Secure Multi-party Summation Protocols: Are They Secure Enough Under Collusion?

Thilina Ranbaduge[*], **Dinusha Vatsalan**[**], **Peter Christen**[*]

[*]Research School of Computer Science, The Australian National University, Canberra ACT 2601, Australia.

[**]Information Security and Privacy Group, Data61-CSIRO, Sydney NSW 2015, Australia.

E-mail: `thilina.ranbaduge@anu.edu.au`

Received 7 October 2018; received in revised form 4 July 2019; received in revised form 8 November 2019; accepted 16 December 2019

Abstract. To enable data analytics that provides valuable insights, data that are distributed across several organisations increasingly need to be shared before they can be analysed. However, sharing data from different sources can raise privacy and confidentiality concerns. Organisations are often unwilling or not allowed to share their sensitive data, such as personal details or health or financial data, with other parties because this potentially violates the privacy of individuals. *Secure multi-party computation* (SMC) has been introduced as a solution to overcome the problem of performing computations on sensitive data across organisations. SMC allows parties to jointly compute a function over their inputs while preserving the privacy of these inputs. *Secure summation protocols* are an important building block in many SMC applications that can be used under two different SMC models (i.e. with and without the involvement of a third party to conduct the computations). A secure summation protocol is used to compute the summation of private inputs held by different parties. In this paper we study existing secure summation protocols that can be used under different SMC models and then propose three advanced secure summation protocols that use homomorphic encryption. We then consider different scenarios of how parties might collude with each other in secure summation protocols, and the potential collusion risks that occur with these protocols. No such investigation of possible collusion scenarios for secure summation protocols has so far been presented. We analyse each secure summation protocol under different collusion scenarios and evaluate the efficiency of each protocol with different numbers of parties and different input data sizes. Our evaluation shows that our proposed protocols provide improved privacy against collusion risks and they can calculate a sum more efficiently compared to existing secure summation protocols.

Keywords. Secure multi-party computation, homomorphic encryption, collusion, honest but curious, secret sharing, privacy evaluation.

1 Introduction

We are living in a Big Data era and many businesses, government agencies, and research organisations are collecting increasingly vast amounts of data to be analysed for interesting patterns and useful knowledge in support of efficient and quality decision making [25]. Due to the distributed availability of data, different parties or organisations often need to participate in cooperative computations, where all participants jointly conduct computations to calculate a global result for data analysis. The sharing or integration of data across different parties or organisations within data analytics processes is often challenged due to various reasons. Database heterogeneity, data quality, and scalability are some of the major challenges that are common in any data integration project [15, 24, 37], while privacy has become a crucial aspect that needs to be considered if the databases to be integrated contain sensitive data such as personal identifying information (for example names and addresses) [16, 42, 62, 65]. Since the participating parties in data integration protocols can be semi-trusted or untrusted, privacy and confidentially of such sensitive data need to be preserved because the analysis of integrated data can potentially be used by an adversary to infer private information about individuals [14, 40, 45, 65, 73]. Below we highlight some real-world example scenarios where organisations might want to share their data in order to obtain a common result, and at the same time protecting the privacy and security of individuals stored in such data is a challenge.

Health research. Assume a scenario where a group of hospitals wish to collaborate on analysing their patient databases for the purpose of investigating the geographical and temporal effects of diseases and drug usages in certain patient groups. Each hospital database can contain many hundreds of thousands of patient records, where the records across all the databases need to be integrated to identify how patient groups with different illnesses react to different drugs and medical treatments. However, because of privacy and confidentiality reasons, neither of the hospitals is willing or allowed to share its patient database with any other hospital or any other organisation. Hence, it is necessary to find a solution that enables these hospitals to conduct the required analytical operations upon their databases without revealing any sensitive information.

Financial Crimes. Today, financial and transactional data are generated in many different formats and with very large volumes. Analysing such data requires more sophisticated information systems than traditional methods of data analysis [56]. To identify fraudulent transactions from different individuals, for example, there is a need to analyse the flow of money that has been transferred by an individual between financial institutions such as banks. However, due to security and confidentiality reasons financial institutions are commonly not willing or not allowed by legislation to provide information about individuals in their customer databases to other organisations, which makes such an analysis on transactional data challenging.

Internet of Things. Another real-world example would be an *Internet of Things* (IoT) application where various devices (sensors) communicate with one another to execute daily operations with a minimum of human interventions [58, 60]. In IoT applications devices or components that belong to people, objects, or organisations are interconnected and they communicate over public, untrusted networks. Sharing and integration of data generated by each individual device often raises privacy and security concerns (i.e. the data stored in each IoT device might contain information about individuals or households) because the data collected at each individual device needs to be integrated to perform valuable data analysis. In such applications any sensitive information generated by each individual device must not be communicated to other connected devices, but processed collaboratively to employ the required analytical functionalities.

Secure multi-party computation (SMC) has been introduced as a solution to overcome the problem of performing computations on sensitive data across organisations [29]. SMC enables several parties, named *data providers* (DPs) [47], to be involved in a computation with

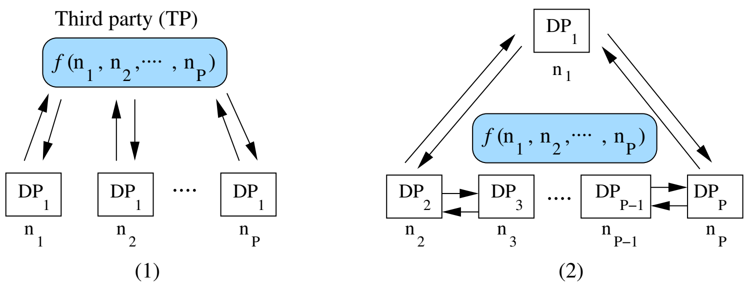

Figure 1: Secure multi-party computation (SMC) models, (1) with a third party (TP) and (2) without a TP. P (≥ 3) data providers (DPs), DP$_1$, DP$_2$, $\cdots$, DP$_P$, participate in a SMC protocol to compute the value of a public function f on their private inputs, $n_1, n_2, \cdots, n_P$, respectively. At the end of a SMC protocol, each DP$_i$ learns $f(n_1,\ n_2, \cdots,\ n_p)$, but it learns nothing about the private inputs n_j of any other DP$_j$, with $i \neq j, 1 \leq i, j \leq P$. Arrows represent the communication between parties.

their private input data without the need to explicitly share or integrate these data, where at the end of the computation no party learns anything about any other party's private input data except the final result of the computation. SMC has been used in different application domains including privacy-preserving data mining [9, 10, 16, 67, 68], privacy-preserving data publishing [27], and privacy-preserving record linkage [65, 73]. SMC enables multiple DPs to compute a function, such as summation, comparison, scalar product, set intersection, and so on, upon their combined private inputs such that at the end of the computation no DP learns anything about any other DP's private input, and all DPs learn the final result. As we will discuss in Section 2, the protocols that are proposed for SMC can be based on garbled circuits [31, 76], secret sharing [17], homomorphic encryption [1, 19, 59], or commutative encryption [2, 71].

The aim of SMC is to allow different parties to compute a common function upon their private data, where any SMC protocol must satisfy two important properties [46]. (1) *Privacy*: this property states that once the protocol finishes neither of the parties should be able to learn anything other than the final result of the computed function. (2) *Correctness*: this property states that each party should always receive the correct output such that the computed function should always produce an accurate result with the inputs provided by the participating parties. Consider an example where two or more banks cooperatively want to know the financial credibility of some customers, however, no bank is willing to disclose the details of their customers to the other banks. Thus, all banks can participate in an SMC protocol to perform the required analysis. This analysis generally involves computation and communication among the banks according to some prescribed specification, where the parties only learn the output as desired. As shown in Figure 1, in SMC, P parties wish to compute a function f of their secret inputs, where $f : \{0,1\}_1{}^* \times \cdots \times \{0,1\}_P{}^* \rightarrow \{0,1\}^*$. The goal is to compute the output $y = f(\mathbf{x})$ ensuring the correctness and privacy, where $\mathbf{x}$ represents an input vector $(n_1, n_2, \cdots, n_P)$.

SMC protocols can be categorised into two different models [47]. (1) SMC with a *third party* (TP): in this model the DPs communicate only with the TP to compute a function f. Once the computation of f is completed the TP sends the final result to all DPs. (2) SMC without a TP: in this model the DPs communicate among themselves to compute the function f. Once the final result is computed one DP (the one that calculates the final result) distributes the final result to all other DPs. The participants of these models can be categorised as either trusted, semi-trusted, or untrusted [47]. Realistically since the parties involved in a SMC protocol cannot always be trusted, SMC protocols must be able to guarantee the privacy of the input of each DP when performing computations and communications between the protocol participants.

SMC protocols are generally designed to follow either the *honest-but-curious* (HBC) or the *malicious* adversary models [47]. The HBC model assumes all parties follow the protocol but aim to learn as much as possible from the data they receive from other parties, while in the malicious model parties can behave arbitrarily. A malicious party may not follow the protocol as specified and can refuse to participate or abort the protocol at any time, and can provide arbitrary values as their inputs [29, 47]. However, in many real-world settings, the assumption regarding the HBC behaviour does not suffice, while security in the presence of malicious adversaries is excessive and expensive to achieve [32, 33, 50].

The more recently proposed *covert* and *accountable computing* models are advanced adversary models that lie between the HBC and malicious models [3, 38, 39]. The covert model matches with many real-world settings by assuming parties may deviate arbitrarily from the protocol specification in an attempt to cheat, and where the honest parties can identify the misbehaviour of an adversary with high probability [3, 32, 47]. The main intention behind the covert model is to capture situations where the penalty for cheating is high relative to the potential gain. Once coupled with a high penalty for cheating, the covert security of the protocol serves to prevent participants from deviating from the protocol [36]. The accountable computing model provides accountability for privacy compromises by an adversary without any of the additional complexity or costs that occurs with malicious adversaries [38, 39].

All secure summation protocols discussed in this paper assume the HBC model. However, in any of these adversary model, collusion between the participating parties needs to be considered [47]. Collusion is an inevitable and serious privacy risk in multi-party computations where a sub-set of (two or more) parties aims to learn the sensitive private input(s) of another party or sub-set of parties. This is generally accomplished by the colluding parties sharing their own data and function parameter settings among themselves. For the two SMC models described above, as shown in Figure 1 (and discussed in Section 3), collusion can occur either between the DPs, or between one or more DPs and the TP. To the best of our knowledge the applicability of secure summation protocols under such different collusion scenarios has not been studied so far.

Contributions: In this paper we study secure summation, which is an essential building block of SMC protocols and is widely used in privacy-preserving multi-party computation over distributed data. As we describe next, in Sections 2.1 to 2.5 we first review five existing secure summation protocols. We then propose three novel secure summation protocols that use homomorphic encryption to compute the summation of the private inputs of a set of data providers. We study different collusion scenarios that are possible with the two SMC models, and the applicability of each secure summation protocol under these scenarios. We analyse the privacy of the presented protocols and show our proposed secure summation protocols are more secure than the existing protocols for different collusion scenarios. We empirically evaluate the complexity (computation and communication) of each protocol for different real-world scenarios. The goal of this paper is not only to analyse secure summation protocols, but also to provide a guideline for researchers and practitioners to identify possible collusion scenarios and appropriate secure summation protocols for their own domains.

Organisation: The rest of this paper is structured as follows. In the following section, we first discuss the generic SMC protocols that use garbled circuits schemes and their applicability on real-world scenarios. Next we provide an overview of five existing secure summation protocols and propose three novel secure summation protocols. For each of these secure summation protocols we provide an algorithmic description to outline the steps of the protocol. In Section 3 we then analyse the privacy of each protocol under different collusion scenarios, and the computational and communication complexities of each protocol. In Section 4 we empirically evaluate each protocol under different real-world scenarios, and we conclude this paper in Section 5 with a summary of our findings.

2 Secure Summation Protocols

The concept of SMC was first introduced by Yao [75, 76] for two parties with the idea of performing computations securely such that at the end of a computation no party knows anything except its own input and the final results of the computed function [11, 47, 57]. The function to be computed is represented as a combinatorial circuit where the participating parties run a short protocol for every gate in the circuit to compute the final result [53]. Yao's approach to secure computation between two parties was extended by Goldreich et al. [31] for multiple parties following the same methodology. Surveys of SMC are available from Franklin and Young [26], Goldreich [30], and Lindell and Pinkas [47].

Secure summation is a commonly used building block in SMC to calculate a global sum of the numerical private inputs held by several DPs that participate in a protocol. The concept of secure summation was presented as an example of SMC protocols in many earlier works [5, 17, 64]. Secure multi-party summation can be used to build more complex protocols for other SMC functions. It has been widely used in privacy-preserving multi-party computation over distributed data, including privacy-preserving data mining [10, 16, 40, 46, 67, 71], privacy-preserving machine learning [51, 52, 63, 68], privacy-preserving collaborative social networks [7, 69, 74], and privacy-preserving record linkage [73].

In a generic SMC protocol, secure summation can be performed using an arithmetic (adder) circuit implementation either based on Yao's garbled circuit or Goldreich, Micali and Wigderson (GMW) protocol schemes [31, 47]. Both Yao's and GMW protocol schemes are proven to be secure assuming the underlying oblivious transfer (OT) protocol they use to construct the circuit is secure [36].

In the Yao's garbled circuit based protocol, an encrypted binary circuit is generated for preserving the privacy of inputs. The circuit generation depends on symmetric cryptography [30] and an oblivious transfer (OT) protocol [64], where one party generates the circuit while the other evaluates it. However, Yao's garbled circuit protocol often has high communication complexity and requires input sizes to be known in advance. This allows a SMC protocol based on Yao's protocol to have all the OTs to be computed in parallel at the beginning of the communication. The number of OTs required in the protocol generally depends on input size, which makes these approaches not scalable to large input sizes.

Similar to Yao's garbled circuit protocols, GMW protocols also use a binary circuit representation but the security evaluation is built on secret sharing [17] rather on encrypted gates [31]. Each party's inputs are shared across parties using a secret sharing scheme allowing each party to conduct computation on the input shares which ensures that at the end of the computation each party is left with a share of the output. The inputs can be shared by simply XORing the shares of their input wires. To evaluate an AND gate the participating parties perform an OT where one party can pre-compute all possible output of the gate and every other party can obtain the output that correspond to their input shares. The output of the circuit can be obtained by exchanging the shares of the output wires. GMW requires one OT for computing each gate but requires less number of communication rounds which is proportional to the depth of the circuit.

However, whether Yao's protocol or the GMW protocol is more efficient will depend on the type of function being evaluated [35, 49]. Most protocols based on garbled circuits are better suited to two or three-party secure computation and do not directly extend to hundreds of parties. The main reason for such scalability issue is the inherent inefficiency which arises due to the number of computation and communication rounds required in a protocol. Further, the number of rounds in the protocol is linear in the depth of the circuit that the parties need to compute [4, 36].

In 2012, Damgård et al. [20] proposed a SMC technique, popularly known as SPDZ, which describes a flavour of GMW scheme for arithmetic circuits. The protocol consists of two phases, *preprocessing* and *online*. In the preprocessing phase it uses a somewhat homomorphic encryption scheme to secretly distribute the shares of inputs of parties and also generates lots of random values which are used by the parties to provide input to the circuit. In the online phase the P parties construct and evaluate the circuit and open the secret corresponding to the circuit output. However, the total overhead of this approach is $O(P \cdot S + P^3)$ where S is the size of the computed arithmetic circuit.

Recently, Lindell et al. [48] proposed a SMC protocol that uses constant-round of computations and communications, and dishonest majority setting. The protocol combines SPDZ with the constant-round protocol (BMR) developed by Beaver, Micali and Rogaway [4]. The protocol is comprised of two phases; a preprocessing phase for securely computing random shares of the garbled circuit where each party locally computes the random values as needed for every gate; an online phase where all the parties construct a single garbled circuit, exchange garbled values on the input wires, and evaluate the garbled circuit. However, the total overhead of this approach is $O(P^3)$ due to the amount of computations and communication required in different phases. Another recent variation of this protocol was proposed by Lindell et al. [49] which uses BMR and somewhat homomorphic encryption to reduce the number of rounds of computations and communications between parties. This protocol has an overhead of $O(S \cdot P^2)$ due to the communications required in the preprocessing phase. However, both these protocols [48, 49] provided no implementation and remained largely theoretical, so it is not clear how concretely efficient they are in real scenarios.

To this end, in this paper we will focus on secure summation protocols that use privacy techniques which have smaller computational and communication cost, while providing privacy against collusion. In Sections 2.1 to 2.5 we describe five existing secure summation protocols where we provide an algorithmic description of each protocol [16, 51, 69, 72, 74]. In Sections 2.6 to 2.8 we then propose three new protocols that exhibit improved privacy with regard to different types of collusion scenarios compared to existing protocols. We assume the two inputs to each protocol are $P \geq 3$, the number of participating DPs, and the private (sensitive) numerical input value of each DP$_i$, n_i, with $1 \leq i \leq P$. Each protocol securely calculates and returns the final sum $\mathcal{E} = \sum_{i=1}^{P} n_i$ without revealing any of the n_i to any other DPs, or any party external to the protocol. We use *E()* and *D()* to represent the functions for encryption and decryption, respectively. We analyse the presented summation protocols in terms of privacy and their computation and communication complexities in Section 3.

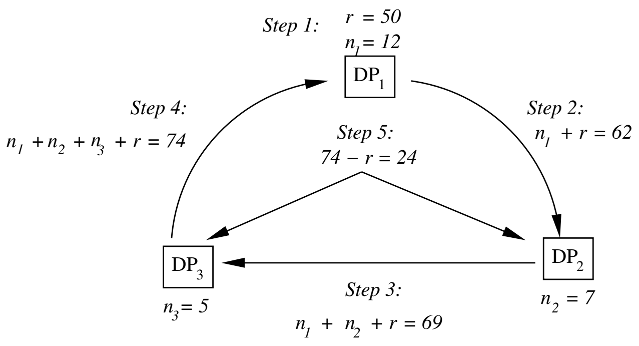

Figure 2: An example of the basic secure summation protocol (BSS) between data providers DP_1, DP_2, and DP_3 to compute their private input values n_1, n_2, and n_3, respectively. In this example, DP_1 initialises the protocol by generating a random number r (step 1) and adding r to its private input n_1 (step 2). In steps 3 and 4, DP_2 and DP_3, respectively, updates the partial sum it received from the previous DP by adding its own private input value and sending the updated partial sum value to the next DP. Finally, in step 5, DP_1 sends the final sum $\mathcal{E} = 24$ to other DPs by subtracting r from the partial sum value it received from DP_3.

2.1 Basic Secure Summation (BSS)

A basic secure summation protocol to securely mine association rules over horizontally distributed data under the honest-but-curious adversary model was initially proposed by Clifton et al. [16]. In horizontally partitioned databases the global support count is equal to the sum of all the local support counts. The authors used a secure summation protocol to compute the global support count of a rule such that the participants collaboratively form the global rule while none of the local support counts is being revealed.

The suggested protocol is based on randomisation which requires the participating parties to communicate in a round robin fashion. Algorithm 1 describes the steps involved in this basic protocol [16, 42, 43]. Initially DP_1 chooses a large random number r (line 1) which it keeps secret from all other DPs. It then adds r to its input n_1 (line 2) to generate the partial sum $s_1 = (n_1 + r)$. Then DP_1 sends s_1 to DP_2 (line 3). Since r is random, DP_2 cannot learn anything about n_1. DP_2 receives the partial sum s_1 (line 5) and then adds its value n_2 to s_1 (line 6), $s_2 = s_1 + n_2$, and sends the result to DP_3 (line 8). This process is repeated (lines 4 to 13) until all the DPs have added their values, and the partial sum $s_P = r + n_1 + \cdots + n_P$ is sent back to DP_1 (line 11). DP_1 subtracts r from s_P (line 15) and the final sum $\mathcal{E}$ is distributed to all other DPs (line 16).

BSS is susceptible to privacy risks where a collusion between two or more adjacent DPs can allow them to learn the private input of a non-colluding DP. We discuss the collusion risk of BSS in detail in Section 3. Figure 2 shows an example of BSS for three DPs.

2.2 Encrypted Secure Summation (ESS)

Encryption is a well-known security technique which is the process of encoding a message or information in such a way that only authorised parties who know the decryption key can access the original message while those who are not authorised cannot [16]. The main limitation of this technique, however, is the requirement of the data to be decrypted before any computation operations can be performed on the data. Thus, it is desirable to have an

Algorithm 1: Basic secure summation protocol (BSS) [16]

```
Input  : $P$ - Number of DPs
         $n_i$ - The secret numerical input value of DP$_i$, $1 \leq i \leq P$
Output: $\mathcal{E}$ - Final sum where $\mathcal{E} \leftarrow \sum_1^P n_i$
1  DP$_1$ generates a random number $r$
2  DP$_1$ computes partial sum $s_1 \leftarrow n_1 + r$
3  DP$_1$ sends $s_1$ to DP$_2$
4  for $i \in [2, 3, \cdots, P]$ do
5      DP$_i$ receives $s_{i-1}$
6      DP$_i$ computes partial sum $s_i \leftarrow s_{i-1} + n_i$   // Compute the partial sum
7      if $i < P$ then                                          // Check if the current DP is not DP$_P$
8          DP$_i$ sends $s_i$ to DP$_{i+1}$                     // Send the partial sum to the next DP
9      end
10     else
11         DP$_i$ sends $s_i$ to DP$_1$                         // Send the final sum to DP$_1$
12     end
13 end
14 DP$_1$ receives $s_i$
15 DP$_1$ computes final sum $\mathcal{E} \leftarrow s_P - r$   // Substract $r$ from final partial sum
16 DP$_1$ sends final sum $\mathcal{E}$ to other DPs
17 Other DPs receive final sum $\mathcal{E}$
```

encryption scheme that allows non-trivial operations on encrypted data without decryption. Such a special encryption scheme is called a homomorphic cryptosystem [1, 19, 59].

Homomorphic encryption schemes allow certain computations to be performed on data that are in an encrypted form (ciphertext) and output an encrypted result [20, 28]. Homomorphic encryption ensures that the decrypted result is identical to the result of the same function performed on the unencrypted input data. In homomorphic encryption schemes a key pair, known as public and private (secret) keys, is used to encrypt and decrypt the private input data accordingly [59]. The public key is kept publicly available to any party that participates in a protocol while a private key is not shared and kept secret by the party that generates the key pair. Successive encryption of the same value using the same public key generates different encrypted values with high probability, while decrypting an encrypted value using a private key always returns the correct original value [1, 20].

For example, assume two given numbers n_1 and n_2 belonging to data providers DP$_1$ and DP$_2$, respectively. First, DP$_1$ and DP$_2$ encrypt their numbers n_1 and n_2 into an encrypted form ε_1 and ε_2, respectively, using an encryption function $E()$, where $\varepsilon_1 \leftarrow E(n_1)$ and $\varepsilon_2 \leftarrow E(n_2)$. Based on a homomorphic addition scheme [55], DP$_1$ and DP$_2$ can compute the summation of ε_1 and ε_2, denoted as $(\varepsilon_1 + \varepsilon_2)$. The decryption of $(\varepsilon_1 + \varepsilon_2)$ is equal to the addition of the plain counterparts n_1 and n_2, denoted as $(n_1 + n_2)$ = $D(\varepsilon_1 + \varepsilon_2)$, where $D()$ is a decryption function.

Homomorphic encryption can be categorised into fully and partially (somewhat) homomorphic schemes [55]. Fully homomorphic schemes can perform addition and multiplication [59] or any arbitrary calculation [28]. However, existing fully homomorphic schemes are computationally not efficient due to their complex encryption and decryption operations. Partially homomorphic encryption schemes only support a limited number of operations on encrypted data, however, they are much faster and thus more practical [55].

The idea of the encryption based secure summation protocol (ESS) is to use a public and

Algorithm 2: Encrypted secure summation protocol (ESS) [72, 74]

```
Input  : $P$ - Number of DPs
         $n_i$ - The secret numerical input value of DP$_i$, $1 \leq i \leq P$
Output: $\mathcal{E}$ - Final sum where $\mathcal{E} \leftarrow \sum_1^P n_i$
TP generates $pk$ and $sk$ of public and private key pair, respectively
TP sends $pk$ to all DPs
DPs receive $pk$
for $i \in [1, 2, \cdots, P]$ do                              // All DPs follow the same steps
    $\varepsilon_i \leftarrow E(n_i, pk)$                                  // Encrypt the secret input
    if $i == 1$ then                                  // Check if the current DP is DP$_1$
        DP$_1$ sends the partial sum $s_1 \leftarrow \varepsilon_1$ to DP$_2$
    end
    else
        DP$_i$ receives $s_{i-1}$           // DP receives the partial sum from the previous DP
        DP$_i$ computes partial sum $s_i \leftarrow s_{i-1} + \varepsilon_i$                // Compute the partial sum
        if $i < P$ then                           // Check if the current DP is not DP$_P$
            DP$_i$ sends $s_i$ to DP$_{i+1}$                  // Send the partial sum to the next DP
        end
        else
            DP$_P$ sends $s_P$ to TP                  // Send the final partial sum to the TP
        end
    end
end
TP receives $s_P$
TP gets the $\mathcal{E} \leftarrow D(s_P, sk)$                    // Decrypt the received final sum value
TP sends final sum $\mathcal{E}$ to DPs
DPs receive final sum $\mathcal{E}$
```

private key pair for encrypting and decrypting the private inputs, respectively, as detailed in Algorithm 2 [72, 74]. ESS uses the partially homomorphic Paillier cryptosystem [59]. In this protocol a third party (TP) is employed to facilitate the secure summation across P DPs. The TP first generates the public (pk) and private (sk) key pair. The TP then sends the public key pk to all DPs to encrypt their private inputs, while the private key sk is kept secret with the TP (lines 1 and 2).

Similar to BSS, the DPs communicate in a round robin fashion which initiates by DP$_1$. As shown in Algorithm 2, each DP$_i$ encrypts its input n_i using the function *E()* and the public key pk (line 5). The encrypted partial sum s_i of each DP is then sent to the next DP$_{i+1}$ (line 7), which adds s_i to its own encrypted input ε_{i+1} (line 11). The final encrypted sum s_P is then sent back to the TP by DP$_P$ (line 16). Using the function *D()* and its own secret key sk, the TP decrypts s_P to get the final sum $\mathcal{E}$ (line 21), which it then sends to all DPs (line 22).

2.3 Salted Secure Summation (SSS)

Salting is a cryptography technique that has been used to improve privacy against dictionary attacks on one-way hash functions where an additional secret value is concatenated with the value that is to be encrypted or encoded [54, 65, 72]. Salts are closely related to the concept of nonce, which is an arbitrary (random) number used only once in a cryptographic communication [61]. As detailed in Algorithm 3, in the SSS protocol each DP$_i$ adds

Algorithm 3: Salted secure summation protocol (SSS) [72]

```
Input  : $P$ - Number of DPs
         $n_i$ - The secret numerical input value of DP$_i$, $1 \leq i \leq P$
Output: $\mathcal{E}$ - Final sum where $\mathcal{E} \leftarrow \sum_1^P n_i$
1  for $i \in [1, 2, \cdots, P]$ do                        // All DPs follow the same steps
2      DP$_i$ computes a random salting value $r_i$
3      DP$_i$ sends $r_i$ to the TP
4      TP receives $r_i$
5      if $i == 1$ then                                    // Check if the current DP is DP$_1$
6          DP$_1$ computes partial sum $s_1 \leftarrow n_1 + r_1$
7          DP$_1$ sends $s_1$ to DP$_2$
8      end
9      else
10         DP$_i$ receives $s_{i-1}$             // DP receives the partial sum from previous DP
11         DP$_i$ computes $s_i \leftarrow s_{i-1} + n_i + r_i$          // Compute the partial sum
12         if $i < P$ then                          // Check if the current DP is not DP$_P$
13             DP$_i$ sends $s_i$ to DP$_{i+1}$            // Send the partial sum to the next DP
14         end
15         else
16             DP$_P$ sends $s_P$ to TP                  // Send the final partial sum to the TP
17         end
18     end
19 end
20 TP receives $s_P$
21 TP computes $r \leftarrow \sum_{i=1}^P r_i$                      // Add all random salt values
22 TP computes final sum $\mathcal{E} \leftarrow s_P - r$          // Substract salt values from final sum
23 TP sends final sum $\mathcal{E}$ to DPs
24 DPs receive final sum $\mathcal{E}$
```

an additional random (salt) value r_i to its own private input n_i. Similar to ESS, the SSS protocol requires a TP because individual salt values cannot be shared between the DPs in order to prevent collusion risks.

In lines 2 and 3, each DP$_i$ starts by generating a random salt value r_i and sends it to the TP. Then DP$_1$ adds n_1 to r_1 and the resulting partial sum s_1 is sent to DP$_2$ (lines 6 and 7). Following the same steps, each following DP$_i$ (with $i > 1$) receives the previous partial sum s_{i-1} from DP$_{i-1}$ and adds its own private input n_i and random salting value r_i to the partial sum s_{i-1} and then sends the resulting sum s_i to the next DP (lines 9 to 18). For example, DP$_2$ adds its private input n_2 and r_2 with s_1 and the resulting sum s_2 is sent to DP$_3$. Finally, DP$_P$ sends the final partial sum s_P to the TP (line 16). The TP computes the final sum $\mathcal{E}$ by subtracting the random salts of all DPs from s_P (lines 21 and 22) and the resulting $\mathcal{E}$ is sent to all DPs (line 23).

The main weakness in this secure summation protocol is that each DP requires to send its random salt value to the TP. Any collusion between the TP and the participating DPs can compromise the privacy of a private input of a non-colluding DP. We discuss the weaknesses of the SSS protocol in more detail in Section 3.

Algorithm 4: Randomly-shared secure summation protocol (RSS) [69]

```
Input  : $P$ - Number of DPs
         $n_i$ - The secret numerical input value of DP$_i$, $1 \leq i \leq P$
Output: $\mathcal{E}$ - Final sum where $\mathcal{E} \leftarrow \sum_1^P n_i$
1  for $i \in [1, 2, \cdots, P]$ do                          // All DPs follow the same steps
2      DP$_i$ computes a list of random shares $R_i \leftarrow$ computeShares($n_i, P$), such that
       $\sum_{j=1}^P r_i^j \in R_i = n_i$
3      for $j \in [1, 2, \cdots, P]$ do                      // All DPs follow the same steps
4          if $i \neq j$ then                                // Send each share to its corresponding DP
5              DP$_i$ sends a random share $r_i^j$ to DP$_j$
6              DP$_j$ receives $r_i^j$
7          end
8      end
9      DP$_i$ computes $s_i = \sum_{j=1}^P r_j^i$          // Add the received shares with its share $r_i^i$
10     if $i \neq 1$ then                                    // Check if the current DP is not DP$_1$
11         DP$_i$ sends $s_i$ to DP$_1$                      // Send the partial sum value to DP$_1$
12     end
13 end
14 DP$_1$ receives $\forall_{i=2}^P s_i$              // DP$_1$ receives the partial sum values from other DPs
15 DP$_1$ computes final sum $\mathcal{E} \leftarrow \sum_{i=1}^P s_i$       // Add all the partial sum values
16 DP$_1$ sends final sum $\mathcal{E}$ to all other DPs
17 Other DPs receive final sum $\mathcal{E}$
```

2.4 Randomly-shared Secure Summation (RSS)

Secret sharing [17] (also known as secret splitting [64]) refers to methods for distributing a secret amongst a group of participants, each of whom is allocated with a share of the secret. The general idea of the secret sharing scheme is that the secret value of a participating party can be reconstructed only when a sufficient number of shares are combined together. In such settings each individual share is of no use as it cannot be used to construct the private input of a party. Threshold secret sharing schemes were first introduced by Shamir [66] and Blakley [6] in 1979. Different other secret sharing schemes were proposed over the years [41, 44], while their properties were studied in [5, 13, 17, 41].

A summation protocol that uses secret sharing was first proposed by Benaloh [17]. A similar protocol was also proposed in 1993 by Chor and Kushilevitz [12] for computing the summation of private input from P DPs, where this protocol is more efficient (it requires a smaller number of messages) than the one by Benaloh for scenarios where less than $P-2$ parties are colluding. Another similar protocol by Tassa and Cohen [69] recently extended the BSS protocol (as described in Section 2.1) to multiple parties by adapting a secret sharing technique proposed in [17]. As detailed in Algorithm 4, the protocol by Tassa and Cohen, named RSS, uses a set of P random shares of each private input value n_i to compute the final summation [69]. Similar to BSS, the RSS protocol can be used in SMC scenarios where no third party is available.

Each DP$_i$ first computes P random shares (r_i^j, with $1 \leq j \leq P$) of its n_i by using the function *computeShares()* (line 2), where the sum of all its shares equals to n_i, such that $n_i = \sum_{j=1}^P r_i^j$. Each share r_i^j is sent to the corresponding DP$_j$ (lines 3 to 8). Each DP$_i$ then adds all the random shares it receives from the other DPs with the random share of its own

n_i to calculate its partial sum s_i (line 9). Each DP$_i$ sends its partial sum s_i to DP$_1$ (lines 10 to 12). DP$_1$ adds all s_is to compute the final summation $\mathcal{E}$ (line 15). Finally, the computed $\mathcal{E}$ is distributed to all other DPs (line 16).

2.5 Distributed Secure Summation (DSS)

Recently, Mehnaz et al. [51] proposed a secure summation protocol that uses collusion resistant anonymisation to provide privacy to the individual values held by each party. The DSS protocol employs a third party TP to conduct the summation where parties are communicating in a round robin fashion similar to the BSS protocol described in Section 2.1. In the proposed DSS protocol the collusion resistant anonymisation is achieved by each party randomly permuting the input values it received from the previous party before sending these received values to the next party in the protocol. The DSS protocol, as outlined in Algorithm 5, uses the ElGamal cryptosystem [23] where each message communicated between the parties is encrypted and decrypted using a public and private key pair.

The ElGamal cryptosystem, as first described by Taher ElGamal in 1985 [23], is a public key cryptosystem based on the discrete logarithm problem for a group of parties, where every party has a private and public key pair (sk, pk). For example, assume a party named Alice has a prime p and an integer g, whose powers modulo p generate a large number of elements. Hence, Alice generates its private and public key pair (sk, pk), where pk = g^{sk} mod p, where mod is the modulo operation. Suppose there is another party Bob who wishes to send a message m to Alice. Bob first generates a random number k which must be less than p, i.e. $1 \leq k < p$. He then computes c_1 = g^k mod p and c_2 = $m \oplus c_1^k$, where $\oplus$ denotes the bit-wise exclusive-OR (XOR). Bob sends the ciphertext (c_1, c_2) to Alice. Upon receiving the ciphertext, Alice computes m = $(c_1^{sk}$ mod $p) \oplus c_2$.

As detailed in Algorithm 5, the DSS protocol requires each participating party to have its own public and private key pair. First the TP generates its public (pk_{TP}) and private (sk_{TP}) key pair (line 1) and sends pk_{TP} to all DPs (lines 2 and 3). DSS contains three phases: *input preparation* (lines 4 to 20), *anonymisation* (lines 21 to 36), and *sum computation* (lines 37 to 40).

In the input preparation phase, each DP$_i$ first generates a public (pk_i) and private (sk_i) key pair and sends its public key pk_i to the TP and every other DPs (lines 5 to 8). Then each DP$_i$ divides its private input n_i into δ ($\geq$ 1) segments (lines 9 and 10). In line 9, each DP$_i$ uses the function *computeSecretShares()* to generate a list of secret shares A_i = $[\alpha_i^1, \alpha_i^2, \cdots, \alpha_i^\delta]$ uniformly and randomly such that $\sum_{j=1}^{\delta} \alpha_i^j = 1$.

These shares are used in the function *computeSegments()* to divide n_i into δ segments N_i where $N_i = [s_1, s_2, \cdots, s_\delta]$ such that $\sum_{j=1}^{\delta} s_j = n_i$ and $s_j = \alpha_i^j \cdot n_i$ (line 10). In lines 12 to 18, each DP$_i$ encrypts segments in N_i. At first each DP$_i$ encrypts each segment in N_i using TP's public key pk_{TP} (line 13). Next, each encrypted segment ϵ is encrypted using the public keys of all DPs in an order of DP$_1$ to DP$_P$ (lines 14 to 16). Each encrypted segment is added into a list N_i' and is sent to the TP (line 19). Once the TP received the lists of encrypted segments from all DPs, it adds all the segments in each list into a list N' (line 21). Next each segment in N' needs to be decrypted before they can be summed together.

Since each segment has the encryption order from the public keys of DP$_1$ to DP$_P$, the appropriate decryption order is using the corresponding private keys of DP$_P$ to DP$_1$. Hence the TP sends the list N' first to DP$_P$ (line 22). In lines 23 to 35, from DP$_P$ to DP$_1$ decrypts each segment in N' using its private key sk_i. Once the segments are decrypted using their private keys (line 26), each DP$_i$ sends the list N' to DP$_{i-1}$ (lines 28 to 34). Before sending the list N' to DP$_{i-1}$, DP$_i$ randomly re-orders the elements in N' using the function *ran-*

Algorithm 5: Distributed secure summation protocol (DSS) [51]

```
Input  : $P$ - Number of DPs
         $n_i$ - The secret numerical input value of DP$_i$, $1 \le i \le P$
Output: $\mathcal{E}$ - Final sum where $\mathcal{E} \leftarrow \sum_1^P n_i$
1  TP generates $pk_{TP}$ and $sk_{TP}$ of public and private key pair, respectively
2  TP sends $pk_{TP}$ to all DPs
3  DPs receive $pk_{TP}$
4  for $i \in [1, 2, \cdots, P]$ do                          // All DPs follow the same steps
5      DP$_i$ generates $pk_i$ and $sk_i$ of public and private key pair, respectively
6      DP$_i$ sends $pk_i$ to TP                                  // Send public key to TP
7      DP$_i$ sends $pk_i$ to other DPs                           // Send public key to other DPs
8      TP and other DPs receive $pk_i$
9      DP$_i$ computes a list of secret shares $A_i \leftarrow$ computeSecretShares($\delta$)
10     DP$_i$ computes a list of segments $N_i \leftarrow$ computeSegments($n_i, A_i$)
11     $N'_i = []$                          // Initialise a list to keep the encrypted segments
12     for $j \in [1, 2, \cdots, \delta]$ do                      // Encrypt each segment
13         $\varepsilon_j \leftarrow E(N_i[j], pk_{TP})$          // Encrypt the segment with TP's public key
14         for $k \in [1, 2, \cdots, P]$ do
15             $\varepsilon_j \leftarrow E(\varepsilon_j, pk_k)$  // Encrypt each segment with DP$_k$'s public key
16         end
17         $N'_i[j] = \varepsilon_j$                             // Add each encrypted segment to $N'_i$
18     end
19     DP$_i$ sends $N'_i$ to TP                                 // Send the encrypted segments to TP
20 end
21 TP receives $N'_i$ and adds each $N'_i$ to list $N'$, such that $\forall_{i \in [1 \cdots P]} \forall_{j \in [1 \cdots \delta]}$ $N'[i \cdot j] = N'_i[j]$
22 TP sends $N'$ to DP$_P$
23 for $i \in [P, P-1, \cdots, 1]$ do              // All DPs (from last to first) follow the same steps
24     DP$_i$ receives $N'$
25     for $j \in [1, 2, \cdots, |N'|]$ do
26         $N'[j] = D(N'[j],\ sk_i)$                     // Decrypt each segment with the private key
27     end
28     $N'$ = randomShuffle($N'$)                   // Re-order elements of $N'$ by random shuffling
29     if $i > 1$ then
30         DP$_i$ sends $N'$ to DP$_{i-1}$                          // Send $N'$ to the previous DP
31     end
32     else
33         DP$_1$ sends $N'$ to TP                                  // DP$_1$ sends $N'$ to the TP
34     end
35 end
36 TP receives $N'$
37 TP computes $N$, i.e. $N[i] = D(N'[i], sk_{TP}), 1 \le i \le |N'|$
38 TP computes final sum $\mathcal{E} \leftarrow \sum_{i=1}^{|N|} N[i]$          // Add all the segment values
39 TP sends final sum $\mathcal{E}$ to all DPs
40 DPs receive final sum $\mathcal{E}$
```

domShuffle() (line 28). This random shuffling of elements in N' helps to hide the source of each value, i.e. to hide information about the DP that each encrypted value belongs to. Finally DP$_1$ sends the list N' to the TP (line 30).

In the final phase (sum computation), the TP decrypts each segment in N' with its private key sk_{TP} and adds each segment into the list N (line 37). Finally, the TP computes the final sum $\mathcal{E}$ by adding each segment in N (line 38) and $\mathcal{E}$ is sent to all DPs (line 39).

2.6 Two Segmented Secure Summation (TSS)

As outlined in Algorithm 6, we now propose a novel secure summation protocol, named as *two segmented secure summation* (TSS), which combines the homomorphic encryption scheme with a secret sharing scheme. We adapt the secret splitting technique [17, 64], described in Section 2.4, to share a random value anonymously among the DPs which makes our protocol less vulnerable to collusion, as we discuss in Section 3. Similar to both ESS and SSS, the TSS protocol employs a TP to compute the final summation value, where the aim is to compute the final summation with less communication steps between the DPs and at the same time to prevent collusion between upto $P - 2$ parties.

The TSS protocol contains three main steps. In step 1 (lines 1 to 3 in Algorithm 6), the TP generates the required public (pk) and private (sk) key pair for encryption and decryption of messages (line 1), respectively, and distributes pk to all DPs (line 2). In step 2 (lines 4 to 24), similar to SSS, each DP$_i$ first generates a random salting value r_i (line 5). Each DP$_i$ then encrypts the summation of its private input n_i and r_i using pk into ε_i (line 6).

Each DP$_i$ then divides its random salting value r_i into two random segments r'_i and r''_i using the function *divideSalt()* (line 7). The segment r''_i is sent to another DP that is selected randomly (line 8) while the random segment r'_i is not shared with any other party in the protocol. The aim of this splitting of r_i is to keep the random salt value secret from being identified by any DP or the TP in a collusion scenario. To improve the privacy each salting value can be divided into more than two segments, however, this would require more messages to be communicated among the parties. We will provide more details on the security of the TSS protocol in Section 3. Further, to improve the privacy when sending its salt segment each DP can hide its identification information from the sent segment by making the sent message anonymous, such as sending segments using a proxy server[21, 22]. Such anonymisation would prevent the receiving DP from learning about the sender from whom it receives the segment. We refer the reader to [22] for details about such information hiding communication protocols.

Except for DP$_1$, in lines 10 to 12, each DP$_i$ adds its encrypted value ε_i to the partial sum value s_{i-1} it received from the previous DP$_{i-1}$ into the partial sum value s_i (line 15). Each DP$_i$ then sends its new partial sum result s_i to the next DP$_{i+1}$ (line 17). For example, DP$_1$ sends its encrypted partial sum value s_1 to DP$_2$. DP$_2$ computes the encrypted partial sum s_2 by adding its encrypted value ε_2 to s_1 received from DP$_1$. DP$_2$ then sends s_2 to DP$_3$, and so on. Finally DP$_P$ sends s_P to the TP (line 20).

In step 3 (lines 24 to 27), each DP$_i$ adds its remaining random segment r'_i with any random segments r''_j it received from other DP$_j$s into the partially summed random segment r^s_i, where $i \neq j$ (line 25). Each DP$_i$ then sends r^s_i to the TP (line 26). Finally, once the TP receives all r^ss from DPs and the partial sum s_P (line 28) it computes the final sum $\mathcal{E}$ by subtracting the sum of r^ss from the decrypted final partial sum (line 29) and then sends $\mathcal{E}$ to all DPs (line 30). Figure 3 shows an example of TSS for three DPs.

2.7 (*P*-1) Segmented Secure Summation (PSS)

As we discussed in Section 2.6, one limitation of the TSS protocol is the requirement of an anonymised communication network where DPs can send their segments anonymously

Algorithm 6: Two segmented secure summation protocol (TSS)

```
Input  : $P$ - Number of DPs
         $n_i$ - The secret numerical input value of DP$_i$, $1 \leq i \leq P$
Output: $\mathcal{E}$ - Final sum where $\mathcal{E} \leftarrow \sum_1^P n_i$
TP generates $pk$ and $sk$ of public and private key pair, respectively
TP sends $pk$ to all DPs
DPs receive $pk$
for $i \in [1, 2, \cdots, P]$ do                                   // All DPs follow the same steps
    DP$_i$ computes a random salting value $r_i$
    $\varepsilon_i \leftarrow E(n_i + r_i, pk)$         // Encrypt the sum of secret input and the salting value
    $r'_i,\ r''_i \leftarrow divideSalt(r_i)$                // Divide the salt into two random segments
    DP$_i$ sends $r''_i$ anonymously to DP$_j$ selected randomly, where $1 \leq j \leq P \wedge i \neq j$
    DP$_j$ receives $r''_i$
    if $i == 1$ then                                            // Check if the current DP is DP$_1$
        DP$_1$ sends $s_1 \leftarrow \varepsilon_1$ to DP$_2$                    // Send the partial sum $s_1$ to DP$_2$
    end
    else
        DP$_i$ receives $s_{i-1}$                  // Receive the partial sum from the previous DP
        DP$_i$ computes partial sum $s_i \leftarrow s_{i-1} + \varepsilon_i$
        if $i < P$ then                                     // Check if the current DP is not DP$_P$
            DP$_i$ sends $s_i$ to DP$_{i+1}$                          // Send the partial sum to the next DP
        end
        else
            DP$_P$ sends $s_P$ to TP                              // Send the final partial sum to the TP
        end
    end
end
for $i \in [1, 2, \cdots, P]$ do                                   // All DPs follow the same steps
    DP$_i$ computes $r^s_i$, such that $r^s_i = r'_i + \sum_{j=1}^P r''_j$, if $\exists\, r''_j \wedge i \neq j$
    DP$_i$ sends $r^s_i$ to TP                                    // Send the random salts to the TP
end
TP receives $r^s_i$, $i \in [1, \cdots, P]$, and $\varepsilon_P$      // TP receives the random salts and final partial sum
TP computes final sum $\mathcal{E} \leftarrow D(s_P,\ sk) - \sum_{i=1}^P r^s_i$
TP sends final sum $\mathcal{E}$ to DPs
DPs receive final sum $\mathcal{E}$
```

to other participating parties. As an alternative, if such an anonymous network setting is not available, in line 8 in Algorithm 6, each DP can divide its random salt value into $P - 1$ segments. We named this protocol variation as *(P-1) segmented secure summation* (PSS). While keeping one segment to itself each DP sends all the other segments to other DPs except to the DP which it sends its encrypted partial sum to. In the PSS approach no DP would receive both an encrypted partial sum and a segment of a salt value from the same DP. However, this variation of the protocol requires more messages to be communicated between the participating DPs.

Similar to TSS, each DP follows the same steps in Algorithm 6 except in lines 8 and 9. In line 8, each DP$_i$ divides its random salt r_i into a list of $P - 1$ segments, such that $r_i = [r_i^1, r_i^2, \cdots, r_i^{P-1}]$, while in TSS the salt is divided into two segments r'_i and r''_i only. Then in line 9, each DP_i (while keeping the random segment r_i^i secret) sends all the remaining random segments to all other DPs except to the DP which it sends its encrypted partial

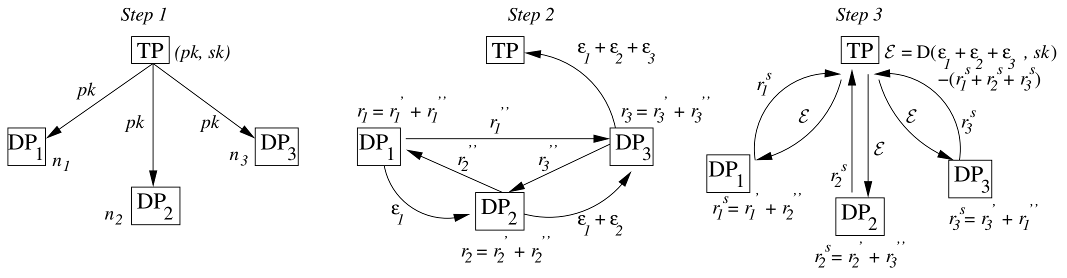

Figure 3: An example of the TSS protocol with its three main steps. Data providers DP$_1$, DP$_2$, and DP$_3$ want to compute the sum of their private input values n_1, n_2, and n_3. Three random salt values r_1, r_2, and r_3, are selected by the three DPs. In step 1, the TP initialises TSS by sending its public key, *pk*, to the three DPs. In step 2, DP$_1$ encrypts its partial sum $(n_1 + r_1)$ using pk into ε_1 and sends ε_1 to DP$_2$. DP$_2$ updates the encrypted sum it received from DP$_1$ by adding its own ε_2, and sends the updated encrypted sum value to DP$_3$. DP$_3$ updates the result it received from DP$_2$ by adding ε_3 and sends the updated partial sum to the TP. DP$_1$ then sends r_1'' to DP$_3$, DP$_2$ sends its r_2'' to DP$_1$, and DP$_3$ sends r_3'' to DP$_2$. In step 3, DP$_1$, DP$_2$, and DP$_3$ send their partially summed random salt values r_1^s, r_2^s, and r_3^s to the TP, where $r_1^s = r_1' + r_2''$, $r_2^s = r_2' + r_3''$, and $r_3^s = r_3' + r_1''$, respectively. The TP computes the final sum $\mathcal{E}$ by subtracting r_1^s, r_2^s, and r_3^s from the decrypted partial sum value it received from DP$_3$ and sends $\mathcal{E}$ to all DPs. Arrows represent the communication between parties.

sum to. For example, DP$_1$ can send its segments to other DPs except to DP$_2$ since DP$_2$ receives ϵ_1 from DP$_1$, while DP$_P$ can send its segments to any DP. Similar to TSS, once the random segments are distributed among the DPs, each DP$_i$ can add the random segments it receives from other DPs with its unshared random segment r_i^i into the partially summed random salt r_i^s and sends r_i^s to the TP (same as in line 26 in Algorithm 6). Finally, once the TP receives all r^ss from DPs and ε_P (same as in line 29 in Algorithm 6) it computes the final sum $\mathcal{E}$ by subtracting the sum of r^ss from the decrypted final partial sum and then sends $\mathcal{E}$ to all DPs. We analyse and empirically evaluate PSS in Sections 3 and 4, respectively.

2.8 Homomorphically Shared Secure Summation (HSS)

As detailed in Algorithm 7, we propose another novel secure summation protocol, named as *homomorphically shared secure summation protocol* (HSS), which uses a partially homomorphic encryption scheme to perform secure communication between the DPs. The HSS protocol employs a TP to compute the summation of the private inputs of the participating DPs. The protocol uses a secret sharing scheme to securely compute the summation as we explain below. The HSS protocol contains three main steps.

As outlined in Algorithm 7, in the first step (lines 1 to 15) each DP$_i$ initiates the protocol by generating its own public and private key pair (pk_i, sk_i), and sends pk_i to other DPs (lines 2 to 4). In line 5 each DP$_i$ divides its private input n_i into P segments using the function *computeSegments()*. Each DP$_i$ then encrypts each of its $P - 1$ value segments separately using the public keys of the other DPs, while keeping a segment to itself without sharing it with any other party in the protocol (lines 7 to 15). Each DP$_i$ also encrypts a value 0 using its own public key pk_i (line 12) and then adds each encrypted segment ε_j to the list of encrypted segments E$_i$ (line 14). The encrypted value 0 is inserted in the corresponding

Algorithm 7: Homomorphically shared secure summation protocol (HSS)

```
Input  : $P$ - Number of DPs
         $n_i$ - The secret numerical input value of DP$_i$, $1 \le i \le P$
Output: $\mathcal{E}$ - Final sum where $\mathcal{E} \leftarrow \sum_1^P n_i$
for $i \in [1, 2, \cdots, P]$ do                                  // All DPs follow the same steps
    DP$_i$ generates $(pk_i, sk_i)$ of public and private key pair, respectively
    DP$_i$ sends $pk_i$ to other DPs
    Other DPs receive $pk_i$
    DP$_i$ computes a list of segments $N_i \leftarrow$ computeSegments$(n_i, P)$, such that $\sum_{j=1}^P s_j \in N_i = n_i$
    E$_i$ = []                                     // Initialise a list to store the encrypted segments
    for $j \in [1, 2, \cdots, P]$ do                                          // Encrypt each segment
        if $j \neq i$ then                                     // Keep a segment to itself without sharing
            $\varepsilon_j \leftarrow E(N_i[j], pk_j)$                     // Encrypt segment $j$ with DP$_j$'s public key
        end
        else
            $\varepsilon_j \leftarrow E(0, pk_i)$                          // Encrypt a value 0 with DP$_i$'s public key
        end
        E$_i$.add$(\varepsilon_j)$                                   // Add each encrypted segment to E$_i$
    end
    DP$_i$ sends E$_i$ to TP                                   // Send the encrypted segments to TP
    TP receives E$_i$ from DP$_i$
end
for $i \in [1, 2, \cdots, P]$ do                                  // All DPs follow the same steps
    $\varepsilon'_i \leftarrow \sum_{j=1}^P$ E$_j[i]$                  // Sum all the encrypted segments assigned to DP$_i$
    TP sends encrypted sum $\varepsilon'_i$ to DP$_i$
end
for $i \in [1, 2, \cdots, P]$ do                                  // All DPs follow the same steps
    DP$_i$ receives $\varepsilon'_i$ from TP                       // DP receives the encrypted partial sum
    $s_i \leftarrow D(\varepsilon'_i, sk_i) + N_i[i]$              // Add its own segment to the decrypted sum value
    DP$_i$ sends partial sum $s_i$ to TP                                 // Send the partial sum to TP
    TP receives $s_i$ from DP$_i$                                      // TP receives the partial sum
end
TP computes final sum $\mathcal{E} \leftarrow \sum_{i=1}^P s_i$              // Add the received partial sum values
TP sends final sum $\mathcal{E}$ to DPs
DPs receive final sum $\mathcal{E}$
```

position of the unshared segment in the list of encrypted segments E$_i$.

Figure 4 shows an example of the HSS protocol for three DPs. As shown in Figure 4, DP$_1$ divides its private input n_1 into three segments n_1^1, n_1^2, and n_1^3, and encrypts n_1^2 and n_1^3 into ε_1^2 and ε_1^3 using the public keys pk_2 and pk_3 of DP$_2$ and DP$_3$, respectively. DP$_1$ does not share the segment n_1^1 with any other party. DP$_1$ generates ε_1^1 by encrypting the value 0 with its public key pk_1 and then adds ε_1^1, ε_1^2 and ε_1^3 to the list E$_1$ such that E$_1$ = [ε_1^1, ε_1^2, ε_1^3].

In the second step (lines 16 to 22) each DP$_i$ first sends its list of encrypted segments E$_i$ to the TP (line 16). Next the TP iterates through each list E$_i$ sent to it by DP$_i$. The TP adds each encrypted segment in the index position i into the encrypted partial sum ε'_i (line 20), and then sends the encrypted partial sum ε'_i to the corresponding DP$_i$ in line 21. For example, as shown in Figure 4, the TP adds ε_1^1, ε_2^1, and ε_3^1 into ε'_1 and sends ε'_1 to DP$_1$. In the third step (lines 23 to 31) each DP$_i$ decrypts the received ε'_i using its private key sk_i and adds its

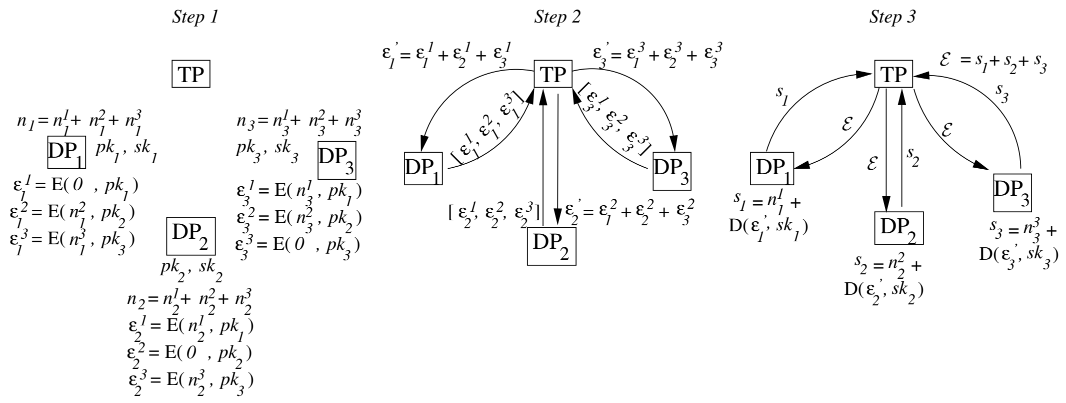
Step 1
TP
$n_1 = n_1^1 + n_1^2 + n_1^3$
DP$_1$ pk_1, sk_1
$\varepsilon_1^1 = E(0, pk_1)$
$\varepsilon_1^2 = E(n_1^2, pk_2)$
$\varepsilon_1^3 = E(n_1^3, pk_3)$
$n_3 = n_3^1 + n_3^2 + n_3^3$
pk_3, sk_3 DP$_3$
$\varepsilon_3^1 = E(n_3^1, pk_1)$
$\varepsilon_3^2 = E(n_3^2, pk_2)$
$\varepsilon_3^3 = E(0, pk_3)$
DP$_2$
pk_2, sk_2
$n_2 = n_2^1 + n_2^2 + n_2^3$
$\varepsilon_2^1 = E(n_2^1, pk_1)$
$\varepsilon_2^2 = E(0, pk_2)$
$\varepsilon_2^3 = E(n_2^3, pk_3)$
Step 2
$\varepsilon_1' = \varepsilon_1^1 + \varepsilon_2^1 + \varepsilon_3^1$
TP
$\varepsilon_3' = \varepsilon_1^3 + \varepsilon_2^3 + \varepsilon_3^3$
$[\varepsilon_1^1, \varepsilon_1^2, \varepsilon_1^3]$
$[\varepsilon_3^1, \varepsilon_3^2, \varepsilon_3^3]$
DP$_1$
DP$_3$
$[\varepsilon_2^1, \varepsilon_2^2, \varepsilon_2^3]$
$\varepsilon_2' = \varepsilon_1^2 + \varepsilon_2^2 + \varepsilon_3^2$
DP$_2$
Step 3
TP
$\mathcal{E} = s_1 + s_2 + s_3$
s_1
$\mathcal{E}$
s_3
$\mathcal{E}$
DP$_1$
$\mathcal{E}$
s_2
$s_1 = n_1^1 +$
$D(\varepsilon_1', sk_1)$
DP$_2$
DP$_3$
$s_3 = n_3^3 +$
$D(\varepsilon_3', sk_3)$
$s_2 = n_2^2 +$
$D(\varepsilon_2', sk_2)$

Figure 4: An example of the HSS protocol, as described in Section 2.8. Data providers DP$_1$, DP$_2$, and DP$_3$ want to compute the sum of their private input values n_1, n_2, and n_3. In this protocol each DP$_i$, $i \in [1, 2, 3]$, has it own public (pk_i) and private (sk_i) keys. In step 1, each DP$_i$ initialises the protocol by segmenting their private input value n_i into three segments, where $n_i = n_i^1$, n_i^2, and n_i^3. Next each DP encrypts its value segments using the public keys of other DPs separately while it encrypts a value 0 using its own public key. In step 2, each DP sends its list of encrypted value segments to the TP. The TP then adds all corresponding encrypted segments in each index position i in these lists together and sends each encrypted partial sum ε_i' to the corresponding DP$_i$. Hence, DP$_1$ receives the encrypted partial sum ε_1' which is the sum of segments $\varepsilon_1^1, \varepsilon_2^1$, and ε_3^1. In step 3, each DP decrypts the received encrypted partial sum from the TP using its private key and adds its remaining segments to compute its partial sum s_i. DP$_1$ decrypts its encrypted partial sum value ε_1' using its private key sk_1 and adds its segment n_1^1 to compute the partial sum s_1. DP$_2$ and DP$_3$ follow the same steps and finally each DP$_i$ sends its partial sum s_i to the TP. The TP computes the final sum $\mathcal{E}$ by adding s_1, s_2, and s_3, together, and sends $\mathcal{E}$ to all DPs.

remaining segment $N_i[i]$ to compute the partial sum s_i (line 25). Each DP$_i$ then sends s_i to the TP (line 26). Finally the TP computes the final sum $\mathcal{E}$ by adding all partial sum results it receives from the DPs (line 29) and then sends $\mathcal{E}$ to all DPs (line 30).

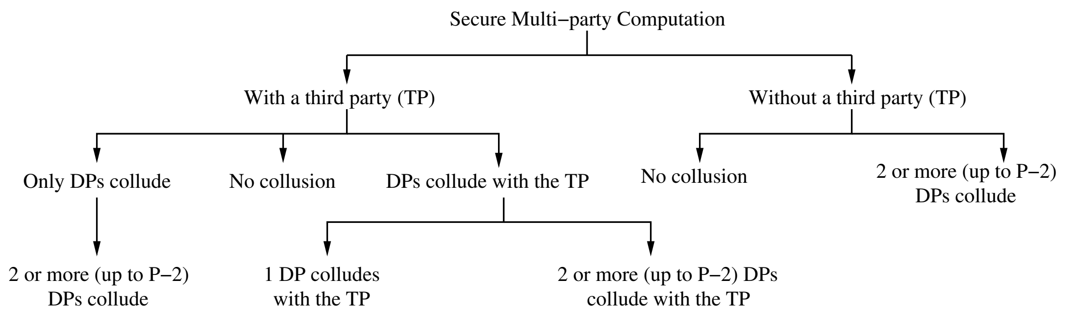
Secure Multi–party Computation
With a third party (TP)
Without a third party (TP)
Only DPs collude
No collusion
DPs collude with the TP
No collusion
2 or more (up to P–2) DPs collude
2 or more (up to P–2) DPs collude
1 DP colludes with the TP
2 or more (up to P–2) DPs collude with the TP

Figure 5: Collusion scenarios that are possible in secure multi-party computation (SMC) applications with and without a third party (TP).

Because of the private value segmentation and encryption of segments, in this protocol neither the TP nor a set of $(P-2)$ DPs is able to identify the private input or a segment value of a non-colluding DP. We analyse the security of HSS under different collusion scenarios in more detail in the following section.

3 Analysis of Protocols

We now analyse the eight presented secure summation protocols with regard to their privacy (Section 3.1), and their computation and communication complexities (Section 3.2). We assume the participating parties (data providers, DPs, and the third party, TP) follow the honest-but-curious (HBC) adversary model [47], as described in Section 1. All parties follow the steps according to the protocol specifications such that at the end of each summation protocol all DPs always receive the correct summed result. Hence the correctness holds for each protocol. Next, as illustrated in Figure 1, we analyse the applicability of each protocol in terms of privacy according to if (1) only DPs are participating or (2) a TP is also involved in the protocol. We assume $P \geq 3$ DPs.

3.1 Privacy Analysis

Collusion between the participating parties is a privacy risk in many real-world multi-party applications [3]. Figure 5 shows the different possible collusion risk scenarios under SMC.

As can be seen from this figure, for the SMC model without the TP, a group of DPs can collude with each other to identify the private input of one or more other DP(s). We consider two possible collusion scenarios under this model: (1) no collusion and (2) collusion between 2 or more DPs (at most $P-2$), where they aim to identify the private input(s) of one or more other non-colluding DP(s) they communicate with. For the SMC model without the TP the risk of collusion between $P-1$ DPs is not solvable if all DPs obtain the final result. For example, in a four-party protocol if three of the DPs collude with each other then they can learn the private input of the remaining DP by subtracting their own input values from the final summation result.

With respect to a subset of participants in the protocol, a SMC protocol can be considered as secure if it does not enable those participants to learn information about the inputs of other participants beyond what is implied by the final output and their own inputs, even if the subset of participants are computationally unbounded. Such a computation is computationally private if it achieves the same goal when the participants are polynomially bounded. We call such a subset of participants that collude in the protocol a *Coalition* [2, 36].

Following the current standard approach for defining security of SMC protocols [30], the property of ensuring the security of a protocol against what a coalition can learn during the computation can be computed based on its input and the output. This property can be proven by building a simulator [30, 47]. This simulator is given the input and output of the colluding parties and it computes a view that is indistinguishable (denoted by $\overset{c}{\equiv}$) from the coalition's view in the real execution of the protocol. Hence, we formally define the security of a SMC protocol under a semi-honest coalition as follows [34].

Definition 1. *Privacy of a SMC protocol against a semi-honest coalition* [30, 34]. Let π be a protocol for computing a SMC function f. The protocol π with P parties securely computes f in the presence of a semi-honest coalition C, if there exists a polynomial-time simulator S_C such that $\{S_C(\mathbf{x}, f(x))\}_\mathbf{x} \overset{c}{\equiv} \{view_C^\pi(\mathbf{x})\}_\mathbf{x}$, where $view_C^\pi(\mathbf{x})$ denotes the view of the coalition C including any input or messages with partial results that the colluding parties may share during the protocol execution and $\mathbf{x} \in \{0,1\}_1{}^* \times \cdots \times \{0,1\}_P{}^*$.

Next we explain each collusion scenario shown in Figure 5 in more details. In the no collusion scenario all the participating parties can be trusted and no actions are carried out by a DP to learn private input of another DP. However, such a scenario might be impractical in some real situations. The second scenario (2 or more DPs collude) is more realistic in real-world situations where multiple DPs (at most $P-2$) might be colluding with each other to learn a private input of another DP [47].

For the SMC model with a TP, collusion can occur either between the DPs, or between the TP and one or more DPs. Apart from the two collusion scenarios described above, DPs can also collude with the TP in two different ways: (1) one DP colludes with the TP to identify the private input of another DP that it communicates with, and (2) two or more (at most $P-2$) DPs collude with the TP to learn the private input(s) of one or more other non-colluding DP(s). Since the TP is assumed to follow the honest-but-curious (HBC) model both of these collusion scenarios are possible in realistic situations. Similar to the SMC model without the TP, the risk of collusion between $P-1$ DPs and the TP is not solvable if all DPs obtain the final result. We next analyse the privacy of each secure summation protocol in terms of these collusion scenarios.

- The BSS protocol [16] in Section 2.1 is susceptible to collusion since the private input of any DP can be identified if its two adjacent DPs collude with each other. For example, the BSS protocol with three DPs has a collusion risk if DP$_1$ and DP$_3$ collude with each other, which enables them to learn the private input of DP$_2$. Hence, the adding of a random value in the BSS initialisation does not provide privacy against collusion for partially summed values computed in the intermediate protocol steps.

  Zhu et al. [77] recently proposed an adapted BSS protocol for four or more parties that masks the private input of a DP by adding or subtracting random numbers based on a random boolean decision (*true* or *false*) sent by another set of DPs who are selected randomly. These masked values are then summed by using a modulo operation. This approach is secure when the adjacent DPs collude with each other to learn the private input of any other DP. However, the private input of a DP can be identified if all DPs which have secretly communicated with the given DP in the masking phase collude. We refer the reader to [77] for details.

- The ESS protocol [74, 72] in Section 2.2 uses a homomorphic encryption scheme to sum the private inputs of the participating DPs. ESS requires a TP to generate the keys (public and private) for encryption and decryption of messages. Any collusion only between DPs does not allow to learn the private input of a non-colluding DP because the private key is only known to the TP. However, a collusion between two adjacent DPs (in the ring communication) and the TP can compromise the privacy of the private input of another DP.

  For example let us assume three DPs, DP$_1$, DP$_2$, and DP$_3$, want to compute the sum of their respective private inputs n_1, n_2, and n_3. Assume the TP and DP$_2$ are colluding and they want to know the private input of DP$_3$. If DP$_2$ reveals to the TP the encrypted partial sum that it has sent to DP$_3$ then the TP can deduct the decrypted value of the partial sum from the final sum to identify the private input of DP$_3$. Hence, the ESS protocol does not provide privacy against collusion between the TP and several participating DPs since each encrypted message with a partially summed result can be decrypted by the TP which can reveal the private input of another DP.

- The SSS protocol [72], described in Section 2.3, allows each DP to generate their own random value (salt) to add to their private input value. Since the DPs do not share their own random value with any other DP, collusion between DPs does not compromise the privacy of a private input value. However, the SSS protocol requires each individual salt value to be sent to the TP to compute the final summed value. Any collusion between DPs and the TP compromises the privacy of SSS because DPs can obtain the private value of another DP.

  For example, assume an SSS protocol with four DPs, DP$_1$ to DP$_4$, where the communication of partial summation results occurs from DP$_1$ to DP$_2$, DP$_2$ to DP$_3$, and DP$_3$ to DP$_4$. In such a setting, a collusion between DP$_1$, DP$_3$, and the TP can identify the private input of DP$_2$. Now let us assume the TP and DP$_1$, respectively, reveal the random salt value of DP$_2$ and the partial sum value to DP$_3$. With this information DP$_3$ can subtract the random salt value of DP$_2$ and the partial sum value of DP$_1$ from the partial sum value it received from DP$_2$ to identify the private input of DP$_2$. Hence, SSS is not secure against collusion between the TP and two or more DPs.

- Compared to BSS, the RSS protocol [69] described in Section 2.4 is secure against collusion between DPs that aim to learn another DP's private input because each DP keeps a random share of its own private value hidden from all the other DPs. Even if DP$_1$ colludes with $P-3$ other DPs, for example, DP$_1$ still cannot identify the individual private value of each remaining non-colluding DP because each non-colluding DP adds its own random share to the partial sum. Hence, the RSS protocol guarantees privacy against collusion between at most $P-2$ DPs.

  In 1993, Chor and Kushilevitz [12] also proposed a similar protocol for computing the summation of the private inputs of P DPs. Similar to RSS, in this protocol each party DP$_i$ first randomly chooses P elements $z_{i,1}, z_{i,2}, \cdots, z_{i,P}$ for its input n_i, such that $n_i = \sum_{j=1}^{P} z_{i,j}$, and then sends $z_{i,j}$ to DP$_j$. Then each DP$_i$ computes the partial sum $w_i = \sum_{j=1}^{P} z_{j,i}$ and sends w_i to DP$_P$. Finally, DP$_P$ computes the final sum $\sum_{i=1}^{P} w_i$. Since each DP$_i$ keeps a random element $z_{i,i}$ of its input n_i without sharing it with any other DPs, this protocol also guarantees privacy against collusion between at most $P-2$ DPs similar to RSS.

  In terms of communication, RSS and the protocol proposed in [12] require the same number of messages to be sent between DPs to achieve privacy against collusion between at most $P - 2$ DPs. However, the protocol by Chor and Kushilevitz requires more communication rounds compared to RSS, because [12] uses a sequential communication (messaging) scheme while sending of messages in RSS is more synchronised and all parties can send their messages at the same time. Hence, communication-wise the protocol by Chor and Kushilevitz [12] is slower than RSS. Furthermore, as we discussed in Section 2.4, another similar protocol was also proposed by Benaloh [17]. However, this protocol is less efficient compared to [12] because of the use of discrete logarithms for computing shares.

Urabe et al. [70] proposed a similar variation of such a protocol for secure summation. In this protocol each DP$_i$ first divides its private input into $(P + 1 - i)$ segments. For example, in a protocol with five DPs, the first DP divides its private input into five segments, the second DP divides its private input into four segments, and so on. Each DP then sends each of its segments to the corresponding other DPs while keeping one segment to itself. Once all the segments are distributed among the DPs, each DP sums all the segments it has received and its remaining local segment, and sends this partial result to the TP. The TP then adds all these partial results together to get the final summed result. In contrast to the RSS protocol [69], this protocol does not provide privacy against collusion by $P-2$ DPs because of the asymmetric (imbalance) secret sharing scheme. We refer the reader to [70] for more details.

- The DSS protocol proposed by Mehnaz et al. [51] uses the ElGamal cryptosystem [23]. Similar to ESS and SSS, this protocol also employs a TP to conduct the summation of private inputs of the participating DPs. Similar to RSS, DSS also uses a segmentation technique to ensure each private input value is divided into $\delta > 1$ segments. Each segment is encrypted using the public keys of all participants in the input preparation phase which requires each segment to be decrypted by each party sequentially in the sum computation phase. Since each DP encrypts all its segments using the TP's public key a collusion between a set of DPs cannot identify the private input of another DP. In the anonymisation phase, after the segments are decrypted by each DP using its own secret key, each DP randomly re-orders the segments before it sends the list of segments to the next DP. Such re-ordering of segments ensures even a collusion between the TP and $P - 2$ DPs cannot learn the private input of a non-colluding DP. This is because the TP cannot identify the individual encrypted value segments of each DP separately in each iteration of the summation phase. Hence the TP cannot learn the correct segments of a non-colluding DP out from all the unencrypted segments of all DPs in the sum computation phase. Therefore the DSS protocol provides privacy against collusion between the TP and $P - 2$ DPs.

  Similar to DSS, another recent work by Bonawitz et al. [9] also uses the ElGamal cryptosystem with a secret sharing scheme to perform the secure summation of numerical data across different numbers of parties. However, this protocol has quadratic computation and communication complexities with regard to the number of parties which makes it not practical in a real-world context with a large number of parties.

- Our proposed TSS protocol improves overall privacy of the secure summation in two ways. First, each DP generates its own random salt value to be added to its private input. The salt value of a given DP is first divided into two segments. To improve the privacy of the random salt value each DP shares only one segment with another DP to which it does not send its encrypted partial summation value, while keeping the other segment with itself. As shown in Figure 4, DP$_1$ sends its segment value r'_1 to DP$_3$ since DP$_2$ receives ε_1 from DP$_1$. Such a division and sharing of a random salt value ensures a collusion between the TP and adjacent DPs of a non-colluding DP does not allow these parties to learn the private input of the non-colluding DP. This also ensures that even a collusion between $P-2$ DPs does not allow them to learn the private input value of another DP, because each DP keeps a segment of its own random salt value hidden from all the other DPs. Without knowing the unshared salt segment value it is not possible for the colluding parties to learn the original salt value of a non-colluding DP.

Second, each DP adds the encrypted sum value of its private and salt values to the encrypted partial sums received from the previous DP. This ensures no DP is able to deduct the private input of a non-colluding party by using the partial summation result. Even a set of DPs that colludes with the TP by receiving its secret key, sk, cannot learn the private input value of a non-colluding DP because each colluding DP does not know from which other DPs it received the segments from because of the anonymisation of messages. Hence our TSS protocol provides privacy against collusion between the TP and $P-2$ DPs, following the formal definition as given in Proposition 1.

Proposition 1. The TSS protocol is secure against collusion between a subset of semi-honest parties as long as there are k non-colluding parties such that $k \geq 2$.

Proof: Let us assume a scenario where P DPs are participating in the TSS protocol and C be a coalition of DPs that collude with the TP, where $C \subset \{$DP$_1$, DP$_2$, $\cdots$, DP$_P\}$, such that $P - |C| = k$.

The view of each DP$_i$ consists of an encrypted partial sum result (ε_i) and a partial summation of salt values (r_i^s) in addition to its private input value n_i, which can be represented as $view_i(n_i, \varepsilon_i, r_i^s)$. Further, as outlined in Algorithm 6, each DP anonymises its message m that contains random salt segment r' before sending it to another DP where each message m is uniformly distributed over a message space M.

To distinguish the private value n_i of a non-colluding DP$_i$, each DP$_j$ in the coalition C can assist the TP by sending their encrypted partial sum results (ε_j) and their partial salt values (r'_j) to the TP, thus creating the coalition C's collaborative view $view_C(\bigcup_{DP_j \in C} view_j(n_j, \varepsilon_j, r_j^s), \mathcal{E})$, where $\mathcal{E}$ denotes the final summation result. This makes $view_C$ to consist of the *incomplete* set of partial sum results and random salt values since it does not contain any information from the non-colluding DPs. To learn a private input value n_i of a non-colluding party DP$_i$, the coalition C can simulate its view during the protocol, by generating random salt values (r^c) for each non-colluding DP$_i$. The simulator S_c's view can be constructed by adding views of all non-colluding DPs which can be generated as $views_S(\bigcup_{DP_j \in C} view_j(n_j, \varepsilon_j, r_j^s),$ $\bigcup_{DP_i \notin C} view_i(\emptyset, \emptyset, r_i^c), \mathcal{E})$, where each $view_i$ of a non-colluding DP$_i$ is equal to $(\emptyset, \emptyset, r_i^c)$ and $r_i^c \in \mathbb{R}$ denotes a randomly generated salt value for DP$_i$.

Since each message that consists a partial salt value sent by a party is anonymised, the probability p that a colluding DP receives a r''_i from a non-colluding DP$_i$ is $p = \frac{1}{k}$, which is at most $p = \frac{1}{2}$ when $k = 2$. However, even if a partial salt value segment r''_i of a non-colluding DP$_i$ has been correctly identified the simulator cannot correctly identify the input value n_i of DP$_i$ since it cannot correctly guess r_i^c that equals r_i^s. This is because salt segment r'_i of DP$_i$ is unknown to the simulator S_c and $(n_i + r'_i) \in$

$\mathbb{R}$. Following Definition 1, $views_S \overset{c}{\equiv} view_C$ given that r'_i of DP$_i$ is computationally indistinguishable. This implies that the TP, which receives the values r^s from all the DPs, cannot learn the n_i of DP$_i$ by subtracting the decrypted ε_i with r^s without knowing r'_i of DP$_i$, thus making n_i indistinguishable.

- Similar to TSS, in our $(P-1)$ segmented protocol (PSS) without knowing the public key pk of the TP no DP can decrypt the partial summation value it receives. Therefore, PSS is secure against any collusion where only DPs are colluding. Furthermore, each DP segments its random salting value into $P-1$ segments and sends each segment to all other DPs except for the DP that it has sent the encrypted partial sum value to, while keeping a segment value to itself without sharing it with other DPs. Because each DP always keeps a salt segment hidden while receiving segments from at least 1 and at most $P-1$ DPs, no DP can learn the random salt value of a non-colluding DP. Moreover, since each DP$_i$ adds the encrypted sum value (ε_i) of its private (n_i) and salt (r_i) values to the encrypted partial sum (ε_{i-1}) received from the previous DP$_{i-1}$, even if several DPs collude with the TP neither the TP nor a DP can calculate the private value of any other DP. Hence our PSS protocol provides privacy against collusion between the TP and $P-2$ DPs as we state in Proposition 2.

Proposition 2. The PSS protocol provides security for the private input n_i of a non-colluding DP$_i$ against a subset of colluding parties as long as there are k non-colluding parties such that $k \geq 2$.

Proof: Similar to Proposition 1, let us assume a scenario where P DPs are participating in the PSS protocol and C be a coalition of DPs colluding with the TP, where $C \subset \{\text{DP}_1, \text{DP}_2, \cdots, \text{DP}_P\}$, such that $P - |C| = k$.

After line 26 in Algorithm 6, the view of each DP$_i$ consists of an encrypted partial summation result, ε_i, the partial salt result, r^s_i, its own salt value, r_i, and its own input value n_i, which can be represented as $view_i(n_i, r^s_i, \varepsilon_i, r_i)$. To distinguish a private value n_i of a non-colluding DP$_i$, each DP$_j$ in the coalition C can assist the TP by sending its encrypted partial sum result (ε_j) and its partial salt value (r^s_j) to the TP, in addition to its own salt and input values, thus creating C's collaborative view $view_C(\{view_j(n_j, r^s_j, \varepsilon_j, r_j)|DP_j \in C\}, \mathcal{E})$. To learn a private input value n_i of a non-colluding party DP$_i$, the coalition C can simulate its view during the protocol, by adding a random value r^r_i for each non-colluding DP$_i$ as its own random salt value r_i. The simulator S_c's view can be constructed by adding views for all non-colluding DPs which can be generated as $views_S(\{view_i(\emptyset, r^s_i, \emptyset, r^r_i)|DP_i \notin C\}$ $\cup \{view_j(n_j, r^s_j, \varepsilon_j, r_j)_{DP_j}|DP_j \in C\}, \mathcal{E})$, where $r^r_i \in \mathbb{R}$.

Since the encrypted partial sum values, ε_j, sent by all DP$_j$s in the coalition C to the TP is a summation of private inputs and salt values of all DPs, the simulator S_c cannot distinguish the n_i of a non-colluding DP$_i$ due to two reasons. First, the simulator S_c cannot identify the correct r^r_i of a DP$_i$ by subtracting $\sum_{j=1}^{|C|} r^s_j$ from $\sum_{i=1}^{P} r^s_i$ because it cannot distinguish the value of r^r_i that is equal to r_i of each DP$_i$, where $r_i \in \mathbb{R}$. Second, the computation of $\mathcal{E}$ - ($D(\varepsilon_P)$ + $\sum_{i=1}^{P} r^s_i$), where the subtraction of all partial salt values r^s received from the DPs plus the decrypted value of the final partial encrypted sum ε_P from DP$_P$, where $D()$ is the decryption function, provides a summation of private inputs of k non-colluding parties. Since each $n_i \in \mathbb{R}$, the TP cannot correctly guess private input n_i of each non-colluding DP$_i$ from $\mathcal{E}$ - ($D(\varepsilon_P)$ + $\sum_{i=1}^{P} r^s_i$). This makes the simulator S_c's view $views_S \overset{c}{\equiv} view_C$ according to Definition 1. This implies that the private input n_i of a non-colluding DP$_i$ is indistinguishable from $\mathcal{E}$ and the PSS protocol provides security for the private input n_i of a non-colluding DP$_i$ against the collusion of the coalition C.

- Similar to TSS and PSS, our proposed HSS protocol, described in Section 2.8, employs a TP to conduct the summation of the private inputs of a set of DPs. Each DP first divides its private input into a list of segments and encrypts the segments using the public keys of other participating DPs, while keeping a value segment to itself without sharing this value with any other party. As detailed in Algorithm 7, HSS requires the TP to compute the summation of the corresponding encrypted segments of a given DP. The computed encrypted partial sums are then sent to the corresponding DPs to decrypt and add their own non-shared value segments. Since each message of a DP is communicated only with the TP, any form of collusion between DPs (i.e. not including the TP) cannot compromise the private value of a non-colluding DP.

  In the HSS protocol, none of the DPs sends their secret share to any other DP nor to the TP. This ensures that even when a colluding DP$_i$ sends the decrypted values of the encrypted segments it receives from a non-colluding DP$_{i-1}$ (in step 2 of HSS) to the TP, the TP still cannot learn the private input value n_{i-1} of DP$_{i-1}$ by summing these decrypted segments. Also, the homomorphic encryption scheme ensures that the encrypted partial sums can only be decrypted by each corresponding DP using its own private key. Further, as we state in Proposition 3 below, even when $P-2$ DPs collude with the TP, the private value n_{i-1} of a non-colluding DP$_{i-1}$ will still remain secret because a segment of DP$_{i-1}$ is kept secret from all the other participating parties. Hence our HSS protocol provides privacy against collusion between the TP and up to $P-2$ DPs.

  Proposition 3. The HSS protocol is secure against collusion between a subset of semi-honest parties making the private input n_i of a non-colluding DP$_i$ computationally indistinguishable by the TP even if it receives the encrypted segments in step 2 from $P-k$ colluding DPs and the unencrypted partial sums in step 3 of the HSS protocol, as long as there are k non-colluding parties such that $k \geq 2$.

  Proof: As with Propositions 1 and 2, let us assume a scenario where P DPs are participating in the HSS protocol and C be a coalition of DPs colluding with the TP, where $C \subset \{\text{DP}_1, \text{DP}_2, \cdots, \text{DP}_P\}$, such that $P - |C| = k$.

  In step 2 of Algorithm 7, each input n is divided into P segments and encrypted using the corresponding DP's public key. To distinguish the private value n_i of a non-colluding DP$_i$, each DP$_j$ in the coalition C can assist the TP by sending its secret key sk_j to the TP in addition to its list of P segments $[n_j^1, \cdots, n_j^P]$. Using these secret keys the TP can decrypt the encrypted segments it receives from each non-colluding DP$_i$ that were encrypted using the public keys of DP$_j$s. This allows the coalition C to generate its collaborative view $view_C(\{(0, [\bigcup_{DP_i \notin C} \varepsilon_i^{k-1}], [\bigcup_{DP_j \in C} n_j^i])_{DP_i} | DP_i \notin C\} \cup \{[n_j^1, \cdots, n_j^P]_{DP_j} | DP_j \in C\})$, where $(0, [\bigcup_{DP_i \notin C} \varepsilon_i^{k-1}], [\bigcup_{DP_j \in C} n_j^i])$ represents the list of segments the TP receives from a non-colluding DP$_i$, and $[\bigcup_{DP_i \notin C} \varepsilon_i^{k-1}]$ represents the list of encrypted segments of other non-colluding DPs.

  As in Propositions 1 and 2, the coalition C can simulate its view during step 3 of the protocol by adding a random value $r_i^i \in \mathbb{R}$ for each non-colluding DP$_i$ as its own segment value. To simulate the list of encrypted segments of other non-colluding DPs the simulator S_c can first compute a partial value $v_i' = s_i - (r_i^i + \sum[\bigcup_{DP_j \in C} n_j^i])$ for

Table 1: Categorisation of secure summation protocols in terms of privacy against collusion between different parties (data providers, DPs, and a third party, TP) participating in a secure multi-party computation (SMC) protocol.

SMC models	Collusion scenarios			
	No collusion	2 or more ($\leq P-2$) DPs collude	1 DP colludes with the TP	2 or more ($\leq P-2$) DPs collude with the TP
Without a TP	BSS RSS	RSS	–	–
With a TP	ESS SSS DSS TSS PSS HSS	ESS SSS DSS TSS PSS HSS	ESS SSS DSS TSS PSS HSS	DSS TSS PSS HSS

each non-colluding DP$_i$, where s_i is the partial summation result of DP$_i$. Then, to simulate the encrypted segments of other non-colluding DPs, the simulator S_c generates a list of $k-1$ random values $[z_i^m, DP_m \notin C]$ of v_i' for each non-colluding DP$_i$, such that $\sum_{DP_m \notin C} z_i^m = v_i'$. Next, to replace each encrypted segment ε_i^m of non-colluding DP$_m$, the simulator S_c encrypts each corresponding z_i^m with the respective public key pk_m of non-colluding DP$_m$ into ϵ_i^m, such that $\epsilon_i^m = E(z_i^m, pk_m)$. This makes the simulator S_c to construct its view as $views_S(\{(r_i^i, [\bigcup_{DP_m \notin C} \epsilon_i^m], [\bigcup_{DP_j \in C} n_j^i])_{DP_i} | DP_i \notin C\} \cup \{[n_j^1, \cdots, n_j^P]_{DP_j} | DP_j \in C\})$.

Since the partial sum values, s, sent by all DPs to the TP are a summation of input segments of all DPs, the simulator S_c cannot distinguish n_i of a non-colluding DP$_i$ due to two reasons. First, as in Proposition 2, the simulator S_c cannot identify the correct segment values of k (≥ 2) non-colluding DPs by subtracting n_j^i from the partial sum values s because it cannot correctly guess r_i^i of each DP$_i$. Second, each partial sum s contains the summation of value segments of k non-colluding DPs that are not known to any other party. Therefore, even if the simulator S_c subtracts the sum of all private values and the unencrypted values of ε_j' of the $P-k$ colluding DPs from the partial sums s_i, the simulator S_c cannot learn the private input n_i of DP$_i$ because each $n_i \in \mathbb{R}$. This makes the simulator S_c's view $views_S \overset{c}{\equiv} view_C$. This implies that the private input n_i of a non-colluding DP$_i$ is indistinguishable even if the TP receives the private input values and partial summation results from all colluding DPs, making the HSS protocol secure against collusion between a subset of semi-honest parties of size at most $(P-2)$.

Table 1 summarises the applicability of each of the presented secure summation protocols under different collusion scenarios. Overall, RSS is the only protocol that can be applied for the secure multi-party computation (SMC) model without a TP when two or more DPs are colluding. For the SMC model with a TP, where collusion is possible between the participating DPs and the TP, the DSS, TSS, PSS, and HSS protocols are more suitable compared to the SSS and ESS protocols as they provide the highest security.

However, as we mentioned before, in any P-party secure summation protocol if $P-1$ participating parties collude with each other they can identify the private input value of the remaining non-colluding DP which will compromise the privacy of any protocol discussed in this paper. For example if $P - 1$ DPs collude with the TP in a summation protocol that requires a TP, then the TP can calculate the private input of the single non-colluding DP. Hence, under the HBC model a secure summation protocol that can guarantee privacy when $P - 1$ parties collude is not logically possible.

3.2 Complexity Analysis

We next analyse the computational complexity of each summation protocol and then discuss the communication complexity in terms of the number of messages communicated and required communication rounds by each participating party in a protocol. In this analysis we assume the encryption and decryption functions performed by the data providers (DPs) or the third party (TP) take constant time to compute.

Computation: In BSS, ESS, and SSS each DP adds its private input to the partial summation result it receives from another DP. Hence BSS, ESS, and SSS have a computational complexity of $O(1)$. For ESS and SSS the TP computes the final summation result and sends it to all DPs which are of $O(1)$ and $O(P)$, respectively. In the RSS protocol each DP computes a list of random shares which is of $O(P)$ complexity. In the DSS protocol each DP encodes δ segments with the public keys of all DPs which is of $O(\delta \cdot P)$. In DSS the TP concatenates the list of segments of each DP into a single list and decrypts each encrypted segment with its own private key. This requires a computational complexity of $O(\delta \cdot P)$.

In TSS and PSS each DP adds all the segments it receives from other DPs which is of constant complexity and $O(P-1)$, respectively. For both these protocols the TP is required to subtract the random salt of each DP from the final partial sum which requires a $O(P)$ complexity. In the first step of the HSS protocol each participating DP computes the list of segments of its private input value and encrypts these segments with the public keys of the other DPs which is of $O(P)$ complexity. In the second step of the HSS protocol the TP iterates through each list of encrypted segments E_i of DP_i to compute the encrypted partial sum by adding corresponding encrypted segments assigned to each DP_i. This requires the TP to have a $O(P^2)$ complexity.

Communication: A participating DP needs to participate in multiple communication rounds to send and receive different numbers of messages in each protocol. In the BSS protocol each DP has to send its computed partial summation result to the next DP in the round robin communication pattern which is of $O(P)$ complexity, i.e. P messages. In the BSS protocol each DP participates in two rounds of communication for sending partial summation result to the subsequent DP and receiving the final summation result from the DP who initiates the protocol.

Similar to BSS, the TP in the ESS and SSS protocols requires a communication complexity of $O(P)$ to send the final computed sum to the participating DPs. Both these protocols require each DP to participate in three communication rounds for sending and receiving messages from other DPs and the TP, while the TP requires to participate in two rounds of communication for receiving partial results from DPs and sending the final summation result to all DPs. In the RSS protocol each DP sends its random shares to other participating DPs which is of $O(P - 1)$ complexity, and each DP requires to participate in two communication rounds to send its random shares to other parties and receive the final result from the DP who adds all random shares. In the DSS protocol each DP sends its public key to all the other participating DPs which is of $O(P - 1)$ complexity, while the TP has a communication complexity of $O(P)$ for sending the final sum to each DP. Further, in each phase of this protocol, as outlined in Algorithm 5, both the TP and a DP require to participate in

Table 2: The computation (Comp) complexity and communication complexity in terms of the number of messages communicated (Comm$_M$) and number of communication rounds required (Comm$_R$) by a single DP and the TP in each secure summation protocol for summing a single numerical value. In here P and δ represent the number of DPs and the number of segments of a value communicated by each DP, respectively.

Complexity	Secure Summation Protocol							
	BSS	ESS	SSS	RSS	DSS	TSS	PSS	HSS
Comp (DP)	$O(1)$	$O(1)$	$O(1)$	$O(P)$	$O(\delta \cdot P)$	$O(1)$	$O(P-1)$	$O(1)$
(TP)	–	$O(1)$	$O(P)$	–	$O(\delta \cdot P)$	$O(P)$	$O(P)$	$O(P^2)$
Comm$_M$ (DP)	$O(P)$	$O(1)$	$O(1)$	$O(P-1)$	$O(P-1)$	$O(1)$	$O(P-2)$	$O(1)$
(TP)	–	$O(P)$	$O(P)$	–	$O(P)$	$O(P)$	$O(P)$	$O(P)$
Comm$_R$ (DP)	$O(2)$	$O(3)$	$O(3)$	$O(2)$	$O(6)$	$O(3)$	$O(3)$	$O(5)$
(TP)	–	$O(2)$	$O(2)$	–	$O(6)$	$O(2)$	$O(2)$	$O(4)$

two rounds of communication for sending and receiving partial results from DPs and the TP, respectively.

In the TSS and PSS protocols the TP sends the final result to each DP which has a communication complexity of $O(P)$, while in PSS each DP requires $P - 2$ messages to send its segments to other DPs. Similar to the ESS and SSS protocols, in these protocols each DP is required to participate in three communication rounds for sending and receiving messages from other DPs and the TP, while the TP is required to participate in two rounds of communication for receiving partial results from DPs and sending the final summation result to all DPs. In the HSS protocol each DP sends a single message to the TP in the second and third steps of the protocol, while the TP sends the partial result of the encrypted segments and the final result, in the second and third steps, respectively, to each DP. Hence, overall a DP has a communication complexity of $O(1)$ while the TP has a communication complexity of $O(P)$ in the HSS protocol. As outlined in Algorithm 7, the TP participates in a total of four communication rounds in steps 2 and 3 of the HSS protocol to send and receive partial results and the final summed result to DPs, while each DP participates in an additional communication round in step 1 to send its public key to other DPs apart from the same set of communication rounds as required by the TP.

Table 2 summarises the computation and communication complexities of each secure summation protocol required by a single DP and the TP for calculating the summation of a single numerical value. It is important to note that some steps in these algorithms can be parallelised, but we do not describe because it is outside the main scope of this paper.

4 Experimental Evaluation

We theoretically analysed the complexities in Section 3.2, by assuming each DP has a single private value that needs to be summed with values from other DPs. However, realistically secure summation protocols are usually used to compute the sum of a list of values [51, 69]. In such a context, to compute the summation of a list of values, each protocol can be run once to compute the summation of an entire list (which would require large messages to be communicated) or multiple times to compute the sum of each value in the list (using small messages) separately. For example, assume a set of DPs each with a list containing 1,000 input values, where each value needs to be summed separately. Hence, a summation protocol can be called once to compute the sum of all values in the entire list in a single run

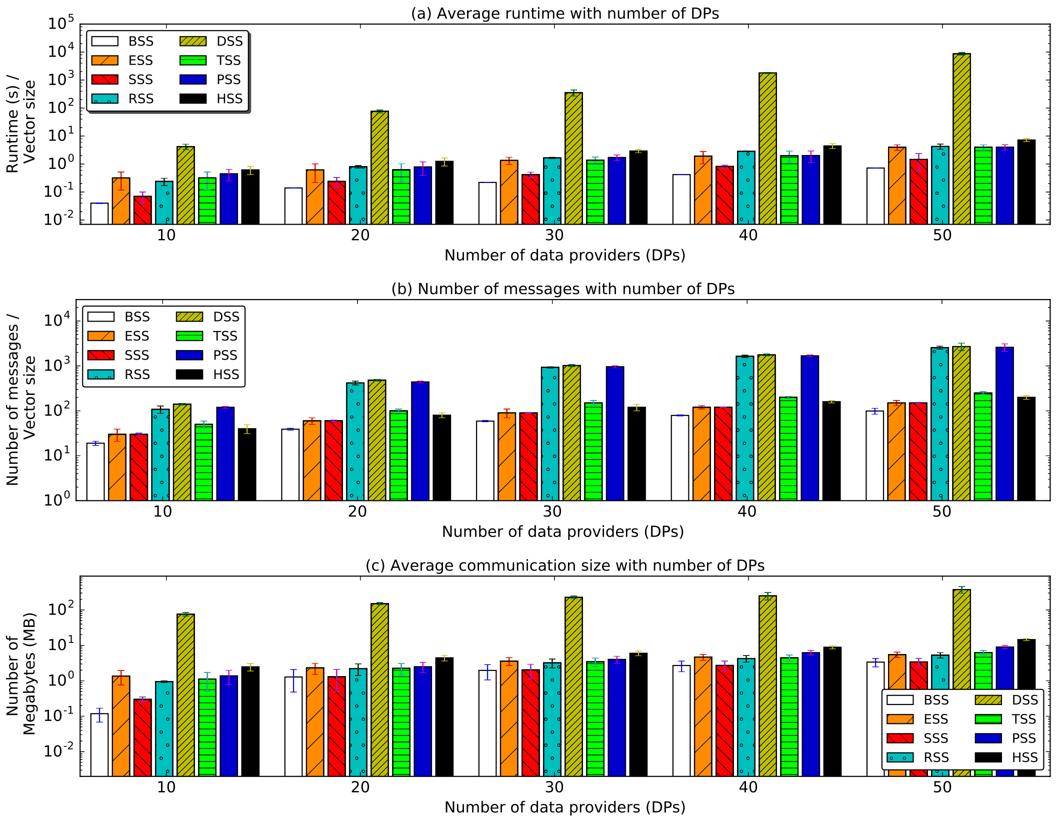

Figure 6: (a) Average runtime (in seconds) with different number of DPs, (b) total number of messages communicated between data providers (DPs) and the TP, and (c) average number of Megabytes (MB) communicated to sum a single input value.

(i.e. the messages communicated between DPs contain the whole list), the protocol can be run 1,000 times to compute the sum of each value in the list independently (i.e. a message only contains a single value), or (as another example) the protocol can be called 10 times with 100 values in each list. We next empirically evaluate each secure summation protocol with different number of data providers, input sizes, and splits of the inputs into sub-lists.

4.1 Experimental Setup

We evaluate the complexities of each presented protocol in terms of computation and communication. We evaluate the computation complexities in terms of total and average runtime to compute the final sum results, and the average number of computations required by each participating party. We use the number of messages communicated and the total message size communicated by each party to evaluate the communication complexities of each protocol.

With regard to the example scenarios discussed in Section 1, we conduct the experiments under three different scenarios with numerical values. In the first scenario we assume all the participating DPs have vectors with integer values. Such numerical data are usually

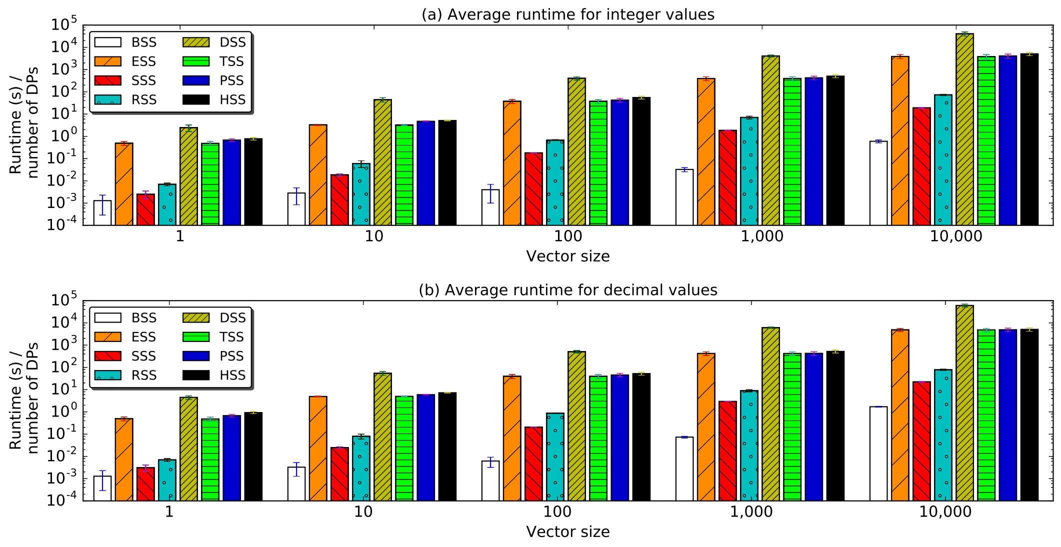

Figure 7: Average runtime (in seconds) for a DP for computing summation of different vector sizes with (a) integer and (b) decimal values.

used in medical data analysis with patients' attributes such as age, weight, height, blood pressure level, etc. In the second scenario we assume the DPs need to compute the summation over numerical vectors with decimal values. Such numerical data is usually used in transactional financial data analysis in banks over customer databases. In the third scenario we assume domestic houses in an area are connected together in an IoT application and each household (considered as a DP) wants to know whether they consume more or less electricity compared to the average electricity consumption of the whole area. Such a scenario might contain numerical vectors with mixed integer and decimal numbers.

We evaluated each summation protocol with 10, 20, 30, 40, and 50 DPs. We ran experiments for vectors with 1 to 10,000 numerical values. For ESS, TSS, PSS, and HSS we set the public and private key length to 128 bits and use partial homomorphic encryption [55]. We implemented all eight protocols in the Python programming language (version 2.7). We ran all experiments on a server with 64-bit Intel Xeon (2.4 GHz) CPUs, 128 GBytes of memory and running Ubuntu 16.04. The programs are available from the authors.

4.2 Results and Discussion

As shown in Figure 6, we measured the average runtime and the number and size of messages communicated for each protocol with different number of DPs. As Figure 6 (a) shows, DSS required a much longer average runtime per input value communicated compared to the other protocols because of the encryption and decryption steps performed by each participating party. Due to the large amount of memory required by the encryption steps we were unable to run experiments for input vectors with more than 3,000 values for DSS.

BSS required the least amount of runtime to compute the summation of private inputs of a set of DPs, however, this protocol does not provide enough privacy against collusion, as we discussed in Section 3.1. SSS is more efficient than the protocols that use homomorphic encryption and decryption functions on messages, while RSS required similar runtime as

Table 3: Average runtime in seconds for summing vectors of 10 data providers (DPs) each with 10,000 integer values for different secure summation protocols.

Vector size x number of runs	BSS	ESS	SSS	RSS	DSS	TSS	PSS	HSS
1 x 10,000	15	3,115	19	73	40,498	3,535	3,602	3,856
10 x 1,000	1.9	382	2.5	10	8,314	2,209	2,403	3,572
100 x 100	0.68	37	0.86	4.5	5,139	1,809	2,009	3,326
1,000 x 10	0.52	3.9	0.66	4.2	4,829	1,278	1,778	3,213
10,000 x 1	0.43	0.43	0.72	4.5	4,630	973	1,473	2,967

Table 4: Average runtime for summing vectors of 10 data providers (DPs) each with 10,000 decimal values for different secure summation protocols.

Vector size x number of runs	BSS	ESS	SSS	RSS	DSS	TSS	PSS	HSS
1 x 10,000	16	3,445	22	75	41,723	3,615	3,672	3,927
10 x 1,000	3.1	388	2.7	12	8,545	2,325	2,493	3,614
100 x 100	1.7	39	0.97	7.1	5,217	1,879	2,039	3,475
1,000 x 10	0.98	4.3	0.72	6.9	4,877	1,345	1,878	3,341
10,000 x 1	0.75	0.48	0.80	6.5	4,828	995	1,595	3,028

ESS because of the communicationally expensive secret sharing scheme. Comparatively, our proposed TSS, PSS, and HSS protocols run slower than the ESS and SSS protocols, but they have a runtime of 2 to 4 magnitudes faster than DSS with an increasing number of DPs and larger input data sizes.

As shown in Figure 6 (b) and (c) we measured the total number of messages communicated between the participants in each protocol and the average message size communicated by each participating party, respectively. As Figure 6 (b) shows, RSS, DSS, and PSS require a much larger number of messages to be communicated (as the number of DPs increases) because random shares (or segments) have to be exchanged between each pair of DPs. Hence in real-world scenarios these protocols would possibly require much longer runtime due to communication delays between many parties.

As shown in Figure 6 (c), the DSS protocol requires much larger message sizes compared to all other secure summation protocols because each individual random segment of each DP is encrypted separately and shared among the participating parties. However, compared to the BSS, SSS, and RSS protocols, ESS, TSS, PSS, and HSS communicate larger messages between the participating DPs since each message contains a homomorphically encrypted value. The total message size of RSS and PSS also increases quadratically with the number of DPs because of the pair-wise communication patterns of RSS and PSS. This potentially becomes expensive for large numbers of DPs.

As shown in Figure 7 (a) and (b) we measured the average runtime per DP for computing the summation of vectors with different numbers of values. As can be seen, secure summation protocols that use homomorphic encryption require larger runtime compared to BSS, SSS, and RSS. However, in real scenarios RSS, DSS, and PSS will possibly require longer runtime because of communication delays that occur when sending large numbers of messages between the parties, as shown in Figure 7 (b). Our proposed TSS, PSS, and HSS protocols require one order of magnitude less runtime for an individual DP compared to DSS while providing the same privacy guarantees against collusion risks.

As shown in Tables 3 to 5 we ran each protocol to compute the sum of 10,000 input values

Table 5: Average runtime for summing vectors of 10 data providers (DPs) each with 10,000 mixed (decimal and integer) values for different secure summation protocols.

Vector size x number of runs	BSS	ESS	SSS	RSS	DSS	TSS	PSS	HSS
1 x 10,000	16	3,198	19	74	41,146	3,581	3,651	3,917
10 x 1,000	2.7	381	2.7	12	8,445	2,275	2,447	3,589
100 x 100	0.9	39	0.87	6.1	5,107	1,801	2,011	3,421
1,000 x 10	0.58	4.1	0.68	6.7	4,864	1,213	1,802	3,283
10,000 x 1	0.65	0.43	0.77	6.0	4,708	978	1,493	2,988

of 10 DPs where in each protocol run we varied the size of the input value vector and the number of times the protocol needs to be run to compute the sum. As shown in these tables, all the protocols run more efficiently when the input vector contains more values compared to running the protocols multiple times to sum each value separately. This is also applicable in real scenarios because the protocols will incur significant communication overhead due to the total number of messages required to sum each value separately.

Overall, RSS and SSS are more suitable for SMC scenarios with and without a TP, respectively, in terms of efficiency. However, for scenarios where collusion is possible between the participating parties (i.e. a collusion up to $P - 2$ DPs and the TP) our proposed TSS, PSS, and HSS are more suitable compared to all the other secure summation protocols as they provide strong privacy guarantees against collusion with a smaller communication overhead.

5 Conclusions and Future Work

Secure summation is an important building block in secure multi-party computation (SMC) to calculate the sum of private inputs held by P data providers (DPs) while not revealing these values to any party. In this paper we have first presented five existing protocols (with and without a third party, TP) for securely summing values across multiple DPs, and then proposed three novel protocols that combine homomorphic encryption with random salting and sharing. We analysed each protocol in terms of privacy against different collusion scenarios that are possible in a real-world context. Our privacy analysis showed that our three novel protocols provide equal or better privacy against collusion between the participating parties compared to existing secure summation protocols. We also conducted an empirical evaluation with different number of DPs and input values which showed our novel protocols can compute the sum of large input vectors efficiently with less communication overhead while guaranteeing privacy even when up to $P - 2$ DPs collude with the TP.

As future work, we aim to develop secure summation protocols under the covert adversary model [3] for SMC applications where $P - 1$ parties might be colluding. The aim of such a protocol would be to allow non-colluding parties to identify any possible collusion between other parties before the final summation result is computed. This would guarantee that the private input of a non-colluding party cannot be identified using any partial result computed at a given intermediate step even when $P - 1$ participating parties collude in the protocol. We aim to investigate the applicability of threshold homomorphic encryption [18] and commutative encryption [2, 71] schemes under malicious and covert adversary model. We also aim to investigate the applicability of different security frameworks (such as Sharemind [8]) to compute the final sum in a shared form such that the plain sum result is not observed by any party that participates in a summation protocol. Finally, we aim to analyse the privacy of other SMC protocols, including secure set intersection and secure set union protocols, under different collusion risk scenarios [33].

Acknowledgements

This work was funded by the Australian Research Council under Discovery Project DP160-101934. We also like to thank Kee Siong Ng for his valuable feedback.

References

[1] A. Acar, H. Aksu, A. S. Uluagac, and M. Conti. A survey on homomorphic encryption schemes: Theory and implementation. *ACM Computing Surveys (CSUR)*, 51(4), 2018.

[2] R. Agrawal, A. Evfimievski, and R. Srikant. Information sharing across private databases. In *ACM SIGMOD*, pages 86–97, 2003.

[3] Y. Aumann and Y. Lindell. Security against covert adversaries: Efficient protocols for realistic adversaries. *Journal of Cryptology*, 23(2):281–343, 2010.

[4] D. Beaver, S. Micali, and P. Rogaway. The round complexity of secure protocols. In *Proceedings of the annual ACM Symposium on Theory of Computing*, pages 503–513, 1990.

[5] A. Beimel. *Secure schemes for secret sharing and key distribution*. PhD thesis, 1996.

[6] G. R. Blakley et al. Safeguarding cryptographic keys. In *Proceedings of the national computer conference*, volume 48, 1979.

[7] G. Blosser and J. Zhan. Privacy preserving collaborative social network. In *International Conference on Information Security and Assurance*, pages 543–548. IEEE, 2008.

[8] D. Bogdanov, S. Laur, and J. Willemson. Sharemind: A framework for fast privacy-preserving computations. In *European Symposium on Research in Computer Security*, pages 192–206. Springer, 2008.

[9] K. Bonawitz, V. Ivanov, B. Kreuter, A. Marcedone, H. B. McMahan, S. Patel, D. Ramage, A. Segal, and K. Seth. Practical secure aggregation for privacy-preserving machine learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1175–1191, 2017.

[10] J. Bringer, H. Chabanne, and A. Patey. Privacy-preserving biometric identification using secure multiparty computation: An overview and recent trends. *IEEE Signal Processing Magazine*, 30(2):42–52, 2013.

[11] C. Cheng, Y.-l. Luo, C.-x. Chen, and X.-k. Zhao. Research on secure multi-party ranking problem and secure selection problem. In *IEEE Conference on Web Information Systems and Mining*, 2010.

[12] B. Chor and E. Kushilevitz. A communication-privacy tradeoff for modular addition. *Information Processing Letters*, 45(4):205–210, 1993.

[13] B. Chor and E. Kushilevitz. Secret sharing over infinite domains. *Journal of Cryptology*, 6(2):87–95, 1993.

[14] P. Christen, R. Schnell, D. Vatsalan, and T. Ranbaduge. Efficient cryptanalysis of Bloom filters for privacy-preserving record linkage. In *PAKDD*, 2017.

[15] P. Christen, D. Vatsalan, and V. S. Verykios. Challenges for privacy preservation in data integration. *Journal of Data and Information Quality (JDIQ)*, 5(1-2):4, 2014.

[16] C. Clifton, M. Kantarcioglu, J. Vaidya, X. Lin, and M. Zhu. Tools for privacy preserving distributed data mining. *SIGKDD Explorations*, 2002.

[17] J. Cohen Benaloh. Secret sharing homomorphisms: Keeping shares of a secret secret. In *CRYPTO*, 1987.

[18] R. Cramer, I. Damgård, and J. B. Nielsen. Multiparty computation from threshold homomorphic encryption. In *International conference on the theory and applications of cryptographic techniques*, pages 280–300. Springer, 2001.

[19] M. S. H. Cruz, T. Amagasa, C. Watanabe, W. Lu, and H. Kitagawa. Secure similarity joins using fully homomorphic encryption. In *Proceedings of the 19th International Conference on Information Integration and Web-based Applications & Services*, pages 224–233. ACM, 2017.

[20] I. Damgård, V. Pastro, N. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. In *Advances in Cryptology–CRYPTO*, pages 643–662. Springer, 2012.

[21] G. Danezis and J. Clulow. Compulsion resistant anonymous communications. In *Information Hiding*, volume 3727, pages 11–25. Springer, 2005.

[22] G. Danezis and C. Diaz. A survey of anonymous communication channels. Technical report, MSR-TR-2008-35, Microsoft Research, 2008.

[23] T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE transactions on information theory*, 31(4):469–472, 1985.

[24] C. K. Emani, N. Cullot, and C. Nicolle. Understandable big data: a survey. *Computer science review*, 17:70–81, 2015.

[25] U. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy. *Advances in knowledge discovery and data mining*. American Association for Artificial Intelligence, 1996.

[26] M. Franklin and M. Yung. The varieties of secure distributed computation. In *Sequences II*, pages 392–417. Springer, 1993.

[27] B. Fung, K. Wang, R. Chen, and P. Yu. Privacy-preserving data publishing: A survey of recent developments. *ACM Computing Surveys (CSUR)*, 2010.

[28] C. Gentry. *A fully homomorphic encryption scheme*. Stanford University, 2009.

[29] O. Goldreich. Secure multi-party computation. Technical report, Department of Computer Science and Applied Mathematics, Weizmann Institute of Science, Israel, 2002.

[30] O. Goldreich. *Foundations of cryptography: volume 2, basic applications*. Cambridge university press, 2009.

[31] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *ACM Symposium on Theory of Computing*, 1987.

[32] V. Goyal, P. Mohassel, and A. Smith. Efficient two party and multi party computation against covert adversaries. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 289–306. Springer, 2008.

[33] C. Hazay and Y. Lindell. Efficient protocols for set intersection and pattern matching with security against malicious and covert adversaries. In *Theory of Cryptography Conference*, pages 155–175. Springer, 2008.

[34] C. Hazay and Y. Lindell. *Efficient secure two-party protocols: Techniques and constructions*. Springer Science & Business Media, 2010.

[35] C. Hazay and M. Venkitasubramaniam. Scalable multi-party private set-intersection. In *IACR International Workshop on Public Key Cryptography*, pages 175–203. Springer, 2017.

[36] B. Hemenway, W. Welser, and D. Baiocchi. *Overview of Secure Multiparty Computation*, pages 5–20. RAND Corporation, 2014.

[37] H. Jagadish, J. Gehrke, A. Labrinidis, Y. Papakonstantinou, J. M. Patel, R. Ramakrishnan, and C. Shahabi. Big data and its technical challenges. *Communications of the ACM*, 57(7):86–94, 2014.

[38] W. Jiang and C. Clifton. AC-framework for privacy-preserving collaboration. In *Proceedings of the SIAM International Conference on Data Mining*, pages 47–56. SIAM, 2007.

[39] W. Jiang, C. Clifton, and M. Kantarcioglu. Transforming semi-honest protocols to ensure accountability. *Data & Knowledge Engineering*, 65(1):57–74, 2008.

[40] M. Kantarcioglu and C. Clifton. Privacy-preserving distributed mining of association rules on horizontally partitioned data. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 16(9):1026–1037, 2004.

[41] E. Karnin, J. Greene, and M. Hellman. On secret sharing systems. *IEEE Transactions on Information Theory*, 29(1):35–41, 1983.

[42] A. Karr, X. Lin, A. Sanil, and J. Reiter. Analysis of integrated data without data integration. *CHANCE*, 2004.

[43] A. F. Karr. Secure statistical analysis of distributed databases, emphasizing what we don't know. *Journal of Privacy and Confidentiality*, 1(2):5, 2010.

[44] S. C. Kothari. Generalized linear threshold scheme. In *Workshop on the Theory and Application of Cryptographic Techniques*, pages 231–241. Springer, 1984.

[45] M. Kuzu, M. Kantarcioglu, A. Inan, E. Bertino, E. Durham, and B. Malin. Efficient privacy-aware record integration. In *Proceedings of the 16th International Conference on Extending Database Technology*, pages 167–178. ACM, 2013.

[46] Y. Lindell and B. Pinkas. Privacy preserving data mining. In *Advances in Cryptology-CRYPTO*, pages 36–54. Springer, 2000.

[47] Y. Lindell and B. Pinkas. Secure multiparty computation for privacy-preserving data mining. *Journal of Privacy and Confidentiality*, 2009.

[48] Y. Lindell, B. Pinkas, N. P. Smart, and A. Yanai. Efficient constant round multi-party computation combining bmr and spdz. In *CRYPTO*, pages 319–338. Springer, 2015.

[49] Y. Lindell, N. P. Smart, and E. Soria-Vazquez. More efficient constant-round multi-party computation from bmr and she. In *Theory of Cryptography Conference*, pages 554–581. Springer, 2016.

[50] A. Lysyanskaya and N. Triandopoulos. Rationality and adversarial behavior in multi-party computation. In *Annual International Cryptology Conference*, pages 180–197. Springer, 2006.

[51] S. Mehnaz, G. Bellala, and E. Bertino. A secure sum protocol and its application to privacy-preserving multi-party analytics. In *Proceedings of the 22nd ACM Symposium on Access Control Models and Technologies*, pages 219–230, 2017.

[52] S. Mehnaz and E. Bertino. Privacy-preserving multi-party analytics over arbitrarily partitioned data. In *Proceedings of the IEEE International Conference on Cloud Computing (CLOUD)*, 2017.

[53] P. Mohassel, M. Rosulek, and Y. Zhang. Fast and secure three-party computation: The garbled circuit approach. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 591–602, 2015.

[54] R. Morris and K. Thompson. Password security: A case history. *Communications of the ACM (CACM)*, 1979.

[55] M. Naehrig, K. Lauter, and V. Vaikuntanathan. Can homomorphic encryption be practical? In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, pages 113–124, 2011.

[56] M. J. Nigrini. Identifying fraud using abnormal duplications within subsets. *Forensic Analytics: Methods and Techniques for Forensic Accounting Investigations*, pages 233–262, 2011.

[57] C. OKeefe, M. Yung, L. Gu, and R. Baxter. Privacy-preserving data linkage protocols. In *ACM Workshop on Privacy in the Electronic Society*, 2004.

[58] V. Oleshchuk. Internet of things and privacy preserving technologies. In *1st International Conference on Wireless Communication, Vehicular Technology, Information Theory and Aerospace Electronic Systems Technology*, pages 336–340, 2009.

[59] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT*, 1999.

[60] C. Perera, A. Zaslavsky, P. Christen, and D. Georgakopoulos. Context aware computing for the internet of things: A survey. *IEEE communications surveys & tutorials*, 16(1):414–454, 2013.

[61] P. Rogaway. Nonce-based symmetric encryption. In *International Workshop on Fast Software Encryption*, pages 348–358. Springer, 2004.

[62] M. Scannapieco, I. Figotin, E. Bertino, and A. K. Elmagarmid. Privacy preserving schema and data matching. In *ACM SIGMOD*, pages 653–664, 2007.

[63] N. Schlitter. A protocol for privacy preserving neural network learning on horizontal partitioned data. *PSD*, 2008.

[64] B. Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, Inc., 1995.

[65] R. Schnell. Privacy-preserving record linkage. In *Methodological Developments in Data Linkage*. John Wiley & Sons, Inc., 2015.

[66] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[67] E. Shi, H. Chan, E. Rieffel, R. Chow, and D. Song. Privacy-preserving aggregation of time-series data. In *Annual Network & Distributed System Security Symposium*. Internet Society, 2011.

[68] K. Shimizu, K. Nuida, H. Arai, S. Mitsunari, N. Attrapadung, M. Hamada, K. Tsuda, T. Hirokawa, J. Sakuma, G. Hanaoka, and K. Asai. Privacy-preserving search for chemical compound databases. *BMC Bioinformatics*, 16(18), 2015.

[69] T. Tassa and D. Cohen. Anonymization of centralized and distributed social networks by sequential clustering. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 2013.

[70] S. Urabe, J. Wang, E. Kodama, and T. Takata. A high collusion-resistant approach to distributed privacy-preserving data mining. *IPSJ Digital Courier*, 3:442–455, 2007.

[71] J. Vaidya, C. W. Clifton, and Y. M. Zhu. *Privacy preserving data mining*, volume 19. Springer Science & Business Media, 2006.

[72] D. Vatsalan, P. Christen, and E. Rahm. Scalable privacy-preserving linking of multiple databases using counting Bloom filters. In *IEEE ICDMW*, 2016.

[73] D. Vatsalan, Z. Sehili, P. Christen, and E. Rahm. Privacy-preserving record linkage for big data: Current approaches and research challenges. In *Handbook of Big Data Technologies*, pages 851–895. Springer, 2017.

[74] B. Yang, I. Sato, and H. Nakagawa. Privacy-preserving em algorithm for clustering on social network. In *The Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD)*, 2012.

[75] A. C. Yao. Protocols for secure computations. In *23rd Annual Symposium on Foundations of Computer Science (SFCS)*, pages 160–164. IEEE, 1982.

[76] A. C. Yao. How to generate and exchange secrets. In *IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, 1986.

[77] Y. Zhu, L. Huang, W. Yang, and X. Yuan. Efficient collusion-resisting secure sum protocol. *Chinese Journal of Electronics*, 20(3):407–413, 2011.