cryptography

MDPI

Article

Fair and Secure Multi-Party Computation with Cheater Detection

Minhye Seo

Department of Cyber Security, Duksung Women's University, Seoul 01369, Korea; mhseo@duksung.ac.kr

Abstract: Secure multi-party computation (SMC) is a cryptographic protocol that allows participants to compute the desired output without revealing their inputs. A variety of results related to increasing the efficiency of SMC protocol have been reported, and thus, SMC can be used in various applications. With the SMC protocol in smart grids, it becomes possible to obtain information for load balancing and various statistics, without revealing sensitive user information. To prevent malicious users from tampering with input values, SMC requires cheater detection. Several studies have been conducted on SMC with cheater detection, but none of these has been able to guarantee the fairness of the protocol. In such cases, only a malicious user can obtain a correct output prior to detection. This can be a critical problem if the result of the computation is real-time information of considerable economic value. In this paper, we propose a fair and secure multi-party computation protocol, which detects malicious parties participating in the protocol *before* computing the final output and prevents them from obtaining it. The security of our protocol is proven in the universal composability framework. Furthermore, we develop an enhanced version of the protocol that is more efficient when computing an average after detecting cheaters. We apply the proposed protocols to a smart grid as an application and analyze their efficiency in terms of computational cost.

Keywords: secure multi-party computation; cheater detection; universal composability; fairness; smart grid

Citation: Seo, M. Fair and Secure Multi-Party Computation with Cheater Detection. *Cryptography* **2021**, *5*, 19. https://doi.org/10.3390/cryptography5030019

Academic Editors: Huaxiong Wang and Josef Pieprzyk

Received: 30 April 2021
Accepted: 10 August 2021
Published: 12 August 2021

1. Introduction

Secure multi-party computation (SMC) is a set of cryptographic techniques that allows a set of mutually distrusting parties to compute a predefined function on their private inputs and obtain an output without revealing the inputs. Due to this property, the SMC protocol has been used in many applications, such as electronic auctions, electronic voting, and privacy-preserving statistical analysis, as a building block [1–6]. Privacy-preserving data aggregation protocols that use SMC have been proposed in the literature to deal with user privacy in smart grids [7,8].

A smart grid is a convergence technology that combines a traditional grid and information communications technology (ICT). As the next generation of power grids, it allows two-way transmission between generation plants and customers. Using this bidirectional communication, power suppliers (e.g., utilities) and other service providers offer various convenient services to the customer and optimize energy consumption and cost. Smart meters collect customers' data in real time. Meaningful information for beneficial services may be obtained by aggregating data collected by smart meters. For example, information such as power consumption patterns and time-of-use rates enable consumers to find a solution to sustain energy efficiency while reducing the cost of electricity. With time-of-use pricing for electricity, consumers can schedule energy-intensive activities for off-peak or mid-peak hours. Moreover, the power supplier utilizes the power consumption patterns of geographical areas to manage energy supply with the aim of load balancing. However, aggregating or collecting consumers' energy usage data incurs privacy issues. Energy consumption patterns might contain very sensitive information because they reflect

consumers' daily life activities. For example, if such information is revealed, an attacker may be able to determine the number of residents in a house when the house is empty, the types of electronic devices in the house, and other details. Therefore, it is necessary to build a privacy-preserving data aggregation protocol in a smart gird that guarantees the privacy of the data related to each consumer. SMC is a proper solution to handle privacy issues, and several studies have explored the application of the SMC protocol to smart grids [7,8].

However, if malicious consumers tamper with their smart meters while performing computations, the utility as well as honest consumers will be unable to obtain accurate information (e.g., for load balancing or reducing energy cost). Thus, cheater detection is required in the SMC protocol. Cheater detection in SMC has been explored in [9–11], but the schemes proposed there have a few weaknesses. In these schemes [9–11], a malicious party can share its input with honest parties and create a final share by itself. (Each party performs the pre-defined computation using initial shares, the part of other parties¡¯ inputs, in order to create a *final share*. We can obtain a *final output* of the computation by reconstructing, e.g., summing up, final shares of all parties.) The malicious party can then broadcast a random value instead of the final share. In this case, only the malicious party obtains an accurate output by using the final shares of honest parties along with its own final share in the reconstruction phase. The honest parties in such a case obtain an inaccurate output because they reconstruct the values, including a random value by the malicious party. Although the values released by all parties may be used to detect the malicious parties, the final output in this case has already been revealed because the detection occurs *following* the computation. Therefore it is necessary to detect malicious parties *before* computing the final output. With this property, SMC guarantees *fairness*. (Roughly speaking, *fairness* means that either everyone participating in the SMC protocol can obtain an accurate result of computations or none can.)

In this paper, we propose an SMC protocol that guarantees *fairness* by detecting a cheater prior to computing the final output. The proposed protocol provides cheater detection and security even when $(n-1)$ of n parties in the protocol are corrupted. Our proposed protocol is based on the SPDZ protocol [12], which is an efficient SMC protocol that has been proven to be secure in the malicious model. (The SPDZ (Smart, N., Pastro, V., Damgard, I., Zakarias, S.) protocol is an SMC protocol for arithmetic circuits with a highly efficient online phase. The online phase of the SPDZ protocol is derived by preprocessing the task of the online phase in the offline phase of the protocol.) In this paper, we provide a definition of universally composable (UC) security for fair multi-party computation and prove the security of the proposed protocol according to this definition. Moreover, we propose an enhanced version of our protocol in case of computing the average, one of the most widely used computations in the smart grid that improves efficiency. We also analyze the efficiency of the enhanced version of our protocol.

Organization

The remainder of this paper is organized as follows: In Sections 2 and 3, we review related works and briefly explain preliminaries in this area, respectively. In Section 4, we define the relevant notations and introduce the security model for fair SMC in the UC setting. We describe the proposed protocol in Section 5, followed by its security proof in Section 6. In Section 7, we apply the proposed protocol to a smart grid and describe the enhanced version of our protocol through an efficiency analysis. We offer our conclusions in Section 8.

2. Related Work

- **Cheater Detection.** Detecting cheaters on the Internet has been an important issue, such as astroturfing detection [13–15], and a diversity of techniques concerning this matter have been proposed so far. Several studies have been carried out on cheater detection in secret sharing schemes [16–19]. The results of these techniques are only applicable to secret sharing schemes and thus may not be used to verify the correct-

ness of the new shares computed by secure multi-party computation (SMC) protocol. Several SMC protocols that can detect malicious parties have been proposed so far. Damgård and Orlandi proposed an SMC protocol where every party broadcasts the computed final shares and checks its commitments following the computation of its own final shares [10]. Baum et al. proposed an SMC protocol with an additional audit algorithm [20]. Since the SPDZ protocol was proposed, several SMC protocols have been built on top of the SPDZ protocl to achieve cheater detection and identification [21,22]. Furthermore, there have been efforts to improve the efficiency of SMC protocols with identifiable abort [23]. However, neither of these protocols can prevent malicious parties from learning the final result of computation, whereas honest parties may not. In other words, malicious parties can only be detected following the broadcast of the final shares or the reconstruction of the final output. In such a case, a malicious party can broadcast an incorrect value and obtain all correct final shares of the honest parties as well as its own. The malicious party would then be the only one that can reconstruct the correct output of the computation.

- **Fair and Secure Multi-party Computation.** It is known that any functionality can be fairly computed in the case of honest majority [24–27]. However, fairness is impossible to be guaranteed with corrupted majority [28]. Consequently, a number of definitions of security do not consider fairness, even in a Universal Composability (UC) framework, or only consider partial fairness [29–32]. To overcome this impossibility result, a few approaches to achieve fairness have been proposed. The gradual release approach makes parties take turns releasing their outputs in each round [33–35]. However, this approach is still somewhat unfair. Another approach involves employing semi-trusted third parties or physical assumptions [36–38]. As part of this approach, a technique using Bitcoin has been proposed [39,40]. It was recently shown that fair, secure, multi-party computation can be achieved by applying a multi-party fair exchange protocol to any SMC protocol [41]. Moreover, several results have been insisted on the possibility of utilizing reputation systems [42], public bulletin boards [43], and trusted hardware [44] to achieve fairness.
- **Secure Multi-party Computation in Smart Grid.** A number of studies on privacy-preserving aggregation of information for metering or billing have been conducted [45–50]. SMC techniques have recently been applied to smart grids to preserve user privacy [7,8]. The protocol proposed in [7] can be applied to smart grids but requires that all participating parties decrypt the final output together in an interactive manner. In [8], Clark and Hopkinson proposed an optimized SMC protocol called transferable multi-party computation (T-MPC) for smart grid networks. To improve efficiency and scalability, T-MPC allows small groups of users to compute the local results of sub-functions. However, it is unable to detect malicious parties and guarantee fairness at the same time.
- **Implementations of Secure Multi-party Computation.** Implementation and compiler design of SMC is an active area of research. In the early days of research, compilers were developed that convert code written in a high-level language called secure function definition language (SFDL) into a circuit representation (e.g., FairplayMP [3], VIFF [5], SEPIA [6]). Since then, research has been underway for implementations that are more suitable for real-world applications, that is, fast enough to evaluate complex functions and large data sets. For secure two-party computation, most protocols have been designed using circuit garbling [51], such as ABY [52], EzPC [53], and ABY2.0 [54]. For true multi-party computation, a variety of compilers that execute SMC in arbitrary functions have been developed, and representative examples include PICCO [55,56] and MP-SPDZ [57].

In this paper, we propose an SMC protocol that provides cheater detection and fairness. To prevent a malicious party from solely obtaining the final output, in our proposed protocol, detection occurs *prior to* the computation of the final output. Thus, the proposed protocol guarantees *fairness*. We also prove the security of our fair SMC protocol in the UC

setting. Moreover, we apply our protocol to a smart grid for privacy-preserving aggregation of customers' data. We also propose an enhanced version of our protocol, in the case of computing the average, and analyze its efficiency.

Contributions

Fairness is an important factor in SMC, and there have been numerous studies to detect malicious parties participating in the protocol. However, most of them focus on *detecting* and do not fully consider the *benefits* that malicious attacker can obtain from cheating. If malicious parties share invalid values while others share valid ones, they could be the only ones who can reconstruct a valid final output. Even detecting malicious parties, if they (and only they) can obtain the final output, it can in some cases cause considerable economic damage or invasion of privacy. The underlying cause of this problem is that cheater detection is performed by each party in the protocol individually checking whether the intermediate values (i.e., shares) received from other parties are fair or not. In other words, the detection of malicious parties in previous works occurs *after* or *during* reconstruction of the final shares, meaning that it could be after the malicious parties have already obtained the final output. Therefore, it is required that the detection should be executed *before* the reconstruction of the final output so that malicious parties could not obtain it.

In this paper, we assume that there exists a (semi-honest) server, which searches out malicious parties before the final output is computed. The server should not be able to obtain the secret input values of each party or the final output of the protocol. To this end, we adopt cryptographic techniques including broadcast encryption, commitment, and non-interactive zero-knowledge proof system. After all parties in the protocol have completed their computation individually, they encrypt their own final share using a broadcast encryption scheme and create a commitment to the final share. Then they generate a non-interactive zero-knowledge proof to prove that they committed the encrypted value through broadcast encryption. Finally, they encrypt these three values using the server's public key and send them to the server. The server then detects the malicious parties by verifying the validity of the values sent from each party, and if all values are valid, the server sends the ciphertexts of each final share to all parties.

Our protocol prevents malicious parties from obtaining an accurate output because the server detects malicious parties prior to computing the final output. Following the execution of the proposed protocol, every honest party may obtain the correct output of the relevant computation whereas malicious parties may not. The proposed protocol guarantees fairness even when $(n-1)$ of n parties are corrupted. If all parties are corrupted, no one can obtain the correct output of the computation. We extend the definition of fair, secure, multi-party computation in the UC setting and give the security proof of our fair SMC protocol in the UC model.

Finally, we develop an enhanced version of our protocol in case of computing averages. The average may be re-computed by slightly modifying each honest party's final share. Honest parties are not required to restart the protocol from the beginning. The final output is the accurate result of the computation using only fair shares of honest parties. In comparison with the version that involves restarting the protocol, our enhanced version reduces communication- and computation-based overhead in recomputing the final output following the detection of the malicious party. We apply this approach to a smart grid as an application.

3. Preliminaries

3.1. Secure Multi-Party Computation

Research on secure multi-party computation (SMC) can be divided into two groups, the garbled circuit group and the secret sharing group. In this paper, we focus on the secret sharing group because it is more commonly used for $n > 2$. We provide a brief explanation of SMC based on the additive secret sharing scheme in particular on carrying out addition and multiplication using shares [10,58,59].

In this setting, each party splits its secret value x_i into shares $x_{i,j}$ adding up to the secret value, i.e., $x_i = \sum_j x_{i,j}$, and distributes them to other parties. All parties participating in the protocol jointly compute the secret values using shares sent from the other parties (and one share of its own secret) without revealing any intermediate or final results. More precisely, each party computes the final share using its own shares, and the final result of the computation is obtained by reconstructing the final shares of all parties.

Every computation is represented as a combination of addition and multiplication. Hence, it is sufficient to introduce the manner in which each party computes a new share for an addition and a multiplication of two secret values using its shares.

3.1.1. Addition

In terms of addition, each party can locally compute a new share. The new share is derived by adding the shares of each secret, e.g., the new share for $(x_1 + x_2)$ of party P_j can be computed by $(x_{1,j} + x_{2,j})$. Similarly, in case of a new share for $(a \cdot x_1)$, where a is a constant, party P_j can derive this by computing $(a \cdot x_{1,j})$ for itself.

3.1.2. Multiplication

The multiplication of two secrets requires interactions among the parties. To reduce the numbers of required interactions, all parties pre-share a number of triples prior to the execution of the computation. Triples (a_i, b_i, c_i) of party P_i satisfy the equation $\sum_i c_i = \sum_i a_i + \sum_i b_i$ and are independent of the computation to be performed.

Using these triples, parties can compute the multiplication of two secrets, x_1 and x_2, through a single round of interaction. To obtain a new share for $(x_1 \cdot x_2)$ of party P_i, P_i first computes and reveals its shares of $\varepsilon_i = x_{1,i} - a_i$ and $\delta_i = x_{2,i} - b_i$. The parties can then reconstruct $\varepsilon = \sum_i \varepsilon_i$ and $\delta = \sum_i \delta_i$. P_i then locally computes its new share $t_i = c_i + \delta \cdot a_i + \varepsilon \cdot b_i + \varepsilon \cdot \delta$. This enables us to calculate any arithmetic circuit using the same number of interactions as the multiplicative depth of the circuit. For more details, please refer to [10,60].

3.2. Public-Key Broadcast Encryption

Public-key broadcast encryption (PKBE) allows a sender to securely distribute messages to a dynamically changing set of users over an insecure channel in the public key setting [61,62]. The PKBE system consists of three randomized algorithms:

Setup (n). Takes as input the number of receivers n and outputs a public key PK as well as n private keys $d_1, \ldots, d_n$.

Encrypt (S, PK). Takes a subset $S \subseteq \{1, \ldots, n\}$ and a public key PK as input, and outputs a pair (Hdr, K), where Hdr is the header and $K \in \mathcal{K}$ is a message encryption key. We often refer to Hdr as the broadcast ciphertext.

Let M be a message to be broadcasted to set S and C_M be the encryption of M under symmetric key K. A sender broadcasts (S, Hdr, C_M) to users in S, where the pair (S, Hdr) is often called the full header and C_M is often called the broadcast body.

Decrypt (d_i, S, Hdr, PK). Takes as input private key d_i for user i, a subset $S \in \{1, \ldots, n\}$, a header Hdr, and the public key PK. If user i is in S, the algorithm outputs the message encryption key $K \in \mathcal{K}$, which is used to decrypt broadcast body C_M and obtain message M.

3.3. Non-Interactive Zero Knowledge

Non-interactive zero knowledge (NIZK) is a kind of zero-knowledge proof system where no interaction is required between a prover and a verifier. The NIZK protocol usually assumes an initial setup that generates a common reference string (CRS) to eliminate any interaction. The CRS is a publicly shared random string between a prover and a verifier and is given to both a prover and a verifier in advance. Parameterized with relation R, the NIZK protocol proceeds as follows:

Prove. Takes as input (x, w) if $(x, w) \notin R$ outputs $\perp$. Otherwise, it outputs (x, π), where π is proof of the statement that $(x, w) \in R$.

Verify. Takes as input (x, π) and outputs 1 to accept the proof. Otherwise, it outputs 0.

In this paper, we use the UC-secure NIZK protocol proposed in [63].

3.4. Universally Composable Security

In general, we prove the security of the protocol executed in isolation. However, in the real world, many executions occur simultaneously, and some protocols can be used as a sub-function of others. In a universal composability (UC) framework, one can guarantee the security of a protocol even when it is used as a sub-routine of any other protocol running concurrently in the system. The framework for UC was first proposed by Canetti [64].

In the UC framework, there is a "trusted party" that obtains the inputs of all parties and provides them with the desired outputs. A set of instructions for a trusted party describes the functionality of the protocol. Informally, a protocol securely carries out a given task if running the protocol amounts to "emulating" an ideal process, where the parties provide their inputs to a trusted party with the appropriate functionality and obtain their outputs from it without any other interaction. The algorithm operated by the trusted party is called an *ideal functionality*.

The UC framework is an enhanced version of the real–ideal security model, which includes parties and an adversary **A**. The notion of emulation in the UC framework is considerably stronger than that in the real–ideal security model because the adversarial entity, called the environment **Z**, is additionally adopted. The environment **Z** generates the inputs of all parties, reads all outputs, and interacts with the adversary **A** in an arbitrary manner throughout the computation. A protocol is said to securely realize a given ideal functionality $\mathcal{F}$ if, for any adversary **A**, there exists an "ideal-process adversary" **S** such that *no environment* **Z** can tell whether it is interacting with **A** and parties executing the protocol, or with **S** and parties interacting with $\mathcal{F}$ in the ideal process. In a sense, **Z** here serves as an "interactive distinguisher" between a run of the protocol in the real world and the process with access to $\mathcal{F}$ in the ideal world. In summary, a protocol is UC secure if, for every real-world adversary **A**, there exists an ideal-world adversary (simulator) **S** such that the environment **Z** cannot distinguish between a real execution with **A** and an ideal execution with **S**.

4. Definitions

4.1. Notation

We define the notation used in this paper to clarify the representations of our protocol. The $[\![x]\!]$ representation of x is defined as $[\![x]\!] = (x_1, \ldots, x_n)$, where $x = \sum_{i=1}^{n} x_i$. Each party P_i will hold its own share x_i of such a representation. For $x, y, e \in \mathbb{Z}_p$, we define the operations of this representation as follows:

$$[\![x]\!] + [\![y]\!] := (x_1 + y_1, \ldots, x_n + y_n)$$
$$e \cdot [\![x]\!] := (e \cdot x_1, \ldots, e \cdot x_n)$$
$$e + [\![x]\!] := (x_1 + e, x_2, \ldots, x_n)$$

Definition 1. *Let $x, r \in \mathbb{Z}_p$ and $g, h \in G$, where both g and h are generators of group G. We use the Pedersen commitment [65] $pc(x, r) = g^x h^r$ and define the commitment $Com([\![x]\!])$ as follows:*

$$Com([\![x]\!]) = \Big[pc(x_i, r_i)\Big]_{i \in n}, \text{ where } r_i \in_R \mathbb{Z}_p$$

Each party P_i creates its own commitment $pc(x_i, r_i)$ of such a representation. For $x, y, x_{rand}, y_{rand}, a \in \mathbb{Z}_p$, we define the operations as follows:

$$Com(x) \cdot Com(y) = pc(x, x_{rand}) \cdot pc(y, y_{rand})$$
$$Com(x)^a = (pc(x, x_{rand}))^a$$

4.2. UC-Secure Fair Multi-Party Computation

In secure multi-party computation (SMC), a group of parties with their private inputs x_i desire to compute a function ϕ. We can guarantee *fairness* in this case if either all of the parties learn the final output of the computation or none of them learns it. This is formalized by real–ideal world simulations [41]. We extend the real–ideal paradigm to the UC framework defined below, and prove the fairness and security of our protocol in the UC setting in Section 6.

In Figure 1, we present the ideal functionality of our fair SMC protocol in the UC setting. The ideal functionality is the protocol where a trusted party communicates with participants over a secure channel and computes the desired output. In the UC framework, running protocols in the real world is compared to an ideal functionality in the ideal world.

The ideal functionality $\mathcal{F}_{SMC}$

Initialize: On input (`Init`, C, p) (where C is a circuit to be computed consisting of addition and multiplication gates over $\mathbb{Z}_p$ and $p \in \mathbb{P}$) from all parties:

1. Wait until **A** sends the set $P_C \subseteq \{1, \ldots, n\}$ (corrupted parties)

Input: On input (`Input`, x_i) from each party P_i (where x_i is the secret value of party P_i):

1. If $i \notin P_C$ then store (P_i, x_i). Else let **A** choose x' and store (P_i, x').

Compute: On input (`Compute`) from all parties:

1. If an input gate of C has no value assigned, stop here.
2. Compute $y_c = C(x_1, \ldots, x_n)$

Output: If **Compute** was executed,

1. The functionality sends $(\texttt{Output}, y_c)$ to all parties.

Figure 1. The ideal functionality that describes the online phase.

Real World: Let P be a set of n parties, $P = P_C \bigcup P_H$. It consists of an adversary **A** that compromises the set P_C of $m (m < n)$ corrupted parties, the set P_H of remaining honest parties, and a server that detects malicious behavior during the execution of the protocol. The pair of outputs of the honest parties $\in P_H$ and **A** in the real execution of the protocol π, employing the server, is denoted by $\text{REAL}_{\pi,\textbf{Server},\textbf{A}(aux),\textbf{Z}}(\lambda, z, x_1, x_2, \ldots, x_n)$, where $\{x_i\}_{1 \leq i \leq n}$ are the private inputs of each party, aux is an auxiliary input of **A**, λ is the security parameter, and z is the input of the environment.
Ideal World: It consists of an adversary **S** that controls the set P_C, the set P_H of remaining honest parties, and the ideal functionality $\mathcal{F}$ (*not* the server). $\mathcal{F}$ receives inputs $\{x_i\}_{\{i \in P_C\}}$ or the message ABORT from **S** and $\{x_j\}_{\{j \in P_H\}}$ from the honest parties.

- If the inputs are invalid or **S** sends the message ABORT, $\mathcal{F}$ sends $\perp$ to all parties and halts.
- Otherwise, $\mathcal{F}$ computes $\phi(x_1, \ldots, x_n) = \Big(\phi_1(x_1, \ldots, x_n), \ldots, \phi_n(x_1, \ldots, x_n)\Big)$. Let $\phi_i = \phi_i(x_1, \ldots, x_n)$ be the i-th output. Then, $\mathcal{F}$ sends $\{\phi_i\}_{\{i \in P_C\}}$ to **S** and $\{\phi_j\}_{\{j \in P_H\}}$ to the corresponding honest parties.

The outputs of the parties in an ideal execution involving the honest parties and **S**, where $\mathcal{F}$ computes ϕ, is denoted by $\text{IDEAL}_{\mathcal{F},\textbf{S}(aux),\textbf{Z}}(\lambda, z, x_1, x_2, \ldots, x_n)$, where $x_1, x_2, \ldots, x_n$, aux, λ, and z are as above.

Definition 2 (UC-secure Fair Multi-party Computation). *Let π be a probabilistic polynomial time (PPT) protocol and $\mathcal{F}$ be a PPT multi-party functionality computing ϕ. We say that π*

computes ϕ ***fairly and securely*** *if, for every non-uniform PPT real-world adversary* **A** *attacking π, there exists a non-uniform PPT ideal world simulator* **S** *such that for every $x_1, x_2, \ldots, x_n$, $\lambda \in \{0,1\}^*$, the environment* **Z** *with input z cannot distinguish between a real execution with* **A** *and an ideal execution with* **S***:*

$$\begin{aligned}&\left\{\mathrm{REAL}_{\pi,\mathbf{Server},\mathbf{A}(aux),\mathbf{Z}}(\lambda, z, x_1, x_2, \ldots, x_n)\right\}\\ &\quad\equiv_c \left\{\mathrm{IDEAL}_{\mathcal{F},\mathbf{S}(aux),\mathbf{Z}}(\lambda, z, x_1, x_2, \ldots, x_n)\right\}\end{aligned}$$

Note that since the server does not exist in the ideal world, the simulator should also simulate its behavior.

5. FSMC Protocol with Cheater Detection

We propose a fair and secure multi-party computation (FSMC) protocol that can detect malicious parties. In FSMC protocol, there are n parties that perform a predefined computation and a semi-honest server that detects malicious parties. The parties corrupted by an adversary are regarded as malicious. The malicious parties do not follow the protocol as described, thereby obtaining the inputs of honest parties or disrupting the computation so that the honest parties may not obtain a correct output.

Once the malicious parties are detected, they may not obtain any information regarding the final output. The protocol consists of three phases: the setup, the offline, and the online phases. In order to improve efficiency of the actual computation, the values used in the online phase were pre-computed in the offline phase.

We use $\mathcal{F}_{BE}$ and $\mathcal{F}_{NIZK}$ to satisfy the fairness of this protocol. Specifically, the functionality $\mathcal{F}_{BE}$ is used to conceal any information regarding the final output. The functionality $\mathcal{F}_{NIZK}$ is used to guarantee the connection between the commitment and the ciphertext generated by each party.

In this protocol, each party uses $\mathcal{F}_{NIZK}$ to generate the proof of the statement that the commitment and the ciphertext are generated based on the same value, namely the final share it creates. More formally, each party executes **Prove** with the following relation R:

$$R = \left\{ \left((\mathrm{Com}, \mathrm{CT}), [\![y]\!] \right) \middle| \ \mathrm{Com} = Com[\![y]\!], \mathrm{CT} = BE[\![y]\!] \right\}$$

5.1. Setup

Let $p \in \mathbb{P}$ be a prime number and G be a group of order p. The Discrete Logarithm Problem (DLP) is difficult to solve in group G. Let $g \in G$ be a generator of G. Choose $s \in \mathbb{Z}_p^*$ uniformly at random and set $h = g^s$. There is a server that verifies the correctness of each party's values and detects malicious parties. This server must not be able to obtain any secret values of the parties or the final output of the protocol. We assume that a secure channel between the server and each party can be established and that a broadcast encryption functionality $\mathcal{F}_{BE}$ (Figure 2) and a non-interactive zero knowledge functionality $\mathcal{F}_{NIZK}$ (Figure 3) are available for every party. The notations in the functionalities $\mathcal{F}_{BE}$ and $\mathcal{F}_{NIZK}$ represent the same meanings as assigned to them in Sections 3.2 and 3.3.

5.2. Offline Phase

In the offline phase, as a preprocessing stage, some values are pre-computed in order for parties to execute the online phase more efficiently. In Figure 4, we define functionality $\mathcal{F}_{Setup}$, which describes the offline phase. In **Setup**, the values to be used for commitments are generated. These values are used in **Compute.Send** in the online phase to generate commitments and **Output** in the online phase to verify them. In **RandomValues**, the values used to divide each party's secret into shares are generated. These values are used in the **Input** of the online phase. In **Triples**, the multiplication triples and their commitments are generated. The multiplication triples are used in **Compute.Multiply** in the online phase to compute multiplications with minimal interaction. The commitments are used in **Output** in the online phase to detect malicious parties.

The functionality $\mathcal{F}_{BE}$

Setup: On input (`Setup`,n) from all parties, the functionality generates a public key, PK_{BE}, and n private keys $sk_1, \ldots, sk_n$. Then the functionality sends (PK_{BE}, sk_i) to each party P_i.

Encrypt: On input (`Encrypt`, S, PK_{BE}) from the party P_i, the functionality generates a pair (Hdr, K) where Hdr is the header and K is a message encryption key. Then the functionality sends (Hdr, K) to the party P_i.

Decrypt: On input (`Decrypt`, sk_i, S, Hdr, PK_{BE}) from the party P_i, the functionality generates the message encryption key K and sends it to P_i.

Figure 2. The ideal functionality for broadcast encryption.

The functionality $\mathcal{F}_{NIZK}$

Parameterized with relation R and being executed with parties $P_1, \ldots, P_n$.

Prove: On input (`Prove`, x, w) from party P_i, the functionality ignores it if $(x, w) \notin R$; otherwise, the functionality generates the proof π and stores (x, π). Then the functionality sends (`proof`, π) to party P_i.

Verify: On input (`Verify`, x, π) from the verifier, the functionality checks whether (x, π) is stored. If (x, π) has been stored, the functionality sends (`verification`, 1) to the verifier. Else the functionality sends (`verification`, 0) to the verifier.

Figure 3. The ideal functionality for non-interactive zero knowledge.

The functionality $\mathcal{F}_{Setup}$

Setup: On input (`Setup`,p) from all parties, the functionality stores the prime p. The adversary **A** chooses the set of corrupted parties $P_C \subseteq \{1, \ldots, n\}$.

1. Choose $g \in G$ and $s \in \mathbb{Z}_p^*$. Set $h = g^s$ and send g, h to **A**.
2. Send g, h to P_i, $i \notin P_C$.

RandomValues: On input (`RandomValues`,n) from all parties:

1. For $i \notin P_C$, the functionality chooses uniformly random $\mathbf{r}_i \leftarrow \mathbb{Z}_p^n$ and sends these to the party $\mathbf{P}_i$.
2. For $i \in P_C$, **A** inputs $\mathbf{r}_i \in \mathbb{Z}_p^n$.
3. Set $[\![\mathbf{r}]\!] \leftarrow (\mathbf{r}_1, \ldots, \mathbf{r}_n)$.
4. Return $([\![\mathbf{r}]\!])$.

Triples: On input (`Triples`,m) from all parties:

1. For $i \notin P_C$, the functionality samples $a_i, b_i \in \mathbb{Z}_p^m$ at random and sends them to P_i.
2. For $i \notin P_C$, the functionality computes $Com(a_i), Com(b_i)$ and sends them to the server.
3. For $i \in P_C$, **A** inputs $a_i, b_i, c_i \in \mathbb{Z}_p^m$.
4. For $i \in P_C$, the functionality computes $Com(a_i), Com(b_i)$ and $Com(c_i)$, and sends them to the server.
5. Define $a = \sum_{j=1}^{n} a_j$, $b = \sum_{j=1}^{n} b_j$.
6. Let $j \notin P_C$ be the smallest index of an honest player. For all $i \notin P_C$, $i \neq j$ choose $c_i \in \mathbb{Z}_p^m$ uniformly at random. For P_j let $c_j = ab - \sum_{i \in [n], i \neq j} c_i$. Send c_i to P_i, $i \notin P_C$.
7. For $i \notin P_C$, the functionality computes $Com(c_i)$ and sends them to the server.

Figure 4. The ideal functionality that describes the offline phase.

5.3. Online Phase

The online phase in our protocol is executed as shown in Figure 5. In **Initialize**, the parties obtain some pre-computed values through the functionalities, which are subsequently used to calculate a predefined circuit during the online phase of this protocol. Furthermore, the server obtains the commitments through the functionality which are used to detect malicious parties. In **Input**, each party creates the initial shares of its secret value and sends them to the other parties. Each party also obtains the shares of every other party. In **Compute**, all parties calculate the circuit and send the encrypted final shares, which are used to reconstruct the output of the computation, to the server. They also create some additional values to guarantee the fairness of this protocol. In **Output**, the server checks the validity of the final shares sent by all parties. If there is no malicious party, the server sends all encrypted final shares to each party. Otherwise, the server notifies the honest parties of the existence of the malicious party.

Protocol $\prod_{FSMC}$

Initialize: On input (`Init`, n, C, p) (where C is a circuit with n inputs and one output and consists of addition and multiplication gates over $\mathbb{Z}_p$ and $p \in \mathbb{P}$) from all parties:

1. Every party sends (`Setup`, p) to $\mathcal{F}_{Setup}$ and obtains the values used to generate and verify the commitments.
2. Every party sends (`RandomValues`, n) to $\mathcal{F}_{Setup}$ and obtains the random values used to divide its secret values.
3. Every party sends (`Triples`, m) to $\mathcal{F}_{Setup}$ (where m is the number of multiplication gates of the circuit C) and obtains m multiplication triples. The server obtains the commitments of each multiplication triple from $\mathcal{F}_{Setup}$.
4. Every party sends (`Setup`, n) to $\mathcal{F}_{BE}$ (where n is the number of parties participating in the protocol) and obtains a public key, PK_{BE}, as well as its own private key sk_i $(i = 1, \ldots, n)$.

Input: On input (`Input`, P_i, x_i) from each party P_i (where x_i is the secret value of party P_i):

1. $[\![r]\!]$ is privately opened as r to P_i.
2. The party P_i broadcasts $\epsilon = x_i - r$.
3. Each party locally computes $[\![x_i]\!] = [\![r]\!] + \epsilon$ and creates $Com([\![x_i]\!])$ ($[\![x_i]\!]$ is the share of the secret value of party P_i).
4. Party P_i sends the commitment it creates to the server.

Compute: On input (`compute`) from all parties, if **Initialize** has been executed and inputs for all input wires of C have been assigned, evaluate C gate by gate as follows:

Add: For two values $([\![r]\!], [\![s]\!])$, each party locally computes $[\![t]\!] = [\![r]\!] + [\![s]\!]$.

Multiply: To multiply two values $([\![r]\!], [\![s]\!])$ (using the multiplication triple $([\![a]\!], [\![b]\!], [\![c]\!])$):

1. Each party calculates $[\![\gamma]\!] = [\![r]\!] - [\![a]\!]$, $[\![\delta]\!] = [\![s]\!] - [\![b]\!]$.
2. The parties publicly reconstruct γ, δ.
3. Each party locally calculates $[\![t]\!] = [\![c]\!] + \delta[\![a]\!] + \gamma[\![b]\!] + \gamma\delta$.

Send: For the final shares $[\![y]\!]$ of the output, each party creates the value used to verify its correctness as follows:

1. Each party encrypts the final share by taking as input a recipient set, all the parties participating in the protocol, and a public key for a broadcast encryption system. Let $BE([\![y]\!])$ be the encryption of the final share using a broadcast encryption system.
2. Each party creates $Com([\![y]\!])$, the commitment of the final share.
3. Each party generates the non-interactive zero-knowledge proof $NIZK([\![y]\!])$, proving the equality of the value used in encryption and the commitment.
4. Each party sends $(BE([\![y]\!]), Com([\![y]\!]), NIZK([\![y]\!]))$ to the server.

Output: **If** the server receives the value created in **Compute.Send** from all the parties:

1. To verify the commitment, the server follows the computation gates of the evaluated circuit C in the same order as they were computed.
   Add: The parties added $[\![r]\!]$ and $[\![s]\!]$ to $[\![t]\!]$. Set $Com([\![t]\!]) = Com([\![r]\!]) \cdot Com([\![s]\!])$
   Multiply: The parties multiplied $[\![r]\!]$ and $[\![s]\!]$ to $[\![t]\!]$ using multiplicative triples $([\![a]\!], [\![b]\!], [\![c]\!], [\![\gamma]\!], [\![\delta]\!])$.
   (a) Set $Com([\![t]\!]) = Com([\![c]\!]) \cdot Com([\![a]\!])^{\delta} \cdot Com([\![b]\!])^{\gamma} \cdot pc(\gamma \cdot \delta, 0)$.
   (b) Check that $Com([\![r]\!]) \cdot Com([\![a]\!])^{-1} =?Com([\![\gamma]\!])$
   and $Com([\![s]\!]) \cdot Com([\![b]\!])^{-1} =?Com([\![\delta]\!])$. If not, output REJECT.
2. The server verifies the non-interactive zero-knowledge proof.
3. **If** malicious parties are detected, the server and the honest parties do the following:
   (a) The server sends the identities of the malicious parties to the honest parties.
   (b) Each honest party restarts the protocol from the beginning.

   If there is no malicious party,
   (a) The server sends all $BE([\![y]\!])$s to each party.
   (b) Each party decrypts all $BE([\![y]\!])$s and reconstructs the output using the final shares.

Figure 5. The protocol for the online phase.

6. Security in the Online Phase

In this section, we prove the security of the online phase of our protocol. We assume at least one honest party participating in this protocol. We prove the computational security of our protocol, which means that every probabilistic polynomial-time (PPT) adversary succeeds in breaking the scheme with only negligible probability in a reasonable amount of time. We prove that for any polynomial-time adversary **A**, there exists a simulator $\mathcal{S}_{ONLINE}$ that makes protocol $\prod_{FSMC}$ indistinguishable from the functionality $\mathcal{F}_{SMC}$ to the polynomial–time environment **Z**.

Theorem 1. *In the $\mathcal{F}_{Setup}, \mathcal{F}_{BE}, \mathcal{F}_{NIZK}$-hybrid model with a random oracle, the protocol $\prod_{FSMC}$ fairly implements $\mathcal{F}_{SMC}$ with computational security against any static adversary corrupting up to $(n-1)$ parties, corresponding to Definition 2, if the DLP is hard in the group G.*

Proof. We prove the above theorem by constructing the simulator $\mathcal{S}_{ONLINE}$ in Figure 6. The simulator simulates a server and honest parties in the real world and corrupted parties in the ideal world. It runs an instance $\prod$ of $\prod_{FSMC}$ with simulated honest parties and those controlled by the environment **Z**. For **Initialize**, the simulator and the corrupted parties perform the same steps as in $\prod_{FSMC}$. During **Input**, the corrupted parties execute **Input** in $\prod_{FSMC}$ while a simulator simulates the honest parties with their inputs set to 0. The simulator extracts the input values of the corrupted parties from $\prod$ and sends them to $\mathcal{F}_{SMC}$. Moreover, the honest parties send their input values to $\mathcal{F}_{SMC}$. In view of environment **Z**, since the shares of each party's input value are uniformly random and do not reveal any information regarding the input values, this stage, **input**, is indistinguishable from real execution.

During **Compute**, the simulator and the corrupted parties execute **Compute.Add** and **Compute.Multiply** in the same manner as in $\prod_{FSMC}$. For **Compute.send**, the simulator generates final shares of the simulated honest parties for $\prod$ as follows: first, the simulator computes the output of $\prod$, y', using all inputs from both corrupted and honest parties for $\prod$. Second, for any party P_i of the simulated honest parties, it modifies the final share of P_i by adding the value $(y - y')$. For all honest parties except P_i, the simulator keeps their final values intact. The distribution of the shares of simulated honest parties is the same as in a real execution of the protocol. Finally, the simulator executes Step 1 of **Compute.send** in $\prod_{FSMC}$ with the modified final shares of the simulated honest parties. The corrupted parties execute the same steps as in **Compute.send** in $\prod_{FSMC}$. Furthermore, if **Z** decides to stop the execution, a simulator $\mathcal{S}_{ONLINE}$ forwards this to the ideal functionality $\mathcal{F}_{SMC}$. As in the real execution, **Z** will not obtain any additional information.

During **Output**, the simulator acts as a server in the real world. It executes **Output** in $\prod_{FSMC}$ with the corrupted parties. The simulator performs Steps 1 to 3 with values sent from the corrupted parties in **Compute.Send**. For Step 4, if any failures occur in Steps 2 or 3, the simulator outputs $\perp$. Otherwise, it sends the ciphertexts of all final shares in $\prod$ to the corrupted parties.

Finally, we need to show that the probability that adversary **A** can cheat is negligible in real protocol execution. If **A** is able to generate a commitment with a random value, R, which can pass the verification, this random value is verified as a correct value in the **Output** stage. Following this, if **A** generates the ciphertext of R and the non-interactive zero-knowledge proof with respect to the ciphertext, the server has no choice but to verify R as a correct final share. However, since the *DLP* is hard in G, it is nearly impossible for **A** to generate faulty commitments that can pass verification. Therefore, the probability that **A** can cheat in real protocol execution is negligible. □

Simulator $\mathcal{S}_{ONLINE}$

The simulator waits for the set P_C of corrupted parties from the environment **Z**. The values g, h are provided to the server as a CRS by this simulator.

Initialize: On input (`Init`, n, C, p) from **Z**:

1. Start a local instance $\prod$ of $\prod_{FSMC}$ with the corrupted parties in P_C and simulated honest parties.
2. Run **Setup**, **RandomValues** and **Triples** of $\mathcal{F}_{Setup}$ as in $\prod_{FSMC}$. The simulated honest parties and **Z** communicate with $\mathcal{F}_{Setup}$ through the simulator.
3. Run **Setup** of $\mathcal{F}_{BE}$ as in $\prod_{FSMC}$. The simulated honest parties and **Z** communicate with $\mathcal{F}_{BE}$ and $\mathcal{F}_{NIZK}$ through the simulator.

Input: On input (`Input`, $P_i, \cdot$) from each party P_i:

If P_i is corrupted, extract the input value x_i from $\prod$ and send it to $\mathcal{F}_{MPC}$.
If P_i is honest, execute the **Input** of $\prod_{FSMC}$ for a fake input 0.

Compute: On input (`Compute`) from **Z**, if **Initialize** has been executed and inputs for all input gates of C have been provided, calculate C gate by gate as follows:

Add: Execute **Compute.Add** in $\prod_{FSMC}$.
Multiply: Execute **Compute.Multiply** in $\prod_{FSMC}$.
Send: Obtain the output y from $\mathcal{F}_{MPC}$ and simulate $\prod_{FSMC}$ as follows:

1. Generate the final shares for the simulated honest parties for $\prod$:
   (a) Let y' be the output of $\prod$ with **Z** at any given time.
   (b) For any P_i, one of the simulated honest parties, modify the final share of P_i by adding the value $(y - y')$.
   (c) For all honest parties except P_i, keep the final values intact.
2. Execute Step 1 of **Compute.Send** in $\prod_{FSMC}$ to generate the value to be sent to the corrupted parties.

Output: Execute **Output** in $\prod_{FSMC}$ with corrupted parties.

If there exist any failures of verification, output $\perp$.
Otherwise send the ciphertexts of all the final shares to the corrupted parties.

Figure 6. Simulator for the online phase.

7. Application

In this section, we provide a privacy-preserving data aggregation mechanism for advanced metering infrastructures in smart grids by applying our FSMC protocol. In Section 7.1, we propose a privacy-preserving data aggregation mechanism for general circuits. In other words, we may compute any function, not merely the sum or the standard deviation, by using this mechanism in smart grids. With our FSMC protocol, this privacy-preserving data aggregation mechanism prevents malicious parties from revealing the valid output of computation and disturbing load balancing. In Section 7.2, we propose an enhanced version of our FSMC protocol in the case of *computing the average*. We then analyze the efficiency of the enhanced protocol by comparison with the original FSMC protocol.

7.1. Applying FSMC Protocol to the Smart Grid

As shown in Figure 7, there are three types of entities in a smart grid: *utility*, *gateway*, and *smart meters*. The smart meters, $SM_i (1 \leq i \leq n)$, collect the real-time usage data of each user. The data of users belonging to a specific area are relayed to a local gateway, GW. GW aggregates the collected data into compacted data and forwards this information to users and the *utility*. In this process, the privacy of users' data must be preserved because

this information can reveal their living patterns (the number of people in a household, the hours at which they are typically at home and away, their sleeping patterns, etc.).

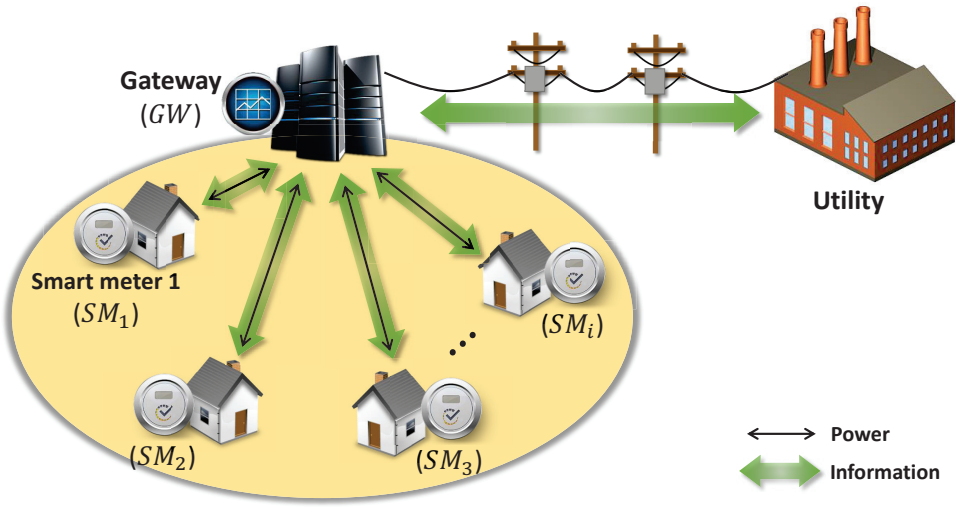


Figure 7. Smart grid architecture.

Our mechanism consists of four phases: (1) System Setup, (2) User Data Sharing, (3) Privacy-preserving Data Aggregation, and (4) Secure Data Retrieval. The details of each phase are as follows.

7.1.1. System Setup

We assume that there exists a trusted third party that generates key pairs for the public-key broadcast encryption (PKBE) scheme, and that the key pair used to execute the PKBE scheme in a local area should be embedded in each SM_i in the manufacturing stage. In order to set up the system, the *utility* executes the following steps:

- **Step 1.** Choose $g \in G$ and $s \in \mathbb{Z}_p^*$. Set $h = g^s$ and send g, h to every SM_i. These values are used to compute commitment.
- **Step 2.** Choose uniformly random $\mathbf{r}_i \leftarrow \mathbb{Z}_p^n$ and send these values to each SM_i. These values are used to share the secret input of each SM_i.
- **Step 3.** Choose $a_i, b_i, c_i \in \mathbb{Z}_p^m$ such that $a = b \cdot c$, where $a = \sum_{j=1}^{n} a_i, b = \sum_{j=1}^{n} b_i$, and $c = \sum_{j=1}^{n} c_i$. Send the triple (a_i, b_i, c_i) to each SM_i. These values are used as multiplication triples.
- **Step 4.** Create the commitments of the multiplication triples, $\left[Com(a_i), Com(b_i), Com(c_i)\right]$ and send them to GW. These are used to detect malicious SM_i in the Secure Data Retrieval phase. $\left(Com(x) = \left(g^x h^r\right)$ where r is a random value$\right)$.

7.1.2. User Data Sharing

In order to perform computations with metering data collected by SM_i while keeping them secret, each SM_i splits its secret data into n shares using the following steps:

- **Step 1.** $[\![r]\!]$ is privately opened as r to SM_i, where $[\![\mathbf{r}]\!] \leftarrow (\mathbf{r}_1, \ldots, \mathbf{r}_n)$.
- **Step 2.** Broadcast $\epsilon_i = x_i - r$, where x_i is the metering data for each SM_i.
- **Step 3.** Compute $[\![x_i]\!] = [\![r]\!] + \epsilon_i$ locally, where $[\![x_i]\!]$ is the share of x_i for each SM_i.
- **Step 4.** Create the commitments of all shares, $Com\left([\![x_i]\!]\right)$, and send them to GW. These commitments are used to detect malicious SM_i in the Secure Data Retrieval phase.

7.1.3. Privacy-Preserving Data Aggregation

Since each secret input data $x_i (i \in n)$ is divided into shares $[\![x_i]\!]$, we can guarantee the privacy of each SM_i's metering data. In order to perform aggregation by using the shares of each metering data $[\![x_i]\!]_{i \in n}$, each SM_i executes the following steps:

- **Step 1.** Run **Compute.Add** and **Compute.Multiply** in $\prod_{FSMC}$ to compute the final share of the output for computation.
- **Step 2.** Encrypt the final share using a PKBE scheme under the recipient set, including all $SM_i (i \in n)$, and the *utility* in a local area.
- **Step 3.** Create the commitments and the non-interactive zero-knowledge proof as in **Compute.Send** of $\prod_{FSMC}$. Send them to GW.

In **Step 1**, each SM_i computes the final share for the output of the pre-defined computation. The final share of any statistical function, not merely the sum or the standard deviation, can be computed. In **Step 2**, each SM_i encrypts the final share. Since the ciphertexts of the final shares are encrypted under the recipient set, including all the $SM_i (i \in n)$ and the *utility* only, GW is unable to obtain any information regarding the final shares. In **Step 3**, each SM_i creates the values used to detect the malicious SM_j in the Secure Data Retrieval phase and guarantee the fairness of the protocol.

7.1.4. Secure Data Retrieval

In order to detect malicious SM_i, GW executes the following steps:

- **Step 1.** Follow Step 1 of **Output** in $\prod_{FSMC}$ to verify the commitments sent from each SM_i.
- **Step 2.** Verify the non-interactive zero-knowledge proof sent from each SM_i.
- **Step 3.**
  - *If* all verifications yield true, send all ciphertexts sent from each SM_i in the Privacy-preserving Data Aggregation phase to all $SM_i (i \in n)$ and the *utility* in a local area.
  - *Otherwise*, notify each honest SM_i of malicious $SM_j (j \neq i)$s identities.

Then, each honest SM_i copes with the following situations as follows:

- **Step 1.**
  - *If* there is no malicious SM_i, decrypt the ciphertexts from GW and reconstruct the output using the final shares.
  - *Otherwise*, restart the protocol from the beginning excluding malicious SM_j.

To detect malicious SM_i, (GW) carries out the verification of each step of computation. Since a single GW is in charge of a whole specific area, we may assume high computation power of GW enough to perform this verification.

To facilitate a better understanding, we describe a simple example of the overall process with three smart meters participating in the protocol in Figure 8.

7.2. Analysis

7.2.1. Improving Efficiency (in terms of **Average**)

In Section 7.1, if a malicious SM_j is detected, each honest SM_i should restart the protocol from the beginning. In this case, additional interactions among the honest SM_i are required. In this subsection, we provide an enhanced version of the original protocol described in Section 7.1 to compute the *average*. In the enhanced version, when each honest SM_i performs a re-computation following cheater detection, no interactions are required among honest users and, thus, the computation overhead is drastically reduced.

This enhanced version of the protocol is executed in the same manner as before detecting malicious SM_j. Hence, the three phases of System Setup, User Data Sharing, and Privacy-preserving Data Aggregation are identical to those in the original protocol in Section 7.1. If a malicious SM_j is detected by GW, each honest SM_i reuses the pre-

viously computed final share to *locally* generate the new one. A detailed explanation is provided below.

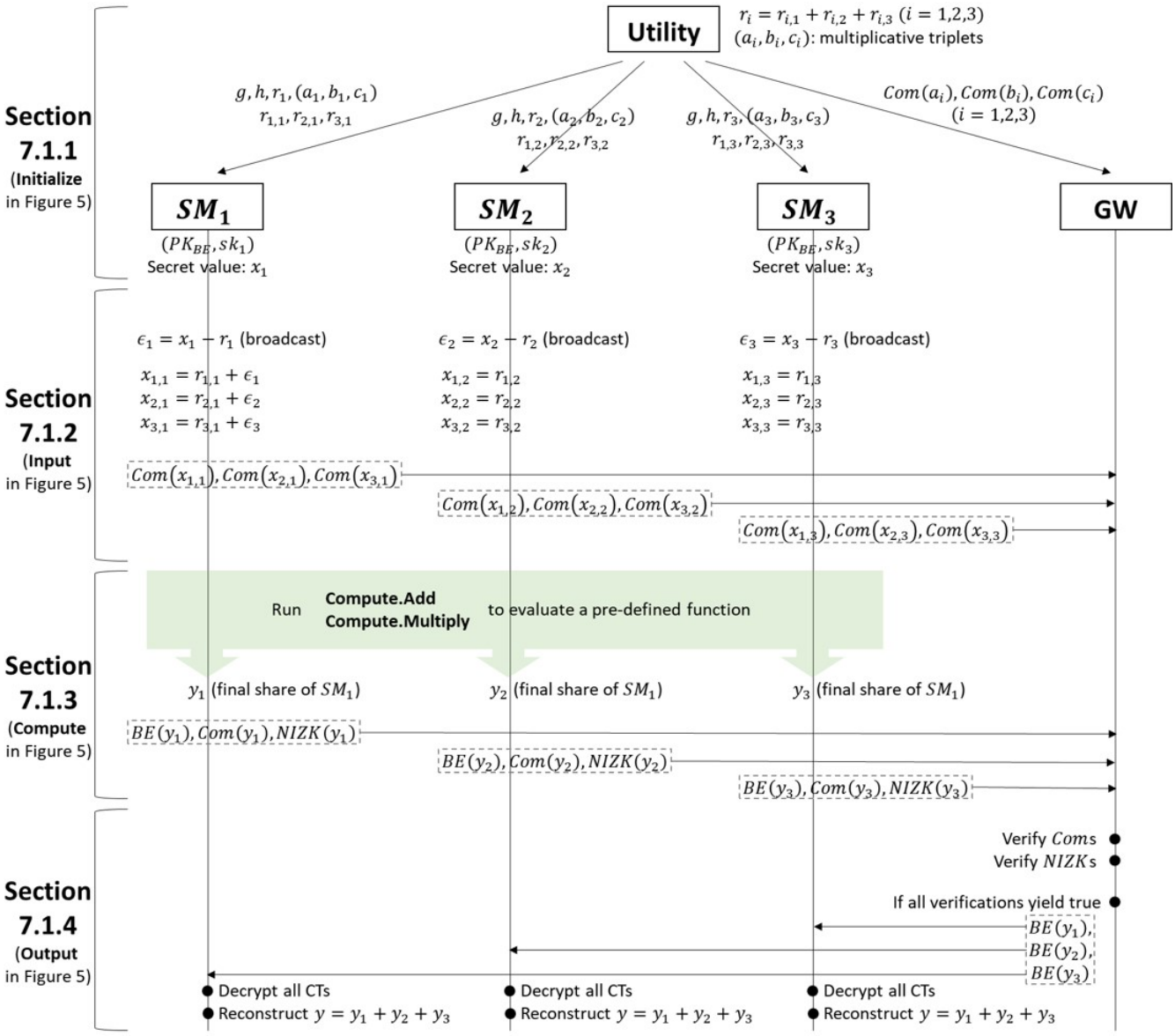


Figure 8. Simple example of the protocol in Section 7.1.

- **Secure Data Retrieval**
  In order to detect the malicious SM_j, GW executes the following steps:
  - **Step 1.** Follow Step 1 of **Output** in $\prod_{FSMC}$ to verify the commitments sent from each SM_i.
  - **Step 2.** Verify the non-interactive zero-knowledge proof sent by each SM_i.
  - **Step 3.**
    - *If* all verification outputs are true, send all ciphertexts sent by each SM_i in the Privacy-preserving Data Aggregation phase to all $SM_i (i \in n)$ and the $utility$ in a local area.
    - *Otherwise*, notify each honest SM_i of the identities of the malicious $SM_j (j \neq i)$s.

  If there is no malicious SM_j, each SM_i decrypts the ciphertexts sent by GW and reconstructs the average by adding all final shares. If there exists at least one malicious SM_j, each honest SM_i executes the following steps:
  (Let P_C be the set of identities of the malicious SM_j.)

- **Step 1.** Compute $s_i' = n \cdot s_i - \sum_{j \in P_C} x_{j,i} + \sum_{j \in P_C} x_{i,j}$, where s_i is a final share of SM_i computed prior to detection and $x_{j,i}$ is an initial share that SM_j assigned to SM_i.
- **Step 2.** Set the new final share $s_i'' = \frac{1}{n-k} \cdot s_i'$, where $k = |P_C|$.
- **Step 3.** Create the commitment of its new final share, s_i'', and the non-interactive zero-knowledge proof as in **Compute.Send** of $\prod_{FSMC}$. Send these to GW.

This phase should be repeated until no malicious party exists. Once GW detects a malicious SM_j, each honest SM_i *locally* executes additional computations to generate the new final share. From the previously computed final share, each honest SM_i removes shares $x_{j,i}$, which were provided by the malicious SM_j, and adds shares $x_{i,j}$, provided to the malicious SM_j. Since the parts related to the malicious SM_j are removed through this process, the new final share s_i'' is correct for all honest SM_i.

We can improve the efficiency of our FSMC protocol described in Section 5 in the same manner when computing the average. As mentioned above, in the enhanced version of the protocol, the **Initialize**, **Input**, and **Compute** phase are the same as in the original protocol $\prod_{FSMC}$. Figure 9 represents the **Output** phase in the enhanced version of the FSMC protocol for computing the average.

Protocol $\prod_{Average}$

Output: **If** the server receives the value created in **Compute.Send** from all parties:

1. This step is executed in the same manner as in $\prod_{FSMC}$.
2. This step is executed in the same manner as in $\prod_{FSMC}$.
3. **If** malicious parties are detected, the server and the honest parties do the following:
   (a) The server sends the identities of the malicious parties to the honest parties. (Let P_C be the set of the identities of malicious parties.)
   (b) Each honest party P_i computes $s_i' = n \cdot s_i - \sum_{j \in P_C} x_{j,i} + \sum_{j \in P_C} x_{i,j}$ (where s_i is the final share of P_i computed prior to detection and $x_{i,j}$ is an initial share that party P_i gave to party P_j).
   (c) Set the new final share of P_i
   $s_i'' = \frac{1}{n-k} \cdot s_i'$ (where $k = |P_C|$).
   (d) Every honest party runs **Compute.Send** with their new final shares.

   If there is no malicious party,
   (a) The server sends all $BE([\![y]\!])$s to each party.
   (b) Each party decrypts all $BE([\![y]\!])$s and reconstructs the output using the final shares.

Figure 9. The enhanced version of the FSMC protocol for computing the average.

7.2.2. Efficiency Analysis

In this section, we compare the computation overhead incurred by each party in case of performing operations in the User Data Sharing phase and the Privacy-preserving Data Aggregation phase. (In terms of communication cost, our proposed protocol is less efficient than the SPDZ protocol [12]. We have combined the SPDZ protocol with a public-key broadcast encryption (PKBE) scheme, a commitment scheme, and a non-interactive zero-knowledge (NIZK) proof system to provide fairness and prevent malicious parties from obtaining the final output. This results in additional values to be shared by each party (i.e., public key of PKBE, commitment, and proof of NIZK), which incurs extra communication overhead).

We conduct an experiment using the PBC [66] and GMP [67] libraries on a laptop with an Intel Core i5-4200U CPU and 4 GB of RAM in order to calculate operation costs. In the original protocol, when each honest SM_i executes the User Data Sharing phase during re-computation, it should generate the commitment of each share. The generation of a

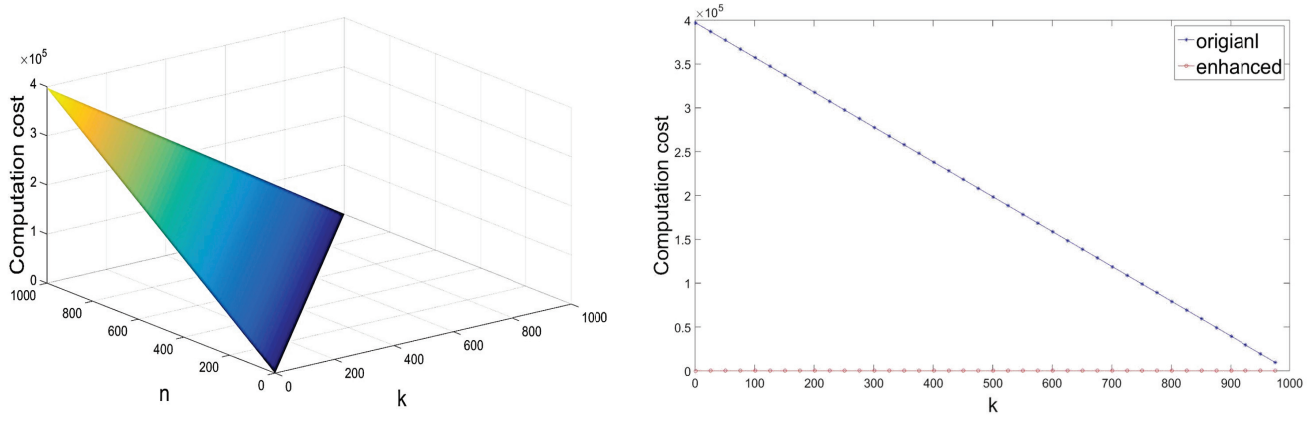
commitment, $g^x h^r$, requires two exponentiation operations in $\mathbb{Z}_p(|p| = 1024$ bits) and a multiplication operation in $\mathbb{G}$ with 160 bits. On the contrary, in the enhanced version, there is no need to re-run the User Data Sharing phase.

We analyze the computation overhead of these protocols by dividing the cases into two types. In Section 7.2.2, we experiment with a case where detection occurs only once and the number of malicious SM_j varies. In Section , we experiment with cases where only one malicious SM_j is detected and the number of detections varies.

- **One detection: k malicious users**
  In this case, we compare the computational overhead of two protocols when detection occurs only once and k malicious SM_j are detected. The number of generating commitments and computing addition operations are listed in Table 1, where n is the number of smart meters. According to the experimental results, a single addition operation in $\mathbb{Z}_p$ and a single commitment generation cost 0.0004 ms ($\mathcal{T}_{add}$) and 397.3102 ms ($\mathcal{T}_{com}$), respectively. Since the User Data Sharing phase is not required during re-computation in the enhanced version of the protocol, only the Privacy-preserving Data Aggregation phase influences the computational cost of the enhanced version.
  The computational cost of original protocol depends on both k and n. We show the variation in computational overhead in terms of both k and n in Figure 10a. Given that n is fixed, the greater the value of k, the smaller the computational cost of the protocol. On the contrary, computational cost of the enhanced version of the protocol depends only on k. We thus depict the difference in computational overhead between two protocols in terms of k in Figure 10b, given that n = 1000. From the figure, we see that the enhanced version is more efficient until k becomes approximately 999.99. This means that the enhanced version of the protocol is more efficient than the original one, excluding the case where all smart meters are detected as malicious.

Table 1. Comparison of Computational Complexity-One detection, k malicious users.

	Original Protocol	**Enhanced Version**
Input (commitment)	$(n-k) \times \mathcal{T}_{com}$	-
Compute (addition)	$(n-k) \times \mathcal{T}_{add}$	$2k \times \mathcal{T}_{add}$



(a) Computational cost of original protocol (b) Comparison of computational overheads

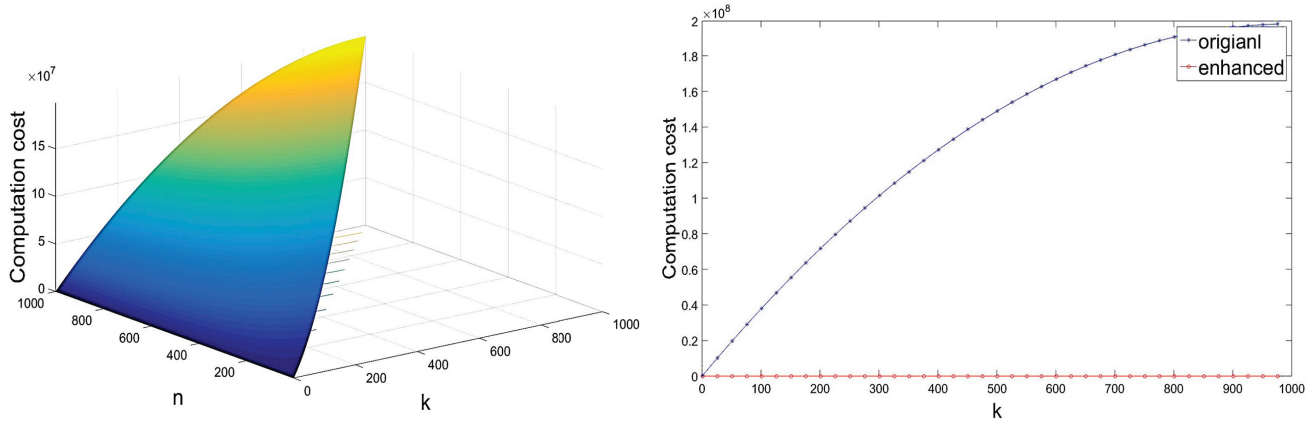
Figure 10. Computational overhead-one detection, k malicious users.

- **One malicious user: k detection**
  In this case, we compare the computation overhead of two protocols when detection occurs k times and one malicious SM_j is detected at a time. The number of generating commitments and computing addition operations is listed in Table 2, where n is the

number of smart meters. (As above, $\mathcal{T}_{add}$ = 0.0004 ms and $\mathcal{T}_{com}$ = 397.3102 ms.) We show the variation in computational costs of the original protocol in terms of both k and n in Figure 11a, where detection occurs k times and only one malicious SM_j is detected at a time. Given that n is fixed, the greater the value of k, the greater the computational cost of the protocol. On the contrary, computational cost of the enhanced version of the protocol depends only on k. We thus depict the difference in computational overhead between two protocols in terms of k in Figure 11b, given that n = 1000. From the figure, we see that the enhanced version of the protocol is more efficient than the original one in all cases.

Table 2. Comparison of Computational Complexity-One malicious user, k detections.

	Original Protocol	Enhanced Version
Input (commitment)	$\left\{nk - \frac{1}{2}k(k+1)\right\} \times \mathcal{T}_{com}$	-
Compute (addition)	$\left\{nk - \frac{1}{2}k(k+1)\right\} \times \mathcal{T}_{add}$	$2k \times \mathcal{T}_{add}$



(a) Computational cost of original protocol (b) Comparison of computational overheads

Figure 11. Computational overhead-one malicious user, k detections.

8. Conclusions and Future Research Directions

In this paper, we have proposed a secure multi-party computation (SMC) protocol that guarantees fairness and provides cheater detection. The proposed protocol has an efficient online phase because it is based on the SPDZ protocol and is secure if at least one party of n is honest. Our protocol guarantees both the correctness of the output of computation by detecting malicious parties as well as the fairness of the protocol by performing detection prior to deriving the final output. We provided a definition of UC-secure fair SMC and proved the security of fair SMC in the UC setting. Moreover, we proposed an enhanced version of our protocol, in terms of efficiency, for smart grids and analyzed the difference between the original protocol and an improved one.

In future work, we plan to construct an enhanced version of protocols for various kinds of operations, as well as reduce the overall communication overhead. The multi-party computation protocol requires a lot of communication by its nature, and it would be better to combine various kinds of techniques to reduce the amount of communication with our proposed protocol.

Author Contributions: Conceptualization, M.S.; methodology, M.S.; software, M.S.; validation, M.S.; formal analysis, M.S.; investigation, M.S.; resources, M.S.; writing—original draft preparation, M.S.; writing—review and editing, M.S.; visualization, M.S. The author have read and agreed to the published version of the manuscript.

Funding: This research was supported by the Basic Research Program through the National Research Foundation of Korea(NRF) funded by the MSIT(grant number: 2021R1A4A502890711).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Naor, M.; Pinkas, B.; Sumner, R. Privacy Preserving Auctions and Mechanism Design. In Proceedings of the 1st ACM Conference on Electronic Commerce, Denver, CO, USA, 3–5 November 1999; EC'99, pp. 129–139.
2. Bogetoft, P.; Christensen, D.; Damgård, I.; Geisler, M.; Jakobsen, T.; Krøigaard, M.; Nielsen, J.; Nielsen, J.; Nielsen, K.; Pagter, J.; et al. Secure Multiparty Computation Goes Live. In *Financial Cryptography and Data Security*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2009; Volume 5628, pp. 325–343.
3. Ben-David, A.; Nisan, N.; Pinkas, B. FairplayMP: A system for secure multi-party computation. In Proceedings of the 15th ACM Conference on Computer and Communications Security, New York, NY, USA, 27–31 October 2008; pp. 257–266.
4. Bogdanov, D.; Laur, S.; Willemson, J. Sharemind: A framework for fast privacy-preserving computations. In *European Symposium on Research in Computer Security*; Springer: Berlin/Heidelberg, Germany, 2008; pp. 192–206.
5. Damgård, I.; Geisler, M.; Krøigaard, M.; Nielsen, J.B. Asynchronous multiparty computation: Theory and implementation. In *International Workshop on Public Key Cryptography*; Springer: Berlin/Heidelberg, Germany, 2009; pp. 160–179.
6. Burkhart, M.; Strasser, M.; Many, D.; Dimitropoulos, X. SEPIA: Privacy-preserving aggregation of multi-domain network events and statistics. *Network* **2010**, *1*, 15–32.
7. Peter, A.; Tews, E.; Katzenbeisser, S. Efficiently Outsourcing Multiparty Computation Under Multiple Keys. *IEEE Trans. Inf. Forensics Secur.* **2013**, *8*, 2046–2058. [CrossRef]
8. Clark, M.; Hopkinson, K. Transferable Multiparty Computation With Applications to the Smart Grid. *IEEE Trans. Inf. Forensics Secur.* **2014**, *9*, 1356–1366. [CrossRef]
9. Kerschbaum, F. Adapting Privacy-Preserving Computation to the Service Provider Model. In Proceedings of the CSE'09, International Conference on Computational Science and Engineering, Vancouver, BC, Canada, 29–31 August 2009; Volume 3, pp. 34–41.
10. Damgård, I.; Orlandi, C. Multiparty Computation for Dishonest Majority: From Passive to Active Security at Low Cost. In *Advances in Cryptology-CRYPTO 2010*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2010; Volume 6223, pp. 558–576.
11. Hirt, M.; Tschudi, D. Efficient General-Adversary Multi-Party Computation. In *Advances in Cryptology-ASIACRYPT 2013*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2013; Volume 8270, pp. 181–200.
12. Damgård, I.; Pastro, V.; Smart, N.; Zakarias, S. Multiparty Computation from Somewhat Homomorphic Encryption. In *Advances in Cryptology-CRYPTO 2012*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2012; Volume 7417, pp. 643–662.
13. Peng, J.; Choo, R.K.K.; Ashman, H. Astroturfing detection in social media: Using binary n-gram analysis for authorship attribution. In Proceedings of the 2016 IEEE Trustcom/BigDataSE/ISPA, Tianjin, China, 23–26 August 2016; pp. 121–128.
14. Mahbub, S.; Pardede, E.; Kayes, A.; Rahayu, W. Controlling astroturfing on the internet: a survey on detection techniques and research challenges. *Int. J. Web Grid Serv.* **2019**, *15*, 139–158. [CrossRef]
15. Mahbub, S.; Pardede, E.; Kayes, A. Detection of Harassment Type of Cyberbullying: A Dictionary of Approach Words and Its Impact. *Secur. Commun. Netw.* **2021**, *2021*. [CrossRef]
16. Cabello, S.; Padro, C.; Saez, G. Secret Sharing Schemes with Detection of Cheaters for a General Access Structure. *Des. Codes Cryptogr.* **2002**, *25*, 175–188. [CrossRef]
17. Araki, T. Efficient (k,n) Threshold Secret Sharing Schemes Secure Against Cheating from n–1 Cheaters. In *Information Security and Privacy*; Lecture Notes in Computer Science; Pieprzyk, J., Ghodosi, H., Dawson, E., Eds.; Springer: Berlin/Heidelberg, Germany, 2007; Volume 4586, pp. 133–142.
18. Harn, L.; Lin, C. Detection and identification of cheaters in (t, n) secret sharing scheme. *Des. Codes Cryptogr.* **2009**, *52*, 15–24. [CrossRef]
19. Obana, S. Almost Optimum t-Cheater Identifiable Secret Sharing Schemes. In *Advances in Cryptology-EUROCRYPT 2011*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2011; Volume 6632, pp. 284–302.
20. Baum, C.; Damgård, I.; Orlandi, C. Publicly Auditable Secure Multi-Party Computation. In *Security and Cryptography for Networks*; Lecture Notes in Computer Science; Springer: Cham, Switzerland, 2014; Volume 8642, pp. 175–196.
21. Spini, G.; Fehr, S. Cheater detection in SPDZ multiparty computation. In *International Conference on Information Theoretic Security*; Springer: Berlin/Heidelberg, Germany, 2016; pp. 151–176.
22. Cunningham, R.; Fuller, B.; Yakoubov, S. Catching MPC cheaters: Identification and openability. In *International Conference on Information Theoretic Security*; Springer: Berlin/Heidelberg, Germany, 2017; pp. 110–134.
23. Baum, C.; Orsini, E.; Scholl, P.; Soria-Vazquez, E. Efficient constant-round MPC with identifiable abort and public verifiability. In *Annual International Cryptology Conference*; Springer: Berlin/Heidelberg, Germany, 2020; pp. 562–592.
24. Goldreich, O.; Micali, S.; Wigderson, A. How to Play ANY Mental Game. In Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing, New York, NY, USA, 25–27 May 1987; STOC'87, pp. 218–229.

25. Ben-Or, M.; Goldwasser, S.; Wigderson, A. Completeness Theorems for Non-cryptographic Fault-tolerant Distributed Computation. In Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing, Chicago, IL, USA, 2–4 May 1988; STOC'88, pp. 1–10.
26. Chaum, D.; Crépeau, C.; Damgard, I. Multiparty Unconditionally Secure Protocols. In Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing, Chicago, IL, USA, 2–4 May 1988; STOC'88, pp. 11–19.
27. Rabin, T.; Ben-Or, M. Verifiable Secret Sharing and Multiparty Protocols with Honest Majority. In Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing, Seattle, WA, USA, 14–17 May 1989; STOC'89, pp. 73–85.
28. Cleve, R. Limits on the Security of Coin Flips when Half the Processors Are Faulty. In Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing, Berkeley, CA, USA, 28–30 May 1986; STOC'86, pp. 364–369.
29. Beaver, D.; Goldwasser, S. Multiparty Computation with Faulty Majority. In *Advances in Cryptology-CRYPTO 1989*; Lecture Notes in Computer Science; Springer: New York, NY, USA, 1990; Volume 435, pp. 589–590.
30. Goldwasser, S.; Levin, L. Fair Computation of General Functions in Presence of Immoral Majority. In *Advances in Cryptology-CRYPTO 1990*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 1991; Volume 537, pp. 77–93.
31. Fitzi, M.; Gottesman, D.; Hirt, M.; Holenstein, T.; Smith, A. Detectable Byzantine Agreement Secure Against Faulty Majorities. In Proceedings of the Twenty-First Annual Symposium on Principles of Distributed Computing, Monterey, CA, USA, 21–24 July 2002; PODC'02, pp. 118–126.
32. Goldwasser, S.; Lindell, Y. Secure Computation without Agreement. In *Distributed Computing*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2002; Volume 2508, pp. 17–32.
33. Even, S.; Goldreich, O.; Lempel, A. A Randomized Protocol for Signing Contracts. *Commun. ACM* **1985**, *28*, 637–647. [CrossRef]
34. Boneh, D.; Naor, M. Timed Commitments. In *Advances in Cryptology-CRYPTO 2000*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2000; Volume 1880, pp. 236–254.
35. Garay, J.; MacKenzie, P.; Prabhakaran, M.; Yang, K. Resource Fairness and Composability of Cryptographic Protocols. In *Theory of Cryptography*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2006; Volume 3876, pp. 404–428.
36. Asokan, N.; Shoup, V.; Waidner, M. Optimistic fair exchange of digital signatures. In *Advances in Cryptology-EUROCRYPT'98*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 1998; Volume 1403, pp. 591–606.
37. Cachin, C.; Camenisch, J. Optimistic Fair Secure Computation. In *Advances in Cryptology-CRYPTO 2000*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2000; Volume 1880, pp. 93–111.
38. Lepinski, M.; Micali, S.; Peikert, C.; Shelat, A. Completely Fair SFE and Coalition-safe Cheap Talk. In Proceedings of the Twenty-third Annual ACM Symposium on Principles of Distributed Computing, St. John's, NL, Canada, 25–28 July 2004; PODC'04, pp. 1–10.
39. Andrychowicz, M.; Dziembowski, S.; Malinowski, D.; Mazurek, L. Secure Multiparty Computations on Bitcoin. In Proceedings of the 2014 IEEE Symposium on Security and Privacy, Berkeley, CA, USA, 18–21 May 2014; SP '14, pp. 443–458.
40. Bentov, I.; Kumaresan, R. How to Use Bitcoin to Design Fair Protocols. In *Advances in Cryptology-CRYPTO 2014*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2014; Volume 8617, pp. 421–439.
41. Kılınç, H.; Küpçü, A. Optimally Efficient Multi-Party Fair Exchange and Fair Secure Multi-Party Computation. In *Topics in Cryptology-CT-RSA 2015*; Lecture Notes in Computer Science; Springer: Cham, Switzerland, 2015; Volume 9048, pp. 330–349.
42. Asharov, G.; Lindell, Y.; Zarosim, H. Fair and Efficient Secure Multiparty Computation with Reputation Systems. In *Advances in Cryptology-ASIACRYPT 2013*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2013; Volume 8270, pp. 201–220.
43. Choudhuri, A.R.; Green, M.; Jain, A.; Kaptchuk, G.; Miers, I. Fairness in an unfair world: Fair multiparty computation from public bulletin boards. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, Dallas, TX, USA, 30 October–3 November 2017, pp. 719–728.
44. Paul, S.; Shrivastava, A. Efficient fair multiparty protocols using Blockchain and trusted hardware. In *International Conference on Cryptology and Information Security in Latin America*; Springer: Berlin/Heidelberg, Germany, 2019; pp. 301–320.
45. Li, F.; Luo, B.; Liu, P. Secure Information Aggregation for Smart Grids Using Homomorphic Encryption. In Proceedings of the 2010 First IEEE International Conference on Smart Grid Communications (SmartGridComm), Gaithersburg, MD, USA, 4–6 October 2010; pp. 327–332.
46. Kursawe, K.; Danezis, G.; Kohlweiss, M. Privacy-Friendly Aggregation for the Smart-Grid. In *Privacy Enhancing Technologies*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2011; Volume 6794, pp. 175–191.
47. Lu, R.; Liang, X.; Li, X.; Lin, X.; Shen, X. EPPA: An Efficient and Privacy-Preserving Aggregation Scheme for Secure Smart Grid Communications. *IEEE Trans. Parallel Distrib. Syst.* **2012**, *23*, 1621–1631.
48. Yang, L.; Xue, H.; Li, F. Privacy-preserving data sharing in Smart Grid systems. In Proceedings of the 2014 IEEE International Conference on Smart Grid Communications (SmartGridComm), Venice, Italy, 3–6 November 2014; pp. 878–883.
49. Mustafa, M.A.; Cleemput, S.; Aly, A.; Abidin, A. An MPC-based protocol for secure and privacy-preserving smart metering. In Proceedings of the 2017 IEEE PES Innovative Smart Grid Technologies Conference Europe (ISGT-Europe), Turin, Italy, 26–29 September 2017; pp. 1–6.
50. Mustafa, M.A.; Cleemput, S.; Aly, A.; Abidin, A. A secure and privacy-preserving protocol for smart metering operational data collection. *IEEE Trans. Smart Grid* **2019**, *10*, 6481–6490. [CrossRef]

51. Yao, A.C.C. How to generate and exchange secrets. In Proceedings of the 27th Annual Symposium on Foundations of Computer Science (sfcs 1986), Toronto, ON, Canada, 27–29 October 1986; pp. 162–167.
52. Demmler, D.; Schneider, T.; Zohner, M. *ABY-A Framework for Efficient Mixed-Protocol Secure Two-Party Computation*; NDSS: San Diego, CA, USA, 8–11 February 2015.
53. Chandran, N.; Gupta, D.; Rastogi, A.; Sharma, R.; Tripathi, S. EzPC: Programmable and efficient secure two-party computation for machine learning. In Proceedings of the 2019 IEEE European Symposium on Security and Privacy (EuroS&P), Stockholm, Sweden, 17–19 June 2019; pp. 496–511.
54. Patra, A.; Schneider, T.; Suresh, A.; Yalame, H. ABY2. 0: Improved mixed-protocol secure two-party computation. In Proceedings of the 30th {USENIX} Security Symposium ({USENIX} Security 21), Virtual Event, 11–13 August 2021.
55. Zhang, Y.; Steele, A.; Blanton, M. PICCO: A general-purpose compiler for private distributed computation. In Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, Berlin, Germany, 4–8 November 2013; pp. 813–826.
56. Zhang, Y.; Blanton, M.; Almashaqbeh, G. Implementing support for pointers to private data in a general-purpose secure multi-party compiler. *ACM Trans. Priv. Secur. (TOPS)* **2017**, *21*, 1–34. [CrossRef]
57. Keller, M. MP-SPDZ: A versatile framework for multi-party computation. In Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, 9–13 November 2020; pp. 1575–1590.
58. Shamir, A. How to share a secret. *Commun. ACM* **1979**, *22*, 612–613. [CrossRef]
59. Goldreich, O.; Micali, S.; Wigderson, A. How to play any mental game, or a completeness theorem for protocols with honest majority. In *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali*; Association for Computing Machinery: New York, NY, USA, 2019; pp. 307–328.
60. Beaver, D. Efficient multiparty protocols using circuit randomization. In *Annual International Cryptology Conference*; Springer: Berlin/Heidelberg, Germany, 1991; pp. 420–432.
61. Naor, D.; Naor, M.; Lotspiech, J. Revocation and tracing schemes for stateless receivers. In *Annual International Cryptology Conference*; Springer: Berlin/Heidelberg, Germany, 2001; pp. 41–62.
62. Dodis, Y.; Fazio, N. Public key broadcast encryption for stateless receivers. In *ACM Workshop on Digital Rights Management*; Springer: Berlin/Heidelberg, Germany, 2002; pp. 61–80.
63. Groth, J.; Ostrovsky, R.; Sahai, A. Perfect Non-interactive Zero Knowledge for NP. In *Advances in Cryptology-EUROCRYPT 2006*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2006; Volume 4004, pp. 339–358.
64. Canetti, R. Universally composable security: A new paradigm for cryptographic protocols. In Proceedings of the 42nd IEEE Symposium on Foundations of Computer Science, Washington, DC, USA, 14–17 October 2001; pp. 136–145.
65. Pedersen, T. Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing. In *Advances in Cryptology-CRYPTO 1991*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 1992; Volume 576, pp. 129–140.
66. pbc Library. 2012. Available online: https://crypto.stanford.edu/pbc/ (accessed on 15 March 2021).
67. gmp Library. 2014. Available online: https://gmplib.org (accessed on 15 March 2021).