Secure Multiparty Computation with Lazy Sharing

Shuaishuai Li[*1], Cong Zhang[†2(✉)], and Dongdai Lin[‡3,4]

[1]Zhongguancun Laboratory, Beijing, China
[2]Institute for Advanced Study, BNRist, Tsinghua University, Beijing, China
[3]Key Laboratory of Cyberspace Security Defense, Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
[4]School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China

Abstract

Secure multiparty computation (MPC) protocols enable n parties, each with private inputs, to compute a given function without leaking information beyond the outputs. One of the main approaches to designing efficient MPC protocols is to use secret sharing. In general, secret sharing based MPC contains three phases: input sharing, circuit evaluation, and output recovery. If the adversary corrupts at most t parties, the protocol typically uses (t, n) threshold secret sharing to share the inputs. In this work, we consider a weaker variant of threshold secret sharing called lazy threshold secret sharing (or simply lazy sharing) and show that

- Lazy sharing can serve as a viable alternative to threshold secret sharing in MPC without compromising security.
- Lazy sharing could be generated more efficiently than threshold secret sharing.

As a result, replacing threshold secret sharing with lazy sharing can lead to a more efficient input sharing phase. Moreover, we propose that the efficiency of the circuit evaluation phase can also be further improved. To support this claim, we apply lazy sharing to several state-of-the-art MPC protocols and analyze the efficiency gain in various settings. These protocols include the GMW protocol (Goldreich et al., STOC 1987), the AFLNO protocol (Araki et al., CCS 2016), and the SPDZ protocol (Damgård et al., CRYPTO 2012). By doing so, we analyze the efficiency gains in various settings and highlight the advantages of incorporating lazy sharing into MPC protocols.

1 Introduction

Secure multiparty computation (MPC) enables individuals to compute a function based on their private inputs without revealing any information beyond the final outputs. Since its inception by Yao [Yao82], MPC has been the subject of extensive research, resulting in significant advancements in both theoretical and practical aspects. Various seminal works [GMW87, CDvdG87, GHY87, BGW88, CCD88, RB89, Bea91b] have demonstrated the feasibility of MPC in diverse settings.

A key category of MPC protocols is generic MPC protocols, which facilitate the computation of arbitrary circuits comprising addition and multiplication gates. To date, two primary methodologies are employed for constructing efficient generic MPC protocols: the secret sharing approach [GMW87, BGW88, CCD88] and the garbled circuit approach [Yao86, BMR90]. The former necessitates interactions for each layer containing multiplication gates in the circuit and boasts a low communication cost, while the latter entails a constant number of rounds but incurs a higher communication cost. Consequently, the secret sharing approach is

[*]liss@zgclab.edu.cn
[†]zhangcong@mail.tsinghua.edu.cn
[‡]ddlin@iie.ac.cn

more suitable for low-latency networks like local area networks, whereas the garbled circuit approach excels in high-latency networks such as wide area networks, albeit with increased communication overhead.

In this study, we focus on MPC protocols based on the secret sharing approach, operating under the assumption that the adversary may corrupt at most t parties. In this context, a (t, n) threshold secret sharing scheme is employed to distribute the inputs[1]. Generally, secret sharing based MPC protocols include three phases. During the *Input Sharing* phase, the parties generate (t, n) threshold secret sharings for their private inputs. Subsequently, in the *Circuit Evaluation* phase, the parties compute the circuit gate-by-gate in a predetermined topological order. For each gate, the value in its output wire is computed as a (t, n) threshold secret sharing. Finally, in the *Output Recovery* phase, the parties recover the output sharings, distributing them to the designated parties who are supposed to obtain the outputs.

Currently, MPC primarily relies on three secret sharing schemes: additive secret sharing, Shamir secret sharing [Sha79], and replicated secret sharing (also known as CNF secret sharing) [ISN89, CDI05]. In practice, additive secret sharing is predominantly utilized for scenarios involving all-but-one corruptions, where the adversary may corrupt up to $n - 1$ parties. On the other hand, the other two secret sharing schemes are primarily employed in the context of an honest majority, where more than half of the parties are assumed to be honest[2]. Notably, replicated secret sharing is commonly employed in MPC protocols involving a small number of parties [AFL$^+$16, FLNW17].

Protocol Models. To date, MPC is mainly considered in two models: the standard MPC model [Yao82, GMW87] and the client-server model [CDI05, DI05, DI06]. In the standard model[3], inputs are provided by internal parties who engage in the entire computation, and outputs are received by designated internal parties. Conversely, in the client-server model, inputs originate from external parties (referred to as clients, who do not actively participate in the circuit evaluation phase), and outputs are transmitted to specific clients. A *trivial* advantage of the standard model over the client-server model is that during input sharing, only $n - 1$ shares need to be sent, resulting in communication savings in the input sharing phase by a factor of $(n-1)/n$. However, this advantage may appear relatively minor. Our interest is piqued by the following question:

What are the benefits of constructing MPC in the standard model beyond the trivial advantage?

1.1 Our Contribution

In this work, we demonstrate a significant disparity in efficiency between the standard model and the client-server model when developing MPC protocols. Specifically, we reveal that employing threshold secret sharing is excessive for designing MPC protocols in the standard model. We introduce a unique non-threshold secret sharing scheme termed "lazy threshold secret sharing" (or simply "lazy sharing"), which relaxes the privacy requirements of threshold secret sharing. Our findings indicate that

For MPC protocols in the standard model, we can replace threshold secret sharing with lazy sharing without compromising security.

To substantiate our claim, we consider the additive and replicated secret sharing schemes and demonstrate that their lazy variants can be generated more efficiently. This enables us to achieve a more efficient input sharing phase in MPC protocols built from these two secret sharing schemes. Specifically, for additive secret sharing, we observe that its lazy variant (referred to as lazy additive sharing) can be generated locally[4]. Furthermore, for replicated secret sharing, its lazy variant (termed lazy replicated sharing) can be generated with reduced communication by a factor of at least $(n-1)/(n-t-1)$ in the statistical setting. In the computational setting, using pseudorandom secret sharing [CDI05], replicated secret sharing can be generated at a very low amortized communication cost, albeit requiring an expensive setup phase involving

[1] A (t, n) threshold secret sharing scheme generates the sharing of an input in such a way that any $t + 1$ parties can reconstruct the input, while any t parties remain oblivious to the input.

[2] Recent works [GPS22, EGP$^+$23] have explored the use of (packed) Shamir secret sharing to design efficient MPC protocols in settings with a dishonest majority, where a constant fraction (not exceeding 0.5) of parties are honest.

[3] We follow the work of [DI05] to use the term "standard" to represent that the inputs come from the internal parties.

[4] We can humorously describe this situation that the parties are too "lazy" to interact to share the inputs, which is the reason why we use the term of "lazy sharing".

the distribution of $\binom{n}{t} - 1$ pseudorandom function (PRF) keys to the parties. Lazy replicated sharing, on the other hand, features a less costly setup phase, distributing at most $\binom{n-1}{t} - 1$ PRF keys. We refer to Section 6 for further details.

In addition to obtaining a more efficient input sharing phase, we argue that using lazy sharing can also improve the efficiency of the circuit evaluation phase. To see this, we apply lazy sharing to several state-of-the-art MPC protocols and analyze the efficiency gain.

Applying to the GMW Protocol [GMW87]. We first consider the passively secure GMW protocol, which is a fundamental and generic MPC protocol based on additive secret sharing. To facilitate a more comprehensive analysis of the efficiency gain, we consider the arithmetic version of GMW [IPS09], and we assume that the parties can invoke an oblivious linear evaluation[5] (OLE) functionality $\mathcal{F}_{\mathsf{ole}}$. Moreover, we will measure the communication cost by the number of calls to the OLE functionality and the number of field elements sent by the parties[6].

By replacing additive secret sharing with lazy additive sharing, we derive a variant of the GMW protocol called lazy GMW (LGMW). Compared to GMW, LGMW enjoys many better efficiency features, as we described below.

- Sharing the inputs is performed locally, that is to say, the communication cost of the input sharing phase is zero.
- The computation of a layer-λ multiplication gate (i.e., a multiplication gate in the λ-th layer of the circuit) requires no more than $\min\{4^{\lambda-1}, 4^{\lambda-1} + n - 2^{\lambda}, n^2 - n\}$ calls to the OLE functionality, while the GMW protocol always requires $n^2 - n$ calls to the OLE functionality. In particular, if λ is a constant, then the computation of a layer-λ multiplication gate only requires a constant number of calls to the OLE functionality.
- Similar to GMW, LGMW can also be extended to the preprocessing model. In particular, LGMW has a cheaper online phase than GMW for the computation of many circuits.

To provide a clearer demonstration of the efficiency improvements of LGMW over GMW, we consider the computation of various common circuits. Specifically, we observe that LGMW achieves an $O(n)$ improvement when computing the product circuit. Additionally, for shallow circuits like the inner product circuit, LGMW can even achieve an impressive $O(n^2)$ improvement. Furthermore, we examine a real-world application of GMW. We show that by substituting LGMW in place of GMW, the efficiency can be further improved by a factor of $O(n)$.

Applying to the AFLNO Protocol [AFL$^+$16]. The AFLNO protocol is a highly effective three-party computation protocol that relies on replicated secret sharing. It provides perfect security against any semi-honest adversary who corrupts up to one party. This protocol boasts a remarkably low bandwidth requirement, with a multiplication gate computation or output recovery (to all parties) necessitating each party to transmit only a *single* field element. However, the input sharing phase of this protocol seems not optimal, as it requires the input owner to send two field elements to each party. In the standard model, the communication cost of sharing an input amounts to *four* field elements. This is less efficient compared to MPC protocols based on Shamir secret sharing, such as the BGW protocol [BGW88], where the input owner only needs to send a *single* element to each party. In other words, Shamir secret sharing offers better efficiency for input sharing, while replicated secret sharing excels in efficiency for computing multiplication gates (MPC based on Shamir secret sharing requires at least *two* field elements per multiplication gate per party [DN07, GLO$^+$21]). Surprisingly, with lazy sharing, we can achieve the best of both worlds.

By replacing replicated secret sharing with lazy replicated sharing, we obtain a variant of the AFLNO protocol that we called the lazy AFLNO (LAFLNO) protocol. In comparison to AFLNO, LAFLNO enjoys a more efficient input sharing phase: if the input is held by a single party, then sharing this input only requires the input owner to send a *single* element to each party; if the input is held by two different parties,

[5] An OLE functionality receives two values a, b from some party P_i and a value x from another party P_j, and sends the value $ax + b$ to P_j.

[6] We remark that our results also hold for GMW over the binary field $\mathbb{F} = \mathbb{F}_2$, where the OLE functionality is replaced with an oblivious transfer (OT) functionality which receives two bits x_0, x_1 from some party P_i and a bit b from another party P_j, and sends the bit x_b to P_j.

then sharing this input requires no communication! We remark that if the input is held by all three parties, then this input is essentially a constant. Additionally, for some specific multiplication gates, LAFLNO may have lower communication than AFLNO, and we refer to Section 6 for more details. Given the effectiveness of AFLNO, we believe that LAFLNO achieves a nice improvement over AFLNO, especially for circuits with many inputs (e.g., $f((x_1, y_1), (x_2, y_2), (x_3, y_3)) = (x_1 + x_2 + x_3)(y_1 + y_2 + y_3)$).

Applying to the SPDZ Protocol [BDOZ11, DPSZ12]. Finally, in Appendix B, we further demonstrate how to apply our idea to the SPDZ protocol, one of the state-of-the-art generic MPC protocols with malicious security in the dishonest majority setting. Like GMW, SPDZ is also based on additive secret sharing, and moreover, it works in the preprocessing model. The key idea of SPDZ for achieving malicious security is to authenticate each sharing with an information-theoretic message authentication code (MAC) in a pairwise manner (the BDOZ-style MAC [BDOZ11]) or a global manner (the SPDZ-style MAC [DPSZ12]). We apply lazy additive sharing to SPDZ and derive a variant called lazy SPDZ (LSPDZ). LSPDZ works in the circuit-dependent preprocessing model, where the parties know the circuit topology in the offline phase. We will focus on the online cost and view the offline phase as a black-box that generates the required correlated randomness. LSPDZ mainly differs from SPDZ in the following three aspects.

- When sharing an input, LSPDZ requires a different precomputed random sharing, which however does not influence the online cost.
- When computing a multiplication gate, opening a sharing only requires the parties whose shares decide the shared value to send their shares, which saves communication.
- When recovering an output, an additive sharing of zero is used to randomize the output sharing.

We remark that LSPDZ has the same communication cost as SPDZ for the input sharing and output recovery phases. For the circuit evaluation phase, when computing a layer-λ multiplication gate, LSPDZ requires at most $2n + \mathsf{min}\{2^\lambda, 2n\} - 4 \leq 4n - 4$ field elements of communication, while SPDZ always requires $4n - 4$ field elements of communication. We can say that LSPDZ offers a cheaper online phase than SPDZ, except for an extreme scenario where the inputs of every multiplication gate depend on all parties. In particular, for small λ (i.e., $\lambda \ll \log n$), the communication cost of computing a layer-λ multiplication gate is almost halved.

Lastly, it is worth noting that beyond the enhancements made to the aforementioned protocols (GMW, AFLNO, and SPDZ), our findings underscore the advantages of internal parties carrying inputs. This systematic optimization is not tied to any particular technology. That means, for any MPC protocol, we can apply our idea as long as the protocol is designed in the standard model[7]. However, it is important to note that the potential for efficiency gains hinges on the particular use cases. As we will elaborate in the conclusion section, when our approach is applied to generic protocols that leverage the Shamir secret sharing, the anticipated efficiency improvements may not be realized.

1.2 Technical Overview

In the previous secret sharing based MPC framework, the parties generate (t, n) threshold secret sharings for the inputs. A (t, n) threshold secret sharing requires that any t shares are independent of the input, which guarantees that any t parties cannot know the input. However, we observe that this level of secrecy is overkill for MPC in the standard model, where the input owner is one of the computing parties. Specifically, when one of the t shares belongs to the input owner (this is always the case in the standard MPC model), we find that it is unnecessary to require that the t shares are independent of the input because the input owner already possesses the knowledge of the input. Building on this insight, we introduce lazy sharing, which relaxes the privacy requirements of threshold secret sharing and therefore could be generated more efficiently. Now, let us delve into the details of two specific threshold secret sharing schemes: additive secret sharing and replicated secret sharing.

[7]Our techniques do not work for the client-server model, where inputs are dispersed among the parties and remain undisclosed to any unauthorized subset of participants. For instance, in threshold cryptography, the secret key is distributed among the parties in such a manner that it is inaccessible to any unauthorized group.

Lazy Additive Sharing. Assume that P_i wants to share its input x using additive secret sharing, then it samples n random values $\{x_j\}_{j\in[n]}$ subject to $\sum_{j\in[n]} x_j = x$. It is obvious any $n-1$ of the n values $\{x_j\}_{j\in[n]}$ are independent of x. Our idea is that we do not require privacy for the share of the input owner P_i. Namely, we only require that the $n-1$ values $\{x_j\}_{j\in[n]\setminus\{i\}}$ are independent of x. At the same time, we also require that $\sum_{j\in[n]} x_j = x$ for guaranteeing correctness. Such a sharing is called lazy additive sharing. Obviously, an additive secret sharing is a lazy additive sharing. However, lazy additive sharing has a much simpler form: $x_i = x$, and for each $j \neq i$, $x_j = 0$. It is clear that this sharing is a lazy additive sharing, and it could be generated *locally*! As we will see, replacing additive secret sharing with lazy additive sharing in MPC can also improve the efficiency of the circuit evaluation phase.

Lazy Replicated Sharing. If a party P_i wants to generate a (t, n) replicated secret sharing for its input x using replicated secret sharing, it samples $\binom{n}{t}$ random values $\{X_T\}_{T\subseteq[n],|T|=t}$ conditioned on $x = \sum_{T\subseteq[n],|T|=t} X_T$ and the j-th share is $x_j = (X_T)_{|T|=t,j\notin T}$. Note that when $t = n/2$, each share consists of super-polynomial (in n) elements, which is why replicated secret sharing can only used for small n. Note that for any $T \subseteq [n]$ with $|T| = t$, the parties in T know nothing about x because they do not know X_T. Lazy replicated sharing relaxes the privacy requirement. Concretely, if some t shares contain the share of P_i, we will not require that these t shares are independent of x. Let us consider the special case of $(1, 3)$ replicated secret sharing, which is used in the AFLNO protocol [AFL+16].

To generate a $(1, 3)$ replicated secret sharing of an input x, the input owner (say, P_1) first samples three random values X_1, X_2, X_3 subject to $x = X_1 + X_2 + X_3$, and the i-th share is $x_i = (X_{i-1}, X_{i+1})$. It is obvious that any single x_i leaks nothing about x. Based on our idea, we do not require privacy for x_1, which allows us to generate the sharing more efficiently. Concretely, P_1 just chooses two random values X_2, X_3 subject to $x = X_2 + X_3$ and sets $X_1 = 0$. Then, P_1 sends X_3 to P_2 and X_2 to P_3. And the sharing of x is $(X_3, X_2), (X_1, X_3), (X_2, X_1) = (X_3, X_2), (0, X_3), (X_2, 0)$. It is easy to see that both the share of P_2 and P_3 are independent of x. Now, we consider the case that x is held by two different parties (say, P_1, P_2). In this setting, we only require that share of P_3 is independent of x. To do this, the parties just locally compute the sharing as $(X_3, X_2), (X_1, X_3), (X_2, X_1) = (x, 0), (0, x), (0, 0)$. Note that the correctness of the protocol will be guaranteed as long as we have $X_1 + X_2 + X_3 = x$.

The above idea can be extended to any number of parties, and we refer to Sections 3.2 and 6.5 for more details. In Section 6, we further discuss how the protocol will proceed when replacing replicated secret sharing with lazy replicated sharing. In particular, we will show that for some specific multiplication gates (where one input to the gate is held by two different parties), we can also improve the efficiency (see Section 6.2).

2 Preliminaries

Basic Notations. For any integer n, we denote $[n]$ the set $\{1, \cdots, n\}$. Let $\mathbb{F}$ be a finite field, and we always consider circuits over $\mathbb{F}$.

2.1 Threshold Secret Sharing

Definition 2.1 (Threshold Secret Sharing)**.** *Let $\mathbb{F}$ be a finite field and t, n, m be three integers with $t \in [n-1]$. A (t, n) threshold secret sharing scheme over $\mathbb{F}$ contains two efficient algorithms* (Share, Recover)*, where* Share *takes a field element $x \in \mathbb{F}$ as input and outputs $(x_1, \ldots, x_n) \in (\mathbb{F}^m)^n$. Moreover, we require the following properties.*

- ***Correctness.*** *For any $A \subseteq [n]$, if $|A| \geq t+1$, then we have*

$$\mathsf{Recover}(\{x_j\}_{j\in A}) = x.$$

- ***Privacy.*** *For any $U \subseteq [n]$, if $|U| \leq t$, then the distribution of $\{x_j\}_{j\in U}$ is independent of x.*

To date, there are three main threshold secret sharing schemes used in MPC protocols: additive secret sharing, Shamir secret sharing [Sha79], and replicated secret sharing [ISN89, CDI05]. In this work, we focus on additive and replicated secret sharing, and we describe these two schemes as follows.

Additive Secret Sharing. Additive secret sharing is an $(n-1,n)$ threshold secret sharing scheme. To share a value x, the dealer samples n random values $x_1,\ldots,x_n$ subject to $\sum_{j\in[n]} x_j = x$. The share of P_j is x_j. Note that to share a value, the dealer needs to send a single element to each party.

Replicated Secret Sharing. Replicated secret sharing is a (t,n) threshold secret sharing scheme for any $t \in [n-1]$, and it is mainly used for small n. To generate a (t,n) replicated secret sharing for a value x, the dealer first selects $\binom{n}{t}$ random values $\{X_T\}_{T\subseteq[n],|T|=t}$ subject to $x = \sum_{T\subseteq[n],|T|=t} X_T$. Then the share of party P_j is $x_j = (X_T)_{j\notin T}$. Note that for any $T \subseteq [n]$ with $|T| = t$, no party in T knows X_T, hence the parties in T cannot recover the shared value x. From the description of replicated secret sharing, we know that the dealer needs to send $\binom{n-1}{t}$ elements to each party.

We remark that additive secret sharing is actually a special case of replicated secret sharing (with $t = n-1$). However, these two schemes are often used with different corruption thresholds: additive secret sharing is used when the number of corrupted parties is at most $n-1$, while replicated secret sharing is used in scenarios where the number of corrupted parties is less than $n/2$ and n is small. For this reason, we describe these two schemes separately.

2.2 The Universal Composability Framework

Throughout this work, we prove the security of our protocols in the universal composability (UC) framework [Can01]. In this framework, a probabilistic polynomial time (PPT) adversary called the environment $\mathcal{Z}$ chooses inputs for the honest parties and gets their outputs. It also can corrupt the parties and take control over their actions. We say that a protocol Π realizes an ideal functionality $\mathcal{F}$ in the UC framework if there exists a PPT simulator $\mathcal{S}$ such that $\mathcal{Z}$ cannot distinguish the following real and ideal executions.

Real Execution. $\mathcal{Z}$ generates inputs for all parties. The parties execute the protocol Π and return the outputs to $\mathcal{Z}$. Additionally, $\mathcal{Z}$ can interact arbitrarily with the corrupted parties during the protocol execution.

Ideal Execution. $\mathcal{Z}$ generates inputs for all parties. The parties forward their inputs to the functionality $\mathcal{F}$ and return the outputs to $\mathcal{Z}$. Additionally, the simulator $\mathcal{S}$ interacts with $\mathcal{Z}$ and $\mathcal{F}$.

3 Lazy Sharing: Definitions and Constructions

In this section, we introduce a weaker variant of threshold secret sharing called lazy threshold secret sharing (or simply lazy sharing). Lazy sharing has an additional parameter $\mathcal{L} \subseteq [n]$, and we call $\mathcal{L}$ the lazy set. The formal definition of lazy sharing is in the following.

Definition 3.1 (Lazy Sharing)**.** *Let $\mathbb{F}$ be a finite field and t,n,m be three integers with $t \in [n-1]$. Let $\mathcal{L} \subseteq [n]$ be some set (which is called the lazy set). A $(t,n,\mathcal{L})$ lazy sharing scheme over $\mathbb{F}$ contains two efficient algorithms* $(\mathsf{Share},\mathsf{Recover})$*, where* Share *takes a field element $x \in \mathbb{F}$ as input and outputs $(x_1,\ldots,x_n) \in (\mathbb{F}^m)^n$. Moreover, we require the following properties.*

- ***Correctness.*** *For any $A \subseteq [n]$, if $|A| \geq t+1$, then we have*

$$\mathsf{Recover}(\{x_j\}_{j\in A}) = x.$$

- ***$\mathcal{L}$-Privacy.*** *For any $U \subseteq [n]\backslash\mathcal{L}$, if $|U| \leq t$, then the distribution of $\{x_j\}_{j\in U}$ is independent of x.*

It is easy to verify that for any $\mathcal{L}_1 \subseteq \mathcal{L}_2 \subseteq [n]$, a $(t,n,\mathcal{L}_1)$ lazy sharing is always a $(t,n,\mathcal{L}_2)$ lazy sharing. Moreover, a (t,n) threshold secret sharing is in fact a $(t,n,\varnothing)$ lazy sharing. This implies that a (t,n) threshold secret sharing is always a $(t,n,\mathcal{L})$ lazy sharing for any $\mathcal{L} \subseteq [n]$. We remark that a $(t,n,\mathcal{L})$ lazy sharing could be viewed as a non-threshold secret sharing, but we view it as a weaker variant of threshold secret sharing because we use it as a replacement of threshold secret sharing in secret sharing based MPC. Now we introduce two lazy sharing schemes called lazy additive sharing and lazy replicated sharing. These two schemes are the lazy variants of additive and replicated secret sharing.

3.1 Lazy Additive Sharing

In lazy additive sharing, we always assume that $t = n - 1$. Let $\mathcal{L}$ be the lazy set and $l = |\mathcal{L}|$, we describe a $(n-1, n, \mathcal{L})$ lazy additive sharing scheme as follows.

Share. To share a value x, the dealer samples l values $\{x_j\}_{j\in\mathcal{L}}$ subject to $\sum_{j\in\mathcal{L}} x_j = x$ and sets $x_j = 0$ for each $j \in [n]\backslash\mathcal{L}$. Note that we make no requirements on the distribution of $\{x_j\}_{j\in\mathcal{L}}$ except that their sum is x. The sharing is $(x_1, \ldots, x_n)$.

Recover. Using the shares $x_1, \ldots, x_n$, we can recover the shared value by computing $\sum_{j\in[n]} x_j$.

It is easy to verify that $(x_1, \ldots, x_n)$ is an $(n-1, n, \mathcal{L})$ lazy additive sharing of x. Moreover, we say that the shares of the parties in $\mathcal{L}$ are *valid* (their shares decide the shared value), and the shares of the parties in $[n]\backslash\mathcal{L}$ are *invalid*. We use $\langle x\rangle_{\mathcal{L}}$ to represent an $(n-1, n, \mathcal{L})$ lazy additive sharing of x. We say that a lazy additive sharing $\langle x\rangle_{\mathcal{L}} = (x_1, \ldots, x_n)$ is *uniform* if $\{x_i\}_{i\in\mathcal{L}}$ is a $(|\mathcal{L}|-1, |\mathcal{L}|)$ additive sharing of x (i.e., any $|\mathcal{L}|-1$ shares in $\{x_i\}_{i\in\mathcal{L}}$ are independent of x).

Example 3.2. *For a value $x \in \mathbb{F}$, $\langle x\rangle_{\{1\}} = (x, 0, \ldots, 0)$ is an $(n-1, n, \{1\})$ lazy additive sharing of x.*

We remark that in lazy additive sharing, the parties in the lazy set can cooperate to recover the shared value, rather than that all parties in the lazy set know the shared value.

3.2 Lazy Replicated Sharing

Let $\mathcal{L}$ be the lazy set and $l = |\mathcal{L}|$, we describe a $(t, n, \mathcal{L})$ lazy replicated sharing scheme as follows.

Share. To share a value x, the dealer first samples $\binom{n-l}{t}$ random values $\{X_T\}_{T\subseteq[n]\backslash\mathcal{L},|T|=t}$ subject to $x = \sum_{T\subseteq[n]\backslash\mathcal{L},|T|=t} X_T$ and then sets $X_T = 0$ for each $T \subseteq [n]$ with $|T| = t$ and $\mathcal{L} \cap T \neq \varnothing$. The j-th share is defined as $x_j = (X_T)_{|T|=t, j\notin T}$. The sharing is $(x_1, \ldots, x_n)$.

Recover. Using $t+1$ shares $\{x_j\}_{j\in A}$ where $|A| = t+1$, we can recover the shared value by computing $\sum_{T\subseteq[n]\backslash\mathcal{L},|T|=t} X_T$. Note that for each $T \subseteq [n]\backslash\mathcal{L}$ with $|T| = t$, at least one party in A knows X_T because at most t parties do not have X_T.

Note that for any $T \subseteq [n]\backslash\mathcal{L}$ with $|T| = t$, the parties in T do not know the value X_T, hence they know nothing about x. Therefore, $(x_1, \ldots, x_n)$ is a $(t, n, \mathcal{L})$ lazy replicated sharing of x. To generate a $(t, n, \mathcal{L})$ lazy replicated sharing, the dealer needs to send $\binom{n-l}{t}$ field elements to P_i for each $i \in \mathcal{L}$ and $\binom{n-l-1}{t}$ field elements to P_j for each $j \in [n]\backslash\mathcal{L}$.

Example 3.3. *For a value $x \in \mathbb{F}$, let X_2 be a random value and $X_3 = x - X_2$, then $(X_3, X_2), (0, X_3), (X_2, 0)$ is a $(1, 3, \{1\})$ lazy replicated sharing of x.*

4 Lazy GMW: GMW with Lazy Additive Sharing

In this section, we improve the efficiency of the semi-honest GMW protocol [GMW87] by replacing the additive secret sharing scheme with lazy additive sharing. For simplicity, we directly consider the arithmetic variant of GMW [IPS09] and assume the parties can invoke the oblivious linear evaluation (OLE) functionality $\mathcal{F}_{\mathsf{ole}}$. The formal definition of $\mathcal{F}_{\mathsf{ole}}$ can be found in Appendix A. We consider the GMW protocol in the $\mathcal{F}_{\mathsf{ole}}$-hybrid model and measure the cost with the number of calls to the functionality $\mathcal{F}_{\mathsf{ole}}$ and the number of field elements sent by the parties. In Appendix C.1, we present a formal description of the GMW protocol, which uses the following notations for the computed circuit f.

- Let M_{in} (resp. M_{out}) be the number of the inputs (resp. outputs) of f. Moreover, for each $j \in [n]$, we use M_{in}^j (resp. M_{out}^j) to denote the number of the inputs (resp. outputs) belong to the party P_j.
- Let C_{add} be the number of the addition gates in f and C_{mul} be the number of the multiplication gates in f. Let $C = C_{\mathsf{add}} + C_{\mathsf{mul}}$ be the size of f.
- Let D be the multiplicative depth[8] of f.

[8]The multiplicative depth of a circuit is the number of the layers containing multiplication gates.

- Let E be the depth of f.
- For each $\lambda \in [E]$, let $C^{\lambda}_{\mathsf{add}}$ be the number of the addition gates in the λ-th layer of f and $C^{\lambda}_{\mathsf{mul}}$ be the number of the multiplication gates in the λ-th layer of f. Let $C_{\lambda} = C^{\lambda}_{\mathsf{add}} + C^{\lambda}_{\mathsf{mul}}$ be the total number of the gates in the λ-th layer of f.

4.1 Overview of the Techniques

Our starting point is that if the input x is offered by some party P_i, then for the sharing of x, we do not require that the share of P_i is independent of x. In other words, instead of generating an $(n-1,n)$ additive secret sharing, we let P_i generate an $(n-1,n,\{i\})$ lazy additive sharing. By the discussion in Section 3.1, an $(n-1,n,\{i\})$ lazy additive sharing for any $i \in [n]$ could be generated locally.

After sharing the inputs, we now describe how to compute the circuit. For the computation of addition gates, we just let each party locally add its input shares as the output share. As a result, the sum of an $(n-1,n,\mathcal{L}_0)$ lazy additive sharing and an $(n-1,n,\mathcal{L}_1)$ lazy additive sharing will be an $(n-1,n,\mathcal{L}_0 \cup \mathcal{L}_1)$ lazy additive sharing. For example, the resulting sharing of adding two lazy additive sharings $\langle x\rangle_{\{1\}} = (x,0,\ldots,0)$ and $\langle y\rangle_{\{n\}} = (0,\ldots,0,y)$ is

$$\langle z\rangle_{\{1,n\}} = (x,0,\ldots,0,y).$$

It is clear that $\langle z\rangle_{\{1,n\}}$ is an $(n-1,n,\{1,n\})$ lazy additive sharing of $x+y$.

Now we consider the computation of multiplication gates. Let $\langle x\rangle_{\mathcal{L}_0} = (x_1,\ldots,x_n)$ and $\langle y\rangle_{\mathcal{L}_1} = (y_1,\ldots,y_n)$ be the input sharings. It is obvious that we can just do as in GMW: for each $(i,j) \in [n]^2$ with $i \neq j$, P_i and P_j additively share x_iy_j (see the formal description of GMW in Appendix C.1). This requires $n(n-1)$ calls to the OLE functionality. However, for each $(i,j) \in [n]^2 \backslash (\mathcal{L}_0 \times \mathcal{L}_1)$ we know $x_iy_j = 0$, which implies that P_i and P_j can additively share x_iy_j without any interaction: $x_iy_j = 0+0$. In other words, only when $(i,j) \in \mathcal{L}_0 \times \mathcal{L}_1$ and $i \neq j$, P_i and P_j need to interactively share x_iy_j, which only requires $|\mathcal{L}_0| \cdot |\mathcal{L}_1| - |\mathcal{L}_0 \cap \mathcal{L}_1|$ calls to the OLE functionality.

Now we use a simple example to show how our protocol will proceed. Assume that three parties P_1, P_2, P_3 want to compute the circuit $f(x_1,x_2,x_3) = x_1x_2 + x_3$, where each P_i holds x_i. They do the following steps.

1. *Input Sharing.* P_1, P_2, P_3 locally share their inputs: the sharing of x_1 is $(x_1,0,0)$, the sharing of x_2 is $(0,x_2,0)$, and the sharing of x_3 is $(0,0,x_3)$.
2. *Circuit Evaluation.* The parties compute the circuit as follows.
   (a) Computing $G_1(x_1,x_2) = x_1x_2$: P_1 samples a random value r_1, and then P_1 and P_2 invoke the OLE functionality, where P_1 playing the sender takes $x_1, -r_1$ as inputs and P_2 playing the receiver takes x_2 as input; P_2 receives $r_2 = x_1x_2 - r_1$ as output; the resulting sharing is $(r_1,r_2,0)$.
   (b) Computing $G_2(x_1x_2,x_3) = x_1x_2 + x_3$: P_3 just takes x_3 as its share and the final sharing is (r_1,r_2,x_3).

At the end of the above computation, the parties obtain an $(n-1,n,[3])$ lazy additive sharing of the output $x_1x_2+x_3$. The left problem is how to recover the output.

How to Securely Recover an Output. In GMW, since the output of each gate is computed as an additive secret sharing, the parties will get additive sharings of the outputs of the computed circuit. Note that a major feature of additive sharing is that any $n-1$ shares are independent of the shared value. Therefore, to recover an output, the parties just send their shares to the party who is supposed to obtain the output, which does not hurt the privacy of the inputs. However, this situation is different in our protocol. A lazy additive sharing cannot guarantee that any $n-1$ shares are independent of the shared value, which means that the shares may contain information about the inputs that should not be leaked to the adversary. Therefore, we need a more secure way to recover the outputs. Our solution is that the parties securely add up all the shares. To maintain the statistical security of our protocol in the $\mathcal{F}_{\mathsf{ole}}$-hybrid model, we let the parties use the GMW protocol to securely compute the sum of the shares. We note that a secure sum protocol can

be made more efficiently by pre-preparing a batch of pseudorandom additive sharings of zero. When the parties want to recover an output, they locally add an additive sharing of zero to the output sharing, and then they can send their shares to recover the outputs as in the GMW protocol.

4.2 GMW with Lazy Additive Sharing: LGMW

In this section, we describe lazy GMW (LGMW), which replaces additive secret sharing in GMW with lazy additive sharing. LGMW requires a secure sum protocol for recovering the outputs. We assume that the parties can invoke the sum functionality $\mathcal{F}_{\mathsf{sum}}^{\mathcal{L},r}$, and the formal definition of $\mathcal{F}_{\mathsf{sum}}^{\mathcal{L},r}$ can be found in Appendix A. In this work, we assume that $\mathcal{F}_{\mathsf{sum}}^{\mathcal{L},r}$ is realized using the GMW protocol, and the communication cost is $|\mathcal{L}|^2 - \tau$ field elements, where τ equals 1 if $r \in \mathcal{L}$ and 0 otherwise.

Now we describe the LGMW protocol in the $(\mathcal{F}_{\mathsf{ole}}, \mathcal{F}_{\mathsf{sum}}^{\mathcal{L},r})$-hybrid model.

Protocol 4.1 (The LGMW Protocol)**.** *Let $f : \mathbb{F}^{M_{\mathsf{in}}^1} \times \cdots \times \mathbb{F}^{M_{\mathsf{in}}^n} \to \mathbb{F}^{M_{\mathsf{out}}^1} \times \cdots \times \mathbb{F}^{M_{\mathsf{out}}^n}$ be the computed circuit. Each party P_i has M_{in}^i private inputs and M_{out}^i private outputs.*

1. ***Input Sharing.*** *For each input x belonging to P_i, the parties locally compute an $(n-1, n, \{i\})$ lazy additive sharing $\langle x \rangle_{\{i\}} = (x_1, \ldots, x_n)$, where for each $j \in [n]$, x_j is equal to x if $j = i$ and 0 otherwise.*
2. ***Circuit Evaluation.*** *For each gate G, assume that the input sharings are $\langle u \rangle_{\mathcal{L}_0} = (u_1, \ldots, u_n)$ and $\langle v \rangle_{\mathcal{L}_1} = (v_1, \ldots, v_n)$. Let $\mathcal{L} = \mathcal{L}_0 \cup \mathcal{L}_1$. The parties do the following steps.*
   - *If G is an addition gate, each party P_i computes $w_i = u_i + v_i$ as its share. The resulting sharing is $\langle w \rangle_{\mathcal{L}} = (w_1, \ldots, w_n)$.*
   - *If G is a multiplication gate, The parties do the following steps.*
     (a) *For each $i \in \mathcal{L}_0$ and $j \in \mathcal{L}_1 \backslash \{i\}$, P_i samples a random value $r_{i,j}$. P_i and P_j invoke $\mathcal{F}_{\mathsf{ole}}$ where P_i acting as the sender takes u_i and $-r_{i,j}$ as inputs and P_j acting as the receiver takes v_j as input. P_j receives $s_{j,i} = u_i v_j - r_{i,j}$ from $\mathcal{F}_{\mathsf{ole}}$.*
     (b) *For each $i \in \mathcal{L}_0 \backslash \mathcal{L}_1$, P_i computes $w_i = \sum_{j \in \mathcal{L}_1} r_{i,j}$. For each $i \in \mathcal{L}_0 \cap \mathcal{L}_1$, P_i computes $w_i = u_i v_i + \sum_{j \in \mathcal{L}_0 \backslash \{i\}} s_{i,j} + \sum_{j \in \mathcal{L}_1 \backslash \{i\}} r_{i,j}$. For each $i \in \mathcal{L}_1 \backslash \mathcal{L}_0$, P_i computes $w_i = \sum_{j \in \mathcal{L}_0} s_{i,j}$. For each $i \in [n] \backslash (\mathcal{L}_0 \cup \mathcal{L}_1)$, P_i sets $w_i = 0$.*
     (c) *The resulting sharing is $\langle w \rangle_{\mathcal{L}} = (w_1, \ldots, w_n)$.*
3. ***Output Recovery.*** *To recover a sharing $\langle y \rangle_{\mathcal{L}} = (y_1, \ldots, y_n)$ to some party P_i, the parties invoke the sum functionality $\mathcal{F}_{\mathsf{sum}}^{\mathcal{L},i}$ where each party P_j in $\mathcal{L}$ takes y_j as input.*

Theorem 4.1. *For any n-party circuit $f : \mathbb{F}^{M_{\mathsf{in}}^1} \times \cdots \times \mathbb{F}^{M_{\mathsf{in}}^n} \to \mathbb{F}^{M_{\mathsf{out}}^1} \times \cdots \times \mathbb{F}^{M_{\mathsf{out}}^n}$, Protocol 4.1 securely computes f against a passive adversary statically corrupting any number of parties in the $(\mathcal{F}_{\mathsf{ole}}, \mathcal{F}_{\mathsf{sum}}^{\mathcal{L},r})$-hybrid model.*

The proof is deferred to Appendix I.1.

4.3 Complexity Analysis of LGMW

In this section, we analyze the cost of the LGMW protocol, which depends on the sizes of the lazy sets of the lazy additive sharings generated in the protocol. By estimating the size of the lazy sets, we give an upper bound for the cost of computing a multiplication gate.

Before showing our results, we first introduce the concept of multiplication-immune (MI) depth, which is used in Corollary 1.

Definition 4.2 (Multiplication-Immune Depth)**.** *We define the multiplication-immune depth of a circuit f to be the maximal integer λ which satisfies that there exists at least one multiplication gate in the λ-th layer of f.*

For any circuit f, let $\mathsf{M}(f), \mathsf{D}(f)$ and $\mathsf{MI}(f)$ be its multiplicative depth, depth and MI depth, respectively, then we have $\mathsf{M}(f) \leq \mathsf{MI}(f) \leq \mathsf{D}(f)$.

Example 4.3. *For the circuit $f(x_1, x_2, x_3, x_4, x_5) = (x_1 + x_2)(x_3 + x_4) + x_5$, we have $\mathsf{M}(f) = 1, \mathsf{MI}(f) = 2, \mathsf{D}(f) = 3$.*

Cost of LGMW. In the LGMW protocol, for each multiplication gate G, let $\mathcal{L}_0$ and $\mathcal{L}_1$ be the two lazy sets of its input sharings, then the parties invoke the OLE functionality $|\mathcal{L}_0| \cdot |\mathcal{L}_1| - |\mathcal{L}_0 \cap \mathcal{L}_1|$ times to compute G. Therefore, if we let $G^{\circledast}_{\lambda,k}$ be the k-th multiplication gate in the λ-th layer of the circuit and $\mathcal{L}^{\circledast,0}_{\lambda,k}, \mathcal{L}^{\circledast,1}_{\lambda,k}$ be the two lazy sets of the input sharings of $G^{\circledast}_{\lambda,k}$, then the cost of the circuit evaluation phase will be

$$\sum_{\lambda \in [E]} \sum_{k \in [C^{\lambda}_{\mathsf{mul}}]} (|\mathcal{L}^{\circledast,0}_{\lambda,k}| \cdot |\mathcal{L}^{\circledast,1}_{\lambda,k}| - |\mathcal{L}^{\circledast,0}_{\lambda,k} \cap \mathcal{L}^{\circledast,1}_{\lambda,k}|)$$

calls to the OLE functionality. In the output recovery phase, for each $i \in [n]$ and $j \in [M^i_{\mathsf{out}}]$, let $y_{i,j}$ be the j-th output of P_i and $\mathcal{L}^{\mathsf{out}}_{i,j}$ be the corresponding lazy set. To recover $y_{i,j}$ to P_i, the parties need to invoke the sum functionality $\mathcal{F}^{\mathcal{L}^{\mathsf{out}}_{i,j},i}_{\mathsf{sum}}$ that is realized by the GMW protocol. Therefore, the communication cost of the output recovery phase is $\sum_{i \in [n]} \sum_{j \in [M^i_{\mathsf{out}}]} (|\mathcal{L}^{\mathsf{out}}_{i,j}|^2 - \tau_{i,j})$ field elements where $\tau_{i,j}$ equals 1 if $i \in \mathcal{L}^{\mathsf{out}}_{i,j}$ and 0 otherwise.

Estimating the Sizes of the Lazy Sets. For each circuit layer[9] $\lambda \in [E]$ and each $k \in [C_\lambda]$, let $G_{\lambda,k}$ be the k-th gate in the λ-th layer of f and $\mathcal{L}_{\lambda,k}$ be the lazy set of the output sharing of $G_{\lambda,k}$. We have the following claim.

Claim 4.4. *For any $\lambda \in [E], k \in [C_\lambda]$, it holds that $|\mathcal{L}_{\lambda,k}| \leq \mathsf{min}\{2^\lambda, n\}$.*

The proof is deferred to Appendix J.1.

An Upper Bound for the Cost of Computing a Layer-λ Multiplication Gate. For a layer-λ multiplication gate G with $\mathcal{L}_0, \mathcal{L}_1$ being the lazy sets of its input sharings, we know that the cost of computing G is $|\mathcal{L}_0| \cdot |\mathcal{L}_1| - |\mathcal{L}_0 \cap \mathcal{L}_1|$ calls to the OLE functionality. We give an upper bound for $|\mathcal{L}_0| \cdot |\mathcal{L}_1| - |\mathcal{L}_0 \cap \mathcal{L}_1|$ using the following Claim 4.5.

Claim 4.5. *If $\mathcal{L}_0, \mathcal{L}_1$ are the lazy sets of the input sharings of a layer-λ multiplication gate, then it holds that*

$$|\mathcal{L}_0| \cdot |\mathcal{L}_1| - |\mathcal{L}_0 \cap \mathcal{L}_1| \leq \mathsf{min}\{4^{\lambda-1}, 4^{\lambda-1} + n - 2^\lambda, n^2 - n\}.$$

The proof is deferred to Appendix J.2.

Claim 4.5 states that the cost of computing a layer-λ multiplication gate is no more than $\mathsf{min}\{4^{\lambda-1}, 4^{\lambda-1} + n - 2^\lambda, n^2 - n\}$ calls to the OLE functionality. Intuitively, a multiplication gate of small depth will require a few calls to the OLE functionality because λ is small. We in fact have the following simple corollary.

Corollary 1. *For any n-party circuit $f : \mathbb{F}^{M^1_{\mathsf{in}}} \times \cdots \times \mathbb{F}^{M^n_{\mathsf{in}}} \to \mathbb{F}^{M^1_{\mathsf{out}}} \times \cdots \times \mathbb{F}^{M^n_{\mathsf{out}}}$ with constant MI depth, Protocol 4.1 securely computes f with $O(C_{\mathsf{mul}})$ calls to the OLE functionality and $O(M_{\mathsf{out}} \cdot n^2)$ field elements of communication. Moreover, if f has constant depth, then the cost will be $O(C_{\mathsf{mul}})$ calls to the OLE functionality and $O(M_{\mathsf{out}})$ field elements of communication.*

The proof is deferred to Appendix J.3.

Remark. Circuits with constant depth (and bounded fan-in) capture an important complexity class called NC^0 in computational complexity theory. In particular, the works of [AIK04, AIK06] showed the existence of one-way function (OWF) and pseudorandom generator (PRG) in NC^0.

4.4 Comparison to the GMW Protocol

The efficiency of LGMW depends on the topology of the computed circuit. In particular, its improvements over GMW are more significant for computing small-depth multiplication gates (i.e., multiplication gates of small depth) than large-depth multiplication gates. In fact, for the circuits with only a few number of small-depth multiplication gates[10], the LGMW protocol may have limited improvements over GMW. However, for many natural and common circuits, we can show that the LGMW protocol achieves a nice

[9]Counting in both multiplications and additions.

[10]In other words, every party contributes input(s) to most multiplication gates in the circuit. For example, for the circuit $f((x_1, y_1), \ldots, (x_n, y_n)) = (x_1 + \cdots + x_n)(y_1 + \cdots + y_n)$, it is easy to verify that using LGMW to compute f requires the same number of calls to OLE functionality as using GMW.

improvement over GMW. More concretely, we compare LGMW with GMW for the computation of the sum, product, and inner product circuits. These circuits have important applications. For instance, in a contest, secure sum allows the total score to be computed without disclosing the score from each referee. Moreover, secure product allows us to determine whether a proposal has been unanimously passed without revealing which party rejected it. For inner product, it has important applications in template matching, which is very useful in image filtering [CG13], edge detection [JLW+22], feature extraction [CZS08], and other tasks in digital image processing. Template matching enabling authentication involves computing circuits of the form $f = \sum_{i\in[l]}(x_i - y_i)^2$, where $(x_1, \ldots, x_l)$ is the features of authorized users, and $(y_1, \ldots, y_l)$ is the features of the person being tested. When f is less than a certain threshold c, the tested person is considered to have passed the feature matching. In some scenarios where the target feature set $(x_1, \ldots, x_l)$ and the tested feature set $(y_1, \ldots, y_l)$ should be private, we need to compute f securely. Secure inner product allows us to do this. Note that $f = \sum_{i\in[n]}(x_i^2 + y_i^2 - 2x_iy_i) = \sum_{i\in[n]} x_i^2 + \sum_{i\in[n]} y_i^2 - 2\sum_{i\in[n]} x_iy_i$. By securely computing the inner product $z = \sum_{i\in[n]} x_iy_i$ to obtain an additive sharing (z_0, z_1) of z, the parties can locally compute $(\sum_{i\in[n]} x_i^2 - 2z_0, \sum_{i\in[n]} y_i^2 - 2z_1)$, which is exactly an additive sharing of f.

We defer the detailed complexity analysis to Appendix E.1 and summarize the results in Table 1. Our results show that compared to GMW, LGMW reduces the number of calls to the OLE functionality by a factor of $O(n)$ for the product circuit and $O(n^2)$ for the inner product circuit.

Circuit Type	GMW	LGMW
Sum	$n^2 - 1$ FEs	$n^2 - 1$ FEs
Product	$n^2 - 1$ FEs & $n(n-1)^2$ OLEs	$n^2 - 1$ FEs & $n(n-1)/2$ OLEs
Inner Product	$(2l+1)(n-1)$ FEs & $ln(n-1)$ OLEs	$n^2 - 1$ FEs & l OLEs

Table 1: The comparison of the efficiency between LGMW and GMW for computing the sum, product, and inner product circuits. Note that we abbreviate 'field elements' by 'FEs'.

So far, we have only discussed the computation of low-depth circuits. In Appendix F, we further discuss the improvements for large-depth circuits. Moreover, in Appendix G, we show the efficiency improvements in the real-world application of online marketplaces.

Remark. We want to note that our optimization for GMW allows us to find better tradeoffs between round complexity and communication cost. Suppose that n parties want to securely compute some function f. To compute f using generic MPC protocols, the parties need to represent f as a boolean or arithmetic circuit. There may exist two circuits C_1, C_2 that compute f, and moreover, C_1 and C_2 are of different circuit topologies (e.g., $(x+y)z$ and $xz+yz$ compute the same function). In particular, C_1 has fewer multiplication gates and a larger multiplicative depth than C_2. It seems that the protocol will suffer from a large number of rounds (using C_2) or a high communication cost (using C_1). However, with our techniques, since C_2 has a smaller depth, computing the multiplication gates in C_2 may have a similar communication cost as in C_1. In this way, the protocol will enjoy both a small number of rounds and a low communication cost (using C_2 to compute f).

5 Extending to the Preprocessing Model: Lazy GMW with Preprocessing

Assuming the parties have access to the OLE functionality, any arithmetic circuit consisting of addition and multiplication gates can be securely computed. However, the realization of the OLE functionality relies on expensive public-key primitives, which makes the resulting MPC protocols ineffective in practice. To handle this, we consider the preprocessing model, where the computation is separated into two phases: an offline phase and an online phase. The offline phase is independent of the inputs, and the online phase makes use of the inputs. By relocating the offline phase to a much earlier stage before the parties offer their inputs, the parties can execute a much more efficient online phase to complete the computation.

In this section, we discuss how to extend the LGMW protocol to the preprocessing model (in Appendix D, we review the GMW protocol in the preprocessing model). For LGMW, we consider two preprocessing settings: circuit-independent and circuit-dependent. Circuit-independent preprocessing means that the parties do not know the inputs and the computed circuit in the offline phase, while circuit-dependent preprocessing allows the parties to know the topology of the computed circuit in the offline phase.

Throughout this section, we use $\langle x \rangle$ to represent an additive sharing of x and $\langle x \rangle_{\mathcal{L}}$ to represent a lazy additive sharing of x with lazy set $\mathcal{L}$. Recall that we say that a lazy additive sharing $\langle x \rangle_{\mathcal{L}}$ is *uniform* if for any set $U \subseteq \mathcal{L}$ with $|U| \leq |\mathcal{L}| - 1$, the shares of the parties in U are independent of x.

5.1 LGMW with Circuit-Independent Preprocessing

In this section, we study LGMW in the circuit-independent preprocessing model, where the parties know nothing about the topology of the computed circuit in the offline phase.

Preprocessing Input Sharing. Note that the input sharing phase of LGMW could be executed locally, therefore we make no changes to this phase.

Preprocessing Circuit Evaluation. In LGMW, for a multiplication gate with input sharings $\langle x \rangle_{\mathcal{L}_0}$ and $\langle y \rangle_{\mathcal{L}_1}$, the parties need to generate a lazy additive sharing $\langle xy \rangle_{\mathcal{L}}$ where $\mathcal{L} = \mathcal{L}_0 \cup \mathcal{L}_1$. To preprocess the LGMW multiplications in this setting, we let the parties generate a Beaver triple $(\langle a \rangle, \langle b \rangle, \langle c \rangle)$, where $c = ab$. In the online phase, the parties compute the multiplication by two subphases. In the first subphase, the parties generate $(\langle a \rangle_{\mathcal{L}_0}, \langle b \rangle_{\mathcal{L}_1}, \langle c \rangle_{\mathcal{L}})$ from $(\langle a \rangle, \langle b \rangle, \langle c \rangle)$ such that $\langle a \rangle_{\mathcal{L}_0}, \langle b \rangle_{\mathcal{L}_1}, \langle c \rangle_{\mathcal{L}}$ are uniform lazy additive sharings[11].

- For each $\langle \eta \rangle_{\mathcal{L}'} \in \{\langle a \rangle_{\mathcal{L}_0}, \langle b \rangle_{\mathcal{L}_1}, \langle c \rangle_{\mathcal{L}}\}$, the parties compute $\langle \eta \rangle_{\mathcal{L}'}$ from $\langle \eta \rangle = (\eta_1, \ldots, \eta_n)$ as follows.
  1. Let k be the smallest index in $\mathcal{L}'$. For each $i \in [n] \backslash \mathcal{L}'$, P_i sends its share η_i to P_k and updates its own share to 0.
  2. P_k computes $\eta'_k = \eta_k + \sum_{i \in [n] \backslash \mathcal{L}'} \eta_i$ as its share in $\langle \eta \rangle_{\mathcal{L}'}$.
  3. For each $i \in \mathcal{L}' \backslash \{k\}$, P_i takes $\eta'_i = \eta_i$ as its share in $\langle \eta \rangle_{\mathcal{L}'}$.
  4. The resulting sharing is $\langle \eta \rangle_{\mathcal{L}'} = \{\eta'_i\}_{i \in \mathcal{L}'}$.

It is easy to verify that the generated $\langle a \rangle_{\mathcal{L}_0}, \langle b \rangle_{\mathcal{L}_1}, \langle c \rangle_{\mathcal{L}}$ are uniform lazy additive sharings. In the second subphase, the parties compute a lazy additive sharing $\langle xy \rangle_{\mathcal{L}}$ as follows.

1. The parties locally compute $\langle \alpha \rangle_{\mathcal{L}_0} = \langle x \rangle_{\mathcal{L}_0} - \langle a \rangle_{\mathcal{L}_0}$ and $\langle \beta \rangle_{\mathcal{L}_1} = \langle y \rangle_{\mathcal{L}_1} - \langle b \rangle_{\mathcal{L}_1}$.
2. The parties open α to the parties in $\mathcal{L}_1$ and β to the parties in $\mathcal{L}_0$ as follows.
   (a) If $\mathcal{L}_0 \cap \mathcal{L}_1 \neq \varnothing$, let k be the smallest index in $\mathcal{L}_0 \cap \mathcal{L}_1$, otherwise, let k be the smallest index in $\mathcal{L}_1$. Then, for each $i \in \mathcal{L}_0 \backslash \{k\}$, P_i sends its shares of α to P_k. Finally, P_k recovers and sends α to each other party in $\mathcal{L}_1$.
   (b) If $\mathcal{L}_0 \cap \mathcal{L}_1 \neq \varnothing$, let k' be the smallest index in $\mathcal{L}_0 \cap \mathcal{L}_1$, otherwise, let k' be the smallest index in $\mathcal{L}_0$. Then, for each $i \in \mathcal{L}_1 \backslash \{k'\}$, P_i sends its shares of β to $P_{k'}$. Finally, $P_{k'}$ recovers and sends β to each other party in $\mathcal{L}_0$.
3. The parties locally compute $\langle z \rangle_{\mathcal{L}} = \alpha\beta + \alpha\langle b \rangle_{\mathcal{L}_1} + \beta\langle a \rangle_{\mathcal{L}_0} + \langle c \rangle_{\mathcal{L}}$.

The correctness of the above construction is implied by the following equation:

$$xy = (x - a + a)(y - b + b) = (\alpha + a)(\beta + b) = \alpha\beta + \alpha b + \beta a + ab.$$

As for the security, we note that the adversary only obtains the values α and β. Firstly, α leaks nothing if the adversary does not corrupt all parties in $\mathcal{L}_0$ because a is a random value. And if the adversary corrupts all parties in $\mathcal{L}_0$, then it knows the value of x before the execution of the above construction. In other words,

[11] In fact, for small n, we can directly prepare $(\langle a \rangle_{\mathcal{L}_0}, \langle b \rangle_{\mathcal{L}_1}, \langle c \rangle_{\mathcal{L}})$ in the offline phase for all possible $\mathcal{L}_0, \mathcal{L}_1$. In this way, the first subphase does not need to be performed.

leaking α to the adversary does not hurt the privacy of x. By a similar discussion, leaking β to the adversary does not hurt the privacy of y.

Preprocessing Output Recovery. For the output recovery phase, to recover an output sharing $\langle x\rangle_{\mathcal{L}}$ to some party P_i, we need to securely add up the shares of the parties in $\mathcal{L}$ and let P_i obtain the output. Note that if $|\mathcal{L}| = 2$ and $i \in \mathcal{L}$, we just let the other party in $\mathcal{L}$ send its share to P_i. Now we consider the case of $|\mathcal{L}| \geq 3$ or $i \notin \mathcal{L}$. If using GMW to add up the shares, then the communication cost will be $|\mathcal{L}|^2 - \tau$ field elements, where τ equals 1 if $i \in \mathcal{L}$ and 0 otherwise. We consider the following way to recover the output.

1. In the offline phase, the parties prepare a batch of uniform lazy additive sharings of zero with lazy set $[n]$ (for small n, the parties can directly prepare uniform lazy additive sharings of zero for all possible lazy sets).

2. In the online phase, to recover an output sharing $\langle x\rangle_{\mathcal{L}}$ to P_i, the parties locally add a uniform lazy additive sharing of zero (with the lazy set being $[n]$ if n is large and $\mathcal{L}$ if n is small) to $\langle x\rangle_{\mathcal{L}}$, and then the resulting sharing will be a uniform lazy additive sharing of the output. Then, the parties can recover the output by letting the parties in the lazy set send their shares to P_i.

For large n, the online communication cost of the above approach is $n - 1$ field elements. For small n, the online communication cost of the above approach is $|\mathcal{L}| - \tau$ field elements, where τ equals 1 if $i \in \mathcal{L}$ and 0 otherwise.

Based on our analysis, for small n, we always use our approach to recover the output. For large n, if $|\mathcal{L}|^2 \leq n - 1$, we choose to use the GMW protocol to recover the output, and if $|\mathcal{L}|^2 > n - 1$, we choose to use our approach to recover the output.

5.2 LGMW with Circuit-Dependent Preprocessing

Circuit-dependent preprocessing means that the parties are given the computed circuit in the offline phase (but without knowing the inputs). Given the circuit, the parties can compute the lazy sets in the offline phase, which allows us to obtain a more efficient online phase.

Preprocessing Input Sharing. We make no changes to this phase.

Preprocessing Circuit Evaluation. In LGMW, for a multiplication gate with input sharings $\langle x\rangle_{\mathcal{L}_0}$ and $\langle y\rangle_{\mathcal{L}_1}$, the parties need to generate a lazy additive sharing $\langle xy\rangle_{\mathcal{L}}$ where $\mathcal{L} = \mathcal{L}_0 \cup \mathcal{L}_1$. To preprocess an LGMW multiplication in this setting, we let the parties generate a Beaver triple of form $(\langle a\rangle_{\mathcal{L}_0}, \langle b\rangle_{\mathcal{L}_1}, \langle c\rangle_{\mathcal{L}})$ where $c = ab$ and $\mathcal{L} = \mathcal{L}_0 \cup \mathcal{L}_1$. In the online phase, the parties just perform the second subphase of the circuit-independent preprocessing.

Preprocessing Output Recovery. In the output recovery phase, for each output sharing $\langle x\rangle_{\mathcal{L}}$ belonging to some party P_i, we let the parties prepare a uniform lazy additive sharing $\langle 0\rangle_{\mathcal{L}}$ in the offline phase, and in the online phase, the parties locally add $\langle 0\rangle_{\mathcal{L}}$ to $\langle x\rangle_{\mathcal{L}}$. Then, the parties can recover the output by letting the parties in $\mathcal{L}$ send their final shares to P_i. It is clear that the online communication cost for recovering an output is $|\mathcal{L}| - \tau$ field elements, where τ equals 1 if $i \in \mathcal{L}$ and 0 otherwise.

5.3 Complexity Analysis and Comparison

We give a simple analysis of the communication cost of computing multiplication gates for the GMW and LGMW protocols (see Appendix D for the description of the GMW protocol in the preprocessing model).

Online Cost of GMW with Preprocessing. For the computation of a multiplication gate, the parties need to open two additive sharings in the online phase. Note that the communication cost of opening an additive sharing is $2(n-1)$ field elements. Therefore, the online cost of GMW for computing a multiplication gate is $4(n-1)$ field elements.

Online Cost of LGMW with Circuit-Independent Preprocessing. For the computation of a multiplication gate with input sharings $\langle x\rangle_{\mathcal{L}_0}, \langle y\rangle_{\mathcal{L}_1}$, the parties first generate $(\langle a\rangle_{\mathcal{L}_0}, \langle b\rangle_{\mathcal{L}_1}, \langle c\rangle_{\mathcal{L}})$ from $(\langle a\rangle, \langle b\rangle, \langle c\rangle)$ in the first subphase, which results in

$$(n - |\mathcal{L}_0|) + (n - |\mathcal{L}_1|) + (n - |\mathcal{L}|) = 3n - (|\mathcal{L}_0| + |\mathcal{L}_1| + |\mathcal{L}|)$$

field elements of communication. Then, in the second subphase, the parties need to open a lazy additive sharing with lazy set $\mathcal{L}_0$ to the parties in $\mathcal{L}_1$ and a lazy additive sharing with lazy set $\mathcal{L}_1$ to the parties in $\mathcal{L}_0$, which results in communication $2(|\mathcal{L}_0| + |\mathcal{L}_1| - \tau)$ field elements, where τ equals 1 if $\mathcal{L}_0 \cap \mathcal{L}_1 = \varnothing$ and 2 otherwise. We remark that if $|\mathcal{L}_0| + |\mathcal{L}_1| > n$, then it must hold that $\mathcal{L}_0 \cap \mathcal{L}_1 \neq \varnothing$. Therefore, the total online communication cost is

$$2(|\mathcal{L}_0| + |\mathcal{L}_1| - \tau) + 3n - (|\mathcal{L}_0| + |\mathcal{L}_1| + |\mathcal{L}|) = 3n - 2\tau + |\mathcal{L}_0 \cap \mathcal{L}_1|$$

field elements. Now we show that LGMW always has a cheaper online phase than GMW. Concretely, we have the following claim.

Claim 5.1. *For any two non-empty sets $\mathcal{L}_0, \mathcal{L}_1 \subseteq [n]$, we have*

$$3n - 2\tau + |\mathcal{L}_0 \cap \mathcal{L}_1| \leq 4(n-1).$$

The proof is deferred to Appendix J.4.

Now, let us give an upper bound for the communication cost of computing a layer-λ gate in the circuit-independent online phase. By Claim 4.4, we know that if $\mathcal{L}_0, \mathcal{L}_1$ are the lazy sets of the input sharings of a layer-λ multiplication gate, then we have

$$3n - 2\tau + |\mathcal{L}_0 \cap \mathcal{L}_1| \leq 3n - 2 + \min\{2^{\lambda-1}, n\}.$$

Online Cost of LGMW with Circuit-Dependent Preprocessing. For the computation of a multiplication gate with input sharings $\langle x \rangle_{\mathcal{L}_0}, \langle y \rangle_{\mathcal{L}_1}$, the parties only need to perform the second subphase of LGMW in the circuit-independent preprocessing model, which requires $2(|\mathcal{L}_0| + |\mathcal{L}_1| - \tau)$ field elements of communication, where τ equals 1 if $\mathcal{L}_0 \cap \mathcal{L}_1 = \varnothing$ and 2 otherwise. It is obvious that the online cost of circuit-dependent preprocessing is no more than that of circuit-independent preprocessing. By Claim 4.4, if $\mathcal{L}_0, \mathcal{L}_1$ are the lazy sets of the input sharings of a layer-λ multiplication gate, then we have

$$2(|\mathcal{L}_0| + |\mathcal{L}_1| - \tau) \leq \min\{2^{\lambda+1}, 4n\} - 2.$$

Comparison for Specific Circuits. We give a comparison of the online communication cost between GMW and LGMW in the preprocessing model for computing the sum, product, and inner product circuits. We defer the detailed analysis to Appendix E.2, and the results are summarized in Table 2. Our results show that with circuit-dependent preprocessing, we can reduce the online communication by a factor of $O(n/\log_2 n)$ for computing the product circuit and $O(n)$ (assume $l = O(n)$) for computing the inner product circuit.

Circuit Type	GMW (CI)	LGMW (CI)	LGMW (CD)
Sum	$n-1$	$n-1$	$n-1$
Product	$(n-1)(4n-3)$	$(n-1)(3n-1)$	$n(2\log_2 n - 1) + 1$
Inner Product	$(4l+1)(n-1)$	$(3l+1)n - 2l - 1$	$2l + n - 1$

Table 2: The online communication costs (in the number of field elements) of GMW and LGMW in the circuit-independent preprocessing model and LGMW in the circuit-dependent preprocessing model for computing the sum, product, and inner product circuits. Note that 'CI' means 'circuit-independent' and 'CD' means 'circuit-dependent'.

6 Lazy AFLNO: AFLNO with Lazy Replicated Sharing

We have described how to use lazy sharing to improve the GMW protocol. GMW is based on additive secret sharing. In this section, we consider MPC protocols based on replicated secret sharing. In particular, we consider the AFLNO protocol [AFL+16] (the formal description of AFLNO can be found in Appendix C.2), which is an effective three-party computation protocol and is secure against any semi-honest adversary corrupting up to one party. This protocol has a very low communication cost, and computing a multiplication

gate or recovering an output (all parties obtain the output) only requires each party to send a *single* field element (addition gates can be computed locally). However, in AFLNO, sharing an input requires the owner to send *four* field elements, which seems not optimal.

In this section, we introduce lazy AFLNO (LAFLNO), which is based on lazy replicated sharing. LAFLNO improves AFLNO in two aspects: the inputs could be shared with less communication; some specific multiplication gates could be computed with less communication. We assume that the parties in the protocol are P_1, P_2, P_3.

6.1 More Efficient Input Sharing

In AFLNO, to share an input x (assume that P_1 is the owner), P_1 samples two random values x_1, x_2 and computes $x_3 = x - x_1 - x_2$. Then it sends (x_1, x_3) to P_2 and (x_2, x_1) to P_3. The sharing of x is $(x_3, x_2), (x_1, x_3), (x_2, x_1)$. It is clear that a single party cannot recover x, and the communication cost of sharing an input is *four* field elements. We show how to improve this in two cases: the input is held by a single party, and the input is held by two different parties.

The Input is Held by A Single Party. In AFLNO, a sharing of the input satisfies that any single share is independent of the input. In LAFLNO, we throw away this requirement for the share belonging to the input owner. If P_1 wants to share its input x, then we only require that both the share of P_2 and P_3 are independent of x. Using the scheme described in Section 3.2, P_1 samples two random values x_2, x_3 subject to $x_2 + x_3 = x$ and sets $x_1 = 0$. Then P_1 sends x_3 to P_2 and x_2 to P_3, and the sharing of x is $(x_3, x_2), (x_1, x_3), (x_2, x_1) = (x_3, x_2), (0, x_3), (x_2, 0)$, which is a $(1, 3, \{1\})$ lazy replicated sharing. It is clear that the communication cost of sharing an input is *two* field elements, which is halved compared to the AFLNO protocol.

The Input is Held by Two Different Parties. We can further reduce the communication if the input is held by two different parties[12]. In this case, we want to generate a $(1, 3, \mathcal{L})$ lazy replicated sharing for the input, where $|\mathcal{L}| = 2$. Assume that the input x is held by P_1 and P_2, then using the scheme described in Section 3.2, we can set $x_1 = 0, x_2 = 0, x_3 = x$ and let $(x_3, x_2), (x_1, x_3), (x_2, x_1) = (x, 0), (0, x), (0, 0)$ be the sharing. It is obvious that this sharing is a $(1, 3, \{1, 2\})$ lazy replicated sharing of x. Moreover, it can be generated locally, i.e., we reduce the communication to zero. Finally, we remark that if the input is known to all three parties, then this input is in fact a public constant.

6.2 Reducing Communication for Specific Multiplication Gates

After sharing the inputs using lazy replicated sharing, the parties can execute the circuit evaluation phase as in AFLNO to compute the circuit. However, we show that for specific multiplication gates, we can further reduce the communication. Recall that in AFLNO, given an additive sharing of zero (which can be generated very efficiently), computing a multiplication gate requires each party to send a single field element, i.e., the communication cost is *three* field elements. We show that for a multiplication gate, if one input of this gate is held by two different parties, then we can reduce the communication cost to *two* field elements.

Assume that the parties want to compute a multiplication gate with inputs x, y. In particular, we assume that x is held by two parties (say, P_1, P_2). In this setting, the sharing of x is $(x, 0), (0, x), (0, 0)$. Let $(y_3, y_2), (y_1, y_3), (y_2, y_1)$ be the sharing of y. We assume that P_1, P_2 have two common random values s, r (which are not known to P_3), then the parties can compute a sharing of xy as follows.

1. P_1 computes $z_3 = s$ and $z_2 = xy_2 + xy_3 + r$, and sends z_2 to P_3.
2. P_2 computes $z_1 = xy_1 - r - s$ and $z_3 = s$, and sends z_1 to P_3.
3. The final sharing is $(z_3, z_2), (z_1, z_3), (z_2, z_1)$.

The correctness follows from that $z_1 + z_2 + z_3 = xy$. Moreover, given that s, r are two random values, we know that z_1, z_2 are also two random values. Therefore, P_3 knows nothing about xy from z_1, z_2. Finally, we remark that we only require two field elements of communication.

[12]The situation of multiple parties (but not all parties) holding the same input has been considered in the context of private information retrieval [CGKS95, CKGS98], where multiple non-colluding servers take the same database as input.

6.3 AFLNO with Lazy Replicated Sharing: LAFLNO

In this section, we present the full description of LAFLNO. Similar to AFLNO, our protocol also assumes that the parties can access the functionality $\mathcal{F}_{\mathsf{zero}}$ which generates additive zero-sharings (see Appendix C.2 for the definition of $\mathcal{F}_{\mathsf{zero}}$). Moreover, we define the functionality $\mathcal{F}_{\mathsf{coin}}^{i,j}$ which generates common random values for the two parties P_i, P_j. We defer the formal definition of $\mathcal{F}_{\mathsf{coin}}^{i,j}$ to Appendix A.

We remark that $\mathcal{F}_{\mathsf{coin}}^{i,j}$ can be realized without any interaction beyond a short initial setup. Let $\mathsf{Prf} : \{0,1\}^\kappa \times \{0,1\}^\kappa \to \mathbb{F}$ be a pseudorandom function (PRF). In the setup phase, each pair of parties P_i, P_j agree on a random key $k_{i,j}$. For each $\mathsf{id} \in \{0,1\}^\kappa$, P_i and P_j can locally agree on a pseudorandom value r_{id} by computing $r_{\mathsf{id}} = \mathsf{Prf}(k_{i,j}, \mathsf{id})$.

Now we can describe the LAFLNO protocol in the $(\mathcal{F}_{\mathsf{coin}}^{i,j}, \mathcal{F}_{\mathsf{zero}})$-hybrid model.

Protocol 6.1 (The LAFLNO Protocol)**.** *The parties in the protocol are P_1, P_2, P_3, and they offer the inputs of a public circuit f.*

1. ***Input Sharing.*** *For each input x, its sharing is generated as follows.*
   - *If x is held by a single party (say, P_1), then P_1 samples two random values x_2, x_3 subject to $x_2 + x_3 = x$. Next, P_1 sends x_3 to P_2 and x_2 to P_3. The sharing of x is $(x_3, x_2), (0, x_3), (x_2, 0)$.*
   - *If x is held by two different parties (say, P_1, P_2), then the parties locally compute the sharing as $(x, 0), (0, x), (0, 0)$.*
2. ***Circuit Evaluation.*** *The parties compute the circuit in a gate-by-gate manner. For each gate G with input sharings $(x_3, x_2), (x_1, x_3), (x_2, x_1)$ (the corresponding value is x) and $(y_3, y_2), (y_1, y_3), (y_2, y_1)$ (the corresponding value is y). If G is an addition gate, then the parties just locally compute $(x_3 + y_3, x_2 + y_2), (x_1 + y_1, x_3 + y_3), (x_2 + y_2, x_1 + y_1)$, which is a sharing of $x + y$. If G is a multiplication gate, then the parties do the following steps.*
   - *If x or y is an input held by two different parties (e.g., x is held by P_1, P_2, and the sharing of x is $(x, 0), (0, x), (0, 0)$), then the parties compute the gate as follows.*
     (a) *P_1 and P_2 ask for two common random values s, r from the functionality $\mathcal{F}_{\mathsf{coin}}^{1,2}$.*
     (b) *P_1 computes $z_3 = s$ and $z_2 = xy_2 + xy_3 + r$, and sends z_2 to P_3.*
     (c) *P_2 computes $z_1 = xy_1 - r - s$ and $z_3 = s$, and sends z_1 to P_3.*
     (d) *The final sharing is $(z_3, z_2), (z_1, z_3), (z_2, z_1)$.*
   - *Otherwise, the parties compute the gate as follows.*
     (a) *The parties ask for an additive sharing (r_1, r_2, r_3) of zero from the functionality $\mathcal{F}_{\mathsf{zero}}$.*
     (b) *Each party P_i computes $z_{i-1} = x_{i-1}y_{i-1} + x_{i-1}y_{i+1} + x_{i+1}y_{i-1} + r_i$ and sends z_{i-1} to P_{i+1}. Note that (z_1, z_2, z_3) is an additive sharing of xy.*
     (c) *The final sharing is $(z_3, z_2), (z_1, z_3), (z_2, z_1)$.*
3. ***Output Recovery.*** *For a sharing $(z_3, z_2), (z_1, z_3), (z_2, z_1)$, to recover it, each P_i sends z_{i+1} to P_{i+1}, and then each P_i computes $z = z_1 + z_2 + z_3$.*

Theorem 6.1. *In the $(\mathcal{F}_{\mathsf{coin}}^{i,j}, \mathcal{F}_{\mathsf{zero}})$-hybrid model, Protocol 6.1 securely computes the circuit f against any static, passive adversary corrupting up to one party.*

The proof is deferred to Appendix I.2.

6.4 Comparison to AFLNO

We give a comparison of the communication cost between AFLNO and LAFLNO for computing the following sum, product, inner product, and chain circuits (with P_1 obtaining the output).

- The sum circuit is $x_1 + x_2 + x_3$ with each P_i holding x_i.
- The product circuit is $x_1x_2x_3$ with each P_i holding x_i.

- The inner product circuit is $\sum_{i\in[l]} x_i y_i$. For each $i \in [l]$, x_i and y_i are privately held by two different parties (otherwise, this party just takes $x_i y_i$ as input).
- The chain circuit is $((y_0 + x_1)y_1 + x_2)y_2$, where for $i < 3$, each party P_i has a private input x_i, and P_3 has 3 private inputs y_0, y_1, y_2.

For the sum circuit, there are 3 inputs and no multiplications, hence the communication cost of AFLNO is $4 \cdot 3 + 1 = 13$ elements, and the communication cost of LAFLNO is $2 \cdot 3 + 1 = 7$ elements. For the product circuit, there are 3 inputs and 2 multiplications, hence the communication cost of AFLNO is $4 \cdot 3 + 3 \cdot 2 + 1 = 19$ elements, and the communication cost of LAFLNO is $2 \cdot 3 + 3 \cdot 2 + 1 = 13$ elements. For the inner product circuit, there are $2l$ inputs and l multiplications, hence the communication cost of AFLNO is $4 \cdot 2l + 3 \cdot l + 1 = 11l + 1$ elements and the communication cost of LAFLNO is $2 \cdot 2l + 3 \cdot l + 1 = 7l + 1$ elements. For the chain circuit, there are 5 inputs and 2 multiplications, hence the communication cost of AFLNO is $4 \cdot 5 + 3 \cdot 2 + 1 = 27$ elements, and the communication cost of LAFLNO is $2 \cdot 5 + 3 \cdot 2 + 1 = 17$ elements.

The results are summarized in Table 3.

Circuit Type	AFLNO	LAFLNO
Sum	13	7
Product	19	13
Inner Product	$11l + 1$	$7l + 1$
Chain	27	17

Table 3: The communication costs (in the number of elements) of AFLNO and LAFLNO for computing the sum, product, and inner product circuits.

6.5 Extending to Any Number of Parties

Our results can be generalized to MPC based on replicated secret sharing with n parties. Concretely, to share an input x that is held by some set $\mathcal{L}$ of parties, instead of generating a (t, n) replicated sharing, we let the parties compute a $(t, n, \mathcal{L})$ lazy replicated sharing. As we discussed in Section 3.2, the dealer needs to send $\binom{n-l}{t}$ elements to each party in $\mathcal{L}$ and $\binom{n-l-1}{t}$ elements to each party in $[n]\backslash\mathcal{L}$. Note that all parties in $\mathcal{L}$ know x, hence one of them can serve as the dealer, and the total communication cost is

$$(l-1) \cdot \binom{n-l}{t} + (n-l) \cdot \binom{n-l-1}{t} = (n-t-1) \cdot \binom{n-l}{t}$$

elements. We can further reduce the communication based on that all parties in $\mathcal{L}$ know x. In the lazy sharing scheme in Section 3.2, the dealer samples $\binom{n-l}{t}$ random values with their sum being x and then sends all the $\binom{n-l}{t}$ values to each party in $\mathcal{L}$. However, due to that all the parties in $\mathcal{L}$ know x, the dealer only needs to send $\binom{n-l}{t} - 1$ values to each of them, and then they can infer the last value. In this way, we reduce the communication by $l - 1$ field elements (one party in $\mathcal{L}$ serves as the dealer). As a result, the communication cost of sharing x will be $(n-t-1) \cdot \binom{n-l}{t} - (l-1)$ field elements. This matches our results for $t = 1, n = 3$. We note that if using (t, n) replicated sharing, the communication cost of sharing an input will be $(n-1) \cdot \binom{n-l}{t}$ field elements.

By our result, if $|\mathcal{L}| = l \geq n - t$ (i.e., the input is known to at least $n - t$ parties), then the parties can share the input locally! In fact, if $l = n - t$, we have $(n-t-1) \cdot \binom{n-l}{t} - (l-1) = 0$, and if $l > n - t$, we just choose a size-$(n-t)$ subset of $\mathcal{L}$ as the new lazy set to locally generate the sharing.

Going to the Computational Setting. In Appendix H, we show that using pseudorandom secret sharing [CDI05], we can generate replicated sharing and lazy replicated sharing more efficiently (with computational security). To generate replicated sharings for a batch of inputs that are known to l parties, the parties need to generate $\binom{n}{t} - 1$ PRF keys in the setup phase, and the amortized communication cost of sharing an input is $n-t-l$ field elements if $l < n-t$ and 0 otherwise. On the other hand, to generate lazy replicated sharings for a batch of inputs that are known to l parties for $l < n - t$, the number of the PRF keys generated in the

setup phase is reduced to $\binom{n-l}{t} - 1$ and the amortized communication cost remains $n - t - l$ field elements. Recall that if $l \geq n - t$, lazy replicated sharing could be generated locally *without any setup*.

Summary. We summarize the results in Table 4. Our results show that compared to replicated secret sharing, lazy replicated sharing could be generated with reduced communication cost (by a factor of at least $(n-1)/(n-t-1)$) in the statistical setting or reduced number of PRF keys (by a factor of at least $n/(n-t)$) in the computational setting.

		Key Number	Security Level	Amortized Communication
$l < n - t$	Replicated Sharing	0	Statistical	$(n-1) \cdot \binom{n-1}{t}$
	Lazy Replicated Sharing	0	Statistical	$(n-t-1) \cdot \binom{n-l}{t} - (l-1)$
	Replicated Sharing	$\binom{n}{t} - 1$	Computational	$n - t - l$
	Lazy Replicated Sharing	$\binom{n-l}{t} - 1$	Computational	$n - t - l$
$l \geq n - t$	Replicated Sharing	0	Statistical	$(n-1) \cdot \binom{n-1}{t}$
	Lazy Replicated Sharing	0	Statistical	0
	Replicated Sharing	$\binom{n}{t} - 1$	Computational	0

Table 4: The comparison for generating replicated sharing and lazy replicated sharing, where the input is known to l parties. 'Key Number' represents the required number of PRF keys, and 'Amortized Communication' represents the required number of field elements for sharing an input.

7 Extending to the Malicious Setting

We have shown how to improve two passively secure MPC protocols using lazy sharing. We also extend our techniques to the malicious setting, where the parties may not follow the protocol specification. We will focus on the SPDZ protocol [BDOZ11, DPSZ12] and show to apply lazy sharing to it.

At a high level, the general idea of applying lazy sharing to the malicious scenarios is to consider malicious variants of secret sharing. In the SPDZ protocol, each (additive) sharing is authenticated using message authentication code (MAC). To apply lazy sharing to SPDZ, we instead use MAC to authenticate *lazy additive sharing*. The resulting authenticated lazy additive sharing scheme is much like the original authenticated additive sharing scheme in SPDZ, as we use the same MAC as in SPDZ. Due to the space limits, we refer the readers to Appendix B for the formal description of the lazy variant of SPDZ called lazy SPDZ (LSPDZ). Now, we present the efficiency comparison between SPDZ and LSPDZ for computing the sum, product, and inner product.

Compared to SPDZ. We focus on the online costs of SPDZ and LSPDZ. We defer the detailed complexity analysis to Appendix E.3, and the summary can be found in Table 5. Our results show that for the product circuit, we reduce the online communication by about $1.6\times$, and for the inner product circuit, we can reduce the online communication by about $1.5\times$.

8 Implementations

To estimate the concrete efficiency of our protocol, we ran experiments measuring the cost of our LGMW and LAFLNO protocols and compared them with the GMW and AFLNO protocols, respectively. The experiments were run on an Intel Core i9-12900K processor (Alder Lake architecture with a base clock speed of 3.2 GHz and a max turbo frequency of 5.2 GHz) running Ubuntu 22.04. We simulate the network connection using the Linux command tc, and the protocols were run in the LAN setting with 10Gbps bandwidth and 0.05ms RTT latency.

Circuit Type	SPDZ	LSPDZ
Sum	$(n+2)(n-1)$	$(n+2)(n-1)$
Product	$(5n-2)(n-1)$	$(3n-2)(n-1)+n\log_2 n$
Inner Product	$(6l+2)(n-1)$	$(4l+2)(n-1)$

Table 5: The online communication costs (in the number of field elements) of SPDZ and LSPDZ for computing the sum, product, and inner product circuits.

8.1 Experimental Parameters

The input length we used for our experiments is 64 bits.

GMW and LGMW. When implementing GMW and LGMW, we consider $6, 8, 10$ parties and the following three circuits, including two small-depth circuits (product and inner product circuits) and a large-depth circuit (chain circuit). We do not test the sum circuit since GMW and LGMW proceed the same for the sum circuit.

- Product. Each party P_i has a private input x_i, and the circuit is $f = \prod_{i\in[n]} x_i$. The multiplicative depth of this circuit is $\lceil \log n \rceil$.
- Inner Product. We consider the inner product circuit with $l = n/2$. Concretely, each party P_i has a private input x_i, and the circuit is $f = \sum_{i\in[n/2]} x_{2i-1}x_{2i}$. The multiplicative depth of this circuit is 1.
- The Chain Circuit. For $i < n$, each party P_i has a private input x_i, and P_n has n private inputs $y_0, y_1, \ldots, y_{n-1}$. The circuit is $f = (\cdots((y_0+x_1)y_1+x_2)y_2+\cdots+x_{n-1})y_{n-1}$. The multiplicative depth of this circuit is $n-1$.

AFLNO and LAFLNO. Both AFLNO and LAFLNO are three-party protocols. We run the protocol for the following four circuits.

- Sum. Each party P_i has a private input x_i, and the circuit is $f = x_1 + x_2 + x_3$. The multiplicative depth of this circuit is 0.
- Product. Each party P_i has a private input x_i, and the circuit is $f = x_1x_2x_3$. The multiplicative depth of this circuit is 2.
- Inner Product. We consider the inner product circuit with $l = 3$. Concretely, each party P_i has two private inputs x_i, x_{i+3}, and the circuit is $f = x_1x_2 + x_3x_4 + x_5x_6$. The multiplicative depth of this circuit is 1.
- The Chain Circuit. For $i < 3$, each party P_i has a private input x_i, and P_3 has 3 private inputs y_0, y_1, y_2. The circuit is $f = ((y_0 + x_1)y_1 + x_2)y_2$. The multiplicative depth of this circuit is 2.

8.2 Experimental Details

The details of our protocol implementations are as follows.

- **GMW and LGMW.** We implement the OLE functionality using the protocol in [Gil99], which involves black-box calls to OT. We implement the DDH-based OT proposed in [BM89], which makes use of hash functions and secret-key encryption. For the underlying DDH group, we use the elliptic curve in the SM2 digital signature algorithm [Adm10a]. For the hash functions and secret-key encryption, we use the SM3 hash function [Adm10b] and AES [oSN01], respectively.
- **AFLNO and LAFLNO.** For the tested circuits, the AFLNO and LAFLNO protocols differ only in the input sharing phase. When realizing the protocols, we assume the parties have generated zero sharings required for computing the multiplication gates.

We use the following four metrics to measure the performance of our protocol.

- **Computational cost.** We measure the computational cost by doing all the computation in the protocol using a single thread. In particular, the communication is not considered. When some values need to be sent, we just store them in memory and execute the computational tasks of each party in sequence. This way, we can eliminate the time consumption introduced by interactions. This allows us to evaluate the improvement of our protocols in pure computation.
- **Running time.** We measure the running time in the LAN setting, i.e., 10Gbps bandwidth and 0.05ms RTT latency. Unlike computational cost, runtime takes into account the communication time of the parties, and each party in each round can perform their computing tasks in parallel. The running time reflects the comprehensive performance of the protocols.
- **Communication cost.** We measure the communication cost of the protocol by the amount of data sent by the parties in the protocol.
- **Throughput.** We measure the throughput using the ratio of the number of gates in the circuit (i.e., the circuit size) to runtime. That is, throughput measures the number of gates a protocol can process per (milli)second.

8.3 Performance and Analysis

We compare LGMW and GMW [GMW87], as well as LAFLNO and AFLNO [AFL+16], respectively. We consider the number of participants $n \in \{6, 8, 10\}$ for LGMW and GMW. The detailed comparison between LGMW and GMW is presented in Table 6, while the detailed comparison between LAFLNO and AFLNO is presented in Table 7.

GMW vs. LGMW. Our LGMW protocol outperforms the GMW protocol in terms of computational and communication overhead, which is consistent with the theoretical analysis in Table 1. For example, when there are 10 parties, the computation and communication of our LGMW improves the GMW for computing the inner product circuit with $l = 5$ by roughly $90\times$. In addition to small-depth circuits (product and inner product circuits), our improvements are still effective for large-depth circuits. For example, for the depth-n chain circuit, our LGMW protocol achieves a $9 - 18\times$ improvement in computational cost.

When considering the running time of the protocol, the improvement factor of our protocol decreases because all parties involved in the protocol will perform computations simultaneously in each round, thus amortizing computational costs. Moreover, our (computational and communication) improvement is asymmetric for some of the tested circuits (e.g., the chain circuit), meaning that the cost savings for each party are different. This may further reduce the improvement factor of our protocol, as the running time depends on the party with highest cost in each round. Nevertheless, our experiments show that LGMW still achieves significant improvements compared to GMW, and the reason is that each party (including the party with highest cost) performs fewer OLEs in each round, resulting in a faster runtime. For example, the running time of our LGMW for computing the inner product circuit with 10 parties is 12.7 milliseconds, while GMW requires 456.8 milliseconds, achieving roughly a $36\times$ improvement. Since throughput represents the number of gates (i.e., circuit size) that can be processed per second, the throughput improvement of our LGMW protocol is consistent with the runtime improvement.

AFLNO vs. LAFLNO. Since our LAFLNO protocol only improves the AFLNO protocol in the input sharing phase, while our LGMW protocol improves both the input sharing and circuit evaluation phases, our LAFLNO protocol does not improve the AFLNO protocol as much as our LGMW protocol. Nevertheless, our LAFLNO still achieves notable improvements compared to AFLNO. Our LAFLNO protocol achieves a $1.19-1.31\times$ improvement in computational costs compared to the AFLNO protocol, depending on different computed circuits. This improvement comes from the fact that our LAFLNO protocol performs simpler computations during the input sharing phase. Note that our LAFLNO only needs to share each input into two parts, while the AFLNO protocol needs to share each input into three parts. The running time of our LAFLNO achieves a $1.09 - 1.12\times$ improvement over AFLNO. The reason for the shrinking in improvement ratio is the same as the LGMW protocol. For communication costs, our LAFLNO protocol achieves a $1.46 - 1.86\times$ improvement compared to the AFLNO protocol, which is consistent with our theoretical

analysis in Table 3. The throughput improvement of our LAFLNO protocol is also consistent with the runtime improvement.

Circuit	n	Computation (s)			Runtime (ms)			Communication (MB)			Throughput (gates/s)		
		GMW	LGMW	Imp	GMW	LGMW	Imp	GMW	LGMW	Imp	GMW	LGMW	Imp
Product	6	3.805	0.376	10.12×	248.9	68.7	3.62×	1.465	0.147	9.97×	2.01	7.28	3.62×
	8	9.940	0.683	14.55×	487.6	71.2	6.85×	3.829	0.274	13.97×	1.44	9.83	6.83×
	10	20.729	1.110	18.67×	813.5	125.8	6.47×	7.911	0.440	17.98×	1.11	7.15	6.44×
Inner Product	6	2.318	0.073	31.75×	151.6	11.6	13.07×	0.879	0.029	30.31×	3.30	43.10	13.06×
	8	5.695	0.096	59.32×	279.4	12.1	23.09×	2.188	0.039	56.10×	2.51	57.85	23.05×
	10	11.407	0.125	91.26×	456.8	12.7	35.97×	4.395	0.050	87.90×	1.97	70.87	35.97×
Chain	6	4.425	0.490	9.03×	276.3	96.7	2.86×	1.465	0.147	9.97×	3.62	10.34	2.86×
	8	11.614	0.846	13.73×	556.8	165.9	3.36×	3.829	0.274	13.97×	2.51	8.44	3.36×
	10	24.315	1.357	17.92×	953.6	245.6	3.88×	7.912	0.440	17.98×	1.89	7.33	3.88×

Table 6: Comparision of GMW and LGMW for computing the product, inner product, and chain circuits. "Imp" is an abbreviation for "improvement". The throughput is computed as the number of gates processed by the protocol per second.

Circuit	Computation (ms)			Runtime (ms)			Communication (byte)			Throughput (gates/ms)		
	AFLNO	LAFLNO	Imp	AFLNO	LAFLNO	Imp	AFLNO	LAFLNO	Imp	AFLNO	LAFLNO	Imp
Sum	0.569	0.434	1.31×	0.174	0.156	1.12×	104	56	1.86×	11.49	12.82	1.12×
Product	0.752	0.633	1.19×	0.276	0.253	1.09×	152	104	1.46×	7.25	7.91	1.09×
Inner Product	1.567	1.256	1.25×	0.345	0.313	1.10×	272	176	1.55×	14.49	15.97	1.10×
Chain	1.418	1.104	1.28×	0.385	0.344	1.12×	216	136	1.59×	10.39	11.63	1.12×

Table 7: Comparision of AFLNO and LAFLNO for computing the sum, product, inner product, and chain circuits. "Imp" is an abbreviation for "improvement". The throughput is computed as the number of gates processed by the protocol per second.

9 Conclusion and Discussion

In this work, we introduce lazy sharing and show how to use it to improve MPC protocols that are based on additive or replicated secret sharing. As a result, we can improve the efficiency of the input sharing phase of protocols. To further analyze the efficiency gain when using lazy sharing, we consider several state-of-the-art MPC protocols. Our results show that by using lazy sharing, we may also improve the efficiency of the circuit evaluation phase of the protocols. An interesting point is that it seems difficult to use lazy sharing to improve MPC protocols that are based on Shamir secret sharing, as we discussed below.

Discussions on Shamir Secret Sharing. If we want to apply the idea of lazy sharing to Shamir secret sharing, then we can define lazy Shamir sharing which throws away the privacy requirement for the share of the input owner (say, P_1). One way to generate a lazy Shamir sharing is that P_1 generates a $(t, n-1)$ (instead of (t, n)) Shamir sharing for the parties $P_2, \ldots, P_n$ and then all the parties locally generate their shares. More previsely, P_1 generates a random degree-t polynomial $f(x)$ subject to $f(0)$ equals the shared input x. Then P_1 sends $f(i)$ to P_i for each $i = 2, \ldots, n$. It is obvious that $(f(2), \ldots, f(n))$ is a $(t, n-1)$ Shamir sharing. To locally generate a (t, n) lazy Shamir sharing, we let P_1 takes $f(a)$ as it share for some

$a \notin \{2, \ldots, n\}$, and the final sharing is $(f(a), f(2), \ldots, f(n))$. In particular, we do not require privacy for P_1, implying that P_1 can take $f(0) = x$ as its share. It is obvious that $(f(0), f(2), \ldots, f(n))$ is not a Shamir sharing (as P_1 itself can recover the input), but it is a lazy Shamir sharing.

An important point to consider is that even with the use of lazy Shamir sharing, we are unable to reduce communication overhead, as the input owner is still required to transmit an element to every other party. In other words, despite the relaxed privacy property that lazy Shamir sharing offers, we have not been successful in devising a more efficient method for generating it.

Acknowledgments

We are grateful to Anyu Wang and Zhiyuan Qiu for their insightful discussions and valuable contributions to the implementation of this paper. We also thank the anonymous reviewers of ACM CCS 2024 for their useful comments, whose constructive feedback significantly enhanced the quality and depth of our research.

References

[Adm10a] State Cryptography Administration. Public key cryptographic algorithm sm2 based on elliptic curves. Technical report, Beijing: State Cryptography Administration, 2010.

[Adm10b] State Cryptography Administration. Sm3 cryptographic hash algorithm. Technical report, Beijing: State Cryptography Administration, 2010.

[AFL+16] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. High-throughput semi-honest secure three-party computation with an honest majority. In CCS 2016, pages 805–817, 2016.

[AIK04] Benny Applebaum, Yuval Ishai, and Eyal Kushilevitz. Cryptography in nc^0. In FOCS 2004, pages 166–175, 2004.

[AIK06] Benny Applebaum, Yuval Ishai, and Eyal Kushilevitz. On pseudorandom generators with linear stretch in nc^0. In APPROX 2006 and RANDOM 2006, pages 260–271, 2006.

[BCS19] Carsten Baum, Daniele Cozzo, and Nigel P. Smart. Using topgear in overdrive: A more efficient zkpok for SPDZ. In SAC 2019, pages 274–302, 2019.

[BDOZ11] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. In EUROCRYPT 2011, pages 169–188, 2011.

[Bea91a] Donald Beaver. Efficient multiparty protocols using circuit randomization. In CRYPTO 1991, pages 420–432, 1991.

[Bea91b] Donald Beaver. Secure multiparty protocols and zero-knowledge proof systems tolerating a faulty minority. J. Cryptol., pages 75–122, 1991.

[BGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In STOC 1988, pages 1–10, 1988.

[BM89] Mihir Bellare and Silvio Micali. Non-interactive oblivious transfer and applications. In Gilles Brassard, editor, Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings, volume 435 of Lecture Notes in Computer Science, pages 547–557. Springer, 1989.

[BMR90] Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols (extended abstract). In STOC 1990, pages 503–513, 1990.

[BNO19] Aner Ben-Efraim, Michael Nielsen, and Eran Omri. Turbospeedz: Double your online spdz! improving SPDZ using function dependent preprocessing. In ACNS 2019, pages 530–549, 2019.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In FOCS 2001, pages 136–145, 2001.

[CCD88] David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols (extended abstract). In STOC 1988, pages 11–19, 1988.

[CDI05] Ronald Cramer, Ivan Damgård, and Yuval Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In TCC 2005, pages 342–362, 2005.

[CDvdG87] David Chaum, Ivan Damgård, and Jeroen van de Graaf. Multiparty computations ensuring privacy of each party's input and correctness of the result. In CRYPTO 1987, pages 87–119, 1987.

[CG13] Ruchika Chandel and Gaurav Gupta. Image filtering algorithms and techniques: A review. International Journal of Advanced Research in Computer Science and Software Engineering, 3(10), 2013.

[CGKS95] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. In 36th Annual Symposium on Foundations of Computer Science, Milwaukee, Wisconsin, USA, 23-25 October 1995, pages 41–50. IEEE Computer Society, 1995.

[CHK+12] Seung Geol Choi, Kyung-Wook Hwang, Jonathan Katz, Tal Malkin, and Dan Rubenstein. Secure multi-party computation of boolean circuits with applications to privacy in on-line marketplaces. In CT-RSA 2012, pages 416–432, 2012.

[CKGS98] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private information retrieval. J. ACM, 45(6):965–981, 1998.

[CKR+20] Hao Chen, Miran Kim, Ilya P. Razenshteyn, Dragos Rotaru, Yongsoo Song, and Sameer Wagh. Maliciously secure matrix multiplication with applications to private deep learning. In ASIACRYPT 2020, pages 31–59, 2020.

[CZS08] Feng-ying Cui, Li-jun Zou, and Bei Song. Edge feature extraction based on digital image processing techniques. In 2008 IEEE International Conference on Automation and Logistics, pages 2320–2324. IEEE, 2008.

[DI05] Ivan Damgård and Yuval Ishai. Constant-round multiparty computation using a black-box pseudorandom generator. In CRYPTO 2005, pages 378–394, 2005.

[DI06] Ivan Damgård and Yuval Ishai. Scalable secure multiparty computation. In CRYPTO 2006, pages 501–520, 2006.

[DKL+13] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In ESORICS 2013, pages 1–18, 2013.

[DN02] Ivan Damgård and Jesper Buus Nielsen. Perfect hiding and perfect binding universally composable commitment schemes with constant expansion factor. In CRYPTO 2002, pages 581–596, 2002.

[DN07] Ivan Damgård and Jesper Buus Nielsen. Scalable and unconditionally secure multiparty computation. In CRYPTO 2007, pages 572–590, 2007.

[DPSZ12] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In CRYPTO 2012, pages 643–662, 2012.

[EGP+23] Daniel Escudero, Vipul Goyal, Antigoni Polychroniadou, Yifan Song, and Chenkai Weng. Superpack: Dishonest majority MPC with constant online communication. In EUROCRYPT 2023, pages 220–250, 2023.

[FLNW17] Jun Furukawa, Yehuda Lindell, Ariel Nof, and Or Weinstein. High-throughput secure three-party computation for malicious adversaries and an honest majority. In EUROCRYPT 2017, pages 225–255, 2017.

[GHY87] Zvi Galil, Stuart Haber, and Moti Yung. Cryptographic computation: Secure faut-tolerant protocols and the public-key model. In CRYPTO 1987, pages 135–155, 1987.

[Gil99] Niv Gilboa. Two party rsa key generation. In Annual International Cryptology Conference, pages 116–129. Springer, 1999.

[GLO+21] Vipul Goyal, Hanjun Li, Rafail Ostrovsky, Antigoni Polychroniadou, and Yifan Song. ATLAS: efficient and scalable MPC in the honest majority setting. In CRYPTO 2021, pages 244–274. Springer, 2021.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In STOC 1987, pages 218–229, 1987.

[GPS22] Vipul Goyal, Antigoni Polychroniadou, and Yifan Song. Sharing transformation and dishonest majority MPC with packed secret sharing. In CRYPTO 2022, pages 3–32, 2022.

[IPS09] Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Secure arithmetic computation with no honest majority. In TCC 2009, pages 294–314, 2009.

[ISN89] Mitsuru Ito, Akira Saito, and Takao Nishizeki. Secret sharing scheme realizing general access structure. Electronics and Communications in Japan (Part III: Fundamental Electronic Science), pages 56–64, 1989.

[JLW+22] Junfeng Jing, Shenjuan Liu, Gang Wang, Weichuan Zhang, and Changming Sun. Recent advances on image edge detection: A comprehensive review. Neurocomputing, 503:259–271, 2022.

[KPR18] Marcel Keller, Valerio Pastro, and Dragos Rotaru. Overdrive: Making SPDZ great again. In EUROCRYPT 2018, pages 158–189, 2018.

[Lin11] Yehuda Lindell. Highly-efficient universally-composable commitments based on the DDH assumption. In EUROCRYPT 2011, pages 446–466, 2011.

[LOS14] Enrique Larraia, Emmanuela Orsini, and Nigel P. Smart. Dishonest majority multi-party computation for binary circuits. In CRYPTO 2014, pages 495–512, 2014.

[oSN01] National Institute of Standards and Technology (NIST). Advanced encryption standard (aes), 2001-11-26 00:11:00 2001.

[RB89] Tal Rabin and Michael Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority (extended abstract). In STOC 1989, pages 73–85, 1989.

[Sha79] Adi Shamir. How to share a secret. Commun. ACM, pages 612–613, 1979.

[Yao82] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In FOCS 1982, pages 160–164, 1982.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In FOCS 1986, pages 162–167, 1986.

A Some Important Functionalities

Functionality A.1 ($\mathcal{F}_{\mathsf{ole}}$). *The functionality $\mathcal{F}_{\mathsf{ole}}$ receives two values a, b from a party called the sender and a value x from another party called the receiver. The functionality returns the value $ax + b$ to the receiver.*

Functionality A.2 ($\mathcal{F}_{\mathsf{sum}}^{\mathcal{L},r}$). *For each $i \in \mathcal{L}$, P_i sends its input x_i to the functionality. The functionality returns the value $x = \sum_{i\in\mathcal{L}} x_i$ to P_r.*

Functionality A.3 ($\mathcal{F}_{\mathsf{coin}}^{i,j}$). *The functionality $\mathcal{F}_{\mathsf{coin}}^{i,j}$ samples a random value $r \in \mathbb{F}$ and sends r to P_i, P_j.*

Functionality A.4 ($\mathcal{F}_{\mathsf{prep}}$). *Let f be the circuit to be computed. Let $\mathcal{C}$ be the set of corrupted parties and $\mathcal{H} = [n]\backslash\mathcal{C}$. On input (Start, f) from the honest parties and adversary, the functionality performs the following steps.*

- ***Initialization.*** *Receive a share Δ_j from the adversary for each $j \in \mathcal{C}$ and sample a random Δ_j for each $j \in \mathcal{H}$. Set $\Delta = \sum_{j\in[n]} \Delta_j$.*
- ***Input.*** *For each input gate with P_i being the input owner.*
  1. *If P_i is honest, choose a random value r. Otherwise, receive the value r from the adversary. Send r to P_i.*
  2. *Set $m_r = r\Delta$. Wait for a value m_r^j for each $j \in \mathcal{C}$ from the adversary and sample $|\mathcal{H}|$ random values $\{m_r^j\}_{j\in\mathcal{H}}$ subject to $m_r = \sum_{j\in[n]} m_r^j$. Return m_r^j to P_j for each $j \in [n]$.*
- ***Triple.*** *For each multiplication gate with $\mathcal{L}_0, \mathcal{L}_1$ being the lazy sets of its two input sharings.*
  1. *For each $i \in \mathcal{L}_0 \cap \mathcal{H}$, sample a random value a_i and sends a_i to P_i. For each $i \in \mathcal{L}_1 \cap \mathcal{H}$, sample a random value b_i and sends b_i to P_i.*
  2. *Wait for $\{a_j\}_{j\in\mathcal{L}_0\cap\mathcal{C}}$, $\{b_j\}_{j\in\mathcal{L}_1\cap\mathcal{C}}$ and $\{c_j\}_{j\in\mathcal{L}\cap\mathcal{C}}$ from the adversary.*
  3. *Compute $a = \sum_{j\in\mathcal{L}_0} a_j$, $b = \sum_{j\in\mathcal{L}_1} b_j$ and $c = ab$. Sample random values $\{c_j\}_{j\in\mathcal{L}\cap\mathcal{H}}$ subject to $c = \sum_{j\in\mathcal{L}} c_j$. For each $i \in \mathcal{L} \cap \mathcal{H}$, sends c_i to P_i.*
  4. *Set $m_a = a\Delta, m_b = b\Delta, m_c = c\Delta$. Wait for three values m_a^j, m_b^j, m_c^j for each $j \in \mathcal{C}$ from the adversary and sample $3|\mathcal{H}|$ random values $\{m_a^j, m_b^j, m_c^j\}_{j\in\mathcal{H}}$ subject to $m_a = \sum_{j\in[n]} m_a^j, m_b = \sum_{j\in[n]} m_b^j$ and $m_c = \sum_{j\in[n]} m_c^j$. Return (m_a^j, m_b^j, m_c^j) to P_j for each $j \in [n]$.*
- ***Zero.*** *For each output sharing with lazy set $\mathcal{L}$, wait for a value r_j for each $j \in \mathcal{L} \cap \mathcal{C}$ from the adversary and sample $|\mathcal{L} \cap \mathcal{H}|$ random values $\{r_j\}_{j\in\mathcal{L}\cap\mathcal{H}}$ subject to $0 = \sum_{j\in[n]} r_j$. Return r_j to P_j for each $j \in \mathcal{L}$.*

Functionality A.5 ($\mathcal{F}_{\mathsf{mpc}}$). *Let f be the computed circuit.*

- ***Initialization.*** *The functionality receives input (Start, f) from all parties.*
- ***Input.*** *On input $(\mathsf{Input}, P_i, vid, x)$ from party P_i and input (Input, P_i) from the other parties, where vid is a fresh identifier, the functionality stores (vid, x).*
- ***Addition.*** *On input $(\mathsf{Add}, vid_0, vid_1, vid)$ from all parties, the functionality retrieves (if vid_0, vid_1 are present in memory and vid is not) the values $(vid_0, x), (vid_1, y)$ and stores $(vid, x + y)$.*
- ***Multiplication.*** *On input $(\mathsf{Multiply}, vid_0, vid_1, vid)$ from all parties, the functionality retrieves (if vid_0, vid_1 are present in memory and vid is not) the values $(vid_0, x), (vid_1, y)$ and stores (vid, xy).*
- ***Output.*** *On input (Output, vid) from all honest parties, the functionality retrieves (vid, z) and sends z to the adversary. The functionality waits for a message Abort or $\mathsf{Success}$ from the adversary: if receiving Abort then it aborts, otherwise, it sends z to all parties.*

B Lazy SPDZ: SPDZ with Authenticated Lazy Additive Sharing

In this section, we extend our techniques to the malicious setting, where the adversary can deviate from the protocol arbitrarily. More concretely, we consider the SPDZ protocol [BDOZ11, DPSZ12], which is a concretely efficient MPC protocol with malicious security. SPDZ with its subsequent optimizations [KPR18, BCS19, CKR+20] has made SPDZ one of the most efficient MPC protocols for computing arithmetic circuits. SPDZ works in the preprocessing model and its description can be found in Appendix C.3.

Now we proceed to introduce lazy SPDZ (LSPDZ), which is derived by replacing additive sharing in SPDZ with lazy additive sharing. Note that SPDZ can be viewed as a maliciously secure version of GMW (in the preprocessing model), and to achieve malicious security, SPDZ authenticates each sharing in GMW with an information-theoretic MAC. Similarly, LSPDZ can be viewed as a maliciously secure version of LGMW. Before describing the LSPDZ protocol, we first show how to authenticate lazy additive sharing in Section B.1. The full description of LSPDZ can be found in Section B.2.

B.1 Authenticating Lazy Additive Sharing

In this section, we show how to use BDOZ-style or SPDZ-style MAC to authenticate lazy additive sharing[13].

Lazy BDOZ-Style MAC. For a lazy additive sharing $\langle x\rangle_{\mathcal{L}_0} = (x_1, \ldots, x_n)$, we only authenticate the valid shares $\{x_j\}_{j\in\mathcal{L}_0}$. To do this, each party P_j generates a global key Δ_j. Then, for each $i \in \mathcal{L}_0$ and $j \in [n]\backslash\{i\}$, P_j holds a local key $k^x_{j,i}$, and P_i has the MAC

$$m^x_{i,j} = x_i\Delta_j + k^x_{j,i}.$$

The share of P_j is denoted as

$$[\![x]\!]_{\mathcal{L}_0,j} = (\Gamma^x_j, \Delta_j, \{k^x_{j,i}\}_{i\in\mathcal{L}_0\backslash\{j\}})$$

where Γ^x_j is equal to $\varnothing$ if $j \notin \mathcal{L}_0$ and $(x_j, \{m^x_{j,i}\}_{i\in[n]\backslash\{j\}})$ otherwise. Moreover, the sharing of x is denoted as

$$[\![x]\!]_{\mathcal{L}_0} = ([\![x]\!]_{\mathcal{L}_0,1}, \ldots, [\![x]\!]_{\mathcal{L}_0,n}).$$

Like SPDZ, the parties can compute addition gates locally. Let $[\![y]\!]_{\mathcal{L}_1}$ be another sharing. Assume that $[\![y]\!]_{\mathcal{L}_1} = ([\![y]\!]_{\mathcal{L}_1,1}, \ldots, [\![y]\!]_{\mathcal{L}_1,n})$, where

$$[\![y]\!]_{\mathcal{L}_1,j} = (\Gamma^y_j, \Delta_j, \{k^y_{j,i}\}_{i\in\mathcal{L}_1\backslash\{j\}})$$

and Γ^y_j is equal to $\varnothing$ if $j \notin \mathcal{L}_1$ and $(y_j, \{m^y_{j,i}\}_{i\in[n]\backslash\{j\}})$ otherwise. Then each party P_j can locally compute its share of $x + y$ as

$$\begin{aligned}[\![x+y]\!]_{\mathcal{L},j} = (\Gamma^{x+y}_j, &\Delta_j, \{k^x_{j,i}\}_{i\in\mathcal{L}_0\backslash\mathcal{L}_1},\\ &\{k^x_{j,i} + k^y_{j,i}\}_{i\in\mathcal{L}_0\cap\mathcal{L}_1}, \{k^y_{j,i}\}_{i\in\mathcal{L}_1\backslash\mathcal{L}_0})\end{aligned}$$

where $\mathcal{L} = \mathcal{L}_0 \cup \mathcal{L}_1$ and

$$\Gamma^{x+y}_j = \begin{cases} (x_j, \{m^x_{j,i}\}_{i\in[n]\backslash\{j\}}), & \text{if } j \in \mathcal{L}_0\backslash\mathcal{L}_1\\ (x_j + y_j, \{m^x_{j,i} + m^y_{j,i}\}_{i\in[n]\backslash\{j\}}), & \text{if } j \in \mathcal{L}_0 \cap \mathcal{L}_1\\ (y_j, \{m^y_{j,i}\}_{i\in[n]\backslash\{j\}}). & \text{if } j \in \mathcal{L}_1\backslash\mathcal{L}_0\\ \varnothing. & \text{if } j \notin \mathcal{L}\end{cases}$$

It is easy to verify that $[\![x + y]\!]_{\mathcal{L}} = ([\![x + y]\!]_{\mathcal{L},1}, \ldots, [\![x + y]\!]_{\mathcal{L},n})$ is a lazy sharing of $x + y$.

Lazy SPDZ-Style MAC. For a lazy additive sharing $\langle x\rangle_{\mathcal{L}_0}$, we authenticate it using the following triple.

$$[\![x]\!]_{\mathcal{L}_0} = (\langle x\rangle_{\mathcal{L}_0}, \langle\Delta\rangle, \langle m_x\rangle)$$

[13]Similiar to SPDZ, in following descriptions, the shared value is in $\mathbb{F}$, but all the keys and MACs come from an extension field $\mathbb{E}$ of $\mathbb{F}$ with $\log_2 |\mathbb{E}| \geq \kappa$.

where Δ is the global key that is independent of x and $m_x = x\Delta$ is the MAC of x. Note that $\langle\Delta\rangle, \langle m_x\rangle$ are additive sharings. We can compute addition gates locally. Let $[\![y]\!]_{\mathcal{L}_1} = (\langle y\rangle_{\mathcal{L}_1}, \langle\Delta\rangle, \langle m_y\rangle)$ be another sharing (set $\mathcal{L} = \mathcal{L}_0 \cup \mathcal{L}_1$). Then the parties can locally compute

$$[\![x+y]\!]_{\mathcal{L}} = (\langle x\rangle_{\mathcal{L}_0} + \langle y\rangle_{\mathcal{L}_1}, \langle\Delta\rangle, \langle m_x\rangle + \langle m_y\rangle)$$

where $\langle x\rangle_{\mathcal{L}_0} + \langle y\rangle_{\mathcal{L}_1}$ is exactly the LGMW addition. It is obvious that $[\![x+y]\!]_{\mathcal{L}}$ is a lazy sharing of $x+y$.

Partial Open of Lazy Sharings. The parties partially open an authenticated sharing $[\![x]\!]_{\mathcal{L}} = (\langle x\rangle_{\mathcal{L}}, \langle\Delta\rangle, \langle m_x\rangle)$ as follows: for some $i \in \mathcal{L}$, each party P_j in $\mathcal{L}$ sends its share x_j to P_i, then P_i computes and sends $\bar{x} = \sum_{j\in\mathcal{L}} x_j$ to all other parties. Note that we require that any $|\mathcal{L}|-1$ values in $\{x_j\}_{j\in\mathcal{L}}$ are independent of x (if this is not the case, the parties should use a uniform lazy additive sharing $\langle 0\rangle_{\mathcal{L}}$ to randomize $\langle x\rangle_{\mathcal{L}}$ before opening the sharing). The partial open of an authenticated sharing $[\![x]\!]_{\mathcal{L}}$ takes $n + |\mathcal{L}| - 2$ field elements of communication.

MAC-Checking on Lazy Sharings. To guarantee that the corrupted parties cannot cheat, we still need to check the MACs of the opened values. Note that the lazy BDOZ sharing can be locally converted to the lazy SPDZ sharing using a similar approach in [LOS14], we just show how to check the lazy SPDZ sharing. In fact, we use the same way as in SPDZ to check an opened value $\bar{x}$ of a sharing $[\![x]\!]_{\mathcal{L}} = (\langle x\rangle_{\mathcal{L}}, \langle\Delta\rangle, \langle m_x\rangle)$. The check procedure is given in Appendix C.3 (Procedure C.5).

B.2 SPDZ with Authenticated Lazy Additive Sharing: LSPDZ

Now we describe the LSPDZ protocol in the circuit-dependent model, where the parties know the topology of the circuit in the offline phase. We use the lazy SPDZ-style MAC in our protocol, and for lazy BDOZ-style MAC, the protocol will be similar. Moreover, we focus on the online phase and use the offline phase as a black-box. In our protocol, the offline phase prepares the following correlated randomness.

- The MAC keys $\Delta_1, \ldots, \Delta_n$.
- For each input, let P_i be the input owner, and the parties generate a random sharing $[\![r]\!]_{\{i\}}$.
- For each multiplication gate with $\mathcal{L}_0, \mathcal{L}_1$ being the lazy sets of its two input sharings (let $\mathcal{L} = \mathcal{L}_0 \cup \mathcal{L}_1$), the parties generate an authenticated triple $[\![a]\!]_{\mathcal{L}_0}, [\![b]\!]_{\mathcal{L}_1}, [\![c]\!]_{\mathcal{L}}$ with $c = ab$.
- For each output sharing with lazy set $\mathcal{L}$, the parties generate a *uniform* lazy additive sharing of zero with lazy set $\mathcal{L}$ (without MAC).

More formally, we assume that the parties can invoke the preprocessing functionality $\mathcal{F}_{\mathsf{prep}}$ that is defined in Appendix A.

In this work, we do not discuss the realization of $\mathcal{F}_{\mathsf{prep}}$. However, we believe that it is interesting to study the efficient realization of $\mathcal{F}_{\mathsf{prep}}$ for various circuits. Now we can describe the LSPDZ protocol, which securely realizes the arithmetic MPC functionality $\mathcal{F}_{\mathsf{mpc}}$ (its formal definition can be found in Appendix A) in the $(\mathcal{F}_{\mathsf{com}}, \mathcal{F}_{\mathsf{prep}})$-hybrid model.

Protocol B.1 (The LSPDZ Protocol)**.** *Let f be the computed circuit.*

- ***Initialization.*** *The parties call the functionality $\mathcal{F}_{\mathsf{prep}}$ with input (Start, f) to generate the required correlated randomness.*
- ***Input.*** *To share an input x, a random sharing $[\![r]\!]_{\{i\}}$ is used:*
  1. *The owner P_i broadcasts $\sigma = x - r$ (r is known to P_i).*
  2. *The parties locally compute $[\![x]\!]_{\{i\}} = [\![r]\!]_{\{i\}} + \sigma$.*
- ***Addition.*** *To compute an addition gate with input sharings $[\![x]\!]_{\mathcal{L}_0}, [\![y]\!]_{\mathcal{L}_1}$ (set $\mathcal{L} = \mathcal{L}_0 \cup \mathcal{L}_1$), the parties locally compute $[\![x+y]\!]_{\mathcal{L}} = [\![x]\!]_{\mathcal{L}_0} + [\![y]\!]_{\mathcal{L}_1}$.*

- ***Multiplication.*** *To compute a multiplication gate with input sharings $[\![x]\!]_{\mathcal{L}_0}$ and $[\![y]\!]_{\mathcal{L}_1}$ (set $\mathcal{L} = \mathcal{L}_0 \cup \mathcal{L}_1$), an authenticated triple $[\![a]\!]_{\mathcal{L}_0}, [\![b]\!]_{\mathcal{L}_1}, [\![c]\!]_{\mathcal{L}}$ is used, and the parties do the followings:*

  1. *The parties locally compute $[\![\alpha]\!]_{\mathcal{L}_0} = [\![x]\!]_{\mathcal{L}_0} - [\![a]\!]_{\mathcal{L}_0}$ and $[\![\beta]\!]_{\mathcal{L}_1} = [\![y]\!]_{\mathcal{L}_1} - [\![b]\!]_{\mathcal{L}_1}$, and then partially open these two sharings.*
  2. *The parties locally compute $[\![z]\!]_{\mathcal{L}} = \alpha\beta + \alpha[\![b]\!]_{\mathcal{L}_1} + \beta[\![a]\!]_{\mathcal{L}_0} + [\![c]\!]_{\mathcal{L}}$.*

- ***Output.*** *To recover a sharing $[\![z]\!]_{\mathcal{L}} = (\langle z\rangle_{\mathcal{L}}, \langle\Delta\rangle, \langle m_z\rangle)$ to all parties, the parties do the followings:*

  1. *Run Procedure C.5 to check the MACs on the opened values so far. If some check does not pass, output* Abort.
  2. *Take a uniform lazy additive sharing $\langle 0\rangle_{\mathcal{L}}$ of zero and compute $[\![z_0]\!]_{\mathcal{L}} = (\langle z\rangle_{\mathcal{L}} + \langle 0\rangle_{\mathcal{L}}, \langle\Delta\rangle, \langle m_z\rangle)$.*
  3. *Finally, partially open $[\![z_0]\!]_{\mathcal{L}}$ to get $\bar{z}$ and run Procedure C.5 to check the MAC. If the check does not pass, output* Abort, *otherwise output* Success *and return $\bar{z}$.*

B.3 Security of LSPDZ

We state the security of Protocol B.1 by proving Theorem B.1.

Theorem B.1. *In the $(\mathcal{F}_{\mathsf{com}}, \mathcal{F}_{\mathsf{prep}})$-hybrid model, Protocol B.1 securely realizes the functionality $\mathcal{F}_{\mathsf{mpc}}$ against any static, active adversary corrupting any number of parties.*

The proof is deferred to Appendix I.3.

B.4 Complexity Analysis of LSPDZ

In this section, we analyze the online cost of LSPDZ and then compare it to the online cost of SPDZ.

Online Cost of LSPDZ. In the input sharing phase, sharing an input requires the parties to send $n-1$ field elements. Moreover, computing a multiplication gate with $\mathcal{L}_0, \mathcal{L}_1$ being the lazy sets of its two input sharings requires the parties to partially open a sharing with lazy set $\mathcal{L}_0$ and a sharing with lazy set $\mathcal{L}_1$, which results in $2n + |\mathcal{L}_0| + |\mathcal{L}_1| - 4$ field elements of communication. Finally, in the output recovery phase, the parties need to partially open the output sharing with lazy set $\mathcal{L}$, which takes $n + |\mathcal{L}| - 2$ field elements of communication. Moreover, like SPDZ, the parties also need to run check the MACs of the opened values and the final output. However, we do not count the communication cost of checking the MACs due to the following two reasons.

- LSPDZ mainly improves the efficiency of SPDZ in opening sharings. In fact, LSPDZ has the same efficiency as SPDZ in checking the MACs.
- Using the batch MAC-checking technique of [DKL$^+$13], the communication cost of checking the MACs in the output recovery phase will be independent of the number of inputs and multiplication gates, which makes it less significant for the total communication cost.

Remark. In the work of [BNO19], the authors improved the SPDZ online phase using circuit-dependent preprocessing such that the parties only need to open one sharing, which takes $2(n-1)$ field elements of communication. Concretely, in their protocol, the sharing of a value x is of the form $(x-a, [\![a]\!])$, where a is a random value and $x-a$ is known to all parties. Let $(x-a, [\![a]\!]), (y-b, [\![b]\!])$ be two sharings. Assume the parties have $[\![ab]\!]$ and a random sharing $[\![r]\!]$ (which can be computed in the offline phase), then the parties can compute a sharing of xy as follows.

1. The parties locally compute
$$[\![xy-r]\!] = (x-a)(y-b) + (x-a)[\![b]\!] + (y-b)[\![a]\!] + [\![ab]\!] - [\![r]\!].$$
2. The parties partially open $[\![xy-r]\!]$ and the sharing is $(xy-r, [\![r]\!])$.

Using lazy sharing, the sharing of a value x will be $(x-a, [\![a]\!]_{\mathcal{L}})$, where $\mathcal{L}$ is the lazy set, a is a random value, and $x-a$ is known to all parties. To multiply two sharings $(x-a, [\![a]\!]_{\mathcal{L}_0})$ and $(y-b, [\![b]\!]_{\mathcal{L}_1})$, the parties do the followings ($[\![ab]\!]_{\mathcal{L}}$ and a random sharing $[\![r]\!]_{\mathcal{L}}$ have been prepared in the offline phase).

1. The parties locally compute
$$[\![xy-r]\!]_{\mathcal{L}} = (x-a)(y-b) + (x-a)[\![b]\!]_{\mathcal{L}_1} + (y-b)[\![a]\!]_{\mathcal{L}_0} + [\![ab]\!]_{\mathcal{L}} - [\![r]\!]_{\mathcal{L}}.$$
2. The parties partially open $[\![xy-r]\!]_{\mathcal{L}}$ and the sharing is $(xy-r, [\![r]\!]_{\mathcal{L}})$.

In our protocol, the parties only need to open a sharing with lazy set $\mathcal{L}$, which only takes $n + |\mathcal{L}| - 2$ field elements of communication. We omit the further details because the analysis will be very similar.

C Several MPC Protocols based on Secret Sharing

C.1 The GMW Protocol

The formal description of the GMW protocol in the $\mathcal{F}_{\mathsf{ole}}$-hybrid model is in the following, which uses some notations that are defined in Section 4.

Protocol C.1 (The GMW Protocol [GMW87, IPS09])**.** *Let $f : \mathbb{F}^{M_{\mathsf{in}}^1} \times \cdots \times \mathbb{F}^{M_{\mathsf{in}}^n} \to \mathbb{F}^{M_{\mathsf{out}}^1} \times \cdots \times \mathbb{F}^{M_{\mathsf{out}}^n}$ be the computed circuit over $\mathbb{F}$. Each party P_i has M_{in}^i private inputs and M_{out}^i private outputs.*

1. ***Input Sharing.*** *For each input x belonging to P_i, P_i samples n random values $\{x_j\}_{j\in[n]}$ subject to $\sum_{j\in[n]} x_j = x$, and then P_i sends x_j to P_j for each $j \in [n]\backslash\{i\}$. The sharing of x is $(x_1, \ldots, x_n)$.*
2. ***Circuit Evaluation.*** *For each gate G, let u_i, v_i be the two shares on the input wires of G held by P_i. The parties perform the following steps.*
   (a) *If G is an addition gate, each party P_i computes $w_i = u_i + v_i$ as its final share.*
   (b) *If G is a multiplication gate, The parties perform the following steps.*
      i. *For each $i \in [n]$ and $j \in [n]\backslash\{i\}$, P_i samples a random value $r_{i,j}$. P_i and P_j invoke the OLE functionality $\mathcal{F}_{\mathsf{ole}}$ where P_i acting as the sender takes u_i and $-r_{i,j}$ as inputs and P_j acting as the receiver takes v_j as input. P_j receives $s_{j,i} = u_i v_j - r_{i,j}$ from $\mathcal{F}_{\mathsf{ole}}$.*
      ii. *Each party P_i computes $w_i = u_i v_i + \sum_{j\in[n]\backslash\{i\}}(s_{i,j} + r_{i,j})$ as its final share.*
3. ***Output Recovery.*** *For each output sharing $(y_1, \ldots, y_n)$ with P_i obtaining the output, each party P_j sends its share y_j to P_i, and then P_i computes $y = \sum_{j\in[n]} y_j$.*

Cost of the GMW Protocol. In the input sharing phase, to share an input, the input owner sends a field element to each other party. Note that there are total M_{in} inputs, hence the communication cost of the input sharing phase is $M_{\mathsf{in}} \cdot (n-1)$ field elements. In the circuit evaluation phase, the parties can locally compute addition gates, hence we only consider the communication cost of computing multiplication gates. To compute a multiplication gate, the parties invoke the OLE functionality $n(n-1)$ times. Since there are total C_{mul} multiplication gates, the cost of the circuit evaluation phase is $C_{\mathsf{mul}} \cdot n(n-1)$ calls to the OLE functionality. Finally, in the output recovery phase, to recover an output to some party P_i, each other party sends a field element to P_i. Note that there are total M_{out} outputs, and the communication cost of the output recovery phase is $M_{\mathsf{out}} \cdot (n-1)$ field elements. Overall, the cost of the GMW protocol is $(M_{\mathsf{in}} + M_{\mathsf{out}})(n-1)$ field elements of communication and $C_{\mathsf{mul}} \cdot n(n-1)$ calls to the OLE functionality.

C.2 The AFLNO Protocol

The AFLNO protocol [AFL+16] is an effective three-party computation protocol with semi-honest security in the honest majority setting (i.e., at most one party can be corrupted). This protocol is based on $(1,3)$ replicated secret sharing. Before describing the AFLNO protocol, we first define a functionality $\mathcal{F}_{\mathsf{zero}}$ which generates additive sharings of zero for the parties. We remark that $\mathcal{F}_{\mathsf{zero}}$ could be realized without any interaction beyond a short initial setup (see [AFL+16] for more details).

Functionality C.2 ($\mathcal{F}_{\mathsf{zero}}$). *The functionality $\mathcal{F}_{\mathsf{zero}}$ samples two random values $r_1, r_2 \in \mathbb{F}$ and computes $r_3 = -r_1 - r_2$. Then it sends r_i to P_i for each $i \in [3]$.*

Now we describe the AFLNO protocol[14].

Protocol C.3 (The AFLNO Protocol [AFL+16]). *The parties P_1, P_2, P_3 offer the inputs.*

1. ***Input Sharing.*** *To share an input x, the input owner (say, P_1) samples two random values x_1, x_2 and computes $x_3 = x - x_1 - x_2$. Then it sends (x_1, x_3) to P_2 and (x_1, x_2) to P_3. The sharing of x is $(x_3, x_2), (x_1, x_3), (x_2, x_1)$.*
2. ***Circuit Evaluation.*** *The parties compute the circuit gate-by-gate. Concretely, for each gate G with two input sharings $(x_3, x_2), (x_1, x_3), (x_2, x_1)$ and $(y_3, y_2), (y_1, y_3), (y_2, y_1)$, the parties do the followings.*
   - *If G is an addition gate, each party P_i locally computes $(x_{i-1} + y_{i-1}, x_{i+1} + y_{i+1})$ as its share. The resulting sharing is $(x_3 + y_3, x_2 + y_2), (x_1 + y_1, x_3 + y_3), (x_2 + y_2, x_1 + y_1)$.*
   - *If G is a multiplication gate, the parties perform the following steps.*
     (a) *The parties ask for an additive sharing (r_1, r_2, r_3) of zero from the functionality $\mathcal{F}_{\mathsf{zero}}$.*
     (b) *Each party P_i computes $z_{i-1} = x_{i-1}y_{i-1} + x_{i-1}y_{i+1} + x_{i+1}y_{i-1} + r_i$ and sends z_{i-1} to P_{i+1}. Note that (z_1, z_2, z_3) is an additive sharing of xy.*
     (c) *The final sharing is $(z_3, z_2), (z_1, z_3), (z_2, z_1)$.*
3. ***Output Recovery.*** *To recover a sharing $(z_3, z_2), (z_1, z_3), (z_2, z_1)$ to all parties, each P_i sends z_{i+1} to P_{i+1} and then each P_i computes $z = z_1 + z_2 + z_3$.*

Communication Cost. To share an input, the input owner needs to send *four* field elements. To compute a multiplication gate, each party only sends *one* element (given that an additive sharing of zero has been prepared). To recover an output, each party only sends one element.

C.3 The SPDZ Protocol

The SPDZ protocol [BDOZ11, DPSZ12] is a concretely efficient MPC protocol with malicious security. SPDZ works in the preprocessing model and contains an offline phase and an online phase. In the offline phase, the parties generate correlated randomness before knowing the inputs and the computed circuit, and in the online phase, the parties use the generated correlated randomness to compute the circuit. In SPDZ, to achieve malicious security, an authenticated additive secret sharing scheme is used, which is realized by equipping additive secret sharing with an information-theoretic MAC. Now let us describe two types of MACs used in SPDZ[15].

BDOZ-Style MAC [BDOZ11]. For a value x, it is first additively shared between the parties, i.e., an additive secret sharing $\langle x \rangle = (x_1, \ldots, x_n)$ is generated. Then each share x_i is authenticated by each other party (such that the parties cannot modify their shares). To do this, each party P_j generates a global key Δ_j to authenticate the shares of other parties. Then, for each $i \in [n]$ and each $j \in [n] \backslash \{i\}$, P_j holds a local key $k^x_{j,i}$ and P_i has the MAC

$$m^x_{i,j} = x_i \Delta_j + k^x_{j,i}.$$

The share of P_j is denoted as

$$[\![x]\!]_j = (x_j, \Delta_j, \{k^x_{j,i}, m^x_{j,i}\}_{i \in [n] \backslash \{j\}}).$$

And the sharing of x is

$$[\![x]\!] = ([\![x]\!]_1, \ldots, [\![x]\!]_n).$$

Note that we can compute the addition gates locally. Let $[\![y]\!] = ([\![y]\!]_1, \ldots, [\![y]\!]_n)$ be another sharing, where for each $j \in [n]$,

$$[\![y]\!]_j = (y_j, \Delta_j, \{k^y_{j,i}, m^y_{j,i}\}_{i \in [n] \backslash \{j\}}).$$

[14]The original AFLNO protocol uses a slightly different form of replicated secret sharing, we describe AFLNO using the form we introduced in the preliminary section.

[15]In following descriptions, the shared value is in $\mathbb{F}$, but all the keys and MACs come from an extension field $\mathbb{E}$ of $\mathbb{F}$ with $\log_2 |\mathbb{E}| \geq \kappa$ (If $\log_2 |\mathbb{F}| \geq \kappa$, then $\mathbb{E} = \mathbb{F}$).

Then each party P_j can locally compute its share of $x+y$:

$$[\![x+y]\!]_j = (x_j + y_j, \Delta_j, \{k^x_{j,i} + k^y_{j,i}, m^x_{j,i} + m^y_{j,i}\}_{i\in[n]\setminus\{j\}}).$$

It is easy to verify that $[\![x+y]\!] = ([\![x+y]\!]_1, \ldots, [\![x+y]\!]_n)$ is a sharing of $x+y$.

SPDZ-Style MAC [DPSZ12]. For a value $x \in \mathbb{F}$, the sharing of x has the following form

$$[\![x]\!] = (\langle x\rangle, \langle\Delta\rangle, \langle m_x\rangle),$$

where Δ is the global key that is independent of x and $m_x = x\Delta$ is the MAC of x. Again, using the SPDZ-Style MAC, addition gates can be computed locally. Let $[\![y]\!] = (\langle y\rangle, \langle\Delta\rangle, \langle m_y\rangle)$ another shared value. Then the parties can locally compute

$$[\![x+y]\!] = (\langle x\rangle + \langle y\rangle, \langle\Delta\rangle, \langle m_x\rangle + \langle m_y\rangle).$$

We can verify that $m_x + m_y$ is the MAC of $x+y$ because

$$m_x + m_y = x\Delta + y\Delta = (x+y)\Delta.$$

Partial Open. The parties sometimes need to partially open a sharing $[\![x]\!]$. To do this, each party sends its share x_i to P_1, and then P_1 computes and sends $\bar{x} = \sum_{i\in[n]} x_i$ to other parties. The communication cost is $2(n-1)$ field elements. Note that the reason why we use $\bar{x}$ rather than x is that the corrupted parties can make the opened value incorrect by sending the wrong shares.

MAC-Checking. To check the correctness of the computation, we still need to check the MACs after opening the sharings during the protocol. Due to the reason that the BDOZ-style sharing can be locally converted to the SPDZ-style sharing (see [LOS14] for more details), we just consider checking the SPDZ-style sharings. In the original SPDZ paper [DPSZ12], the authors check the MACs by revealing the global key Δ and the MACs, this however limits the use of the global key to a single evaluation. We adopt the approach described in [DKL$^+$13] to check the MACs without revealing the global key Δ. Checking the MACs requires a commitment scheme, and we assume that the parties can access the following commitment functionality (which could be realized efficiently [DN02, Lin11]).

Functionality C.4 (The Commitment Functionality $\mathcal{F}_{\mathsf{com}}$)**.** *This functionality have two phases.*

- ***Commit.*** *Receive* $(\mathsf{Com}, x, i, h_x)$ *from some party* P_i*, store* (x, i, h_x) *and send* (i, h_x) *to all parties. Note that* h_x *is a handle for the commitment.*
- ***Open.*** *Receive* (Open, i, h_x) *from* P_i *and send* (x, i, h_x) *to all parties. If instead receive* $(\mathsf{NoOpen}, i, h_x)$ *from* P_i*, and* P_i *is corrupt, then send* $(\perp, i, h_x)$ *to all parties.*

Now we show how to check the MAC on an opened value $\bar{x}$ of a sharing $[\![x]\!] = (\langle x\rangle, \langle\Delta\rangle, \langle m_x\rangle)$.

Procedure C.5 (MAC-Checking)**.** *The parties do the following steps*[16]*.*

1. *The parties locally compute* $\langle\sigma\rangle = \langle m_x\rangle - \bar{x}\langle\Delta\rangle$ *and each party calls* $\mathcal{F}_{\mathsf{com}}$ *to commit its share* σ_i *to other parties.*
2. *The parties call* $\mathcal{F}_{\mathsf{com}}$ *to open the committed values* $\sigma_1, \ldots, \sigma_n$ *and check if* $\sum_{i\in[n]} \sigma_i = 0$*. If so, the check passes, otherwise the computation is aborted.*

It can be shown that in Procedure C.5, the probability that the check passes and $\bar{x} \neq x$ is negligible in κ (see [DPSZ12, DKL$^+$13]). Now we describe the SPDZ protocol, which works in the preprocessing model.

Protocol C.6 (The SPDZ Protocol [BDOZ11, DPSZ12, DKL$^+$13])**.** *The SPDZ protocol contains an offline phase and an online phase. The notation* $[\![\cdot]\!]$ *represents a BDOZ or SPDZ sharing.*

1. ***Offline Phase.*** *The parties generate two classes of correlated randomness.*

[16] If we want to check the MACs on a large batch of values, we can use the techniques described in [DKL$^+$13] to do this with higher efficiency than checking the MAC on each value.

- *A random sharing $[\![r]\!]$ for each input (r is known to the input owner).*
- *A random authenticated triple $([\![a]\!], [\![b]\!], [\![c]\!])$ for each multiplication gate where $c = ab$.*

2. ***Online Phase.*** *The parties do the followings.*

   (a) ***Input Sharing.*** *To share an input x:*

      i. *The owner P_i broadcasts $\sigma = x - r$ (recall that r is known to P_i).*
      ii. *The parties locally compute $[\![x]\!] = [\![r]\!] + \sigma$.*

   (b) ***Circuit Evaluation.*** *The parties compute the circuit gate-by-gate. We have described the computation of addition gates. To compute a multiplication gate with inputs $[\![x]\!], [\![y]\!]$, the parties do the followings:*

      i. *The parties locally compute $[\![\alpha]\!] = [\![x]\!] - [\![a]\!]$ and $[\![\beta]\!] = [\![y]\!] - [\![b]\!]$ and partially open these sharings.*
      ii. *The parties locally compute $[\![z]\!] = \alpha\beta + \alpha[\![b]\!] + \beta[\![a]\!] + [\![c]\!]$.*

   (c) ***Output Recovery.*** *To recover an output, the parties first run Procedure C.5 to check the MACs on the opened values so far. If some check does not pass, then the computation is aborted. Otherwise, the parties partially open the output sharing and run Procedure C.5 on the opened value. All parties output the opened value if the check passes and abort the computation otherwise.*

In this work, we do not consider the concrete execution of the offline phase and focus on the online cost of the SPDZ protocol.

Online Cost of SPDZ. To share an input, the input owner needs to send $n - 1$ field elements. To compute a multiplication gate, the parties need to partially open two sharings, resulting in $4(n - 1)$ field elements of communication. To recover an output, the parties need to partially open a sharing, which takes $2(n - 1)$ field elements of communication. Note that in the output recovery phase, the parties also need to check the MACs on all opened values. Due to that our improvements over SPDZ are mainly in opening the sharings during the protocol execution, we do not count the communication cost of checking the MACs[17], which will also allow us to present our results more clearly.

D Reviewing GMW in the Preprocessing Model

In this section, we describe how GMW works in the preprocessing model. We use $\langle x \rangle$ to represent an additive sharing of x. In the preprocessing model, the GMW protocol can achieve very high efficiency. Firstly, the input sharing phase can be executed locally in the online phase: the parties prepare a batch of additive sharings of zero in the offline phase; in the online phase, for each input, the parties consume one sharing of zero, and the input owner just adds its input to its share as the final share. Moreover, the output recovery phase remains the same as in the non-preprocessing model (each party sends its share to the party that is supposed to obtain the output).

Now we recall how to preprocess the circuit evaluation phase. The focal point is to preprocess the computation of a multiplication gate. To do this, the parties produce a multiplication triple (a.k.a. Beaver triple [Bea91a]) consisting of three additive sharings $\langle a \rangle, \langle b \rangle$ and $\langle c \rangle$ where a, b are uniformly random in $\mathbb{F}$ and unknown to the adversary, and $c = ab$. In the offline phase, the parties generate one such tuple for every multiplication gate in the circuit. With such one tuple, the parties can compute a multiplication gate $G(x, y) = xy$ with input sharings $\langle x \rangle, \langle y \rangle$ as follows.

1. The parties locally compute $\langle \alpha \rangle = \langle x \rangle - \langle a \rangle$ and $\langle \beta \rangle = \langle y \rangle - \langle b \rangle$.
2. The parties open the sharings $\langle \alpha \rangle$ and $\langle \beta \rangle$ such that all parties know the values of α and β. Concretely, for each $i \in [n - 1]$, P_i sends its shares of α and β to P_n, and P_n recovers and sends α and β to other parties.
3. The parties locally compute $\langle z \rangle = \alpha\beta + \alpha\langle b \rangle + \beta\langle a \rangle + \langle c \rangle$.

[17] In fact, our protocol will use the same approach to check the MACs as in SPDZ.

It is easy to verify that $\langle z \rangle$ is an additive sharing of xy. In particular, the above construction achieves information-theoretic security and has high efficiency because it only makes use of simple arithmetic operations over $\mathbb{F}$.

Assume that the computed circuit has M_{in} inputs, M_{out} outputs (each party P_i obtains M^i_{out} outputs), and C_{mul} multiplication gates, then the online communication cost of GMW in the preprocessing model is $(4C_{\mathsf{mul}} + M_{\mathsf{out}})(n-1)$ field elements.

E Complexity Analysis for Specific Circuits

In this section, we analyze the efficiency of our protocols and previous protocols for computing the sum, product, and inner product circuits that are defined as follows.

- *Sum*: $f_{\mathsf{sum}}(x_1, \ldots, x_n) = \sum_{i\in[n]} x_i$. Each party P_i has a private input x_i and the party P_1 obtains the output.
- *Product*: $f_{\mathsf{pro}}(x_1, \ldots, x_n) = \prod_{i\in[n]} x_i$. Each party P_i has a private input x_i and the party P_1 obtains the output.
- *Inner Product*: $f_{\mathsf{inp}}(x_1, \ldots, x_l, y_1, \ldots, y_l) = \sum_{i\in[l]} x_i y_i$. We assume that each party has input(s). For each $i \in [l]$, x_i and y_i are privately held by two different parties (otherwise, this party just takes $x_i y_i$ as input), and the party P_1 obtains the output.

E.1 Complexity Analysis for GMW and LGMW

In this section, we analyze the costs of GMW and LGMW (without preprocessing).

Computing the Sum Circuit f_{sum}. For the sum circuit, we have $M_{\mathsf{in}} = n, M_{\mathsf{out}} = 1$ and $C_{\mathsf{mul}} = 0$, hence the communication cost of GMW for computing f_{sum} is $n^2 - 1$ field elements. On the other hand, if using LGMW to compute f_{sum}, then the input sharing and circuit evaluation phases can be executed locally, and the output recovery phase requires the parties to invoke the GMW protocol to compute the sum $\sum_{i\in[n]} x_i$. Therefore, the communication cost of using LGMW to compute f_{sum} is equal to the communication cost of using GMW to compute f_{sum}, i.e., $n^2 - 1$ field elements.

Computing the Product Circuit f_{pro}. We assume that $n = 2^d$ for some integer d. For the product circuit, we have $M_{\mathsf{in}} = n, M_{\mathsf{out}} = 1$ and $C_{\mathsf{mul}} = n - 1$, hence the cost of the GMW protocol for computing f_{pro} is $n(n-1)^2$ calls to the OLE functionality and $n^2 - 1$ field elements of communication. Now we clarify the cost of using LGMW to compute f_{pro} according to the analysis in Section 4.3. Note that there are total $2^{d-\lambda}$ layer-λ multiplication gates for each $\lambda \in [d]$. For each $\lambda \in [d]$ and $k \in [2^{d-\lambda}]$, let $G_{\lambda,k}$ be the k-th gate in the λ-th layer of the circuit. Then it is easy to see that the lazy sets of the input sharings of $G_{\lambda,k}$ are $\mathcal{L}^0_{\lambda,k} = \{(k-1)2^\lambda + 1, \ldots, (2k-1)2^{\lambda-1}\}, \mathcal{L}^1_{\lambda,k} = \{(2k-1)2^{\lambda-1} + 1, \ldots, k2^\lambda\}$. Therefore, the required number of calls to OLE functionality for computing f_{pro} using LGMW is

$$\sum_{\lambda\in[E]} \sum_{k\in[C^\lambda_{\mathsf{mul}}]} (|\mathcal{L}^0_{\lambda,k}| \cdot |\mathcal{L}^1_{\lambda,k}| - |\mathcal{L}^0_{\lambda,k} \cap \mathcal{L}^1_{\lambda,k}|) = \frac{n(n-1)}{2}.$$

Moreover, note that the lazy set of the final output sharing is $[n]$, hence the output recovery phase requires the parties to communicate $n^2 - 1$ field elements (for securely computing the sum of the n shares). Compared to GMW, LGMW reduces the number of calls to the OLE functionality by a factor of $2(n-1)$ for computing the circuit $f_{\mathsf{pro}}(x_1, \ldots, x_n) = \prod_{i\in[n]} x_i$.

Computing the Inner Product Circuit f_{inp}. For the inner product circuit, we have $M_{\mathsf{in}} = 2l, M_{\mathsf{out}} = 1$ and $C_{\mathsf{mul}} = l$, hence the cost of the GMW protocol for computing this circuit is $ln(n-1)$ calls to the OLE functionality and $(2l+1)(n-1)$ field elements of communication. Now we clarify the cost of using LGMW to compute f_{inp}. Note that there are total l layer-1 multiplication gates, and for each multiplication gate, the lazy sets $\mathcal{L}_0, \mathcal{L}_1$ of its input sharings satisfy that $|\mathcal{L}_0| = |\mathcal{L}_1| = 1$ and $\mathcal{L}_0 \cap \mathcal{L}_1 = \varnothing$. Therefore, the number of calls to OLE functionality for computing f_{inp} is l. Moreover, the lazy set of the final output sharing is $[n]$

(each party has at least one input), hence the output recovery phase requires the parties to communicate $n^2 - 1$ field elements. Compared to GMW, the LGMW protocol reduces the number of calls to the OLE functionality by a factor of $n(n-1)$ for computing the inner product.

E.2 Complexity Analysis for GMW and LGMW in the Preprocessing Model

In this section, we analyze the costs of GMW and LGMW in the preprocessing model.

Computing the Sum Circuit f_{sum}. In both the GMW and LGMW protocols in the preprocessing model, the online phase only requires each party to interact to recover an additive sharing. Therefore, the online communication costs of both GMW and LGMW are $n-1$ field elements.

Computing the Product Circuit f_{pro}. Assume that $n = 2^d$ for some integer d. If using GMW to compute the product circuit, the online communication cost will be $(n-1) \cdot 4(n-1) + (n-1) = (n-1)(4n-3)$ field elements. Now we consider using LGMW to compute the product circuit. For each $\lambda \in [d]$ and $k \in [2^{d-\lambda}]$, let $G_{\lambda,k}$ be the k-th gate in the λ-th layer of the circuit. The lazy sets of the input sharings of $G_{\lambda,k}$ are $\mathcal{L}^0_{\lambda,k} = \{(k-1)2^\lambda + 1, \ldots, (2k-1)2^{\lambda-1}\}, \mathcal{L}^1_{\lambda,k} = \{(2k-1)2^{\lambda-1} + 1, \ldots, k2^\lambda\}$. If using circuit-independent preprocessing, then the online communication cost will be

$$n - 1 + \sum_{\lambda \in [d]} \sum_{k \in [2^{d-\lambda}]} (3n-2) = (n-1)(3n-1)$$

field elements. If using circuit-dependent preprocessing, then the online communication cost will be

$$\begin{aligned} &n - 1 + \sum_{\lambda \in [d]} \sum_{k \in [2^{d-\lambda}]} 2(|\mathcal{L}^0_{\lambda,k}| + |\mathcal{L}^1_{\lambda,k}| - 1) \\ =&n - 1 + \sum_{\lambda \in [d]} \sum_{k \in [2^{d-\lambda}]} 2(2^{\lambda-1} + 2^{\lambda-1} - 1) = n(2\log_2 n - 1) + 1 \end{aligned}$$

field elements.

Computing the Inner Product Circuit f_{inp}. Note that f_{inp} has l layer-1 multiplication gates. If using GMW to compute this circuit, the online communication cost will be $l \cdot 4(n-1) + (n-1) = (4l+1)(n-1)$ field elements. Now we consider using LGMW to compute the inner product circuit. For each $i \in [l]$, let G_i be the i-th multiplication gate and $\mathcal{L}^0_i, \mathcal{L}^1_i$ be the lazy sets of its input sharings. We always have $|\mathcal{L}^0_i| = |\mathcal{L}^1_i| = 1$ and $\mathcal{L}^0_i \cap \mathcal{L}^1_i = \varnothing$. If using circuit-independent preprocessing, then the online communication cost will be

$$n - 1 + \sum_{i \in [l]} (3n-2) = (3l+1)n - 2l - 1$$

field elements. If using circuit-dependent preprocessing, then the online communication cost will be

$$n - 1 + \sum_{i \in [l]} 2(|\mathcal{L}^0_i| + |\mathcal{L}^1_i| - 1) = 2l + n - 1$$

field elements.

E.3 Complexity Analysis for SPDZ and LSPDZ

In this section, we analyze the costs of SPDZ and LSPDZ.

Computing the Sum Circuit f_{sum}. If using SPDZ, the online phase requires the parties to share n inputs and partially open a sharing, which results in $n(n-1)+2(n-1) = (n+2)(n-1)$ field elements of communication. If using LSPDZ, the online phase requires the parties to share n inputs and partially open a lazy sharing with lazy set $[n]$, resulting in $n(n-1) + n + n - 2 = (n+2)(n-1)$ field elements of communication. In other words, SPDZ and LSPDZ have the same online communication cost for computing the sum circuit.

Computing the Product Circuit f_{pro}. Assume that $n = 2^d$ for some integer d. If using SPDZ to compute the product circuit, the online communication cost will be $n(n-1) + 4(n-1)^2 + 2(n-1) = (5n-2)(n-1)$ field

elements. Now we consider using LSPDZ to compute the product circuit. For each $\lambda \in [d]$ and $k \in [2^{d-\lambda}]$, let $G_{\lambda,k}$ be the k-th gate in the λ-th layer of the circuit. The lazy sets of the input sharings of $G_{\lambda,k}$ are $\mathcal{L}^0_{\lambda,k} = \{(k-1)2^\lambda + 1, \ldots, (2k-1)2^{\lambda-1}\}, \mathcal{L}^1_{\lambda,k} = \{(2k-1)2^{\lambda-1} + 1, \ldots, k2^\lambda\}$. We know that the online communication cost of LSPDZ is

$$\begin{aligned} & n(n-1) + \sum_{\lambda\in[d]} \sum_{k\in[2^{d-\lambda}]} (2n + |\mathcal{L}^0_{\lambda,k}| + |\mathcal{L}^1_{\lambda,k}| - 4) + (2n-2) \\ =&(n+2)(n-1) + \sum_{\lambda\in[d]} \sum_{k\in[2^{d-\lambda}]} (2n + 2^{\lambda-1} + 2^{\lambda-1} - 4) \\ =&(3n-2)(n-1) + n\log_2 n \end{aligned}$$

field elements.

Computing the Inner Product Circuit f_{inp}. Note that there are total $2l$ inputs and l layer-1 multiplication gates, if using SPDZ to compute this circuit, the online communication cost will be

$$2l \cdot (n-1) + l \cdot 4(n-1) + 2(n-1) = (6l+2)(n-1)$$

field elements. Now we consider using LSPDZ to compute the inner product circuit. For each $i \in [l]$, let G_i be the i-th multiplication gate and $\mathcal{L}^0_i, \mathcal{L}^1_i$ be the lazy sets of its input sharings. We always have $|\mathcal{L}^0_i| = |\mathcal{L}^1_i| = 1$. Therefore, the online communication cost is

$$2l \cdot (n-1) + \sum_{i\in[l]} (2n-2) + 2n - 2 = (4l+2)(n-1)$$

field elements.

F Improvements for Circuits with a Large Depth

Based on our analysis, we claim that our optimization is more suitable for circuits with small depth. For this reason, we have only discussed the computation of low-depth circuits, i.e., the inner product (which is in NC^0) and product (which is in NC^1) circuits. However, for many circuits with a large depth, we can still achieve impressive improvements. When computing a multiplication gate, our improvement depends on the size of the lazy sets of the sharings input to this gate. The smaller the lazy sets, the better our improvements. Since, in general, the size of the lazy sets increases with the depth, we say that the improvements will be smaller for circuits with a larger depth. Nevertheless, an observation is that in many circuits, the sizes of lazy sets may grow slowly with the circuit depth. In this case, our techniques can achieve relatively good savings. Next, we consider two particular circuits with a large depth.

We first consider the following chain circuit

$$\begin{aligned} f_n =& f((y_0, y_1 \ldots, y_n), x_1, \ldots, x_n) \\ =& g(\cdots g(g(y_0 + x_1, y_1) + x_2, y_2) + \cdots + x_n, y_n) \end{aligned}$$

for some function g. When g is a block cipher with $y_1 = \cdots = y_n$ being the key, then $(f_1, \ldots, f_n)$ in fact outputs the ciphertext of $(x_1, \ldots, x_n)$ under the cipher block chaining (CBC) mode. Now we analyze our efficiency obtained for computing the circuit f_n. We assume that each x_i is held by P_i and $y_0, y_1, \ldots, y_n$ are held by an additional party P_0. Moreover, to simplify our analysis, we assume that $g(x, y)$ simply outputs xy.[18] This way, we have $f_1 = (y_0 + x_1)y_1$ and $f_i = (f_{i-1} + x_i)y_i$ for $i > 1$. As a result, the depth of f_n is $O(n)$ and the number of multiplication gates is n. If we run GMW among the parties $P_0, P_1, \ldots, P_n$, then the number of required OLEs is $n(n+1) \cdot n = n^2(n+1)$. However, since each f_i depends only on the parties $P_0, P_1, \ldots, P_i$, if using our LGMW, then the number of required OLEs for computing f_i is i. Hence the total number of required OLEs is $\sum_{i\in[n]} i = n(n+1)/2$. That is, using our technique, we achieve an improvement by a factor of $2n$.

[18] We remark that the efficiency improvement mainly comes from the fact that each f_i only depends on the parties $P_0, P_1, \ldots, P_i$, rather than the choice of g.

We consider another special circuit

$$f(x_1,\ldots,x_n)=\prod_{b_1,\ldots,b_n\in\{0,1\},\prod_{i\in[n]}(1-b_i)\neq 1}(b_1x_1+\cdots+b_nx_n).$$

We assume that each x_i is held by P_i. This circuit has a very large depth $O(n\log n)$. Now we show that we can also improve the efficiency for computing this circuit. We consider $n=4$ and now the circuit is

$$\begin{aligned}&f(x_1,x_2,x_3,x_4)=\\&\quad x_1x_2x_3x_4(x_1+x_2)(x_1+x_3)(x_1+x_4)\\&\quad (x_2+x_3)(x_2+x_4)(x_3+x_4)(x_1+x_2+x_3)\\&\quad (x_1+x_2+x_4)(x_1+x_3+x_4)(x_2+x_3+x_4)(x_1+x_2+x_3+x_4).\end{aligned}$$

Using standard GMW, the number of required OLEs will be $4(4-1)\cdot 14=168$. Using our LGMW, the number of required OLEs will be $1+2+3+6\cdot 6+9\cdot 4+12=90$. That is, we improve the efficiency by a factor of about $168/90\approx 1.86$. In fact, this factor approaches 2 as n increases. To see this, note that if using standard GMW, then the number of required OLEs will be $n(n-1)(2^n-2)$. Now we analyze the efficiency when using our LGMW. If we let $X=(x_1,\ldots,x_n), B=(b_1,\ldots,b_n)$, then we have

$$\begin{aligned}&f(x_1,\ldots,x_n)\\=&\prod_{b_1,\ldots,b_n\in\{0,1\},\prod_{i\in[n]}(1-b_i)\neq 1}(b_1x_1+\cdots+b_nx_n)\\=&\prod_{i\in[n]}\Big(\prod_{B\in\{0,1\}^n,|B|=i}\langle B,X\rangle\Big).\end{aligned}$$

Note that computing $y_1=\prod_{B\in\{0,1\}^n,|B|=1}\langle B,X\rangle$ requires $1+2+\cdots+(n-1)=n(n-1)/2$ calls to OLE. Since y_1 depends on all the inputs, hence computing $y_1\cdot\langle B,X\rangle$ for any B subject to $|B|=i$ requires $i(n-1)$ calls to OLE and the result still depends on all the inputs. Therefore, we can infer that the total number of required OLEs for computing $f(x_1,\ldots,x_n)$ is

$$\begin{aligned}&n(n-1)/2+\sum_{i\in[n-1]}(i+1)(n-1)\cdot\tbinom{n}{i+1}\\=&n(n-1)/2+\sum_{i\in[n-1]}n(n-1)\cdot\tbinom{n-1}{i}\\=&n(n-1)(1/2+2^{n-1}-1)=n(n-1)(2^{n-1}-1/2).\end{aligned}$$

As a result, we reduce the number of required OLEs by a factor of

$$\frac{n(n-1)(2^n-2)}{n(n-1)(2^{n-1}-1/2)}=2-\frac{2}{2^n-1}.$$

G Efficiency Improvements of LGMW over GMW in MPC Applications

In this section, we show the efficiency improvements of LGMW over GMW in real-world applications. Concretely, we consider the work of [CHK+12], which studied applications of MPC in *online marketplaces*, where customers choose the resources that providers advertise, and it is preferred to protect privacy as much as possible. The work of [CHK+12] showed that the semi-honest GMW protocol can outperform previous MPC implementations with $n\geq 3$ parties. We show that using our LGMW protocol, we can further reduce the asymptotic cost.

The formal description of the online marketplace problem can be stated as follows (the details can be found in [CHK+12, Section 3]). Let R (with $|R|=s$) be some set of items. There are a customer P_0 and n providers $P_1,\ldots,P_n$ where each provider P_i has a set $R_i\subseteq R$ of items and P_0 wishes to buy an item (e.g., a

car) from one of these n providers[19]. The customer P_0 is interested in the items that meet some conditions (which is described by the private input x of P_0). Moreover, each provider P_i independently prices each item in R_i. The goal is that the customer P_0 wants to find an acceptable item at the lowest price and P_0 learns the identity of the provider who offers this item. More concretely, the input of each provider P_i is a collection of values for the items in R_i, i.e., P_i's input is of the form $\{V_{i,r}\}_{r\in R_i}$ (e.g., the price of r is included in $V_{i,r}$). Let us consider the marketplaces where the computation can be broken into the following two steps.

1. For each provider P_i and each item r in R_i, compute a scoring function $y_{i,r} = \mathsf{Score}(i, r, V_{i,r}, x)$ (with the size of $y(i, r)$ being L).
2. Next, apply a best-match function BestMatch to $\{y_{i,r}\}_{i\in[n],r\in R_i}$ and the output is given to the customer P_0.

The work of [CHK+12] considered three special cases, i.e., P2P content-distribution services, cloud computing and mobile social network. We focus on the case of P2P content-distribution services, and the analyses for the other two cases are similar. In the P2P content-distribution services setting, the scoring function Score contains $O(L)$ multiplication gates, and the best-match function BestMatch contains $O(n(L + \log n))$ multiplication gates (more details can be found in [CHK+12, Section 4.2])[20]. Using the GMW protocol, the asymptotic costs for computing Score and BestMatch are $O(n^2L)$ and $O(n^3(L + \log n))$ calls to the OT functionality, respectively[21]. Note that we need to compute $\sum_{i\in[n]} |R_i| = O(n)$ times of Score and one time of BestMatch. Therefore, the total cost is

$$O(n) \cdot O(n^2L) + O(n^3(L + \log n)) = O(n^3(L + \log n))$$

calls to the OT functionality. Using our LGMW protocol, we can improve the performance for computing Score and BestMatch as follows.

1. Using our protocol, the scoring function Score can be computed by two parties (some provider P_i and the customer P_0) instead of $n+1$ parties because only P_i and P_0 provide inputs (P_i takes $V_{i,r}$ for $r \in R_i$ as input and P_0 takes x as input) for each computation of Score. This allows us to reduce the number of calls to the OT functionality from $O(n^2L)$ to $O(L)$, achieving an $O(n^2)$ improvement.
2. The best-match function BestMatch takes $m = \sum_{i\in[n]} |R_i| = O(n)$ values as inputs and outputs the index of the input with the maximum score. Concretely, the function BestMatch can be written as

$$\mathsf{BestMatch}(\{y_{i,r}\}_{i\in[n],r\in R_i}) = \bigodot_{i\in[n],r\in R_i} (r, y_{i,r})$$

where the $\odot$ gate contains $O(L + \log n)$ multiplication gates and outputs the input with larger score[22]. An important observation is that each input $y_{i,r}$ comes from two parties (because it is the output of the scoring function), i.e., each $y_{i,r}$ is additively shared by two parties. Using a similar method for computing the product function, the parties need to compute $m/2$ layer-1 $\odot$ gates, $m/4$ layer-2 $\odot$ gates, $m/8$ layer-3 $\odot$ gates, etc. Note that a layer-λ $\odot$ gate requires $O(4^\lambda(L + \log n))$ calls to the OT functionality (because at most $2^{\lambda+1}$ parties participate in the computation of a layer-λ $\odot$ gate). Therefore, using our protocol, the cost will be

$$\sum_{\lambda\in[\log m]} (m/(2^\lambda))O(4^\lambda(L + \log n)) = O(n^2(L + \log n))$$

calls to the OT functionality, which improves the performance by a factor of $O(n)$ compared to using the GMW protocol (which requires $O(n^3(L + \log n))$ calls to the OT functionality).

[19]To better show our results, we assume that $s = O(n)$ and $\sum_{i\in[n]} |R_i| = O(n)$. This is reasonable, as in many applications, the number of items held by each provider P_i is mainly influenced by its own wealth rather than the number of providers in the market.

[20]The underlying field of these multiplication gates is $\mathbb{F}_2$ and the parties can access the OT functionality. Note that our LGMW protocol also works in $\mathbb{F}_2$ with the OLE functionality being replaced by the OT functionality.

[21]We ignore other costs because the main cost is in the calls to the OT functionality.

[22]For example, $(r, y_{i,r}) \odot (r', y_{i',r'})$ equals $(r, y_{i,r})$ if $y_{i,r} \geq y_{i',r'}$ and $(r', y_{i',r'})$ otherwise. We refer to [CHK+12, Section 4.2]) for more details.

Overall, using the LGMW protocol, the total cost is

$$O(n) \cdot O(L) + O(n^2(L + \log n)) = O(n^2(L + \log n))$$

calls to the OT functionality. As a result, we can improve the performance by a factor of $O(n)$.

H (Lazy) Replicated Sharing in the Computational Setting

H.1 Replicated Sharing in the Computational Setting

Replicated secret sharing could be generated with much lower amortized communication cost when we only require computational security, which is achieved by using pseudorandom secret sharing [CDI05]. Recall that to generate a (t, n) replicated sharing of a value x, the input owner first samples $\binom{n}{t}$ random values $\{X_T\}_{T\subseteq[n],|T|=t}$ subject to their sum is x. Note that any $\binom{n}{t} - 1$ of these values are uniformly random values that are independent of x, and each of them will be sent to $n - t$ parties. If we let these $n - t$ parties agree on a key of a PRF, then they can locally agree on an unlimited number of pseudorandom values, which can be used to share a large batch of inputs. After obtaining these $\binom{n}{t} - 1$ pseudorandom values, the left thing is to compute and distribute the last value. To do this, we let the input owner know all the keys, and then the input owner can compute and distribute the last value.

We give the detailed description for the general case that the input is held by a set $\mathcal{L} \subseteq [n]$ of parties. Let $\mathsf{Prf} : \{0,1\}^\kappa \times \{0,1\}^\kappa \to \mathbb{F}$ be a PRF. The parties first prepare keys in the following setup phase.

- **Setup.** The parties choose a set $T_\mathcal{L} \subseteq [n]$ subject to $|T_\mathcal{L}| = t$. Then for each $T \subseteq [n]$ with $|T| = t, T \neq T_\mathcal{L}$, the parties in $\mathcal{L} \cup ([n]\backslash T)$ agree on a random key k_T[23]. Note that the total number of keys is $\binom{n}{t} - 1$.

Then to share an input x that is held by the parties in $\mathcal{L}$ (let $l = |\mathcal{L}|$), the parties perform the following steps.

1. **Generating $\binom{n}{t} - 1$ Pseudorandom Values.**
   (a) Let $\mathsf{id} \in \{0,1\}^\kappa$ be an identifier that has not been used.
   (b) For each $T \subseteq [n]$ with $|T| = t, T \neq T_\mathcal{L}$, the parties in $\mathcal{L} \cup ([n]\backslash T)$ locally compute $X_T = \mathsf{Prf}(k_T, \mathsf{id})$.
2. **Computing the Last Value.** Every party in $\mathcal{L}$ computes the last value.
$$X_{T_\mathcal{L}} = x - \sum_{T\subseteq[n],|T|=t,T\neq T_\mathcal{L}} X_T.$$
3. **Distributing the Last Value.** For some $j \in \mathcal{L}$, P_j sends $X_{T_\mathcal{L}}$ to the parties in $[n]\backslash(\mathcal{L} \cup T_\mathcal{L})$.
4. **Creating the Final Sharing.** For each $j \in [n]$, the share of P_j is $x_j = (X_T)_{|T|=t,j\notin T}$.

In the above construction, the communication cost is $|[n]\backslash(\mathcal{L} \cup T_\mathcal{L})|$ field elements. Note that the only requirement for $T_\mathcal{L}$ is $|T_\mathcal{L}| = t$. To minimize $|[n]\backslash(\mathcal{L} \cup T_\mathcal{L})|$, we choose $T_\mathcal{L}$ such that $\mathcal{L} \cap T_\mathcal{L} = \varnothing$ or $\mathcal{L} \cup T_\mathcal{L} = [n]$. It is obvious that if $l < n - t$, the communication cost is $n - t - l$ field elements, and if $l \geq n - t$, the communication cost is zero.

H.2 Lazy Replicated Sharing in the Computational Setting

Now we show how to generate lazy replicated sharing more efficiently. Recall that if an input x is held by a set $\mathcal{L}$ of parties (let $l = |\mathcal{L}|$), then if $l \geq n - t$, the parties can generate a $(t, n, \mathcal{L})$ lazy replicated sharing locally *without any setup*. Now we show that if $l < n - t$, we can use pseudorandom secret sharing [CDI05] to further reduce the communication cost.

Recall that to generate a $(t, n, \mathcal{L})$ lazy replicated sharing of a value x, the input owners first sample $\binom{n-l}{t}$ random values subject to their sum is x. Any $\binom{n-l}{t} - 1$ of these values are random and could be generated very efficiently using a PRF. Concretely, in the setup phase, let $\mathsf{Prf} : \{0,1\}^\kappa \times \{0,1\}^\kappa \to \mathbb{F}$ be a PRF, the parties prepare the required keys.

[23] Since the parties in $\mathcal{L}$ are the input owners, we allow them to know all the keys.

- **Setup.** The parties choose a set $T_{\mathcal{L}} \subseteq [n]\backslash\mathcal{L}$ subject to $|T_{\mathcal{L}}| = t$. Then for each $T \subseteq [n]\backslash\mathcal{L}$ with $|T| = t, T \neq T_{\mathcal{L}}$, the parties in $\mathcal{L} \cup ([n]\backslash T)$ agree on a random key k_T. Note that the total number of keys is $\binom{n-l}{t} - 1$.

To share an input that is known to the parties in $\mathcal{L}$ (assume that $l = |\mathcal{L}| < n - t$), the parties perform the following steps.

1. **Generating $\binom{n-l}{t} - 1$ Pseudorandom Values.**
   (a) Let $\mathsf{id} \in \{0,1\}^\kappa$ be an identifier that has not been used.
   (b) For each $T \subseteq [n]\backslash\mathcal{L}$ with $|T| = t, T \neq T_{\mathcal{L}}$, the parties in $\mathcal{L} \cup ([n]\backslash T)$ locally compute $X_T = \mathsf{Prf}(k_T, \mathsf{id})$.
2. **Computing the Last Value.** Every party in $\mathcal{L}$ computes the last value.
$$X_{T_{\mathcal{L}}} = x - \sum_{T \subseteq [n]\backslash\mathcal{L}, |T|=t, T \neq T_{\mathcal{L}}} X_T.$$
3. **Distributing the Last Value.** For some $j \in \mathcal{L}$, P_j sends $X_{T_{\mathcal{L}}}$ to P_k for each $k \in [n]\backslash(\mathcal{L} \cup T_{\mathcal{L}})$.
4. **Creating the Final Sharing.** For each $T \subseteq [n]$ with $|T| = t, \mathcal{L} \cap T \neq \varnothing$, all the parties locally set $X_T = 0$. For each $j \in [n]$, the share of P_j is $x_j = (X_T)_{|T|=t, j \notin T}$.

It is easy to verify that $(x_1, \ldots, x_n)$ is a (pseudorandom) $(t, n, \mathcal{L})$ lazy replicated sharing of x. Moreover, the communication cost is $|[n]\backslash(\mathcal{L} \cup T_{\mathcal{L}})|$ field elements. Note that $T_{\mathcal{L}} \subseteq [n]\backslash\mathcal{L}, |T_{\mathcal{L}}| = t$, hence the communication cost is $n - t - l$ field elements.

I Missing Security Proofs

I.1 Security Proof of Theorem 4.1

Proof. The correctness of Protocol 4.1 is easy to verify. Now we proceed to prove the security. Let $\mathcal{C}$ be the set of corrupted parties and $\mathcal{H} = [n]\backslash\mathcal{C}$ be the set of honest parties. We need to construct a simulator that simulates the view of the corrupted parties given a trusted party P for computing f. The simulator first gives the inputs of the corrupted parties to the trusted party P and receives the outputs $\{y_{i,1}, \ldots, y_{i,M^i_{\mathsf{out}}}\}_{i \in \mathcal{C}}$. The simulator $\mathcal{S}$ simulates the protocol execution as follows.

1. **Simulating the Input Sharing Phase.** Note that the input sharing phase is executed locally by the parties, so the simulator just executes the input sharing phase.
2. **Simulating the Circuit Evaluation Phase.** For each $\lambda \in [E]$ and each gate G in the λ-th layer of the circuit, the simulation proceeds as follows. Let $\langle u \rangle_{\mathcal{L}_0} = (u_1, \ldots, u_n)$ and $\langle v \rangle_{\mathcal{L}_1} = (v_1, \ldots, v_n)$ be the two input sharings. Let $\mathcal{C}_0 = \mathcal{C} \cap \mathcal{L}_0, \mathcal{H}_0 = \mathcal{L}_0\backslash\mathcal{C}_0, \mathcal{C}_1 = \mathcal{C} \cap \mathcal{L}_1$ and $\mathcal{H}_1 = \mathcal{L}_1\backslash\mathcal{C}_1$. The simulation proceeds as follows.
   - If G is an addition gate, then $\mathcal{S}$ follows the protocol execution to compute the shares of the corrupted parties.
   - If G is a multiplication gate, then the simulation proceeds as follows.
     (a) For each $i \in \mathcal{C}_0$ and $j \in \mathcal{H}_1$, the simulation is straightforward because the inputs of P_i for the OLE functionality are already defined and P_i receives no output.
     (b) For each $i \in \mathcal{H}_0$ and $j \in \mathcal{C}_1$, the simulator $\mathcal{S}$ first samples a random value $\alpha_{i,j}$ and then $\mathcal{S}$ simulates P_i invoking the OLE functionality and returns $\alpha_{i,j}$ to P_j.
     (c) For each $i \in \mathcal{C}_0$ and $j \in \mathcal{C}_1$, $\mathcal{S}$ just follows the protocol execution (all inputs are known to the simulator).

3. **Simulating the Output Recovery Phase.** For each output y belonging to P_i, let $\langle y \rangle_{\mathcal{L}} = (y_1, \ldots, y_n)$ be the sharing of y. Note that after simulating the circuit evaluation phase, the simulator has simulated the shares of the honest parties, i.e., $\{y_j\}_{j \in \mathcal{H} \cap \mathcal{L}}$. The simulation proceeds as follows.

   - If $i \in \mathcal{H}$, the simulation is straightforward because the inputs of honest parties for the sum functionality $\mathcal{F}_{\mathsf{sum}}^{\mathcal{L},i}$ are already defined and the corrupted parties receive no output.
   - If $i \in \mathcal{C}$, then $\mathcal{S}$ knows the value y. $\mathcal{S}$ simulates the honest parties invoking the sum functionality $\mathcal{F}_{\mathsf{sum}}^{\mathcal{L},i}$ and returns y to P_i.

It remains to show that no environment $\mathcal{Z}$ can distinguish between the simulated and the real execution. We will prove that our simulation is perfect, which means that the view of the corrupted parties in the real execution and that in the simulated execution are identical.

The difference between the real execution and simulated execution is in the computation of multiplication gates and the execution of the output recovery phase. For the computation of a multiplication gate, in the real execution, each new message received by the corrupted parties is of form $uv + r$ where u is a share of some honest party P_i, r is a uniformly random value sampled by P_i, and v is a share of some corrupted party. In the simulated execution, each new message received by the corrupted parties is simulated with a freshly sampled random value r'. It is obvious that the distributions of $uv + r$ and r' are identical. For the execution of the output recovery phase, the corrupted parties obtain the same outputs in both the real execution and simulated execution (which is guaranteed by the correctness of our protocol). Therefore, the view of the corrupted parties in the real execution and that in the simulated execution are identical, which implies that no environment $\mathcal{Z}$ can distinguish between the simulated and the real executions. □

I.2 Security Proof of Theorem 6.1

Proof. The correctness of Protocol 6.1 is easy to verify. Now we proceed to prove the security. Let P_i be the corrupted party. We need to construct a simulator that simulates the view of P_i given a trusted party P for computing f. The simulator first gives its inputs to the trusted party P and receives the outputs. The simulator $\mathcal{S}$ simulates the protocol execution as follows.

1. **Simulating the Input Sharing Phase.** We need to consider the following three cases.

   - If P_i is the input owner, $\mathcal{S}$ just receives values from P_i.
   - If the input is held by a single party P_j ($j \neq i$), then the simulator $\mathcal{S}$ just simulates P_j sending a random value to P_i.
   - If the input is held by the two parties P_{i-1}, P_{i+1}, then the simulator $\mathcal{S}$ does nothing because there is no interaction.

2. **Simulating the Circuit Evaluation Phase.** If G is an addition gate, then $\mathcal{S}$ follows the protocol execution. If G is a multiplication gate, we consider three cases.

   - If some input of the gate is an input held by P_i and another party P_j, then $\mathcal{S}$ first simulates P_j invoking the functionality $\mathcal{F}_{\mathsf{cr}}^{i,j}$ two times and returns s, r to P_i. Then $\mathcal{S}$ follows the protocol execution.
   - If some input of the gate is an input held by P_{i-1} and P_{i+1}, then $\mathcal{S}$ samples two random values s, r. Then $\mathcal{S}$ simulates P_{i-1} sending s to P_i and simulates P_{i+1} sending r to P_i.
   - If both inputs of G are not held by two parties (but may be held by a single party), then $\mathcal{S}$ first simulates the honest parties invoking the functionality $\mathcal{F}_{\mathsf{zero}}$ and returns the output r to P_i. Then $\mathcal{S}$ samples a random value r' and simulates P_{i-1} sending r' to P_i.

3. **Simulating the Output Recovery Phase.** Let z be the output received from P and (z_{i-1}, z_{i+1}) be the simulated share of P_i, then $\mathcal{S}$ computes $z_i = z - z_{i-1} - z_{i+1}$ and simulates P_{i-1} sending z_i to P_i.

Now we show that no environment $\mathcal{Z}$ can distinguish between the simulated and real executions. We will show that the simulation is perfect, meaning that the simulated execution and the real execution are perfectly indistinguishable.

In the input sharing phase, only when the input is held by a single party P_j for $j \neq i$, P_i receives messages from the honest parties. Note that in both the real and simulated executions, P_i receives a random value, hence the simulation for the input sharing phase is perfect.

In the circuit evaluation phase, only when computing a multiplication gate, P_i will receive messages. For a multiplication gate G, if some input of G is held by P_i and another party P_j, then in both the simulated and real executions, P_i receives two random values. If some input of G is held by P_{i-1} and P_{i+1}, then P_i receives two random values in the simulated execution, while in the real execution, P_i receives two values of form $z_1 = xy_1 - r - s, z_2 = xy_2 + xy_3 + r$. Note that s, r are two random values that are not known to P_i, hence z_1, z_2 are indeed two random values, which implies the simulation is perfect. The final case is that both inputs of G are not held by more than one party. In the simulated execution, P_i receives a random value from the functionality $\mathcal{F}_{\mathsf{zero}}$ and a random value from P_{i-1}, while in the real execution, P_i receives a random value from the functionality $\mathcal{F}_{\mathsf{zero}}$ and a value of form $z + r$ where z is a value known to P_{i-1}, and r is a random value that is not known to P_i. It is easy to see that in both the simulated and real executions, P_i receives two random values, therefore the simulation is perfect.

Finally, we consider the output recovery phase. For the output sharing, let (z_{i-1}, z_{i+1}) be the share of P_i in the real execution and (z'_{i-1}, z'_{i+1}) be the share of P_i in the real execution. Due to that the simulation for the input sharing and circuit evaluation phases are perfect, the distributions of (z_{i-1}, z_{i+1}) and (z'_{i-1}, z'_{i+1}) are identical. In the output recovery phase, P_i receives $z_i = z - z_{i-1} - z_{i+1}$ from P_{i-1} in the real execution, while in the simulated execution, P_i receives $z'_i = z - z'_{i-1} - z'_{i+1}$ from P_{i-1}. Since the distributions of (z_{i-1}, z_{i+1}) and (z'_{i-1}, z'_{i+1}) are identical, the distributions of z_i and z'_i are identical. Therefore, the simulation for the output recovery phase is perfect. □

I.3 Security Proof of Theorem B.1

Proof. To prove the security, we build a simulator $\mathcal{S}$ such that the environment cannot distinguish between the simulated and the real execution. $\mathcal{S}$ works as follows.

1. **Simulating Initialization.** $\mathcal{S}$ just emulates the $\mathcal{F}_{\mathsf{prep}}$ because it knows the computed circuit.
2. **Simulating Input.** If the input owner P_i is honest, $\mathcal{S}$ simulates P_i with a dummy input (e.g., 0). If the input owner is corrupted, $\mathcal{S}$ waits for a value σ from P_i and extracts $x = \sigma + r$ (r is the masked value for this input). Then $\mathcal{S}$ sends x to $\mathcal{F}_{\mathsf{mpc}}$ as the input of P_i.
3. **Simulating Addition.** $\mathcal{S}$ simply follows the protocol execution.
4. **Simulating Multiplication.** Computing a multiplication gate requires the parties to partially open two sharings. For a sharing $[\![\alpha]\!]_{\mathcal{L}_0}$, if all parties in $\mathcal{L}_0$ are corrupted, $\mathcal{S}$ simply execute the partial open because it knows all the shares. If some party in $[\![\alpha]\!]_{\mathcal{L}_0}$ is honest, then $\mathcal{S}$ simulates the honest parties opening random shares.
5. **Simulating Output.** In the output phase, $\mathcal{S}$ first calls $\mathcal{F}_{\mathsf{mpc}}$ to obtain the correct output z. Next, $\mathcal{S}$ simulates the honest parties running Procedure C.5. If some check does not pass, $\mathcal{S}$ sends Abort to $\mathcal{F}_{\mathsf{mpc}}$, otherwise, $\mathcal{S}$ modify the shares of the honest parties such that these shares are consistent with the output z ($\mathcal{S}$ can do this because it knows the global key Δ). Then $\mathcal{S}$ simulates the honest parties running Procedure C.5 with the adversary. If the check fails, $\mathcal{S}$ sends Abort to $\mathcal{F}_{\mathsf{mpc}}$, otherwise it sends Success.

Now we show that the environment $\mathcal{Z}$ cannot distinguish between the simulated and the real executions. Firstly, note that the honest parties always send uniformly random values, which is implied by two points. Firstly, in the input phase, an honest input owner broadcasts a value $\sigma = x - r$ where r is a uniformly random value (so σ is uniformly random). Secondly, in the multiplication phase, each opened sharing $[\![\alpha]\!]_{\mathcal{L}_0}$ is uniform in the sense any $|\mathcal{L}_0| - 1$ shares are independent of the shared value α (the reason is that $[\![\alpha]\!]_{\mathcal{L}_0} = [\![x]\!]_{\mathcal{L}_0} - [\![a]\!]_{\mathcal{L}_0}$ and $[\![a]\!]_{\mathcal{L}_0}$ is uniform). Finally, let us consider the output phase. Note that in both the

simulated and the real executions, the probability that Procedure C.5 outputs Abort is the same. If all checks pass, then the environment $\mathcal{Z}$ sees the right output z (with the corresponding shares of the honest parties) in the simulated execution. We only need to show that $\mathcal{Z}$ sees the right output z in the real execution with overwhelming probability. As we mentioned in Section B.1, if the check passes, then the opened value $\bar{z}$ is equal to the shared value z with overwhelming probability. This implies that the environment $\mathcal{Z}$ sees the same distribution in both the simulated and the real executions. The proof is completed. □

J Missing Mathematical Proofs

J.1 Proof of Claim 4.4

Proof. It is obvious that $|\mathcal{L}_{\lambda,k}| \leq n$. We only need to prove that $|\mathcal{L}_{\lambda,k}| \leq 2^\lambda$ holds for any $\lambda \in [E], k \in [C_\lambda]$. Let $\mathcal{L}^0_{\lambda,k}, \mathcal{L}^1_{\lambda,k}$ be the lazy sets of the two input sharings of $G_{\lambda,k}$. Let us prove the claim using mathematical induction.

If $\lambda = 1$, then for each $k \in [C_1]$, we have $|\mathcal{L}^0_{1,k}| = |\mathcal{L}^1_{1,k}| = 1$ because each value inputs to $G_{1,k}$ is an input from some party, which follows that $|\mathcal{L}_{1,k}| = |\mathcal{L}^0_{1,k} \cup \mathcal{L}^1_{1,k}| \leq 2$. Assume that $|\mathcal{L}_{\lambda,k}| \leq 2^\lambda$ holds for any $\lambda \leq l$ and $k \in [C_\lambda]$, then we only need to prove that for any $k \in [C_{l+1}]$, it holds that $|\mathcal{L}_{l+1,k}| \leq 2^{l+1}$. Since $G_{l+1,k}$ is a gate in the $(l+1)$-th layer of f, there exists a bit b such that $\mathcal{L}^b_{l+1,k}$ is the lazy set of the output sharing of some gate in the l-th layer of f, and $\mathcal{L}^{1-b}_{l+1,k}$ is the lazy set of the output sharing of some gate in the l'-th layer of f for some $l' \leq l$, which implies that both $|\mathcal{L}^b_{l+1,k}|$ and $|\mathcal{L}^{1-b}_{l+1,k}|$ are no more than 2^l. Therefore, we have

$$|\mathcal{L}_{l+1,k}| = |\mathcal{L}^b_{l+1,k} \cup \mathcal{L}^{1-b}_{l+1,k}| \leq |\mathcal{L}^b_{l+1,k}| + |\mathcal{L}^{1-b}_{l+1,k}| \leq 2^{l+1}.$$

The proof is completed. □

J.2 Proof of Claim 4.5

Proof. Note that $|\mathcal{L}_0 \cap \mathcal{L}_1| + |\mathcal{L}_0 \cup \mathcal{L}_1| = |\mathcal{L}_0| + |\mathcal{L}_1|$, hence we have

$$|\mathcal{L}_0| \cdot |\mathcal{L}_1| - |\mathcal{L}_0 \cap \mathcal{L}_1| = (|\mathcal{L}_0| - 1) \cdot (|\mathcal{L}_1| - 1) + |\mathcal{L}_0 \cup \mathcal{L}_1| - 1.$$

By Claim 4.4, both $|\mathcal{L}_0|$ and $|\mathcal{L}_1|$ are no more than $\mathsf{min}\{2^{\lambda-1}, n\}$, which implies that $(|\mathcal{L}_0| - 1) \cdot (|\mathcal{L}_1| - 1) \leq \mathsf{min}\{(2^{\lambda-1} - 1)^2, (n-1)^2\}$ and $|\mathcal{L}_0 \cup \mathcal{L}_1| \leq \mathsf{min}\{2^\lambda, n\}$. Therefore, we have

$$\begin{aligned}
&|\mathcal{L}_0| \cdot |\mathcal{L}_1| - |\mathcal{L}_0 \cap \mathcal{L}_1| \\
\leq &\mathsf{min}\{(2^{\lambda-1} - 1)^2, (n-1)^2\} + \mathsf{min}\{2^\lambda, n\} - 1 \\
= &\mathsf{min}\{4^{\lambda-1}, 4^{\lambda-1} + n - 2^\lambda, n^2 - n\}.
\end{aligned}$$

This completes the proof. □

J.3 Proof of Corollary 1

Proof. By Claim 4.4, if f has constant MI depth, then for each multiplication gate G in f, the lazy sets of its input sharings have constant sizes. By Claim 4.5, we know that the communication cost of computing G is $O(1)$ calls to the OLE functionality. On the other hand, the communication cost of recovering an output is $O(n^2)$ field elements. Note that there are C_{mul} multiplication gates and M_{out} outputs, the total cost is $O(C_{\mathsf{mul}})$ calls to the OLE functionality and $O(M_{\mathsf{out}} \cdot n^2)$ field elements of communication.

Moreover, if f has constant depth (which implies that f has constant MI depth), then by Claim 4.4, the lazy set of the output sharing of each output gate in the circuit f has constant size, which implies recovering an output y only requires $O(1)$ field elements communication. Therefore, the total cost is $O(C_{\mathsf{mul}})$ calls to the OLE functionality and $O(M_{\mathsf{out}})$ field elements of communication. □

J.4 Proof of Claim 5.1

Proof. If $\mathcal{L}_0 \cap \mathcal{L}_1 = \varnothing$ (which implies that $n \geq 2$ and $\tau = 1$), then we have

$$3n - 2\tau + |\mathcal{L}_0 \cap \mathcal{L}_1| = 3n - 2 \leq 4(n-1).$$

And if $\mathcal{L}_0 \cap \mathcal{L}_1 \neq \varnothing$ (which implies that and $\tau = 2$), then we have

$$3n - 2\tau + |\mathcal{L}_0 \cap \mathcal{L}_1| \leq 3n - 4 + n = 4(n-1).$$

This completes the proof. □