A Communication-Efficient Local-Verification Framework for Maliciously Secure MPC with a Two-Thirds Honest Majority*

Hanchao Ku[1(✉)], Hikaru Tsuchida[2], Mingwu Zhang[3], and Takashi Nishide[1]

[1] University of Tsukuba, Tsukuba, Japan
s2330141@u.tsukuba.ac.jp, nishide@risk.tsukuba.ac.jp
[2] Saitama Institute of Technology, Saitama, Japan
h_tsuchida@sit.ac.jp
[3] Hubei University of Technology, Wuhan, China
csmwzhang@gmail.com

Abstract. Secure Multi-Party Computation (MPC) is a cryptographic primitive that enables multiple parties to jointly compute a function over their inputs without revealing the inputs. An MPC protocol is required to provide security against adversarial behavior, typically considered in two classic models: the *semi-honest* model, where adversaries follow the protocol but attempt to learn additional information from the transcript, and the *malicious* model, where adversaries may arbitrarily deviate from the protocol. Protocols secure against semi-honest adversaries are often more efficient, but in many real-world applications the stronger guarantee of malicious security is required.
In this work, we propose an efficient MPC protocol secure against static malicious adversaries controlling at most $t < n/3$ out of n parties. Our protocol builds on Shamir's secret sharing and follows a compiler-based approach: the parties first evaluate the circuit using a semi-honest protocol and then run a verification procedure that enables honest parties to detect cheating except with small statistical error. Our construction injects fresh randomness into the verification procedure to detect cheating even in the presence of incorrect multiplication triples. It also reduces communication overhead by replacing several invocations of multiplication verification with local degree-$2t$ computations and a batched opening, while keeping the local verification cost comparable to or lower than that of prior protocols. We give a concrete instantiation of the compiler and prove that the resulting protocol is secure against malicious adversaries.

Keywords: MPC · Privacy-Preserving computation · Shamir's Secret Sharing.

1 Introduction

Secure Multi-Party Computation (MPC) enables multiple parties to perform distributed computations without revealing their private data. MPC protocols typically provide security properties such as privacy (only the output is revealed), correctness (the computation and output are accurate), input independence (each party chooses its own input), fairness (all parties receive the output simultaneously), and guaranteed output delivery (all parties always receive the output).

* A short preliminary version of this work will appear in the proceedings of ICICS 2026, Springer [22]. Compared with the proceedings version, this expanded full version includes a more detailed security proof, expanded descriptions of the subprotocol instantiations, and microbenchmarks of the local computation cost in the verification phase.

These properties enable MPC to be applied in various scenarios, such as privacy-preserving machine learning [27,24] and electronic auctions [30], where data security is critical. The security of these MPC protocols is formally proven by demonstrating that they emulate an ideal execution with an incorruptible trusted party computing the function [23,6].

Since Yao [28] introduced MPC, it has been proven that any function can be securely computed. In the computational setting, secure computation is achievable for any $t < n$ (where n is the total number of parties and t is the number of parties an adversary may corrupt) [18,29]. In the information-theoretic setting, it is possible for $t < n/2$ with a broadcast channel [26] and for $t < n/3$ without one [4,6].

However, in real-world applications, efficiency is often as critical as security. For instance, in privacy-preserving machine learning, efficiency is essential alongside security [20], making the enhancement of MPC protocols' efficiency crucial for practical adoption.

A major performance bottleneck in many MPC protocols is the verification of multiplication gates under malicious security. Ensuring that a multiplication result is correctly computed while maintaining low communication overhead remains a fundamental challenge in secret-sharing based MPC. One widely used approach for achieving malicious security is based on message authentication codes (MACs), as introduced in the SPDZ framework [14]. In such protocols, each secret share is authenticated with a MAC under a global secret key, enabling the parties to detect incorrect computations when values are opened. This approach has proven highly effective, particularly in the dishonest-majority setting.

In contrast, MPC protocols in the honest-majority setting often rely on algebraic consistency checks over secret sharings rather than MAC authentication. These protocols typically combine efficient semi-honest computation with additional verification procedures to detect malicious behavior. Our work follows this line of research and focuses on improving the efficiency of such verification mechanisms in Shamir secret-sharing based MPC protocols.

Contribution. In this work, we propose an efficient protocol secure against static malicious adversaries controlling up to $t < n/3$ parties.

Our approach follows a standard compiler paradigm: a semi-honest multiplication protocol is first executed to compute candidate multiplication results, and a lightweight verification procedure is then applied to detect malicious deviations with high probability.

Based on Shamir's Secret Sharing (SSS), our protocol extends semi-honest multiplication protocols for $t < n/3$ by incorporating an efficient verification step that provides security against malicious adversaries. Our protocol is designed for *arithmetic circuits* over a finite field $\mathbb{F}$, and evaluates the circuit using a semi-honest multiplication protocol augmented with our verification procedure.

We employ Beaver triples (a, b, c) with $c = a \cdot b$ [2] as part of the verification mechanism. Our framework is compatible with several semi-honest multiplication protocols, including the DN protocol [13] and the GRR protocol [17]. Our verification step builds on the framework of [23], which achieves security for $t < n/2$. However, in contrast to [23], our protocol introduces fresh randomness to detect cheating even when incorrect triples are used and leverages *local* computation to reduce communication costs, at the expense of lowering the corruption threshold to $t < n/3$.

We present an instantiation based on Shamir secret sharing that integrates a semi-honest multiplication protocol such as DN or GRR with our verification procedure to achieve malicious security with improved efficiency.

Related Work. Currently, in the honest-majority setting, two main threshold conditions are considered: $t < n/2$ and $t < n/3$.

Protocols designed for $t < n/2$ generally provide stronger resilience against adversarial corruption, whereas protocols for $t < n/3$ often allow more efficient constructions. In addition to secret-sharing based protocols, another major line of research achieves malicious security using MAC-based authentication. The SPDZ framework introduced by Damgård et al. [14] attaches a message authentication code to each secret share and verifies correctness when values are opened. This paradigm has inspired a large body of follow-up work and is particularly effective in the dishonest-majority model.

Due to the different adversarial models and design goals, direct efficiency comparisons between MAC-based MPC protocols and honest-majority secret-sharing protocols are generally not meaningful. We therefore focus on prior work most closely related to our approach, namely secret-sharing based MPC protocols in the honest-majority setting.

For example, in the $t < n/2$ setting, [19], [11], and [23] propose protocols for efficiently verifying multiplication gates. In particular, [23] presents a highly efficient verification method that even outperforms certain protocols designed for the $t < n/3$ setting [5]. More recently, [7] further improves verification efficiency in the $t < n/2$ setting.

Protocols with threshold $t < n/3$ remain highly relevant in practice since they often achieve better efficiency. For example, the protocols in [13,3] provide malicious security with unconditional guarantees. Furukawa and Lindell [15] further demonstrated that two-thirds honest-majority protocols can achieve near semi-honest efficiency while maintaining malicious security. Their protocol even outperforms several $t < n/2$ constructions such as [23]. Building on this line of research, our work further improves the efficiency of verification mechanisms while maintaining security in the $t < n/3$ setting.

In secret-sharing based MPC, multiplication can be instantiated using several underlying protocols. Among them, the DN protocol [13] and the GRR protocol [17] are two widely used approaches. Prior works such as [9,10] optimize the DN protocol for efficient multiplication. However, these optimizations still require two communication rounds because an intermediate value must be reconstructed and broadcast.

In contrast, the GRR protocol completes multiplication in a single round by avoiding intermediate reconstruction and instead computing output shares through local linear combinations. This difference in round complexity makes GRR preferable in network environments where latency dominates performance.

These variations highlight the importance of flexible MPC frameworks that can incorporate different multiplication subroutines depending on the deployment environment. Our framework naturally accommodates protocols such as DN and GRR as interchangeable multiplication components while providing an efficient verification layer on top.

2 Preliminaries

2.1 Notation

Let $P_1, \ldots, P_n$ denote the n parties, and we assume that the adversary can corrupt at most t parties. In this work, we assume $t < n/3$. Let H denote the subset of honest parties, and $\mathcal{C}$ denote the subset of corrupted parties. Finally, let $\mathbb{F}$ be a finite field, with $|\mathbb{F}|$ denoting its size.

2.2 Linear Threshold Secret Sharing

A secret sharing scheme with threshold t enables n parties to share a secret $v \in \mathbb{F}$ among n parties so that no subset of t parties can learn any information about the secret, whereas any subset of $t+1$ or more parties can reconstruct it. In this work, we employ **Shamir's Secret Sharing** to construct the protocol. We require that the scheme used in our protocol supports the following procedures:

- **share**(v): In this procedure, a dealer shares a value $v \in \mathbb{F}$. We consider *non-interactive* secret sharing, where the dealer D receives v (with some randomness) and outputs shares $v_1, \ldots, v_n$, where v_i is the share for P_i. We denote the sharing of v by $[v]_t$ and $[v]_{2t}$, requiring $t+1$ and $2t+1$ shares, respectively, to reconstruct v (For convenience, we will only introduce the relevant definition of $[v]_t$ later). If the dealer is corrupted, the shares may be incorrect, but we still use $[v]_t$ to denote them.
- **reconstruct**$([v]_t, i)$: Given a sharing $[v]_t$ and an index i, P_i receives the shares and verifies correctness. If incorrect, P_i outputs $\perp$ and **aborts**; otherwise, it outputs v.
- **open**$([v]_d)$ for $d \in \{t, 2t\}$ (following [15]): all parties broadcast their shares. Each recipient checks whether all received points are consistent with a single polynomial of degree at most d. If the check passes, the recipient outputs the constant term v of this polynomial; otherwise, it outputs $\perp$. We stress that the check is performed on the set of received shares, rather than on an arbitrary subset of only $d+1$ shares, since any $d+1$ points can always be interpolated by a polynomial of degree at most d. For correctness against up to t corrupt parties, the honest shares alone must determine a unique polynomial of degree at most d, and therefore $n-t \geq d+1$ must hold. Thus, opening $[\cdot]_t$ requires at least $t+1$ honest shares (hence $t < n/2$), while opening $[\cdot]_{2t}$ requires at least $2t+1$ honest shares (hence $t < n/3$). Since each party broadcasts one field element to the other parties, the communication cost of one opening is $n-1$ field elements per party. A concrete algorithm for both degrees is given in § 4.
- **local operations**: Given correct sharings $[u]_t$, $[v]_t$ and a scalar $\alpha \in \mathbb{F}$, each party can locally compute $[u+v]_t$, $[\alpha \cdot v]_t$, $[v+\alpha]_t$, and $[u \cdot v]_{2t}$ without interaction. These operations are denoted as $[u]_t + [v]_t$, $\alpha \cdot [v]_t$, $[v]_t + \alpha$, and $[u]_t \cdot [v]_t$, respectively.

Now we define the correctness of a sharing. First, a dealer D shares the output shares with the honest parties. Formally, let $H \subseteq \{P_1, \ldots, P_n\}$ denote the set of honest parties. For an honest subset J of size $d+1$, where $d \in \{t, 2t\}$, let $\mathsf{val}([v]_d)_J$ be the value obtained by the parties in J by running **open**$(\cdot)$. The formal definition of correctness is as follows:

Definition 1 (Correctness of a Sharing). *Let $d \in \{t, 2t\}$ and let $H \subseteq \{P_1, \ldots, P_n\}$ denote the set of honest parties. A sharing $[v]_d$ is correct if there exists a value $v' \in \mathbb{F}$ with $v' \neq \perp$ such that, for every subset $J \subseteq H$ with $|J| = d+1$, it holds that $\mathsf{val}([v]_d)_J = v'$. If no such value exists, then $[v]_d$ is called incorrect.*

For Shamir secret sharing, an incorrect sharing is equivalently value-inconsistent: there exist two honest subsets $J_1, J_2 \subseteq H$ of size $d+1$ such that $\mathsf{val}([v]_d)_{J_1} \neq \mathsf{val}([v]_d)_{J_2}$. Thus, a sharing is correct if and only if all honest subsets of size $d+1$ reconstruct the same value. This notion applies to both degree-t sharings $[v]_t$ and degree-$2t$ sharings $[v]_{2t}$.

We use the following standard facts about linear secret sharing.

Claim ([23]). Let $d \in \{t, 2t\}$. If $[u]_d$ and $[v]_d$ are correct sharings, then for every $\alpha \in \mathbb{F}$, the sharing $[w]_d = \alpha \cdot [u]_d + [v]_d$ is also correct.

Taking the contrapositive, we obtain the following corollary.

Corollary 1. *Let $d \in \{t, 2t\}$, and let $[u]_d$ and $[v]_d$ be sharings. If $[w]_d = \alpha \cdot [u]_d + [v]_d$ is not correct for some $\alpha \in \mathbb{F}$, then at least one of $[u]_d$ and $[v]_d$ is incorrect.*

The following lemma is used for checking correctness.

Lemma 1 ([23]). *Let $d \in \{t, 2t\}$. Let $[u]_d$ be an incorrect sharing of a Shamir secret sharing scheme, and let $[v]_d$ be any sharing. If $\alpha \in_R \mathbb{F} \setminus \{0\}$ is chosen uniformly at random, then the probability that $[w]_d = \alpha[u]_d + [v]_d$ is a correct sharing is at most $\frac{1}{|\mathbb{F}|-1}$.*

2.3 Building Blocks and Subprotocols

This section recalls standard ideal functionalities and some protocols we use as black boxes. Our new protocol and its analysis are given in §3. In particular, we summarize the existing verification approach of [23] (Protocol 4) and the ideal functionalities that will be invoked later.

$\mathcal{F}_{rand}$-Generating Random Shares for Threshold t. We define the ideal functionality $\mathcal{F}_{rand}$ to generate a sharing of a random value where we need $t+1$ honest parties to reconstruct it. The ideal functionality will choose a random value that no party knows. The random shares $[r]_t$ will be generated by the ideal functionality and sent to the parties. The functionality is defined as follows:

FUNCTIONALITY 1 *$\mathcal{F}_{rand}$-Generating Random Shares*

The ideal functionality $\mathcal{F}_{rand}$ chooses a random value $r \in \mathbb{F}$, generates a sharing $[r]_t$ by running **share**(r), and sends the share $[r]_t^{(i)}$ to each party P_i, where $[r]_t^{(i)}$ denotes P_i's share.

$\mathcal{F}_{zero}$-Generating 0-shares for Threshold $2t$. We define the ideal functionality $\mathcal{F}_{zero}$ to generate a sharing $[0]_{2t}$ of value 0 where we need $2t+1$ honest parties to reconstruct it. The functionality is defined as follows:

FUNCTIONALITY 2 *$\mathcal{F}_{zero}$-Generating Random Shares*

The ideal functionality $\mathcal{F}_{zero}$ generates a sharing $[0]_{2t}$ by running **share**(0), and sends the share $[0]_{2t}^{(i)}$ to each party P_i, where $[0]_{2t}^{(i)}$ denotes P_i's share.

Π_{coin}-Generating Random Value. The protocol Π_{coin} allows the parties to obtain a public random value r. A standard way to implement Π_{coin} is through $\mathcal{F}_{rand}$ as given in Protocol 1, where the parties must use **open**$(\cdot)$ to obtain r, requiring interaction among them. Instead, we adopt a Pseudo-Random Function (PRF) [25] approach as in Protocol 2 to implement Π_{coin} more efficiently. This technique is well-known and can securely generate random values.

PROTOCOL 1 *Securely Computing Π_{coin} in the $\mathcal{F}_{rand}$-Hybrid Model*

- **Inputs:** None
- **The protocol:**
  1. The parties call $\mathcal{F}_{rand}$ to generate a random sharing $[r]_t$.
  2. The parties run **open**$([r]_t)$ to receive r. If any party received $\perp$ in the execution, then it sends $\perp$ to the other parties and aborts. Otherwise, it outputs r.
- **Output:** random value r.

PROTOCOL 2 *Securely Efficiently Computing Π_{coin} in Plain Model*

- **Inputs:** *seed* and key s held by all parties in advance.
- **The protocol:**
  1. The parties call $\mathsf{PRF}_s(seed)$ to generate a random r.
- **Output:** random value r.

Correctness Check of Shares. We introduce a method [23] for verifying the correctness of a set of m shares. This is to ensure the correctness of the inputs provided by the parties. This idea is to choose the random coefficients and use them to verify the sharings used in the computation. No party can know the coefficient shares before they are generated. In [23], it is proven that if there is any incorrect sharing, the verification will succeed only with a small probability. The protocol is given in Protocol 3.

PROTOCOL 3 ([23]) *Correctness Check of Shares*

- **Inputs:** The parties hold m shares $[x_1]_t, [x_2]_t, \ldots, [x_m]_t$.
- **The protocol:**
  1. The parties call Π_{coin} to generate the random value $\gamma_1, \gamma_2, \ldots, \gamma_m \in \mathbb{F} \setminus \{0\}$.
  2. The parties call $\mathcal{F}_{rand}$ to obtain a sharing $[r]_t$.
  3. The parties locally compute
  $$[v]_t = \gamma_1[x_1]_t + \gamma_2[x_2]_t + \cdots + \gamma_m[x_m]_t + [r]_t.$$
  4. The parties run **open**$([v]_t)$.
  5. If no **abort** message was received, then the parties output **accept**.

2.4 Triple Verification Based on Open Procedure [23]

Suppose we have a multiplication triple of shares $([a]_t, [b]_t, [c]_t)$, where $c = a \cdot b$. We say that a multiplication triple is *random* if $[a]_t$ and $[b]_t$ are sharings of random values in $\mathbb{F}$. The definition of correctness is as follows:

Definition 2 (Correctness). *$([a]_t, [b]_t, [c]_t)$ is a correct multiplication triple if $[a]_t, [b]_t$ and $[c]_t$ are correct sharings and $c = a \cdot b$.*

We recall the protocol [23] that verifies the correctness of a multiplication triple $([x]_t, [y]_t, [z]_t)$ without revealing its values, while consuming an additional random multiplication triple $([a]_t, [b]_t, [c]_t)$. This protocol uses the same method as [14,12]. The detail is given in Protocol 4. The idea behind Protocol 4 is as follows. In this protocol, we have the shares of x, y, z and a, b, c. First, the parties compute and open the shares of $\rho = \alpha \cdot x + a$ and $\sigma = y + b$, where $\alpha \in_R \mathbb{F} \setminus \{0\}$. Because a and b are random, these values will reveal nothing about x and y. In addition, from the definition of **open**$(\cdot)$, the honest parties must hold the same value for ρ and σ. Here, if $([a]_t, [b]_t, [c]_t)$ is a correct triple and $([x]_t, [y]_t, [z]_t)$ is not, then

$$\alpha \cdot [z]_t - [c]_t + \sigma \cdot [a]_t + \rho \cdot [b]_t - \rho \cdot \sigma$$

is also incorrect. In this case, we can consider that the adversary adds some element $\Delta \neq 0$ such that $x \cdot y + \Delta = z$ to let parties get the incorrect result. This means that honest parties can detect cheating and **abort** based on computing and opening these shares. If $([a]_t, [b]_t, [c]_t)$ is also incorrect,

the multiplication of $[z]_t$ by the random α ensures that the probability that the parties obtain a correct sharing of 0 is bounded by $\frac{1}{|\mathbb{F}|-1}$. The communication cost of this protocol depends on the three invocations of **open**$(\cdot)$. The security of this protocol has been formally proven in [23].

We note that Protocol 4, originally from [23], is included here only for completeness and to aid the reader's understanding of our core idea; it is not part of our proposed protocol. In [23], the authors further improve Protocol 4 by employing π_{mult} (defined in Protocol 9) to replace the costly **open**$(\cdot)$ in Protocol 4, thereby reducing the overall communication complexity from $O(n^2)$ to $O(n)$. In contrast, our Protocol 6 (§3.2) achieves a further improvement by transforming the operations of **open**$(\cdot)$ into purely local computations, thus reducing the communication cost from $O(n^2)$ to zero.

PROTOCOL 4 ([23]) *Triple Verification Based on open procedure*

- **Inputs:** The parties hold a triple $([x]_t, [y]_t, [z]_t)$ to verify, and an additional random triple $([a]_t, [b]_t, [c]_t)$.
- **The protocol:**
  1. The parties call Π_{coin} to generate a random $\alpha \in \mathbb{F} \setminus \{0\}$.
  2. Each party locally computes $[\rho]_t = \alpha \cdot [x]_t + [a]_t$ and $[\sigma]_t = [y]_t + [b]_t$.
  3. The parties run **open**$([\rho]_t)$ and **open**$([\sigma]_t)$. If a party receives $\perp$ in the opening, then it sends $\perp$ to all the other parties and outputs **abort** or **accept** ρ and σ.
  4. Each party locally computes
  $$[v]_t = \alpha \cdot [z]_t - [c]_t + \sigma \cdot [a]_t + \rho \cdot [b]_t - \rho \cdot \sigma.$$
  5. The parties run **open**$([v]_t)$ to receive v. If a party receives $\perp$ in the opening, then it sends $\perp$ to all the other parties and outputs **abort**.
  6. Each party checks that $v = 0$. If not, then it sends $\perp$ to the other parties and **abort**.
  7. If no **abort** messages are received, then the parties output **accept**.

2.5 Security Definition

Malicious adversaries: As in [23], our protocol works with a semi-honest multiplication protocol and adds a verification step to help honest parties detect cheating, thereby achieving security against malicious adversaries. The formulation of this notion is provided in [1]. Most standard linear secret sharing-based semi-honest multiplication protocols can meet this condition.

We proceed to define security mostly following [23]. Let $View_{\mathcal{A},I,\pi}(\vec{x}, \kappa)$ denote the view of the adversary $\mathcal{A}$ where $\mathcal{A}$ controls parties $\{P_i\}_{i\in I}$ (with $I \subset \{1, \ldots, n\}$) in a real execution of n-party protocol π, with input $\vec{x} = (x_1, \ldots, x_n)$ and security parameter κ. A protocol is private in the presence of t malicious corrupted parties if the view of the corrupted parties where the input is $\vec{x}$ is computationally indistinguishable from its view when the input is $\vec{x}'$.

Definition 3 ([23]). *Let $f : \mathbb{F}^n \to \mathbb{F}^n$ be an n-party functionality and let π be an n-party protocol. We say that π t-privately computes f in the presence of malicious adversaries if it is correct and if for every non-uniform probabilistic polynomial-time adversary $\mathcal{A}$, every $I \subset \{1, \ldots, n\}$ with $|I| \leq t$, and every two vectors $\vec{x}, \vec{x}' \in \mathbb{F}^n$, $\{View_{\mathcal{A},I,\pi}(\vec{x}, \kappa)\}_{\kappa\in\mathbb{N}} \stackrel{c}{\equiv} \{View_{\mathcal{A},I,\pi}(\vec{x}', \kappa)\}_{\kappa\in\mathbb{N}}$.*

A protocol is private semi-honest if it is private in the presence of malicious adversaries and secure in the presence of semi-honest adversaries.

Security in the presence of malicious adversaries: In this work, we use the standard ideal-real paradigm to prove the security. Our protocol guarantees **security with abort**.

Security up to additive attacks: In the semi-honest setting, all parties follow the protocol exactly as specified, without performing any additional operations. In contrast, a malicious adversary may deviate from the protocol arbitrarily. Our protocol is secure up to *additive attacks*, meaning that the adversary's ability to cheat is limited to adding a value to the output. Specifically, in the case where exactly $t+1$ parties are honest, consider an ideal functionality for multiplication that receives shares of x, y from honest parties and a value Δ from the adversary, and returns shares of $x \cdot y + \Delta$ as the output [4]. A multiplication protocol is said to be secure up to additive attacks against malicious adversaries if it securely realizes this ideal functionality.

3 Our Protocol

In this section, we present our protocol for verifying multiplication results.

- In §3.1, we describe an efficient and secure method for generating multiple PRF seeds.
- In §3.2, we present an efficient protocol to verify the multiplication based on local operations.
- In §3.3, we introduce an efficient protocol for generating multiplication triples.
- Finally, in §3.4, we combine these components to obtain the main protocol, Protocol 9, which can be applied in the real MPC scenarios.

3.1 Seed Generation for PRF with Delayed Opening

Our verification protocols require several fresh public random values. Instead of invoking **open**$(\cdot)$ for each value as in Protocol 1, the parties derive them from a PRF seed. The seed, however, must be fixed before the adversary learns the verification challenges; otherwise, a rushing adversary[5] may bias the randomness.

Protocol 5 addresses this problem through delayed opening. The parties first generate random values whose sum defines the seed, but keep them hidden until the verification phase. When a fresh seed is needed, they open the corresponding values and derive the required randomness locally via Protocol 2. In this way, L random values can be generated with one preparation phase and one opening phase, reducing the round complexity from $O(L)$ to $O(1)$ in the security-with-abort setting.

Compared to Protocol 1, which requires invoking **open**$(\cdot)$ for each random value, our delayed-opening Protocol 5 requires only a single commit phase and a single opening phase, regardless of the number of values generated. This reduces the round complexity of generating L random values from $O(L)$ to $O(1)$, while preserving the same level of security in the **security with abort** setting.

[4] This will result in all parties receiving a correct/valid sharing of an incorrect value, namely, $[x \cdot y + \Delta]_t$ instead of $[x \cdot y]_t$. In this case, the shares have been maliciously altered while the sharing remains structurally valid.

[5] A rushing adversary can adaptively choose its messages after observing those sent by honest parties in the same communication round, potentially biasing the randomness here.

PROTOCOL 5 *Efficiently Sharing a seed with Delayed Opening*

- **The protocol:**
  1. The n parties choose random values $r_i, r_i' \in \mathbb{F}$.
  2. The n parties compute the commitment $\mathsf{Com}(r_i; r_i')$ to r_i where r_i' is the randomness, and send it to other parties.
  3. **(No immediate opening)** After all parties receive $\mathsf{Com}(r_i; r_i')$, the values r_i, r_i' are *not* opened at this stage. The committed values are logically fixed but remain hidden until the end of the verification phase.
  4. The *seed* is defined as $seed = \sum_{i=1}^{n} r_i$ in $\mathbb{F}$, and is fixed after the commit phase. This *seed* will later be used as the seed/input to a PRF to derive one or more random values in subsequent protocols, where a counter-based input is applied to ensure that each derived value is distinct.
  5. **Opening phase:** When the parties need a new *seed*, the n parties send r_i, r_i' to other parties.
  6. Each party verifies $\mathsf{Com}(r_i; r_i')$. If any verification fails, the parties **abort**.
- **Output:** Multiple *seed*'s or **abort**.

3.2 Triple Verification Based on Local Operations

We present a new protocol for verifying the correctness of a multiplication triple. While its structure resembles Protocol 4 in [23], our protocol differs in that all operations can be performed locally without interaction. In settings requiring stronger security guarantees, the approach of [23] may still be preferable. However, we show that efficiency can be further improved by relaxing the corruption threshold from $t < n/2$ to $t < n/3$. At a high level, the setting $t < n/3$ enables opening a sharing of a multiplication result $[\cdot]_{2t}$ in the verification of multiplication triples (see Step 8 in Protocol 6), which in turn allows us to eliminate the costly invocations of **open**$(\cdot)$ on sharings $[\cdot]_t$ (see Step 3 of Protocol 4 in [23]). While this reduces the security threshold, the trade-off can often be acceptable in practice, particularly when the number of parties n is large since the absolute number of tolerated corruptions remains high. A related technique was also proposed in [15] to reduce the cost of multiplication gates. However, their verification step incurs additional overhead because it requires extra executions of **open**$([v]_t)$, an overhead our protocol entirely avoids.

In Protocol 4, the parties compute shares of

$$v = \alpha \cdot z - c + \sigma \cdot a + \rho \cdot b - \rho\sigma,$$

where $\rho = \alpha x + a$ and $\sigma = y + b$, and verify that it equals 0. In Protocol 4, ρ and σ are obtained by running **open**$(\cdot)$. However, these openings incur high communication cost. To address this, [23] proposed a verification protocol that reduces the number of invocations of **open**$(\cdot)$ by leveraging the semi-honest multiplication protocol.

In this work, we show that verification efficiency can be further enhanced. Unlike the prior approach, which still requires multiple **open**$(\cdot)$ operations and semi-honest multiplication, our protocol keeps ρ and σ secret-shared, computes the verification value locally, and invokes **open**$(\cdot)$ only once to obtain v. Another difference is that we employ triples of the form $([a]_t, [b]_t, [c]_{2t})$.

PROTOCOL 6 *Triple Verification Based on Local Operations*

- **Inputs:** The parties hold a triple $([x]_t, [y]_t, [z]_t)$ to verify (which is typically a result of a semi-honest multiplication protocol), and an additional random triple $([a]_t, [b]_t, [c]_{2t})$.
- **The protocol:**
  1. The parties execute $\mathcal{F}_{zero}$ to generate a sharing $[0]_{2t}$.
  2. The parties execute Π_{coin} to generate a random $\alpha \in \mathbb{F} \setminus \{0\}$.
  3. The parties compute $\alpha \cdot [x]_t$ to obtain $[\alpha \cdot x]_t$.
  4. The parties compute $\alpha \cdot [z]_t$ to obtain $[\alpha \cdot z]_t$.
  5. Each party locally computes $[\rho]_t = [\alpha \cdot x]_t + [a]_t$ and $[\sigma]_t = [y]_t + [b]_t$.
  6. Each party locally computes $[a \cdot \sigma]_{2t} = [a]_t \cdot [\sigma]_t$.
  7. Each party locally computes $[\rho \cdot y]_{2t} = [\rho]_t \cdot [y]_t$.
  8. Each party locally computes
  $$[v]_{2t} = [\alpha \cdot z]_t - [c]_{2t} + [a \cdot \sigma]_{2t} - [\rho \cdot y]_{2t} + [0]_{2t}.$$
  9. The parties run the **open**$([v]_{2t})$ procedure to receive v. If a party receives $\perp$ in the opening, then it sends $\perp$ to all the other parties and **abort**.
  10. Each party checks if $v = 0$. If not, then it sends $\perp$ to the other parties and aborts.
  11. If no **abort** messages are received, then output is **accept**.
- **Output: accept** or **abort**.

Definition 4 (Correctness). $([a]_t, [b]_t, [c]_{2t})$ *is a correct multiplication triple if* $[a]_t, [b]_t$ *and* $[c]_{2t}$ *are correct sharings and* $c = a \cdot b$.

We propose our protocol based on local operations in two steps. First, we note that this protocol still needs one opening for each triple verification; namely, the parties open $[v]_{2t}$ to verify that it equals 0. Next, we extend it by showing how to check that many shares are equal to 0 with a single **open**$(\cdot)$ operation.

The first protocol is given in Protocol 6. In this protocol, the values of ρ and σ are not opened, unlike Protocol 4. Instead, we compute the sharing

$$\alpha \cdot [z]_t - [c]_{2t} + [a]_t \cdot [\sigma]_t - [\rho]_t \cdot [y]_t + [0]_{2t}$$

by local operations where a sharing $[0]_{2t}$ is used to hide the polynomial for the sharing $[v]_{2t}$. To improve the efficiency, we use a degree-$2t$ sharing instead of a degree-t sharing. At the same time, we add a sharing of 0 in this protocol. Thus, all operations here are performed without requiring data transmission between the parties. Although decreasing the threshold reduces security, the resulting efficiency improvement makes it worthwhile. All the details are given in Protocol 6.

Lemma 2. *If* $[x]_t$ *and* $[y]_t$ *are correct shares, and* $([x]_t, [y]_t, [z]_t)$ *is not a correct triple, then:*

1. *If* $([a]_t, [b]_t, [c]_{2t})$ *is a correct triple, then all honest parties* ***abort*** *in Protocol 6 with probability 1.*
2. *If* $([a]_t, [b]_t, [c]_{2t})$ *is not a correct triple, then the honest parties output* ***accept*** *in Protocol 6 with probability at most* $\frac{1}{|\mathbb{F}|-1}$.

Proof. To prove Item (1), suppose that $([a]_t, [b]_t, [c]_{2t})$ is a correct triple. It also means that $c = a \cdot b$. Next, we assume that $[x]_t, [y]_t$ are correct shares, and $[z]_t$ is value-consistent, and that the triple

$([x]_t, [y]_t, [z]_t)$ is not correct. This implies that $x \cdot y \neq z$. We assume $z = x \cdot y + \Delta$ where $\Delta \in \mathbb{F} \setminus \{0\}$. Under these assumptions, all parties compute locally

$$\begin{aligned}[v]_{2t} &= [\alpha \cdot z]_t - [c]_{2t} + [a \cdot \sigma]_{2t} - [\rho \cdot y]_{2t} + [0]_{2t} \\ &= [\alpha(z - x \cdot y) - (c - ab)]_{2t}.\end{aligned}$$

Since $c = a \cdot b$, we have $[v]_{2t} = [\alpha(z - x \cdot y)]_{2t}$. Then, we have $v = \alpha \cdot \Delta \neq 0$ where $\Delta \in \mathbb{F} \setminus \{0\}$, so the honest parties will output **abort** with probability 1. Next, we consider the case where $[0]_{2t}$ is an incorrect sharing. We assume that the input at this time is $[h]_{2t}$ where $h \in \mathbb{F} \setminus \{0\}$. Hence, the honest parties output **accept** with probability at most $\frac{1}{|\mathbb{F}|-1}$. We can notice that, even if $[0]_{2t}$ is not a correct sharing, the parties can detect it in the **open**$(\cdot)$ stage. Therefore, for simplicity, in the following proof, we assume that $[0]_{2t}$ is a correct sharing.

Next we prove Item (2). We consider the case where $[z]_t$ and $[c]_{2t}$ are value-consistent. We note that when $[z]_t, [c]_{2t}$ are not value-consistent, the parties have the output $\perp$ from **open**$([v]_{2t})$ and output **abort**. In this case, we can assume that $z = x \cdot y + \Delta_1$ and $c = a \cdot b + \Delta_2$, where $\Delta_1, \Delta_2 \in \mathbb{F} \setminus \{0\}$, and we have

$$\begin{aligned}[v]_{2t} &= [\alpha \cdot z]_t - [c]_{2t} + [a \cdot \sigma]_{2t} - [\rho \cdot y]_{2t} + [0]_{2t} \\ &= [\alpha \cdot \Delta_1 - \Delta_2]_{2t}.\end{aligned}$$

The honest parties output **accept** if and only if $\alpha \cdot \Delta_1 - \Delta_2 = 0$. In other words, if and only if $\alpha = \Delta_2/\Delta_1$ in Protocol 6, the honest parties output **accept**. However, α is distributed uniformly in $\mathbb{F} \setminus \{0\}$ and independent of Δ_1, Δ_2. Hence, the honest parties output **accept** with probability at most $\frac{1}{|\mathbb{F}|-1}$. Taking all conditions into account, the probability that the protocol ends with the honest parties outputting **accept** is bounded by $\frac{1}{|\mathbb{F}|-1}$.

Improving the efficiency of multiple invocations of open procedures. We note that in Protocol 6, verifying multiple multiplication results requires repeatedly using the open procedure. However, in actual applications, we often need to perform multiplication protocols many times. To handle this situation, we design Protocol 7. The idea behind Protocol 7 is simple: First, we compute all $[v_i]_{2t}$ and prepare the same number of random coefficients. Then we multiply each $[v_i]_{2t}$ by the corresponding coefficient and sum the resulting sharings. Finally, we open the sum, and if any $[v_i]_{2t} \neq 0$, the honest parties will abort with the nonzero opened value. Because the coefficients are chosen at random, the result of $[v]_{2t}$ will not equal 0, except the probability $\frac{1}{|\mathbb{F}|-1}$. The protocol is given in Protocol 7.

Lemma 3. *If $\{[x_i]_t, [y_i]_t\}_{i=1}^L$ are correct but there exists some $k \in [1, L]$ such that $[z_k]_t$ is not value-consistent or $([x_k]_t, [y_k]_t, [z_k]_t)$ is an incorrect triple then:*

1. *If $\{([a_i]_t, [b_i]_t, [c_i]_{2t})\}_{i=1}^L$ are correct triples, then all honest parties* **accept** *in Protocol 7 with probability at most $\frac{1}{|\mathbb{F}|-1}$.*
2. *If $\{([a_i]_t, [b_i]_t, [c_i]_{2t})\}_{i=1}^L$ have at least one $k \in [1, L]$ such that $[c_k]_{2t}$ is not correct, then the honest parties output* **accept** *in Protocol 7 with probability at most $\frac{1}{|\mathbb{F}|-1}$.*

Proof. For Item (1), suppose that $\{[x_i]_t, [y_i]_t\}_{i=1}^L$ are correct, and that there exists an index $k \in [1, L]$ such that $[z_k]_t$ is value-consistent but $([x_k]_t, [y_k]_t, [z_k]_t)$ is not a correct multiplication triple. In other words, $x_k \cdot y_k \neq z_k$. Then, we can assume that $x_k \cdot y_k + \Delta_k = z_k$, where $\Delta_k \in \mathbb{F} \setminus \{0\}$.

PROTOCOL 7 *Batch Verification of Triples Based on Local Operations*[a]

- **Inputs:** The parties hold a list of triples $\{([x_i]_t, [y_i]_t, [z_i]_t)\}_{i=1}^{L}$ to verify and a list of random triples $\{([a_i]_t, [b_i]_t, [c_i]_{2t})\}_{i=1}^{L}$.
- **The protocol:**
  1. The parties call $\mathcal{F}_{zero}$ to generate a sharing $[0]_{2t}$.
  2. For i in range $[1, L]$:
     a. The parties execute Π_{coin} to generate a public random value $\alpha_i \in \mathbb{F} \setminus \{0\}$.
     b. The parties compute $\alpha_i \cdot [x_i]_t$ to obtain $[\alpha_i \cdot x_i]_t$.
     c. The parties compute $\alpha_i \cdot [z_i]_t$ to obtain $[\alpha_i \cdot z_i]_t$.
     d. Each party locally computes $[\rho_i]_t = [\alpha_i \cdot x_i]_t + [a_i]_t$ and $[\sigma_i]_t = [y_i]_t + [b_i]_t$.
     e. Each party locally computes $[a_i]_t \cdot [\sigma_i]_t = [a_i \cdot \sigma_i]_{2t}$.
     f. Each party locally computes $[\rho_i]_t \cdot [y_i]_t = [\rho_i \cdot y_i]_{2t}$.
     g. Each party locally computes
     $$[v_i]_{2t} = [\alpha_i \cdot z_i]_t - [c_i]_{2t} + [a_i \cdot \sigma_i]_{2t} - [\rho_i \cdot y_i]_{2t}.$$
  3. The parties execute Π_{coin} to generate the random values $\gamma_1, \gamma_2, \ldots, \gamma_L \in \mathbb{F} \setminus \{0\}$.
  4. The parties locally compute
     $$[v]_{2t} = \gamma_1 \cdot [v_1]_{2t} + \cdots + \gamma_L \cdot [v_L]_{2t} + [0]_{2t}.$$
  5. The parties run the **open**$([v]_{2t})$ procedure to receive v. If a party receives $\perp$ in the opening, then it sends $\perp$ to all the other parties and **abort**.
  6. Each party checks if $v = 0$. If not, then it sends $\perp$ to the others and aborts.
  7. If no **abort** messages are received, then output is **accept**.
- **Output: accept** or **abort**.

[a] We note that this protocol is typically executed only once near the end of the entire circuit evaluation (Protocol 9).

Under these assumptions, if the honest parties output **accept** at the end of the protocol, then in the **open**$(\cdot)$ procedure, $[v]_{2t}$ represents a correct sharing of 0. However, the parties will obtain:

$$\begin{aligned}[v]_{2t} &= \gamma_1 \cdot [v_1]_{2t} + \cdots + \gamma_L \cdot [v_L]_{2t} + [0]_{2t} \\ &= \gamma_k \cdot ([\alpha_k \cdot z_k]_t - [c_k]_t + [a_k \cdot \sigma_k]_{2t} - [\rho_k \cdot y_k]_{2t}) \\ &= \gamma_k \cdot ([\alpha_k \cdot (z_k - x_k \cdot y_k)]_{2t})\end{aligned}$$

Since $z_k = x_k \cdot y_k + \Delta_k$, it follows that $v = \gamma_k \cdot (\alpha_k \cdot \Delta_k)$, where γ_k and α_k are uniformly distributed in $\mathbb{F} \setminus \{0\}$ and independent of Δ_k. Thus, the honest parties will output **accept** with probability 0, implying that all honest parties will **abort** in Protocol 7 with probability 1.

Next, we consider the case where at least one multiplication triple is incorrect. For simplicity, assume that all multiplication triples are incorrect. Let $x_i \cdot y_i + \Delta_i = z_i$ for each $i \in [1, L]$ with $\Delta_i \in \mathbb{F} \setminus \{0\}$. Then, the parties will obtain

$$[v]_{2t} = \gamma_1 \cdot (\alpha_1 \cdot \Delta_1) + \cdots + \gamma_L \cdot (\alpha_L \cdot \Delta_L)$$

where $\gamma_1, \ldots, \gamma_L$ and $\alpha_1, \ldots, \alpha_L$ are uniformly distributed in $\mathbb{F} \setminus \{0\}$ and independent of $\Delta_1, \ldots, \Delta_L$. Thus, the honest parties will output **accept** with probability $\frac{1}{|\mathbb{F}|} < \frac{1}{|\mathbb{F}|-1}$.

For Item (2), let $[u]_{2t} = \sum_{i \in [1,m] \setminus \{k\}} \gamma_i \cdot [v_i]_{2t} + [0]_{2t}$. Then we can get $[v]_{2t} = \gamma_k \cdot [v_k]_{2t} + [u]_{2t}$. Thus, by Lemma 1, the probability that $[v]_{2t}$ is a correct sharing of 0 is at most $\frac{1}{|\mathbb{F}|-1}$.

3.3 Set-up protocol

The set-up protocol is given in Protocol 8. In our protocol, the multiplication result $[c]_{2t}$ (in a Beaver triple) is computed locally. Since the verification in Protocols 6 and 7 requires only random triples of the form $([a_i]_t, [b_i]_t, [c_i]_{2t})$, which does not need to be correct, these triples can be generated efficiently through local computation.

For clarity, in the main protocol, the parties require multiple sets of triples. Specifically, the parties need δ sets of triples (as justified in the following proof). For clarity in notation, let $\vec{d_i} = \{([a_k^{(i)}]_t, [b_k^{(i)}]_t, [c_k^{(i)}]_{2t})\}_{k=1}^N$, where $i = 1, \ldots, \delta$.

PROTOCOL 8 *Generating Random Multiplication Triples*

- **Inputs:** The parties have the number N of triples to generate.
- **The protocol:**
  1. The parties call $\mathcal{F}_{rand}$ to obtain $2N$ random sharings, arranged in a list of the form $\{([a_i]_t, [b_i]_t)\}_{i=1}^N$.
  2. For $i = 1$ to N, the parties compute $[c_i]_{2t} = [a_i]_t \cdot [b_i]_t$ to get $[c_i]_{2t}$.
- **Output:** The parties output $\{([a_i]_t, [b_i]_t, [c_i]_{2t})\}_{i=1}^N$.

3.4 Main protocol

Now, we introduce the main protocol. In the main protocol, every party inputs their own input and computes an arithmetic circuit. This protocol works by evaluating the circuit using a semi-honest protocol and then running a verification step at the end of the entire circuit evaluation. This verification will use the random triples from the offline phase and run after all multiplication gates are over. In the verification step, the parties all compute locally. This means that the parties do not exchange any information during the verification step. In this condition, suppose the additive attack happens in the multiplication gate. The verification will help the parties to detect the attack efficiently. The full protocol is given in Protocol 9.

We note that Protocol 8 does not require an intermediate well-formedness check for $[c]_{2t}$. Even if $[c]_{2t}$ is *not* a valid degree-$2t$ sharing (i.e., not value-consistent), this will be detected by the standard **open** subroutine invoked at the end (Step 5 of Protocol 9): during reconstruction, the opened shares are checked to interpolate to a unique polynomial of degree at most $2t$; otherwise inconsistency is detected and the protocol aborts in our Security-with-Abort setting. Hence, both semantic errors ($c \neq a \cdot b$) and structural errors (invalid $[c]_{2t}$) are caught without any per-gate interaction.

We also note that Step 5 of Protocol 9 must use a *fresh seed*. As established in the proof of Protocol 6, if α (e.g., generated via Protocol 2) were not derived from a fresh *seed* unknown to the adversary until that step, then a rushing adversary, having already fixed Δ when $([x]_t, [y]_t, [z]_t)$ is prepared, could adaptively choose h such that $\alpha = -h/\Delta$, where $x \cdot y = z + \Delta$, thereby invalidating the argument. In general, the verification phase requires a fresh *seed*, as in Protocol 7[6].

Now we will prove the security of Protocol 9.

[6] When the verification phase is executed multiple times separately, Protocol 5 is advantageous because the parties can pre-commit to a small pool of *seed*'s in a single round and, at Step 5, simply *open the corresponding commitment* to reveal a new *seed* and derive α locally via the PRF, thus preserving $O(1)$ rounds and low communication.

PROTOCOL 9 *Computing an Arithmetic Circuit Over Finite Fields (where π_{mult} is a semi-honest multiplication protocol)*

- **Inputs:** Each party P_j ($j \in \{1, \ldots, n\}$) holds an input $x_j \in \mathbb{F}^\ell$.
- **Auxiliary Input**: The parties hold the statistical security parameter σ and the description of an arithmetic circuit C that computes f on inputs of length $\ell \cdot n$.
- **The protocol:**
  (1) *Precomputation*[a]: Each party sets δ to be the smallest value such that $\left(\frac{1}{|\mathbb{F}|-1}\right)^\delta \leq 2^{-(\sigma+1)}$[b], where σ is the statistical security parameter (see Theorem 1), and the parties run δ executions of Protocol 8 with input N where N is the total number of multiplication gates, and obtain vector $\vec{d}_1, \ldots, \vec{d}_\delta$ (see Step 5).
  (2) *Sharing the input*: For each input wire with an input v, the parties run **share**(v) to get the sharing.
  (3) *Correctness checking of inputs*: Let $[v_1]_t, \ldots, [v_m]_t$ be the shares on the input wires generated by the previous step.
  The parties run Protocol 3 δ times on $[v_1]_t, \ldots, [v_m]_t$. If there exists an execution in which a party does not output **accept**, it sends $\perp$ to the other parties and halts.
  (4) *Circuit emulation*: Let $G_1, \ldots, G_L$ be a predetermined topological ordering of the gates of the circuit. For $k = 1, \ldots, L$ the parties work as follows:
  - *If G_k is an addition gate*: Given shares $[x]_t$ and $[y]_t$ on the input wires, the parties locally compute $[x + y]_t$.
  - *If G_k is a multiplication-by-a-constant gate*: Given share $[x]_t$ on the input wire and a public constant $a \in \mathbb{F}$, the parties locally compute $[a \cdot x]_t$.
  - *If G_k is a multiplication gate*: Given shares $[x]_t$ and $[y]_t$ on the input wires, the parties run a semi-honest multiplication π_{mult} (see C) on $[x]_t$ and $[y]_t$, and define the result $([z]_t)$ as their shares on the output wire.
  (5) *Verification stage (this stage is executed after all the gates of the circuit are processed*[c]*)*[d]: If there exists the multiplication gate, let $\{([x_k]_t, [y_k]_t, [z_k]_t)\}_{k=1}^N$ be the triples generated by computing multiplication gates, and let $\vec{d}_i = \{([a_k^{(i)}]_t, [b_k^{(i)}]_t, [c_k^{(i)}]_{2t})\}_{k=1}^N$ be the triples generated in i-th iteration of the offline phase. For $i = 1$ to δ, the parties perform as follows:
  The parties run Protocol 7 on $\{([x_k]_t, [y_k]_t, [z_k]_t)\}_{k=1}^N$ and $\{([a_k^{(i)}]_t, [b_k^{(i)}]_t, [c_k^{(i)}]_{2t})\}_{k=1}^N$ to verify $\{([x_k]_t, [y_k]_t, [z_k]_t)\}_{k=1}^N$. If a party does not output **accept**, it sends $\perp$ to the other parties and output $\perp$.
  (6) If any party received $\perp$ in any of the previous steps, then it outputs $\perp$ and halts.
  (7) *Output reconstruction*: For each output wire of the circuit, the parties run **reconstruct**$([v]_t, j)$, where $[v]_t$ is the sharing of the value on the output wire, and P_j is the party whose output is on the wire.
  (8) If a party received $\perp$ in any call to the reconstruction procedure, then it sends $\perp$ to the other parties, outputs $\perp$, and halts.
- **Output:** If a party has not output $\perp$, then it outputs the values it received on its output wires.

[a] In this step, if the parties choose Protocol 2 to generate random values, each party needs to call Protocol 5 to prepare multiple *seed*'s.

[b] The preliminary ICICS 2026 version [22] used the condition $\left(\frac{1}{|\mathbb{F}|-1}\right)^\delta \leq 2^{-\sigma}$, in line with the proof treatment of [23]. In this version, we instead require the slightly stronger bound $\left(\frac{1}{|\mathbb{F}|-1}\right)^\delta \leq 2^{-(\sigma+1)}$, so that the two soundness-error contributions, from the input-correctness check and from the multiplication verification stage, are each bounded by $2^{-(\sigma+1)}$. A union bound then gives total soundness error at most $2^{-\sigma}$.

[c] It is also possible to execute the verification phase incrementally at intermediate points during the computation, thereby verifying the multiplications performed up to that stage.

[d] If the parties choose Protocol 2 to generate random values in Step 3, this step requires a new *seed*.

Theorem 1. *Let f be an n-party functionality represented by an arithmetic circuit over $\mathbb{F}$. Let π_{mult} be a correct multiplication protocol that is private and secure up to additive attacks. Assume that the public random challenges used in Protocols 3 and 7 are fresh and independent of the adversary's previous messages.*[7] *Let $p = \frac{1}{|\mathbb{F}|-1}$. Let δ be chosen as in Step 1, namely such that $p^\delta \leq 2^{-(\sigma+1)}$, where σ is the statistical security parameter. Then Protocol 9, where Protocol 3 is performed with δ independent random linear checks and Protocol 7 is executed δ times in the verification stage, securely computes f with abort in the $(\mathcal{F}_{zero}, \mathcal{F}_{rand})$-hybrid model, with statistical error at most $2^{-\sigma}$, in the presence of a static malicious adversary corrupting at most $t < n/3$ parties.*

Proof. We prove the theorem by showing correctness, correctness against malicious deviations, and privacy.

Correctness. Assume that all parties follow Protocol 9 honestly. In the input sharing phase, each input value is shared by running **share**$(\cdot)$, and hence each input wire contains a correct SSS sharing. Therefore, all δ independent executions of Protocol 3 accept.

During the circuit emulation phase, we argue by induction over the topological ordering of the gates. Assume that the input wires of a gate contain correct degree-t sharings of their underlying values. If the gate is an addition gate or a multiplication-by-a-constant gate, then the output sharing is obtained by local linear operations. Since SSS is linear, these operations preserve correctness of the sharings. If the gate is a multiplication gate with input sharings $[x_k]_t$ and $[y_k]_t$, then, in the present correctness argument, all parties are assumed to follow Protocol 9 honestly, and therefore they also follow the invoked subprotocol π_{mult}. By the correctness of π_{mult} in an honest execution, the parties obtain a correct degree-t sharing $[z_k]_t$ of $z_k = x_k \cdot y_k$. Hence the invariant that every wire contains a correct sharing of its corresponding circuit value is preserved throughout the circuit emulation phase.

In the precomputation phase, Protocol 8 generates random triples of the form $([a_k^{(j)}]_t, [b_k^{(j)}]_t, [c_k^{(j)}]_{2t})$, where $[c_k^{(j)}]_{2t} = [a_k^{(j)}]_t \cdot [b_k^{(j)}]_t$. Thus, in an honest execution, each auxiliary triple is correct, namely $c_k^{(j)} = a_k^{(j)} \cdot b_k^{(j)}$.

Now consider the j-th execution of Protocol 7 in Step 5. For each multiplication gate k, Protocol 7 computes $[\rho_k^{(j)}]_t = \alpha_k^{(j)}[x_k]_t + [a_k^{(j)}]_t$ and $[\sigma_k^{(j)}]_t = [y_k]_t + [b_k^{(j)}]_t$. It then locally computes

$$[v_k^{(j)}]_{2t} = [\alpha_k^{(j)} z_k]_{2t} - [c_k^{(j)}]_{2t} + [a_k^{(j)}\sigma_k^{(j)}]_{2t} - [\rho_k^{(j)} y_k]_{2t}.$$

Since $z_k = x_k \cdot y_k$ and $c_k^{(j)} = a_k^{(j)} \cdot b_k^{(j)}$, the value underlying $[v_k^{(j)}]_{2t}$ is

$$\alpha_k^{(j)} z_k - c_k^{(j)} + a_k^{(j)}\sigma_k^{(j)} - \rho_k^{(j)} y_k = 0.$$

Therefore, every $[v_k^{(j)}]_{2t}$ is a correct sharing of zero. The random linear combination computed in Protocol 7 is also a correct sharing of zero, and adding the sharing $[0]_{2t}$ does not change the underlying value. Hence, the final opening outputs 0, and all honest parties accept.

Since this holds for every $j = 1, \ldots, \delta$, all executions of Protocol 7 accept in the verification stage. Finally, the output wires contain correct SSS sharings of the circuit output, and the reconstruction

[7] In our instantiation, this condition is achieved by deriving the challenges from seeds generated and fixed by the commitment-and-delayed-opening procedure of Protocol 5, together with the PRF-based coin-generation method of Protocol 2. A single PRF seed can be used to derive multiple sets of challenges by using distinct counters or domain-separated inputs.

procedure outputs the corresponding values to the designated parties. Therefore, Protocol 9 outputs f correctly in an honest execution.

Correctness against malicious deviations. Let $p = \frac{1}{|\mathbb{F}|-1}$. We define the event Bad as the event that the adversary causes some honest party to output an incorrect value without being detected.

Let $\mathsf{Bad}_{\mathsf{in}}$ be the event that at least one incorrect input sharing passes the input-correctness check in Step 3. Protocol 3 checks the correctness of the input sharings by using fresh random coefficients and opening a random linear combination. Since Step 3 runs Protocol 3 with δ independent random linear checks, we have

$$\Pr[\mathsf{Bad}_{\mathsf{in}}] \le p^{\delta}.$$

We next condition on $\neg\mathsf{Bad}_{\mathsf{in}}$. Then all input sharings used in the circuit emulation are correct. Since addition gates and multiplication-by-a-constant gates are computed by local linear operations, they preserve correctness. Thus, any remaining undetected error must come from at least one multiplication gate.

By the assumption that π_{mult} is private and secure up to additive attacks, any malicious deviation in a multiplication gate that affects the output can be represented as an additive error in the output sharing. Namely, for some multiplication gate k, instead of obtaining a sharing of $x_k \cdot y_k$, the parties may hold $[z_k]_t = [x_k \cdot y_k + \Delta_k]_t$ for some $\Delta_k \in \mathbb{F}$. If $\Delta_k = 0$, then the multiplication result is correct. Hence, an incorrect multiplication result corresponds to $\Delta_k \neq 0$.

In Step 5, the parties run Protocol 7 independently for $i = 1, \ldots, \delta$, using the i-th set of random triples $\vec{d}_i = \{([a_k^{(i)}]_t, [b_k^{(i)}]_t, [c_k^{(i)}]_{2t})\}_{k=1}^{N}$. The public random challenges used in these executions are fresh and independent of the adversary's previous messages. Therefore, after the adversary has fixed the multiplication outputs, the random coefficients used in the verification are independent of the additive error value Δ_k. Consequently, by Lemma 3, if at least one multiplication output contains a nonzero additive error, then one execution of Protocol 7 accepts with probability at most $1/(|\mathbb{F}|-1)$. Since the δ executions use fresh independent challenges, the probability that such an error passes all executions is at most $\left(\frac{1}{|\mathbb{F}|-1}\right)^{\delta}$.

Let $\mathsf{Bad}_{\mathsf{mult}}$ be the event that, conditioned on $\neg\mathsf{Bad}_{\mathsf{in}}$, at least one incorrect multiplication result passes all δ executions of Protocol 7. By Lemma 3, one execution of Protocol 7 accepts an incorrect multiplication result with probability at most p. This bound also covers the case where an auxiliary triple generated by Protocol 8 is incorrect: if a degree-$2t$ sharing is not value-consistent, the final opening detects the inconsistency and the protocol aborts; if it is value-consistent but represents an incorrect value, the random linear-combination argument of Lemma 3 applies. Since the δ executions use fresh independent challenges, we have $\Pr[\mathsf{Bad}_{\mathsf{mult}} \mid \neg\mathsf{Bad}_{\mathsf{in}}] \le p^{\delta}$. Therefore,

$$\Pr[\mathsf{Bad}] \le \Pr[\mathsf{Bad}_{\mathsf{in}}] + \Pr[\mathsf{Bad}_{\mathsf{mult}} \mid \neg\mathsf{Bad}_{\mathsf{in}}] \le p^{\delta} + p^{\delta} = 2p^{\delta}.$$

By Step 1, δ is chosen so that $p^{\delta} \le 2^{-(\sigma+1)}$. Thus,

$$\Pr[\mathsf{Bad}] \le 2p^{\delta} \le 2 \cdot 2^{-(\sigma+1)} = 2^{-\sigma}.$$

Therefore, except with probability at most $2^{-\sigma}$, every deviation by the adversary is either detected and causes abort, or does not affect the correctness of the output.

Ideal functionality for f with abort. First we specify the ideal functionality used in the simulation. Let

$$f(x_1, \ldots, x_n) = (y_1, \ldots, y_n)$$

be the functionality computed by the arithmetic circuit, where y_i is the output intended for party P_i. The ideal functionality $\mathcal{F}_f^{\mathsf{abort}}$ receives the input x_i from each honest party P_i and receives the input x_i^* for each corrupted party from the ideal-world adversary/simulator. If the simulator sends abort before the output phase, then $\mathcal{F}_f^{\mathsf{abort}}$ sends $\perp$ to all honest parties and halts. Otherwise, it computes

$$(y_1, \ldots, y_n) = f(x_1', \ldots, x_n'),$$

where $x_i' = x_i$ for honest parties and $x_i' = x_i^*$ for corrupted parties. It sends y_i to each honest party P_i and sends the outputs of the corrupted parties to the simulator.

Simulation and privacy. We now show that the adversary's view can be simulated in the ideal execution with $\mathcal{F}_f^{\mathsf{abort}}$, except with the probability of the event Bad defined above. Let $\mathcal{A}$ be a real-world adversary corrupting a fixed set $\mathcal{C}$ of at most t parties. We construct an ideal-world simulator $\mathcal{S}$ as follows.

In the input sharing phase, $\mathcal{S}$ internally runs $\mathcal{A}$. For the corrupted parties, $\mathcal{S}$ uses the inputs chosen by $\mathcal{A}$. For the honest parties, $\mathcal{S}$ uses dummy inputs, say 0, and generates SSS sharings of these dummy values. Since SSS is perfectly private against at most t corrupted parties, the shares seen by $\mathcal{A}$ are distributed identically to the shares it would see in a real execution with the honest parties' actual inputs.

For the input sharings generated by corrupted parties, $\mathcal{S}$ receives the shares sent by the corrupted parties to the honest parties. If such a sharing is detected as incorrect in Protocol 3, then $\mathcal{S}$ sends abort to the ideal functionality and simulates the honest parties aborting. If an incorrect input sharing is not detected, then the event Bad occurs. Otherwise, each accepted corrupted-party input sharing defines a unique value, and $\mathcal{S}$ extracts this value as the corrupted party's input x_i^* to be sent to $\mathcal{F}_f^{\mathsf{abort}}$.

In the precomputation phase, $\mathcal{S}$ simulates the calls to $\mathcal{F}_{rand}$ and $\mathcal{F}_{zero}$ exactly as in the hybrid execution. Since these are ideal functionalities in the hybrid model, the shares delivered to the corrupted parties have the same distribution as in the real execution. The public random challenges used in Protocol 3 and Protocol 7 are simulated as fresh random field elements, which is consistent with the assumption that these challenges are fresh and independent of the adversary's previous messages.

In the circuit emulation phase, $\mathcal{S}$ simulates all local linear operations using the simulated sharings. For every multiplication gate, $\mathcal{S}$ uses the privacy of π_{mult} to simulate the messages seen by the corrupted parties. By the assumption that π_{mult} is secure up to additive attacks, any malicious deviation that affects the multiplication output can be represented as an additive error in the output sharing. Such errors are handled by the verification stage. Therefore, conditioned on $\neg\mathsf{Bad}$, the simulated view of $\mathcal{A}$ during the circuit emulation phase is indistinguishable from its view in the real execution.

In the verification stage, $\mathcal{S}$ simulates the fresh public challenges and the degree-$2t$ zero sharings used in Protocol 7. If all multiplication outputs are correct, then every execution of Protocol 7 opens a sharing of zero, and $\mathcal{S}$ simulates the opening of 0. If some multiplication output is incorrect and the verification detects it, then $\mathcal{S}$ sends abort to the $\mathcal{F}_f^{\mathsf{abort}}$ and simulates the honest parties aborting. If an incorrect multiplication output is not detected by all δ executions of Protocol 7, then the event Bad occurs.

It remains to simulate the output reconstruction phase. If no abort has occurred and Bad does not occur, then all output-wire sharings are correct sharings of the corresponding output values. The simulator sends the extracted inputs of the corrupted parties to the $\mathcal{F}_f^{\mathsf{abort}}$ and receives the outputs intended for the corrupted parties. When an output is reconstructed to a corrupted party,

$\mathcal{S}$ chooses the reconstruction messages of the honest parties so that they are consistent with the corrupted parties' current shares and with the output value received from the $\mathcal{F}_f^{\mathsf{abort}}$. This is possible because the adversary controls at most t shares, and any t points can be extended to a degree-t SSS sharing of any prescribed secret. Thus, the reconstruction messages seen by the corrupted parties are distributed as in the real execution.

For outputs intended to honest parties, $\mathcal{S}$ receives the messages that the corrupted parties send during reconstruction. If these messages are inconsistent, then $\mathcal{S}$ sends the corresponding abort instruction to the $\mathcal{F}_f^{\mathsf{abort}}$ and simulates the honest party outputting $\perp$. Otherwise, $\mathcal{S}$ lets the honest party receive the output from the $\mathcal{F}_f^{\mathsf{abort}}$.

The only case in which the simulation may differ from the real execution is the event Bad. As shown above, $\Pr[\mathsf{Bad}] \leq 2^{-\sigma}$. Conditioned on $\neg\mathsf{Bad}$, the input sharing phase is perfectly simulated by the privacy of SSS, the multiplication phase is simulated by the privacy of π_{mult}, the verification stage either opens zero or aborts in both executions, and the output reconstruction phase is simulated consistently with the ideal outputs. Hence, the real execution and the ideal execution are indistinguishable up to statistical error $2^{-\sigma}$.

4 Instantiation of Protocol 9 Based on Shamir's Secret Sharing

In SSS, a secret is distributed among n parties with threshold t, by defining a polynomial of degree at most t, and handing each party a point on the polynomial. Formally, given a secret v, the dealer chooses random coefficients $q_1, \ldots, q_t \in \mathbb{F}$ and defines a polynomial $q(x) = v + \sum_{j=1}^{t} q_j x^j$. Then, each party P_i is given the value $q(i)$. Then, no subset of t parties can reconstruct the secret, but a subset consisting of $t+1$ parties can reconstruct the secret. For the threshold of $2t$, we can generate a sharing in the same way.

A sharing $[v]_t$ is *correct* if there exists a single polynomial $q(x)$ of degree at most t such that for every honest party P_i holding value v_i, it holds that $q(i) = v_i$. Thus, a sharing $[v]_t$ is *incorrect* if for every polynomial $q(x)$ of the degree at most t, there exists an honest P_i holding v_i such that $q(i) \neq v_i$. In particular, if the sharing $[v]_t$ is not correct, then it is value-inconsistent. This holds since any subset $t+1$ shares define a polynomial of degree at most t, and thus for any subset of $t+1$ honest parties J, it holds that $\mathsf{val}([v]_t) \neq \perp$. SSS is linear, so it enables parties to locally add shares and multiply them by a constant, as required for our protocol.

The Basic Procedures and Subprotocols.

- **share**(v): The dealer chooses a random polynomial $q(x)$ of degree t under the constraint that $q(0) = v$, and then sends each P_i the point $q(i)$.
- **reconstruct**$([v]_t, i)$: In order to reconstruct to party P_i, each party sends its share to P_i. Then, P_i uses any of the $t+1$ shares to compute the unique polynomial of degree at most t defined by the points, and checks that all other shares lie on the same polynomial. If not, P_i sends $\perp$ to all the other parties and halts. Since there are at least $t+1$ honest parties (whose shares uniquely define a polynomial and so the value to be reconstructed), we are guaranteed that the corrupted parties cannot cause P_i to output an incorrect value.
- **open**$([v]_d)$ for $d \in \{t, 2t\}$ (following [15]): the parties broadcast their shares, and each recipient checks that the polynomial has degree at most d. The recipient outputs v if the check passes, and $\perp$ otherwise. Correctness against up to t corrupt parties requires that the honest points alone fix a unique polynomial of degree at most d, where $n - t \geq d + 1$ must hold. In particular, opening $[\cdot]_t$ needs at least $t+1$ honest parties (hence $t < n/2$), and opening $[\cdot]_{2t}$ needs at

least $2t+1$ honest parties (hence $t < n/3$). The procedure is *detectable although not correctable*: inconsistent shares result in **abort**, never in an incorrect value.[8]

- $\mathcal{F}_{rand}$*-Generating Random Shares for Threshold* t. We describe the method for generating random shares with Pseudo-Random Secret Sharing (PRSS) [8] (if $n > 7$, parties can choose the Vandermonde-matrix-based technique (referred to as VAN) [3]) where n is the total number of the parties), which enables the parties to generate pseudorandom sharings without interaction. This is based on the replicated secret sharing to distribute random keys (for each subset of t parties, all the parties in the complement subset receive a key). Their random keys are used to generate a pseudorandom value and then convert it to SSS. Because this protocol does not require interaction, it is easy to get the correct sharing. This is given in A.
- $\mathcal{F}_{zero}$*-Generating 0-shares for Threshold* $2t$. We describe the method for generating shares of 0 with Pseudo-Random Zero Sharing (PRZS) [8], which enables parties to generate a sharing of 0 with a polynomial of degree at most $2t$. This protocol is basically the same as PRSS, and it uses the replicated secret sharing to distribute the random keys. Then parties use the random keys to generate the pseudorandom value and convert it to SSS. The protocol does not require interaction, and the difference is that this protocol uses the threshold $2t$ rather than t. This protocol is given in B.
- π_{mult} *semi-honest multiplication that is private and secure up to additive attack*. We describe two different semi-honest multiplication protocols. The first is based on GRR [17], and BGW [4]. This protocol enables n parties with $t < \frac{n}{2}$ semi-honest corrupt parties to compute a multiplication with quadratic communication complexity. Specifically, each party computes the multiplication locally. Next, the parties send their results to other parties. Then, each party computes a linear combination of the shares it receives locally.
  The second protocol is the DN protocol [13], which is more efficient than the above protocol, particularly when a large number of parties participate. These protocols are given in C. We note that the performance of our verification method does not depend on the efficiency of the underlying multiplication protocol, and our verification can be combined with any semi-honest π_{mult} beyond the ones mentioned here.

5 Efficiency Analysis and Comparison with Prior Work

We analyze the performance of our protocol. The comparison is based on the instantiation of §4. In our protocol, we use the black-box building blocks (such as $\mathcal{F}_{zero}, \mathcal{F}_{rand}$ and π_{mult}), and compile them into a maliciously secure protocol, and we measure the cost as a function of these building blocks. The communication cost is shown in Table 1 where PRSS is a non-interactive method to implement $\mathcal{F}_{rand}$, and PRZS is a non-interactive method to implement $\mathcal{F}_{zero}$ (see A and B for details). In [23], when there are relatively many parties ($n > 7$), PRSS is replaced with a technique using the VAN to reduce the computational overhead, which will lead to additional communication

[8] We stress that **open**$(\cdot)$ is used here as a detectable opening procedure with abort, rather than as an error-correcting reconstruction procedure based on Reed–Solomon decoding. Error-correcting reconstruction can deal with inconsistent or erroneous shares, but it does not by itself guarantee that a structurally valid sharing encodes the value that should have resulted from a previous computation. For example, a malicious deviation in a multiplication protocol may cause the parties to hold a valid sharing of an incorrect value, namely $[xy + \Delta]_t$ instead of the intended sharing $[xy]_t$ as in additive attacks. Such a sharing may pass the degree-consistency check even though the shared value is wrong. Therefore, this type of deviation must be handled by the verification protocol, not by the opening procedure.

Table 1. Communication cost (number of field elements) per party for each building block

	π_{mult}	$\mathcal{F}_{rand}$	$\mathcal{F}_{zero}$	$\mathbf{open}(\cdot)$
Case of SSS $\mathcal{F}_{rand}$ with PRSS and $\mathcal{F}_{zero}$ with PRZS (with few parties)	6	0	0	$n-1$
Case of SSS $\mathcal{F}_{rand}$ with VAN and $\mathcal{F}_{zero}$ with PRZS (with many parties)	6	2	0	$n-1$

overhead (2 field elements of $\mathcal{F}_{rand}$ in the case of many parties, as shown in Table 1). However, in [21], PRZS is shown to remain efficient in practice even with many parties without relying on the Vandermonde-matrix-based technique [3]. Moreover, $\mathcal{F}_{zero}$ is invoked only once in Protocol 7. Therefore, assuming that $\mathcal{F}_{zero}$ is implemented using PRZS, we set the communication cost of $\mathcal{F}_{zero}$ to 0 for the case of many parties in Table 1. Here we focus on the growth in complexity with respect to the circuit size while ignoring the cost of input sharing and output reconstruction. Since addition and multiplication-by-a-constant gates are computed locally, this leads to measuring the cost per multiplication gate.

For multiplication gates, the protocol requires one invocation of a semi-honest multiplication (for ease of comparison, we use SSS with the DN multiplication protocol, as in [23]). For random multiplication triples, the protocol requires two random sharings generated by $\mathcal{F}_{rand}$.

Next we consider the verification step. In the protocol, we need one more invocation of Π_{coin} to generate α, and need to invoke $\mathcal{F}_{zero}$ to generate a sharing $[0]_{2t}$. Here the cost of Π_{coin} is quite low and could be ignored. Recall that the verification procedure is repeated δ times in order to achieve statistical security $2^{-\sigma}$. Hence, δ denotes the number of random linear checks performed during verification. We stress that all multiplications appearing in the verification protocol are performed locally over degree-$2t$ sharings. Therefore, these operations do not require additional invocations of the semi-honest multiplication protocol π_{mult}. Thus, the overall cost per multiplication gate in the protocol is $1 \cdot \mathsf{cc}(\pi_{mult}) + 2\delta \cdot \mathsf{cc}(\mathcal{F}_{rand}) + \delta \cdot \mathsf{cc}(\mathcal{F}_{zero})$ where $\mathsf{cc}(\cdot)$ is the communication cost of the protocol or ideal functionality. As a comparison, in [23], the overall cost per multiplication gate is $(1 + 5\delta) \cdot \mathsf{cc}(\pi_{mult}) + 3\delta \cdot \mathsf{cc}(\mathcal{F}_{rand})$. The main difference is that the verification procedure in [23] requires additional invocations of the semi-honest multiplication protocol, whereas in our protocol the verification step consists only of local operations and a single invocation of $\mathbf{open}(\cdot)$.

Remark 1 (Comparison with [15]). It is worth noting that [15] presents an optimized variant of the DN protocol for the same threshold regime $t < n/3$, which reduces the communication overhead per multiplication to nearly half that of the original DN protocol and requires only $\frac{8}{3}n$ field elements. Since this threshold matches that assumed by our framework, this optimized DN protocol can in principle be used as the underlying multiplication protocol π_{mult}. In Table 2, however, we use the original DN protocol as the common underlying multiplication protocol for consistency with the $t < n/2$ baselines [7,23]. If the optimized DN protocol of [15] is used as π_{mult}, then our instantiated protocol inherits the same multiplication cost as the optimized DN protocol of [15]. Hence, the difference between [15] and ours lies in the verification step: [15] also introduces a verification protocol with a communication cost of $\frac{20}{3}n$ field elements per party for the entire circuit, whereas our corresponding batched verification incurs a total cost of $\delta \cdot \mathsf{cc}(\mathcal{F}_{zero}) + \delta \cdot \mathsf{cc}(\mathbf{open}(\cdot)) = \delta(n-1)$ field elements per party for the entire batch of multiplication triples under the instantiation considered here. Since δ is typically small in practice, often just 1 or 2, our verification layer can have lower total communication than that of [15].

Table 2. Communication cost (number of field elements) per party for different protocols for one multiplication gate and one verification step. Here the original DN protocol is used as the common underlying multiplication protocol for consistency with the $t < n/2$ baselines [23,7]; the comparison with [15] under the same threshold $t < n/3$ is discussed in Remark 1.

	[23] with $\delta = 1$ (Any Field)	[7] with $\delta = 1$ (Large Field)	[7] with $\delta = 1$ (Any Field)	Protocol 9 with $\delta = 1$ (Any Field)	[7] with $\delta = 2$ (Any Field)	Protocol 9 with $\delta = 2$ (Any Field)
(A)	36	12	12	**6**	18	**6**
(B)	42	12	14	**10**	22	**14**

(A) Case of SSS, $\mathcal{F}_{rand}$ with PRSS and $\mathcal{F}_{zero}$ with PRZS (with few parties).
(B) Case of SSS, $\mathcal{F}_{rand}$ with VAN and $\mathcal{F}_{zero}$ with PRZS (with many parties).

Observe that when using a field $|\mathbb{F}|$ with $|\mathbb{F}| > 2^{\sigma+1}$ (e.g., $\mathbb{F} = \mathbb{Z}_p$ with p being a 40-bit prime, where the allowed statistical error is 2^{-39}), then it suffices to take $\delta = 1$. In this case, the cost of our protocol is $1 \cdot \mathsf{cc}(\pi_{mult}) + 2 \cdot \mathsf{cc}(\mathcal{F}_{rand}) + 1 \cdot \mathsf{cc}(\mathcal{F}_{zero})$, whereas the cost in [23] is $6 \cdot \mathsf{cc}(\pi_{mult}) + 3 \cdot \mathsf{cc}(\mathcal{F}_{rand})$. Considering that the efficiencies of $\mathcal{F}_{rand}$ and $\mathcal{F}_{zero}$ are similar, our protocol is more efficient than [23], requiring five fewer invocations of π_{mult}[9] (see Table 1 for the communication cost of π_{mult}).

In another prior work [7], two protocols that improve upon [23] in terms of efficiency are proposed. The first is suitable for large finite fields, while the second targets small fields. In the first case, the communication cost is $2 \cdot \mathsf{cc}(\pi_{mult})$. We also consider a field $\mathbb{F}$ with $|\mathbb{F}| > 2^{\sigma+1}$ (e.g., $\mathbb{F} = \mathbb{Z}_p$ with p being a 60-bit prime. In this case, setting the statistical security parameter to $\sigma + 1 = 60$ ensures that the statistical error is at most 2^{-59}), and it suffices to take $\delta = 1$. In this case, the cost of our protocol is $1 \cdot \mathsf{cc}(\pi_{mult}) + 2 \cdot \mathsf{cc}(\mathcal{F}_{rand}) + 1 \cdot \mathsf{cc}(\mathcal{F}_{zero})$. Here, we realize $\mathcal{F}_{rand}$ using PRSS and $\mathcal{F}_{zero}$ using PRZS, both of which incur no communication overhead in the online stage. Consequently, our protocol requires one fewer invocation of π_{mult} compared to [7]. In the second case, the communication cost of [7] is $(1 + \delta) \cdot \mathsf{cc}(\pi_{mult}) + \delta \cdot \mathsf{cc}(\mathcal{F}_{rand})$. We use the same field as in [7], namely $\mathbb{F} = \mathbb{Z}_p$, where p is either a 30-bit or a 60-bit prime. In this setting, the statistical error is 2^{-59}, which is sufficiently small to allow setting $\delta = 1$ or $\delta = 2$. When $\delta = 1$, the cost of our protocol is $1 \cdot \mathsf{cc}(\pi_{mult}) + 2 \cdot \mathsf{cc}(\mathcal{F}_{rand}) + 1 \cdot \mathsf{cc}(\mathcal{F}_{zero})$, and the cost of [7] is $2 \cdot \mathsf{cc}(\pi_{mult}) + 1 \cdot \mathsf{cc}(\mathcal{F}_{rand})$. For the same reason as above, our protocol has one fewer invocation of π_{mult} compared to [7]. When $\delta = 2$, the cost of our protocol is $1 \cdot \mathsf{cc}(\pi_{mult}) + 4 \cdot \mathsf{cc}(\mathcal{F}_{rand}) + 2 \cdot \mathsf{cc}(\mathcal{F}_{zero})$, and the cost of [7] is $3 \cdot \mathsf{cc}(\pi_{mult}) + 2 \cdot \mathsf{cc}(\mathcal{F}_{rand})$. Thus our protocol has two fewer invocations of π_{mult} compared to [7]. The detailed communication overhead per party is shown in Table 2.

Verification time cost. In addition to the communication analysis in Table 2, we also implement and evaluate the local computation part of the verification procedures by measuring their time cost. All experiments were conducted on a desktop machine with a 13th Gen Intel Core i9-13900KF CPU at 3.00 GHz and 32 GB RAM, running a 64-bit Windows operating system. The experiment only

[9] First we note that in [23], Protocol 1 is used to generate random public values. While the potential use of $\mathsf{PRF_s}$ is mentioned in [23] as a replacement for Protocol 1, no concrete instantiation is provided. In our protocol, we instantiate the use of $\mathsf{PRF_s}$ to achieve the desired functionality. Since the communication cost of Protocol 1 is not included in the cost analysis of [23], we follow the same cost analysis, and we do not include the communication costs for generating the PRF key and *seed* for $\mathsf{PRF_s}$ as these correspond to the generation of random public values.

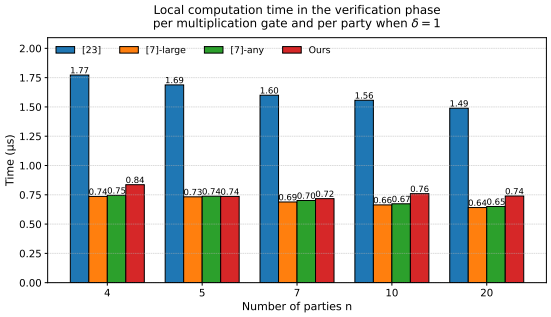

Fig. 1. Local computation time in the verification phase per multiplication gate and per party when $\delta = 1$.

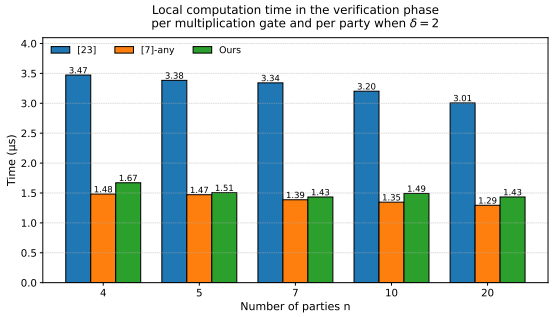

Fig. 2. Local computation time in the verification phase per multiplication gate and per party when $\delta = 2$.

focuses on the verification stage, so all input sharings, auxiliary triples, random challenges, zero sharings, and the masks required by the multiplication protocol are prepared before the measurement. Therefore, the reported time does not include communication, opening/reconstruction, or the generation cost of $\mathcal{F}_{rand}$, $\mathcal{F}_{zero}$, and Π_{coin}. As the main advantage of our protocol over [23] and [7] is the reduction in communication overhead, as shown in Table 2, the purpose of the following experiment is therefore not to provide an end-to-end runtime comparison, but to check whether this communication improvement introduces a significant additional local computation cost.

For the protocols of [23] and [7], the invocations of π_{mult} or $\mathcal{F}_{mult}$ appearing in the verification stage are instantiated by the local computation of the DN multiplication protocol, where the required preprocessing values are already available. For our protocol, we measure the local computation in Protocol 7, including the computation of all $[v_i]_{2t}$ and their random linear combination.

We set $n \in \{4, 5, 7, 10, 20\}$ and $t = \lfloor (n-1)/3 \rfloor$. For all protocols in the experiment, we use the same threshold $t = \lfloor (n-1)/3 \rfloor$ so that the comparison is made under the threshold setting of our protocol. The time cost is reported in microseconds per multiplication gate per party. The results for $\delta = 1$ and $\delta = 2$ are shown in Fig. 1 and Fig. 2, respectively.

The results are consistent with the cost analysis above. Compared with [23], our verification procedure avoids several additional invocations of π_{mult}. Since these invocations are instantiated by the local computation of the DN multiplication protocol in the experiment, this reduction is reflected in the measured local verification time. Therefore, our protocol achieves lower local verification time than [23] in the tested settings.

Compared with [7], our verification procedure uses fewer invocations of π_{mult}: one fewer invocation when $\delta = 1$ and two fewer invocations when $\delta = 2$. However, our protocol also introduces additional local degree-$2t$ computations. As a result, the measured local verification time of our protocol is comparable to that of [7], rather than significantly lower.

Overall, these timing results, together with the communication analysis in Table 2, show that our verification procedure improves over [23] in both local computation and communication: it reduces the local verification time and the communication cost by avoiding several additional invocations of π_{mult}. Compared with [7], the measured local verification time of our protocol remains comparable. Thus, the main benefit of our protocol over [7] is best understood as lower communication overhead with fewer invocations of π_{mult}, while maintaining a comparable level of local computation.

References

1. 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*, pages 805–817. ACM, 2016.
2. Donald Beaver. Efficient multiparty protocols using circuit randomization. In *CRYPTO*, pages 420–432. Springer, 1992.
3. Zuzana Beerliová-Trubíniová and Martin Hirt. Perfectly-secure MPC with linear communication complexity. In *TCC*, pages 213–230. Springer, 2008.
4. Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *STOC*, pages 1–10. ACM, 1988.
5. Eli Ben-Sasson, Serge Fehr, and Rafail Ostrovsky. Near-linear unconditionally-secure multiparty computation with a dishonest minority. In *CRYPTO*, pages 663–680. Springer, 2012.
6. David Chaum, Claude Crépeau, and Ivan Damgard. Multiparty unconditionally secure protocols. In *STOC*, pages 11–19. ACM, 1988.
7. Koji Chida, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Daniel Genkin, Yehuda Lindell, and Ariel Nof. Fast large-scale honest-majority MPC for malicious adversaries. *Journal of Cryptology*, 36(3):15, 2023.
8. Ronald Cramer, Ivan Damgård, and Yuval Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In *TCC*, pages 342–362. Springer, 2005.
9. Anders Dalskov, Daniel Escudero, and Ariel Nof. Fast fully secure multi-party computation over any ring with two-thirds honest majority. In *CCS*, pages 653–666, 2022.
10. Anders Dalskov, Daniel Escudero, and Ariel Nof. Fully secure MPC and zk-FLIOP over rings: new constructions, improvements and extensions. In *CRYPTO*, pages 136–169. Springer, 2024.
11. Ivan Damgård and Yuval Ishai. Scalable secure multiparty computation. In *CRYPTO*, pages 501–520. Springer, 2006.
12. 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*, pages 1–18. Springer, 2013.
13. Ivan Damgård and Jesper Buus Nielsen. Scalable and unconditionally secure multiparty computation. In *CRYPTO*, pages 572–590. Springer, 2007.
14. Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO*, pages 643–662. Springer, 2012.
15. Jun Furukawa and Yehuda Lindell. Two-thirds honest-majority MPC for malicious adversaries at almost the cost of semi-honest. In *CCS*, pages 1557–1571, 2019.
16. Martin Geisler. Cryptographic protocols: theory and implementation. *PhD thesis, University of Aarhus*, 2010.
17. Rosario Gennaro, Michael O Rabin, and Tal Rabin. Simplified VSS and fast-track multiparty computations with applications to threshold cryptography. In *PODC*, pages 101–111. ACM, 1998.
18. Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game. In *STOC*, pages 218–229. ACM, 1987.
19. Martin Hirt and Jesper Buus Nielsen. Robust multiparty computation with linear communication complexity. In *CRYPTO*, pages 463–482. Springer, 2006.
20. Hyunil Kim, Seung-Hyun Kim, Jung Yeon Hwang, and Changho Seo. Efficient privacy-preserving machine learning for blockchain network. *IEEE Access*, 7:136481–136495, 2019.
21. Brian Knott, Shobha Venkataraman, Awni Hannun, Shubho Sengupta, Mark Ibrahim, and Laurens van der Maaten. Crypten: Secure multi-party computation meets machine learning. *NeurIPS*, 34:4961–4973, 2021.
22. Hanchao Ku, Hikaru Tsuchida, Mingwu Zhang, and Takashi Nishide. Fast MPC framework under Shamir's secret sharing with malicious security and two-thirds honest majority. In *ICICS*. Springer, 2026 (to appear).

23. Yehuda Lindell and Ariel Nof. A framework for constructing fast MPC over arithmetic circuits with malicious adversaries and an honest-majority. In *CCS*, pages 259–276. ACM, 2017.
24. Xu Ma, Fangguo Zhang, Xiaofeng Chen, and Jian Shen. Privacy preserving multi-party computation delegation for deep learning in cloud computing. *Information Sciences*, 459:103–116, 2018.
25. Moni Naor and Omer Reingold. Number-theoretic constructions of efficient pseudo-random functions. *Journal of the ACM (JACM)*, 51(2):231–262, 2004.
26. Tal Rabin and Michael Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority. In *STOC*, pages 73–85. ACM, 1989.
27. Anh-Tu Tran, The-Dung Luong, Jessada Karnjana, and Van-Nam Huynh. An efficient approach for privacy preserving decentralized deep learning models based on secure multi-party computation. *Neurocomputing*, 422:245–262, 2021.
28. Andrew C Yao. Protocols for secure computations. In *FOCS*, pages 160–164. IEEE, 1982.
29. Andrew Chi-Chih Yao. How to generate and exchange secrets. In *FOCS*, pages 162–167. IEEE, 1986.
30. Da-Wei Zhou, Su-Zhen Cao, Xiao Zhao, Dan-Dan Xing, and Zheng Wang. Efficient first-price sealed e-auction protocol under secure multi-party computational malicious model. *Journal of Computers*, 35(1):65–81, 2024.

A Protocols For Generating Random Sharings

The PRSS protocol [8]. In this protocol, there is a setup step where the parties generate a random key k_A for each subset $A \subset \{P_1, \ldots, P_n\}$ of parties (known only to the parties in the subset). In addition, for each such subset there is a degree-t polynomial f_A defined by the points: (1) $f_A(0) = 1$; (2) $f_A(i) = 0$ for all i such that $P_i \in \{P_1, \ldots, P_n\} \setminus A$. Then, each party uses the keys it holds to generate random shares without any interaction. The protocol is described in Protocol 10.

PROTOCOL 10 *PRSS protocol [8] for generating random shares*
Let $F_k()$ be a pseudo-random function with security parameter κ.

- **Setup:** For each $A \subset \{P_1, \ldots, P_n\}$, such that $|A| = n-t$, the party in A with the smallest lexicographic index chooses a random key k_A and sends it to all the other parties in A.
- **Upon request** for id: For each $A \subset \{P_1, \ldots, P_n\}$, such that $|A| = n - t$, let $f_A(0) = 1$ be a polynomial of degree at most t defined by the points: (1) $f_A(0) = 1$; (2) $f_A(i) = 0$ for all i such that $P_i \in \{P_1, \ldots, P_n\} \setminus A$. Then, each party P_i computes

$$s_i = \sum_{A \subset \{P_1, \ldots, P_n\} : |A| = n-t, P_i \in A} F_{k_A}(id) \cdot f_A(i)$$

  where id is a public counter that is incremented for each new request, and define s_i as its share where the secret reconstructed from the shares s_i is $\sum_{A \subset \{P_1, \ldots, P_n\} : |A| = n-t} F_{k_A}(id)$.

B Protocols For Generating Sharing of 0

The PRZS protocol [8]. In this protocol for generating $[0]_{2t}$, there is a setup step where the parties generate the random keys $\{k_A^1, \ldots, k_A^t\}$ for each subset $A \subset \{P_1, \ldots, P_n\}$ of $n - t$ parties (known only to the parties in the subset A). In addition, for each such subset A with $|A| = n - t$, there are the polynomials $f_A^1, \ldots, f_A^t$ of degree at most $2t$ defined by the points: (1) $f_A^{i'}(0) = 0$; (2) $f_A^{i'}(i) = 0$ for all i such that $P_i \in \{P_1, \ldots, P_n\} \setminus A$. Then, each party uses the keys it holds to

generate random shares without any interaction. The protocol is described in Protocol 11. In this protocol, we consider a set A of size $n-t$ and consider the set of polynomials $f_A^0 = \{f \mid \deg(f) \leq 2t,\ f(0)=0,\ j \notin A \Rightarrow f(j)=0\}$.

PROTOCOL 11 *PRZS protocol [8] for generating zero shares* $[0]_{2t}$

Let $F_k(\cdot)$ be a pseudo-random function with security parameter κ, where k is a key.

- **Setup:** For each $A \subset \{P_1,\ldots,P_n\}$, such that $|A| = n-t$, the party with the smallest lexicographic index in A chooses the t random keys $\{k_A^1,\ldots,k_A^t\}$ and sends them to all the other parties in A.
- **Upon request** for id: For each $A \subset \{P_1,\ldots,P_n\}$, such that $|A| = n-t$, let $f_A^{i'}$ $(1 \leq i' \leq t)$ be a polynomial of degree at most $2t$ defined by the points: (1) $f_A^{i'}(0) = 0$; (2) $f_A^{i'}(i) = 0$ for all i such that $P_i \in \{P_1,\ldots,P_n\} \setminus A$. Then, each party P_i computes

$$s_i = \sum_{\substack{A \subset \{P_1,\ldots,P_n\}: |A|=n-t, \\ P_i \in A}} \sum_{i'=1}^{t} F_{k_A^{i'}}(id) \cdot f_A^{i'}(i)$$

  where id is a public counter that is incremented for each new request, and define s_i as its share.

If we think of the set of all polynomials of degree at most $2t$ as a $(2t+1)$-dimensional vector space over $\mathbb{F}$, we can see that f_A^0 forms a subspace of dimension $2t+1-(t+1) = t$ since $f \in f_A^0$ has $t+1$ zeros. Thus, we can choose for each A, a basis for f_A^0 consisting of t polynomials $f_A^1,\ldots,f_A^t$ [10].

C Semi-Honest Multiplication Protocols π_{mult}

The GRR protocol [17]. The GRR protocol needs to work by having each party locally multiply its shares of the inputs and share the result with all the other parties. Then we use Lagrange interpolation coefficients to get the new shares. The new sharing of c results from the secret $c = a \cdot b$, and the threshold is t. The detail is shown in Protocol 12.

PROTOCOL 12 *GRR protocol [17]*

- **Inputs:** The inputs for each P_i are $a_i = f_a(i)$ and $b_i = f_b(i)$, where $i \in [1,n]$, $f_a(Z) \in \mathcal{P}^{a,t}$ and $f_b(Z) \in \mathcal{P}^{b,t}$. Here $\mathcal{P}^{a,t}$ and $\mathcal{P}^{b,t}$ are the sets of polynomials of degree at most t with constant term a, b and arbitrary coefficients for the higher powers.
- **Public Input:** Let $\mathfrak{c}_1,\ldots,\mathfrak{c}_n$ be the Lagrange interpolation coefficients, and it holds that $ab = f_a(0)\cdot f_b(0) = \mathfrak{c}_1\cdot(a_1b_1) + \cdots + \mathfrak{c}_n\cdot(a_nb_n)$.
- **The protocol:**
  1. Party P_i computes $k_i = a_i \cdot b_i$.
  2. Party P_i shares k_i with other parties. Let $k_{j,i}$ be the share P_i receives from P_j where $j = 1,\ldots,n$.
  3. Party P_i computes $c_i = \mathfrak{c}_1 \cdot k_{1,i} + \cdots + \mathfrak{c}_n \cdot k_{n,i}$.
- **Output:** Each P_i obtains the share c_i.

The DN protocol [13]. In this protocol, each party needs to send few field element regardless of the number of parties. The protocol has a set-up step where the parties generate two random sharings $[r]_t$ and $[r]_{2t}$ of the same value r, using degree-t and degree-$2t$ polynomials respectively. These shares are then used to multiply x and y which are shared among the parties. The detail is shown in Protocol 13.

[10] In [16, §4.3.2], an efficient concrete method for choosing such a basis is provided.

PROTOCOL 13 *DN protocol [13]*

- **Inputs:** The parties hold sharings $[x]_t$ and $[y]_t$.
- **Setup phase:** The parties generate a list of $n - t$ double random shares $\{[r_k]_t, [r_k]_{2t}\}_{k=1}^{n-t}$ where $[r_k]_t$ is a sharing of r_k using a polynomial of degree at most t and $[r_k]_{2t}$ is a sharing of r_k using a polynomial of degree at most $2t$.
- **The protocol:**
  1. The parties locally compute $[x]_t \cdot [y]_t - [r]_{2t}$ and send the result to party P_1 ($[x]_t \cdot [y]_t$ is locally computed by each party multiplying its own shares together).
  2. Party P_1 uses the first $2t$ shares it received from the parties and its own share to reconstruct $x \cdot y - r$, and then sends it to all the other parties (We note that this is not the same as the **reconstruct** procedure, as there is no correctness check of the shares).
  3. The parties locally compute $[r]_t + (xy - r)$. Each party sets its result to be its share of $x \cdot y$.