mathsf{d})$. Note that for a tuple corresponding to an edge, the first entry denotes the source node, and the second entry denotes the destination node. Both these entries are the same for a tuple representing a node. Further, d is a binary string that encodes the data/state associated with a tuple (node/edge). Thus, we let $\mathcal{L}$ denote the input G comprising the tuples described above. The Gather primitive requires $\mathcal{L}$ to be sorted based on the destination node v such that all tuples corresponding to incoming edges at v appear before the tuple representing node v. Thus a linear scan of $\mathcal{L}$ allows a node to gather information from its incoming edges. Analogously, Scatter requires $\mathcal{L}$ to be sorted based on source node u, such that tuples corresponding to outgoing edges from u appear after the tuple for node u. This facilitates each node to scatter information on outgoing edges via a linear scan. Thus, the paradigm requires securely sorting $\mathcal{L}$ appropriately before each invocation of a Scatter and Gather primitive. In this way, two invocations of a secure sort are required for each round of message passing.

[6] improves the performance of [43] by relying on a secure shuffle primitive instead of repeatedly performing a secure sort. Specifically, [6] introduces the following sequence of operations to securely evaluate a message-passing graph algorithm. As a one-time initialization, the input $\mathcal{L}$ is securely shuffled using a random secret permutation π_A to obtain $\mathcal{L}_A = \pi_A(\mathcal{L})$. Another invocation of secure shuffle with permutation π_B translates $\mathcal{L}_A$ to $\mathcal{L}_B = \pi_B(\mathcal{L}_A)$. Note that the translation must be such that the parties are able to generate new random shares of $\mathcal{L}_B$ from $\mathcal{L}_A$, and vice versa, while ensuring both π_A, π_B remain hidden. Having fixed on the random permutations π_A and π_B, each time a Scatter is required, the parties consider the shares of $\mathcal{L}_A$ and sort its entries based on the source node, as mentioned earlier. Rather than relying on a secure sort for the same, the parties perform a partially-insecure sort wherein the entries in $\mathcal{L}_A$ remain secret shared, but the output of the comparisons made during the sort are revealed to the parties. In this way, the mapping of the entries in $\mathcal{L}_A$ to the *source-sorted*

[15] These two phases of preprocessing and online translate to a real-world scenario as follows. Consider an anonymous blogging site. The user of the system, on creating a pseudonym, receives a set of pre-computed randomness, which makes up the preprocessing phase, and facilitate generation of a certain number of articles in the online phase which will be posted anonymously.

list is made public to the parties. In an alternative view, performing a sort to obtain a *source-sorted* list is equivalent to picking a specific permutation π_{ss} that defines the sorted order. Hence, performing a partially insecure sort translates to performing a secure sort followed by revealing π_{ss} in public. Note that this does not leak any information since the randomly shuffled input does not permit one to determine any associations between entries in the original graph G and the sorted output. Additionally, the above sort needs to be performed only once to determine the mapping (π_{ss}). Subsequent generation of *source-sorted list* from $\mathcal{L}_A$ can be performed locally by relying on the publicly known π_{ss}. Similarly, for Gather, the parties use the shares of $\mathcal{L}_B$ to obtain the mapping π_{ds} of its entries to the *destination-sorted list* using a partially-insecure sort, once. Recall that the translation from *source-sorted list* to *destination-sorted list*, and vice-versa, requires that the permutations π_A, π_B used for generating $\mathcal{L}_A, \mathcal{L}_B$, respectively, remain the same. Hence, the secure shuffle protocol should facilitate the reusing of the permutations as well as their inverse. We remark that this property is guaranteed by our shuffle protocol and hence is an apt fit for GraphSC based applications. A pictorial representation of the various translations described above with an example graph is provided later in §D.1 for clarity. To give a complete picture of securely evaluating message passing graph algorithms, we describe the representative case of breadth first search (BFS) algorithm. We showcase how this captures the need for Composed-Shuffles, and thereby use of Ruffle-2.

Breadth-first search. We consider a secure evaluation of the BFS algorithm to determine all nodes that are within a given distance t from a source node while ensuring the private graph remains hidden. Additionally, we wish to determine the distance of such nodes from the source node u. The input $\mathcal{L}$ is assumed to be such that the data item $\mathsf{d_u}$ corresponding to the source node u is set as 0, while those of every other entry in $\mathcal{L}$ is set to a very high value denoted ∞. The data item $\mathsf{d_v}$ corresponding to node v thus encodes the distance of this node from the source node u. The parties use $\mathcal{L}$ as defined above and proceed as follows.

One-time set up phase: Given that parties have $[\![\cdot]\!]$-shares of $\mathcal{L}$, they invoke Ruffle-2 to generate $[\![\mathcal{L}_A]\!]$ where $\mathcal{L}_A = \pi_A(\mathcal{L})$. They also generate $[\![\mathcal{L}_B]\!]$ from $[\![\mathcal{L}_A]\!]$ via Ruffle-2 where $\mathcal{L}_B = \pi_B(\mathcal{L}_A)$. Further, the parties generate the mapping from $[\![\mathcal{L}_A]\!]$ to $[\![\cdot]\!]$-shares in source-sorted list and $[\![\mathcal{L}_B]\!]$ to $[\![\cdot]\!]$-shares in destination-sorted list, as described earlier. This completes the necessary set-up.

Message-passing phase: To determine the nodes at a distance t from a given source node, the parties evaluate the BFS protocol in t rounds, with each round comprising the following operations.

1. Scatter: Assuming $\mathcal{L}_A$ as the input, obtain the required source-sorted list using the publicly known mapping. Perform a linear scan of the source-sorted list wherein each node v sends out the data $\mathsf{d_v} + 1$ on all of its outgoing edges. Using the publicly known mapping from the source-sorted list to $\pi_A(\mathcal{L})$, obtain $\mathcal{L}_A$ (which now has updated data entries owing to Scatter).
2. Obtaining $\mathcal{L}_B$: Transform shares of updated $\mathcal{L}_A$ into shares of $\mathcal{L}_B$ using the Ruffle-2 protocol with permutation π_B.
3. Gather: Assuming $\mathcal{L}_B$ as the input, obtain the destination-sorted list using the publicly known mapping. Perform a linear scan of the destination-sorted list wherein each node v gathers the data entries among all of its incoming edges and updates its data $\mathsf{d_v}$ to the value minimum among them. Using the publicly known mapping from the destination-sorted list to $\pi_B(\mathcal{L})$, obtain $\mathcal{L}_B$ (which now has updated data entries owing to Gather).
4. Obtaining $\mathcal{L}_A$: Transform shares of updated $\mathcal{L}_B$ into shares of $\mathcal{L}_A$ using the Ruffle-2 protocol with permutation π_B^{-1}.

In this way, each round of message passing propagates distance information through edges and allows nodes to update their distance from the source node as the minimum among the propagated values. Thus, determining the target nodes and their corresponding distances occurs in t rounds. Since each round requires two invocations of secure shuffle, the overall BFS computation requires $2t$ sequential invocations of Ruffle-2. The formal protocol for Scatter and Gather are included for completeness in §D.2.

D.1 Pictorial example

The transformations involved in a GraphSC application are depicted with the help of an example graph in Fig. 10. For this, we consider a graph with 3 nodes ($\mathsf{v}_1, \mathsf{v}_2, \mathsf{v}_3$) and 2 edges ($\mathsf{e}_{12}, \mathsf{e}_{32}$). For the ease of understanding, the entries in the input $\mathcal{L}$ are also named accordingly. However, note that the parties in $\mathcal{P}$ are only aware of the size of $\mathcal{L}$ and not its entries on clear. Same is the case with other lists $\mathcal{L}_A$, $\mathcal{L}_B$ where each of its entries is only known in secret shares among parties in $\mathcal{P}$. The arrows in red denote the invocation of our secure shuffle protocol. On the other hand, the arrows in blue depict the position of each element in the $\mathcal{L}_A$ to its position in the *source-sorted* list. In this way, the blue arrows collectively represent the publicly known permutation that can be obtained by revealing the result corresponding to each comparison made in the secure sort protocol applied on the secret shared entries of $\mathcal{L}_A$. The same applies to the blue arrows between $\mathcal{L}_B$ and *destination-sorted* list.

D.2 Scatter-Gather primitives

We now describe the Scatter and Gather primitives for breadth first search in the GraphSC paradigm. As described earlier, the input to the algorithm is a directed graph where every node and edge in the graph is encoded as a tuple. Let this graph be denoted by G and the i^{th} tuple in the graph be denoted as $\mathsf{G}[i]$. Every vertex tuple is encoded as (u, u, isVertex, buff, dist) and every edge tuple is encoded as (u, v, isVertex, buff, dist) with u, v being the source and destination vertices of an edge. Thus, the entries of isVertex, buff, dist are the components of the data part of each tuple. The isVertex value in a tuple indicates whether the tuple belongs to a vertex and hence is set as 1 for a vertex tuple and 0 for an edge tuple. The dist value holds the minimum distance of a vertex tuple from a given source node. The buff value is used as a buffer for communicating between vertices and edges. During Scatter, the value (dist + 1) corresponding to each vertex is transferred to the buff value of all it's outgoing edges by performing a linear scan of all the entries in G. During Gather, the minimum of all the buff values among all the incoming edges to a vertex is used to assign the minimum distance of that vertex. The steps required to perform such a Scatter and Gather are provided formally in Fig. 11. This provides the clear text algorithm for the primitives and the secure variants can be easily obtained by using secure operations for additions, comparison as well as secure realizations of if – else conditions as done in [6, 43]. Further, we note that

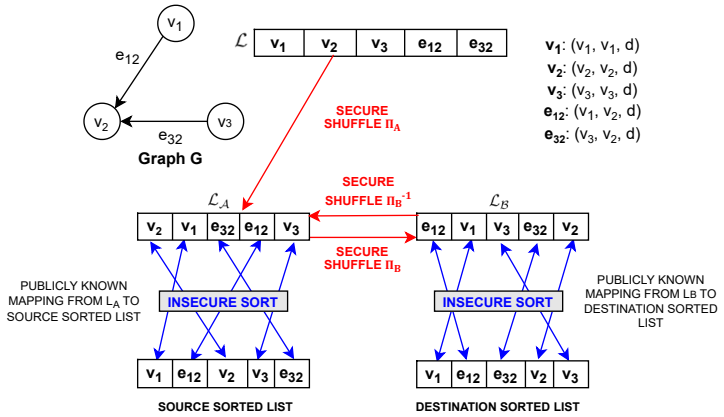


Figure 10: Example for translations in GraphSC.

for simplicity, the input to both the primitives is denoted as G. However, the assumption is that the graph G is *source-sorted* for Scatter and is *destination-sorted* for Gather. Further, the steps in Fig. 11 are provided for a single processor setting. They can be easily translated to the multi-processor setting using the work of [43] thereby providing a solution with number of rounds in the order of $\frac{|V|+|E|}{P} + \log(|V|+|E|)$, where P is the number of processors used for running the parallel algorithm. This is unlike the linear algorithm which has a round complexity in the order of |V| + |E| as described in Fig. 11.

Scatter(G)
val = 0
for i = 1 to |V| + |E| do
 if G[i].isVertex then
  val = G[i].dist + 1
 else
  G[i].buff = val

Gather(G)
val = ∞
for i = 1 to |V| + |E| do
 if G[i].isVertex then
  if val < G[i].dist
   G[i].dist = val
  val = ∞
 else
  val = min(val, G[i].buff)

Figure 11: Scatter and Gather for BFS

E BENCHMARK DETAILS

With respect to the application of anonymous broadcast, we observe that our Python based implementation of [20] has an overhead compared to the public implementation of [20] (GO-based). We believe this is due to the use of Python based implementation. For example, time taken to generate a random 16000 byte value is 9.881×10^{-5} seconds in Clarion's GO-based implementation whereas the same task takes 6.915×10^{-3} seconds in our Python-based implementation, under the same system configuration.

We believe that similar trends apply to the GraphSC framework of [6] as well. We could not get the exact values since [6] does not provide public implementation of their work. Further, we note that the higher cost reported here is also due to the fact that the variant of BFS we consider here is different from the one considered in [6]. The latter considered the simplistic case of determining all nodes reachable within t-rounds which only requires Boolean operations. Making their protocol complaint with arithmetic operations, as required for our BFS implementation, introduces additional overheads.

F SECURITY OF OUR PROTOCOLS

Lemma F.1. *Whenever a* TTP *is identified, it is always honest.*

Proof. Consider the instances in our shuffle protocol (Fig. 4) where we proceed to identify a TTP. We will show that in each such case, a malicious party is never identified as the TTP.

Preprocessing phase. During this phase, a TTP is identified whenever a Shuffle-Pair fails. Without loss of generality, let the first failed Shuffle-Pair instance be with respect to π_{ij}, where TTP = P_k. Since P_i and P_j are involved in communication, failure occurs when either of them sends an incorrect message. Hence, the malicious party lies in $\{P_i, P_j\}$ set. Thus, P_k, which is identified as the TTP, is honest as only one out of the three parties could be malicious.

Online phase. During the online phase, consider the first instance where a receiver P_k broadcasts ("accuse", P_i, P_j, c_i, c_j), where P_i, P_j are the senders who send x, c_j, respectively, to P_k, and $c_i = \mathsf{H}(x)$. We consider all corruption scenarios and prove that the TTP identified in each such case is an honest party.

– P_i *is corrupt:* In this case, when P_k broadcasts ("accuse", P_i, P_j, c_i, c_j), it is due to an incorrect message sent by P_i to P_k. Party P_j will not broadcast ("accuse", P_k), since P_j, P_k are both honest and there will not be a mismatch in their messages. Thus, the malicious P_i will never be identified as a TTP. If P_i broadcasts ("accuse", P_k), then TTP is set as P_j which is an honest party. Else, if P_i does not broadcast ("accuse", P_k), then TTP is set as P_k which is also an honest party.

– P_j *is corrupt:* A similar argument follows for this case.

– P_k *is corrupt:* In this case, since P_i, P_j are honest, if P_k broadcasts ("accuse", P_i, P_j, c_i, c_j), then P_k has wrongfully accused P_i and P_j. In the broadcast message, $c_i = c_j$, then other parties know that P_k is the malicious party and set TTP = P_i. Else, if $c_i \neq c_j$, at least one of P_i or P_j will broadcast ("accuse", P_k) if the value broadcast by P_k is different from what they sent. Hence, TTP $\in \{P_i, P_j\}$ will be set as an honest party.

□

Lemma F.2. *The shuffle protocol,* Π_{Ruffle} *(Fig. 4) securely realizes the functionality* $\mathcal{F}_{\mathsf{Shuffle}}$ *(Fig. 1) against a malicious adversary that corrupts at most one party in* $\mathcal{P}$*, in the* $\mathcal{F}_{\mathsf{setup}}$*-hybrid model.*

Simulator $\mathcal{S}^{P_0}_{\Pi_{\mathsf{Ruffle}}}$

$\mathcal{S}^{P_0}_{\Pi_{\mathsf{Ruffle}}}$ proceeds as follows.

Preprocessing:

(1) Using the keys commonly held with $\mathcal{A}$ (generated as part of $\mathcal{F}_{\mathsf{setup}}$), sample the common randomness.
(2) Simulate the steps of Shuffle-Pair pair using π_{02}. Receive the corresponding message from $\mathcal{A}$. If the received message is incorrect ($\mathcal{S}^{P_0}_{\Pi_{\mathsf{Ruffle}}}$ can verify the correctness of the received message since it possesses all the randomness used by $\mathcal{A}$ to send this message as $\mathcal{S}^{P_0}_{\Pi_{\mathsf{Ruffle}}}$ emulates $\mathcal{F}_{\mathsf{setup}}$), set $\mathsf{flag}_{02} = 1$. Honestly simulate the steps of Set-Equality protocol.
(3) Honestly simulate the steps of Shuffle-Pair using π_{12} followed by simulating the steps of Set-Equality.
(4) Analogous to the case of Shuffle-Pair with π_{02}, simulate the steps of Shuffle-Pair pair using π_{01}. If an incorrect message is received from $\mathcal{A}$, set $\mathsf{flag}_{01} = 1$.
(5) If $\mathsf{flag}_{02} = 1$, set $\mathsf{TTP} = P_1$. Else, if $\mathsf{flag}_{01} = 1$, $\mathsf{TTP} = P_2$.

Online:

(1) Let $\left[\alpha_{\mathsf{T_o}}\right]_{01}, \left[\alpha_{\mathsf{T_o}}\right]_{02}$ denote the partial shares of T_o generated towards $\mathcal{A}$ during preprocessing. Let $\beta_{\mathsf{T_o}} \in \mathbb{Z}_{2^\ell}^N$ be sampled randomly. Invoke the ideal functionality $\mathcal{F}_{\mathsf{Shuffle}}$ with $\mathcal{A}$'s $[\![\cdot]\!]$-shares of the table–$\left[\alpha_{\mathsf{T_o}}\right]_{01}, \left[\alpha_{\mathsf{T_o}}\right]_{02}, \beta_{\mathsf{T_o}}$.
(2) Receive $\mathsf{H}(\delta_{02})$ from $\mathcal{A}$ on behalf of P_1. Set $\mathsf{flag}_1 = 1$ if the received message is incorrect ($\mathcal{S}^{P_0}_{\Pi_{\mathsf{Ruffle}}}$ can verify the correctness of the received message since it possess all the randomness used by $\mathcal{A}$ to send this message as $\mathcal{S}^{P_0}_{\Pi_{\mathsf{Ruffle}}}$ emulates $\mathcal{F}_{\mathsf{setup}}$).
(3) Receive δ_{01} from $\mathcal{A}$ on behalf of P_2.
(4) If $\mathsf{flag}_1 = 1$, broadcast ("accuse", P_2, P_0, c_2, c_0), where $c_0 = \mathsf{H}(\delta_{02})$ as received from $\mathcal{A}$, and $c_2 = \mathsf{H}(\pi_{02}\,(\beta_{\mathsf{T}} \oplus \mathsf{R}_{02}))$ is honestly computed by $\mathcal{S}^{P_0}_{\Pi_{\mathsf{Ruffle}}}$.
   - If $\mathcal{A}$ broadcasts ("accuse", P_1), set $\mathsf{TTP} = P_2$.
   - Else set $\mathsf{TTP} = P_1$.
(5) Else, if $\mathsf{flag}_1 = 0$, proceed as follows. Set $\delta_{12} = \beta_{\mathsf{T_o}}$ (as defined in step 1) and compute $\mathsf{H}(\delta_{12})$. Send $\delta_{12}, \mathsf{H}(\delta_{12})$ to $\mathcal{A}$ on behalf of honest P_1, P_2, respectively. Compute $c_1 = \mathsf{H}(\pi_{01}\,(\pi_{02}\,(\beta_{\mathsf{T}} \oplus \mathsf{R}_{02}) \oplus \mathsf{R}_{01}))$ and compare it with $c_0 = \mathsf{H}(\delta_{01})$ where δ_{01} was received from $\mathcal{A}$. If $c_0 \neq c_1$, set $\mathsf{flag}_2 = 1$.
(6) If $\mathsf{flag}_2 = 1$, broadcast ("accuse", P_0, P_1, c_0, c_1).
   - If $\mathcal{A}$ broadcasts ("accuse", P_2), set $\mathsf{TTP} = P_1$.
   - Else set $\mathsf{TTP} = P_2$.
(7) If $\mathsf{flag}_2 = 0$, and if $\mathcal{A}$ broadcasts ("accuse", P_1, P_2, c_1, c_2), then
   - If $c_1 = c_2$, set $\mathsf{TTP} = P_1$.
   - Else, if only c_1 is not the same as $\mathsf{H}(\delta_{12})$, where δ_{12} was sent by $\mathcal{S}^{P_0}_{\Pi_{\mathsf{Ruffle}}}$, broadcast ("accuse", P_0) on behalf of P_1, and set $\mathsf{TTP} = P_2$. Analogously for c_2.
(8) If at any point during the simulation a TTP is identified, then send the output shares $[\![\mathsf{T}_o]\!]_0 = \left(\beta_{\mathsf{T_o}}, \left[\alpha_{\mathsf{T_o}}\right]_{01}, \left[\alpha_{\mathsf{T_o}}\right]_{02}\right)$ to $\mathcal{A}$ on behalf of TTP.

Figure 12: Simulator $\mathcal{S}^{P_0}_{\Pi_{\mathsf{Ruffle}}}$ for corrupt P_0

Proof. Let $\mathcal{A}$ denote the real-world adversary and $\mathcal{S}_{\Pi_{\mathsf{Ruffle}}}$ denote the corresponding ideal-world adversary. At a high level, $\mathcal{S}_{\Pi_{\mathsf{Ruffle}}}$ begins by first emulating $\mathcal{F}_{\mathsf{setup}}$ during which common keys are established with $\mathcal{A}$ that are used to sample the common randomness required throughout the protocol. Thus, $\mathcal{S}_{\Pi_{\mathsf{Ruffle}}}$ is aware of all the randomness used by $\mathcal{A}$ using which it can extract the input of $\mathcal{A}$ (specifically, $\mathcal{S}_{\Pi_{\mathsf{Ruffle}}}$ knows $[\alpha_{\mathsf{T}}], \beta_{\mathsf{T}}$ held by $\mathcal{A}$) as well as verify the correctness of messages sent by $\mathcal{A}$. Following this, it simulates the steps of the shuffle protocol. The simulation steps for a corrupt P_0 are provided in Fig. 12, where the corresponding simulator is denoted as $\mathcal{S}^{P_0}_{\Pi_{\mathsf{Ruffle}}}$. Analogously the corruption of P_1, P_2 can also be simulated.

Observe that in the real-world, during the preprocessing phase, $\mathcal{A}$ receives messages that are computed as part of Shuffle-Pair and Set-Equality, where the messages are masked using some randomness to hide the missing permutation as well as information regarding the missing share at $\mathcal{A}$. During the online phase, $\mathcal{A}$ receives $\delta_{12}, \mathsf{H}(\delta_{12})$ where δ_{12} is a randomized using the random mask R_{12}. In this way, observe that the messages received by $\mathcal{A}$ in the real-world are random and uniform. In the ideal world too, observe that $\mathcal{A}$ receives messages that are sampled randomly from the uniform distribution. Moreover, $\mathcal{S}^{P_0}_{\Pi_{\mathsf{Ruffle}}}$ can verify the correctness of the messages sent by $\mathcal{A}$, as described earlier. This allows $\mathcal{S}^{P_0}_{\Pi_{\mathsf{Ruffle}}}$ to also simulate all the accuse messages as done in the real world. In this way, real-world and ideal-world executions are indistinguishable. □