Don't Eject the Impostor: Fast Three-Party Computation With a Known Cheater (Full Version*)

Andreas Brüggemann*, Oliver Schick*, Thomas Schneider*, Ajith Suresh† and Hossein Yalame*
* *Technical University of Darmstadt (TUD), Germany*
† *Technology Innovation Institute (TII), Abu Dhabi*

***Abstract*—Secure multi-party computation (MPC) enables (joint) computations on sensitive data while maintaining privacy. In real-world scenarios, asymmetric trust assumptions are often most realistic, where one somewhat trustworthy entity interacts with smaller clients. We generalize previous two-party computation (2PC) protocols like MUSE (USENIX Security'21) and SIMC (USENIX Security'22) to the three-party setting (3PC) with one malicious party, avoiding the performance limitations of dishonest-majority inherent to 2PC.**

We introduce two protocols, AUXILIATOR and SOCIUM, in a machine learning (ML) friendly design with a fast online phase and novel verification techniques in the setup phase. These protocols bridge the gap between prior 3PC approaches that considered either fully semi-honest or malicious settings. AUXILIATOR enhances the semi-honest two-party setting with a malicious helper, significantly improving communication by at least two orders of magnitude. SOCIUM extends the *client-malicious* setting with one malicious client and a semi-honest server, achieving substantial communication improvement by at least one order of magnitude compared to SIMC.

Besides an implementation of our new protocols, we provide the first open-source implementation of the semi-honest 3PC protocol ASTRA (CCSW'19) and a variant of the malicious 3PC protocol SWIFT (USENIX Security'21).

***Index Terms*—Multi-Party Computation, Client-Malicious Setting, 3PC, Asymmetric Trust, MPC**

1. Introduction

In the current digital age, safeguarding sensitive user data, including financial and health information, is of utmost importance. People are increasingly conscious of how businesses use their data and the potential risks of some of their most sensitive information falling into the wrong hands. To address these concerns, laws like the General Data Protection Regulation (GDPR) and the California Consumer Privacy Act (CCPA) enforce stringent rules for data handling. While these regulations address privacy concerns, they also impose additional challenges in implementing systems that benefit both businesses and customers while still accessing private data.

*This article is the full and extended version of an article published at IEEE S&P'24 [18].

For a compelling real-world application, consider the case of a business model where a service provider (SP) aims to offer machine learning inference-as-a-service (MLaaS) to its clients, as shown in Fig. 1. In this scenario, a client, such as the owner of an online store, wants to assess the risk of selling to a new customer who placed an invoice-based order. The SP provides a credit scoring service to evaluate the financial solvency of the customer and may offer additional services like payment fraud detection. These services rely on ML inference, which has been covered in various works such as [54], [62], [67], [80] for credit scoring, and by Amazon for online payment fraud detection.[1] The SP possesses a proprietary ML model specifically trained for the task at hand, which it wants to keep confidential. On the other hand, the client aims to safeguard its customer's privacy by avoiding the exchange of personal data with the SP. This helps to comply with regulations that restrict the sharing of such data and prevents the SP from knowing with whom the client conducts business. Additionally, the SP might need large datasets to train its model before offering the desired service. Although other companies could potentially provide such data, a straightforward data exchange raises privacy issues and could even be unlawful.

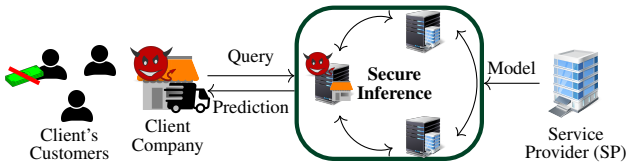

Figure 1: Application where SP offers credit scoring / payment fraud detection to a client that may be malicious. The client applies these services to their customers' data to verify their solvency.

To address the privacy concerns discussed above, Privacy-Enhancing Technologies (PETs) can be deployed. Secure multi-party computation (MPC) [56], [63], [78] is one among such notable PETs, which enables a group of parties to securely compute a joint function on their private data. It ensures that only the outcomes are revealed, preventing disclosure of any data beyond what can be derived from the output. The industry is increasingly adopting MPC[2], especially for privacy-preserving machine

1. Amazon Fraud Detector, https://aws.amazon.com/fraud-detector/
2. See https://www.mpcalliance.org, https://mpc.cs.berkeley.edu

learning (PPML), with major players such as Google [11], Intel [10], Meta [48], and Microsoft [68] utilizing it.

From the perspective of MPC, trust among participants can lead to two primary threat models: In the *semi-honest* model, each party adheres to the protocol but attempts to learn additional information from received messages. In the *malicious* model, parties can arbitrarily deviate from the protocol. In our MLaaS application, we place trust in the service provider (SP) to act semi-honestly, given its established reputation and role as a security solution provider, which allows for the implementation of rigorous security measures to minimize the risk of data breaches or hacks. However, it is a strong assumption to expect that every individual client is semi-honest [55]. A single malicious client gaining insights into the SP's ML model can be a significant threat to its business model.

While there are semi-honest protocols like those in [4], [42] that allow only *additive* errors in the computation when dealing with malicious corruptions [41], [57], these protocols are insufficient for our application. They do not effectively prevent a malicious client from gaining access to the ML model's weights [55] or carrying out poisoning attacks [38]. Therefore, a more direct solution in our scenario involves using a secure two-party protocol (2PC) designed for the malicious setting. However, this approach introduces overhead for safeguarding against a potentially malicious SP, even when assuming it to be semi-honest.

Similar to our application, many real-world scenarios are best represented by an asymmetric trust setting because of the heterogeneous nature of parties and their levels of trustworthiness. As a result, recent works like MUSE [55] and SIMC [23] have adopted the *client-malicious* setting, where the SP is assumed to be semi-honest, but the client can be malicious. However, the classic *client-malicious* setting is limited to two-parties (2PC), and protocols in this setting have significantly more computation and communication overhead than those in less stringent settings, such as three- or four-party scenarios with honest majority [30], [53], [60], [66]. Moreover, in the 2PC setting, security is only guaranteed with the possibility of an *abort* by a malicious party, leading to Denial-of-Service (DoS) attacks [56]. In contrast, settings with an honest majority can offer enhanced security for real-world scenarios, including *fairness*, where honest parties receive output whenever a malicious party does, and *robustness*, which prevents DoS attacks by ensuring the delivery of output to the honest parties [56].

Given the limitations in the 2PC setting, we propose a configuration for our MLaaS application where the client and SP agree to introduce an additional *helper* server, creating a three-party scenario (3PC). The helper server not only enhances computational efficiency, but also contributes to the system's robustness. In our setup, the SP may manage this helper server as well, ensuring non-collusion safeguards, such as geographical separation to mitigate the risk of data breaches during external attacks or outsourcing the helper server to another SP for institutional separation.

The helper server is considered semi-honest, similar to the SP, owing to shared reputation and legal considerations accepted by both the client and the SP. Furthermore, we explore a scenario in which the malicious client acts as a helper, reflecting real-world scenarios involving low-end devices as clients. We show that this configuration significantly enhances efficiency compared to the two-party case where only the client and SP are involved, both maintaining a semi-honest behaviour. These scenarios are further detailed under "Service Plans" in §1.2.

1.1. Related Work

This section provides a concise overview of related works. Firstly, we highlight some of the prominent works in the practical MPC domain. After that, we discuss the recent research concerning non-symmetrical trust settings, which is the main focus of our paper. We refer the reader to [27], [56], [63] for a comprehensive analysis of related work on MPC.

MPC: In current research, most work on MPC falls into two categories: those using Yao's garbled circuits (GC) [79] or linear secret-sharing (SS) [42] as the basis. Over the years, GC-based research has seen notable progress in reducing communication requirements and extending to multiple parties [7], [49], [70]. On the other hand, SS-based research has expanded significantly, resulting in a wide range of protocols [26], [29], [52], including more recent advanced versions like silent-preprocessing [14], [28] and function secret-sharing techniques [13], [33].

In addition to investigating MPC involving an arbitrary number of parties, researchers have also focused on developing efficient optimizations tailored for a smaller number of computation parties, typically up to four [4], [15], [17], [65]. Especially, three-party (3PC) and four-party (4PC) protocols, with an honest majority, have proven to result in practically efficient protocols for advanced applications like private machine learning [30], [53], [60], [66]. This approach can also be used in scenarios with multiple parties, where they can outsource their computations to a select few agreed-upon compute parties [45], [77].

In the literature, there are several techniques and optimizations, such as [16], [19], aimed at enhancing the practical performance of MPC protocols. In our protocols, we leverage a *mixed-protocol* strategy and *function-dependent* preprocessing, following the 3PC protocols employed in [24], [51]. A mixed-protocol strategy combines arithmetic and Boolean protocols from different domains, selecting the most suitable one for each specific operation [34], [53], [71]. In addition, function-dependent preprocessing aims to minimize complexity of the online phase, ensuring optimal performance when actual inputs are provided [8], [20], [43]. This approach is particularly valuable in scenarios like ML-as-a-Service, where computations are repeated over the same circuit, allowing for efficient batching of preprocessing across multiple protocol executions.

Protocols for non-symmetrical trust settings: The standard semi-honest or malicious settings reflect homogeneous trust in all parties which hardly reflects many real-world applications with heterogeneous participants. In the

domain of PPML, this has recently sparked ever-growing interest in what we call a *fixed-corruption* setting where only a fixed and known subset of parties may be corrupted maliciously. Semi-honest protocols offer insufficient protection in such cases [38], [55], while malicious protocols for homogeneous trust provides protection against *any* party being malicious resulting in significant performance overhead.

The non-symmetrical trust assumption we observe here is not a novel concept in the literature. For example, in the semi-honest GC protocol of Yao [79], securing against a malicious evaluator is relatively straightforward by using secure oblivious transfer against a malicious receiver. However, achieving protection against a malicious garbler requires significantly more expensive techniques (see [39] for different techniques). However, this trust assumption remained unexplored from a real-world application perspective until the recent work of MUSE [55], which explored the case of a malicious client in the context of a client-server ML inference protocol.

Subsequently, there were further developments in the two-party (2PC) setting, such as [23], [36], [76], which focused on enhancing efficiency and even examined the opposite corruption scenario involving a malicious server. In an orthogonal direction, [74] considered a three-party setting with semi-honest security, but made the assumption that one party was privileged and held more power than the others. Similarly, in the domain of federated learning (FL), ELSA [69] considered a 2PC semi-honest server setting with malicious clients assisting the servers by providing correlated randomness. Additionally, the recent work of [46] explored a scenario with a dishonest majority and an extra semi-honest helper party, demonstrating significant enhancements in communication efficiency.

This work: We explore a fixed-corruption scenario in the 3PC setting with an honest majority, which, to our knowledge, has not been previously studied. Transitioning from 2PC to an honest-majority case holds the potential for substantial performance improvements, including the removal of public-key operations like oblivious transfers, and enables the attainment of higher security levels [4], [25], [60]. To achieve an efficient online phase, we extend the 3PC protocols used in ASTRA [24] and SWIFT [65], which employ function-dependent preprocessing and a mixed-protocol strategy, representing the state-of-the-art in their respective corruption settings. We proceed by elaborating on our exact contributions.

1.2. Our Contributions

In this work, we introduce the first protocols for honest-majority 3PC with an asymmetric corruption model, where only one dedicated party can be maliciously corrupted. To achieve this, we generalize the *client-malicious* setting from MUSE [55] to the 3PC scenario by splitting the server's task in MUSE into two distinct parties. Our protocols allow one party to act as a helper that is inactive during the majority of the online phase, reducing operating costs. This leads to two novel protocols and associated offerings that the service provider (SP) can provide. The main contributions are summarized as follows:

1) **AUXILIATOR - 3PC with a malicious helper:** Our first novel protocol, AUXILIATOR, is designed for a malicious helper. It builds upon the highly efficient semi-honest 3PC protocol ASTRA [24], and maintains the same efficiency in the online phase. Despite achieving stronger malicious security, AUXILIATOR incurs only an overhead of at most $4\times$ in total communication compared to ASTRA for ML inference. This overhead can be reduced to below 2% with additional local computation.

2) **SOCIUM - 3PC with a malicious evaluator:** Our second protocol, SOCIUM, is designed for a fixed malicious evaluator and is based on the malicious 3PC protocol SWIFT [51]. It achieves a similarly efficient online phase as that of SWIFT and improves total communication for ML inference by more than $8\times$, or over $1.35\times$ when using additional local computation for both protocols.

3) **Streamlined online phase for performance:** Our protocols use the function-dependent preprocessing paradigm [8], [20], [43], ideal for ML inference. The setup can be performed by the SP and client at any time for future protocol runs, while the client needs fast predictions when a potential customer places an order. Additionally, we design the protocols for rings $\mathbb{Z}_{2^\ell}$, natively supported by standard CPU architectures, resulting in more efficient local computation.

4) **Options and insights for efficient setup:** For preprocessing, we offer two options: one optimizes local computation using triple sacrificing techniques [29] and cut-and-choose [3], [40], while the other optimizes communication using distributed zero-knowledge proofs [12]. Our novel combination of triple sacrificing and matrix triples [61] provides both asymptotic and practical improvements in safeguarding matrix multiplication against malicious parties.

5) **Spectrum between semi-honest and malicious 3PC:** AUXILIATOR and SOCIUM bridge the gap between semi-honest and malicious honest-majority 3PC protocols in the symmetric settings, by introducing scenarios where only one fixed party can be malicious. They belong to the same protocol family as ASTRA [24] and SWIFT [51], where one party serves as a helper. The spectrum of protocols ranges from ASTRA to AUXILIATOR and then SOCIUM, up to SWIFT, with the adversary growing stronger when moving from one protocol to the next. As the adversary's strength increases, the protocols become less efficient, allowing for a fine-grained trade-off between adversarial strength and efficiency in asymmetric settings as shown in Table 1 and Table 2 in §6.1.

6) **Machine learning friendly design:** Our protocols offer high-performance for ML tasks, supporting arithmetic for linear and binary computation for non-linear layers, along with corresponding conversions. Additionally, we provide efficient matrix multiplication and free probabilistic truncation protocols. The similarities to ASTRA and SWIFT

enable easy access to further ML functionalities like comparison and fixed-point arithmetic operations (cf. full version (Appendix B) for details).

TABLE 1: Total communication per multiplication (amortized) for our protocols and related 3PC honest-majority protocols with streamlined online phase. Setup can be based on sacrificing (cf. §3.1) or distributed zero-knowledge proofs (DZKP, cf. §3.2). σ denotes the statistical security parameter.

Protocol	Malicious Corruption	Setup		Online
		Sacrifice	DZKP	
ASTRA [24]	none	$1 \times \mathbb{Z}_{2^\ell}$		$2 \times \mathbb{Z}_{2^\ell}$
AUXILIATOR (§4)	helper	$4 \times \mathbb{Z}_{2^{\ell+\sigma}}$	$1 \times \mathbb{Z}_{2^\ell}$	$2 \times \mathbb{Z}_{2^\ell}$
SOCIUM (§5)	evaluator	$5 \times \mathbb{Z}_{2^{\ell+\sigma}}$	$2 \times \mathbb{Z}_{2^\ell}$	$3 \times \mathbb{Z}_{2^\ell}$
SWIFT [51]	any	$9 \times \mathbb{Z}_{2^{\ell+\sigma}}$	$3 \times \mathbb{Z}_{2^\ell}$	$3 \times \mathbb{Z}_{2^\ell}$

7) **Implementation, evaluation, and comparison:** In addition to thoroughly evaluating our protocol's communication, we provide open-source implementations of AUXILIATOR and SOCIUM using triple sacrificing. We also offer the first open-source implementations of ASTRA [24] and SWIFT [51][3] using the same verification technique. Our implementation, based on the MOTION [16] framework, is available at https://encrypto.de/code/MOTION-FD. Notably, this is the first instance where function-dependent preprocessing is implemented in a versatile MPC framework.

Furthermore, we include a comparison with the related 2PC protocols ABY2.0 [65] and SIMC [23]. By extending ABY2.0 with a malicious helper, AUXILIATOR achieves a $2-3$ orders of magnitude improvement in setup communication. On the other hand, SOCIUM with an additional semi-honest helper outperforms SIMC's communication by $14-60\times$ without requiring computationally expensive homomorphic encryption (HE).

8) **Service plans:** For MLaaS application, the choice of the party acting as the helper allows the service provider (SP) to offer various service plans with different pricing. SOCIUM represents a standard plan where the client pays less to the SP but needs to provide computational resources for one evaluator. This setup allows the SP to save costs, as one of its servers acts as the helper and can remain inactive for most of the online phase. On the other hand, AUXILIATOR is an advanced plan where the client pays more but only needs to provide a helper during a batched setup phase. The online phase is fully outsourced to the SP, which is beneficial if the client lacks permanent computational resources but can occasionally use, for example, cloud resources for preprocessing. Additionally, there is a premium plan where the client completely outsources the computation to the SP, which can use a semi-honest 2PC protocol like ABY2.0 [65] or run AUXILIATOR with an external untrusted helper. A comparison of these plans and the setting considered by MUSE [55] and SIMC [23], where the SP has only one server, is provided in Fig. 2.

3. We consider the abort variant of SWIFT in [75].

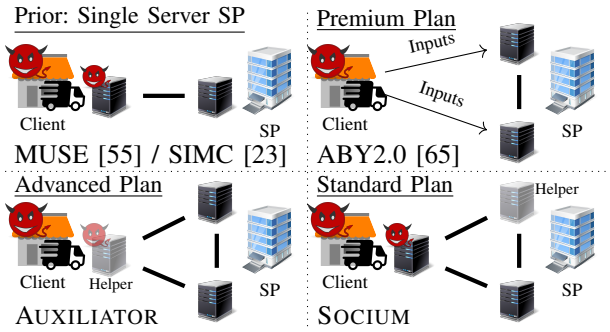

Figure 2: Comparing client-malicious setting: MUSE [55] / SIMC [23] with a single non-colluding server with SP and our service plans: Semi-honest 2PC outsourcing, e.g., using ABY2.0 [65], AUXILIATOR: 3PC with a malicious helper, and SOCIUM: 3PC with a malicious evaluator.

Outline of this work: In §2, we introduce the preliminaries needed for subsequent sections. §3 provides a technical overview of various techniques to protect against a malicious adversary and how they are adapted to our asymmetric setting. We then employ these techniques to construct our protocols AUXILIATOR in §4 and SOCIUM in §5. In §6, we perform both analytical and practical evaluations and present a comprehensive comparison between our protocols and related ones. Additional details relevant to this work are provided in Appendix A, and Appendix B contains more details on ML building blocks.

2. Preliminaries

Our protocols are designed to operate over the ring of integers modulo 2^ℓ for some $\ell \in \mathbb{N}$ that we denote by $\mathbb{Z}_{2^\ell}$. Note that setting $\ell > 1$ corresponds to computation in the arithmetic domain while $\ell = 1$ corresponds to the binary domain. The function to be computed is expressed as a hybrid circuit, i.e., a circuit that may use arithmetic as well as binary domains, and consists of multiplication / AND and addition / XOR gates, as well as more specialized gates such as matrix multiplication that are established throughout the paper. To improve protocol performance, we use input-independent but function-dependent preprocessing [24], [65]. Furthermore, the servers use a collision-resistant hash function, denoted by $\mathsf{H}()$, as well as a pre-shared pseudo-random function (PRF) key setup (see Appendix A.1) that facilitates the non-interactive generation of shared randomness [4], [60].

2.1. Notations

The computation is conducted by three servers $\mathsf{S}_0, \mathsf{S}_1, \mathsf{S}_2$ that are pair-wise connected by private and authentic channels in a synchronous network. Similar to existing 3PC protocols [24], [51], [66], we consider an asymmetric setting where S_0 helps to evaluate the given circuit while $\mathsf{S}_1, \mathsf{S}_2$ have the role of the evaluators. For simplicity, we sometimes write, e.g., S_{i+1} which for $i = 2$ denotes S_0.

We denote the computational security parameter by κ and the statistical security parameter by σ. For some of the secret-sharing schemes we use in this paper, we may switch between domains $\mathbb{Z}_{2^\ell}$ and $\mathbb{Z}_{2^{\ell+\sigma}}$. In this case, extending a sharing to the larger ring stands for padding all shares with σ many zeroes on their left, and reducing a sharing to the smaller ring stands for removing the σ most significant bits on the left, i.e., taking all shares modulo 2^ℓ. We distinguish sharing over different domains by adding an index $\ell + \sigma$ when considering domain $\mathbb{Z}_{2^{\ell+\sigma}}$ by writing, e.g., $\langle v \rangle_{\ell+\sigma}$ instead of $\langle v \rangle$. Regarding secret-sharing, we also use the notation $\langle \vec{\mathbf{A}} \rangle$ to denote a matrix $\vec{\mathbf{A}}$ that is secret-shared element-wise. Furthermore, by $\vec{\mathbf{0}}$ we denote a matrix with zero-entries only.

For applications dealing with real values like ML inference, we use the Fixed-Point Arithmetic (FPA) notation with negative values being encoded in signed two's complement representation [21], [22], [61]. An integer v in FPA encodes the value $v \cdot 2^{-d}$ for a fixed $d \in \mathbb{N}$ similar to works such as [24], [30], [60]. Moreover, operations like truncation after multiplications are handled similar to the 3PC protocols in ASTRA [24] and SWIFT [51] as these protocols form the base for our constructions.

2.2. Security Model

We consider a static malicious adversary $\mathcal{A}$, who gains control over one of the three servers $\mathsf{S}_0, \mathsf{S}_1, \mathsf{S}_2$ at the onset of the protocol. In contrast to a standard corruption scenario, where the adversary can freely choose any server to corrupt as long as it respects the maximum corruption threshold, we focus on a specific setting where the adversary $\mathcal{A}$ is only allowed to corrupt servers from a predetermined set of participating servers. This setting, which is referred to as *fixed-corruption setting* henceforth, has been recently explored in the domain of two-party computation (2PC) in works like [23], [36], [55], [76]. These works identified practical scenarios in secure ML inference, where such restricted corruption settings are applicable and demonstrated improvements in terms of security and performance.

In the fixed-corruption setting with 3 servers, a protocol providing security with abort is sufficient as extending it to provide robustness is very trivial. This is because, once a corruption is detected, the remaining servers can carry out the computation by excluding this server. This also means that achieving the slightly weaker notion of *private robustness* introduced in [30] is straightforward in our setting. However, with respect to the credit scoring application (see Fig. 1), we note a low incentive of the potentially malicious client of trying to abort the protocol execution as it is the sole receiver of outputs. Hence, we focus on security with abort in the remainder of this paper.

Our protocols are not intended to satisfy the recently introduced notion of Friends-and-Foes (FaF) security [2], in which a malicious party is allowed to send its transcript to a semi-honest party, with which the latter can try to break the scheme's privacy. Note that FaF security is proven to be unfeasible for three parties in [2]. Moreover, in the credit scoring application (see Fig. 1), a malicious client company sending its transcript to one of SP's servers contradicts this company's own incentive of protecting its inputs.

3. Protecting Against a Malicious Adversary

The central step of attaining security against a malicious adversary, independent of whether it may corrupt any or just some fixed party, is to ensure the correctness of multiplications, as this operation forms the building block requiring the majority of interaction. Here, we consider multiple popular verification methods over rings for this task that are well suited to being moved to the setup phase. First, verification by **triple sacrificing** [29], [31], [47] (§3.1) offers low computational complexity while requiring communication linear in the number of multiplication gates. Second, verification by **distributed zero-knowledge proofs** [12], [15] (§3.2) only requires sublinear communication at the expense of higher computational complexity. The third approach is the **cut-and-choose** method [3], [40], designed for the binary domain with logical AND for multiplication, presented in §A.2. Thus, we provide an opportunity to opt for either low computation or low communication given a specific network and hardware setting.

In this section, we review these verification methods and discuss how they adapt to our fixed corruption setting where only a single fixed server may turn malicious. Furthermore, we generalize from multiplication to the verification of matrix multiplication due to its significance, e.g., in the evaluation of linear layers in machine learning inference. For now, we simply assume some underlying linear secret-sharing scheme where $\langle v \rangle$ is the sharing of $v \in \mathbb{Z}_{2^\ell}$ between all servers and specify additional required properties where they are required.

3.1. Triple Sacrificing Approach

The original triple sacrifice method [31], [47] works over fields and verifies the correctness of a triple $(\langle a \rangle, \langle b \rangle, \langle c \rangle)$, i.e., that $c = ab$, by *sacrificing* a second correlated and potentially incorrect triple $(\langle \hat{a} \rangle, \langle b \rangle, \langle \hat{c} \rangle)$. Using the sacrifice method in rings requires to use a larger ring to circumvent problems that are introduced by non-zero elements not necessarily being invertible in a ring as discussed in [1], [29]. On a high level, this approach uses triples over the larger ring $\mathbb{Z}_{2^{\ell+\sigma}}$ to verify that a triple is valid in the smaller ring $\mathbb{Z}_{2^\ell}$, i.e., $c \equiv_{2^\ell} ab$ as shown in Proc. 1:

Procedure 1 Multiplication Triple Verification

1: Sample random $r \in \mathbb{Z}_{2^\sigma}$.
2: Compute $\langle v \rangle_{\ell+\sigma} = r\langle a \rangle_{\ell+\sigma} - \langle \hat{a} \rangle_{\ell+\sigma}$.
3: Reconstruct $v \in \mathbb{Z}_{2^{\ell+\sigma}}$.
4: Compute $\langle w \rangle_{\ell+\sigma} = v\langle b \rangle_{\ell+\sigma} - r\langle c \rangle_{\ell+\sigma} + \langle \hat{c} \rangle_{\ell+\sigma}$.
5: Reconstruct $w \in \mathbb{Z}_{2^{\ell+\sigma}}$ and `abort` if $w \not\equiv_{2^{\ell+\sigma}} 0$.

This method verifies the validity of triples by checking if $r(c - ab) \equiv_{2^{\ell+\sigma}} \hat{c} - \hat{a}b$. It fails with high probability if $(\langle a \rangle, \langle b \rangle, \langle c \rangle)$ is incorrect over the smaller ring $\mathbb{Z}_{2^\ell}$.

We observe that this approach can be generalized to matrix triples [61], i.e., triples $(\langle\vec{\mathbf{A}}\rangle_{\ell+\sigma}, \langle\vec{\mathbf{B}}\rangle_{\ell+\sigma}, \langle\vec{\mathbf{C}}\rangle_{\ell+\sigma})$ with $\vec{\mathbf{A}} \in \mathbb{Z}_{2^{\ell+\sigma}}^{u\times w}, \vec{\mathbf{B}} \in \mathbb{Z}_{2^{\ell+\sigma}}^{w\times v}, \vec{\mathbf{C}} \in \mathbb{Z}_{2^{\ell+\sigma}}^{u\times v}$, where we wish to verify that $\vec{\mathbf{C}} \equiv_{2^\ell} \vec{\mathbf{A}} \cdot \vec{\mathbf{B}}$. We generalize triple sacrificing as shown in Proc. 2:

Procedure 2 Matrix Triple Verification

1: Sample random $r \in \mathbb{Z}_{2^\sigma}$.
2: Compute $\langle\vec{\mathbf{V}}\rangle_{\ell+\sigma} = r\langle\vec{\mathbf{A}}\rangle_{\ell+\sigma} - \langle\vec{\hat{\mathbf{A}}}\rangle_{\ell+\sigma}$.
3: Reconstruct $\vec{\mathbf{V}} \in \mathbb{Z}_{2^{\ell+\sigma}}^{u\times w}$.
4: Compute $\langle\vec{\mathbf{W}}\rangle_{\ell+\sigma} = \vec{\mathbf{V}} \cdot \langle\vec{\mathbf{B}}\rangle_{\ell+\sigma} - r\langle\vec{\mathbf{C}}\rangle_{\ell+\sigma} + \langle\vec{\hat{\mathbf{C}}}\rangle_{\ell+\sigma}$.
5: Reconstruct $\vec{\mathbf{W}} \in \mathbb{Z}_{2^{\ell+\sigma}}^{u\times v}$ and `abort` if $\vec{\mathbf{W}} \not\equiv_{2^{\ell+\sigma}} \vec{\mathbf{0}}$.

First, we will discuss optimizations for the verification in Proc. 2, followed by a proof of its correctness.

For optimization, we batch multiple triple sacrifices. Unfortunately, batch verification results for fields as in [9], [64] do not directly generalize to rings. This is due to reliance on the Schwartz-Zippel lemma [72], [81] that itself relies on a non-zero polynomial of degree d over some field $\mathbb{F}$ having at most $d+1$ roots in $\mathbb{F}$. For a ring $\mathbb{Z}_{2^\ell}$, this is clearly not satisfied as $f(X) = 2^{\ell-1}X$ has degree 1 but $2^{\ell-1}$ many roots. In addition, [9], [64] require polynomial interpolation over fields which does not efficiently generalize to rings. Batched verification that outputs triples over $\mathbb{Z}_{2^\ell}$ exists [37], but internally utilizes computation over fields based on the work by [32] which increases computational complexity by further computation modulo a prime.

Instead, we use the batching technique from [37], that is fully compatible with the sacrifice approach from [1], [29]. Similar to [37], we use the same value r sampled in step 1 for all sacrifice instances that are executed in one batch. Here, it only needs to be ensured that r is sampled randomly and that it is unknown to a cheater before all required triples have been generated. Also, the reconstruction of $\vec{\mathbf{W}}$ for comparing all entries to 0 can be substituted with a hash-based approach, contingent on the compatibility of the secret-sharing scheme being utilized here. This approach not only achieves constant communication for an arbitrary number of batched sacrifice instances in step 5 but also ensures that this communication remains independent of matrix dimensions u and v. Further details, including the applied secret-sharing schemes, will be provided later.

The overall communication for one verification mainly comes from reconstructing $\vec{\mathbf{V}}$ in step 3, while the communication of steps 1 and 5 amortize to 0, and other steps are non-interactive. Thus, the communication cost is that of reconstructing uw elements of $\mathbb{Z}_{2^{\ell+\sigma}}$. For $u > v$, the cost can be improved to vw elements by generating the triple $(\langle\vec{\mathbf{B}}\rangle_{\ell+\sigma}^\intercal, \langle\vec{\mathbf{A}}\rangle_{\ell+\sigma}^\intercal, \langle\vec{\mathbf{C}}\rangle_{\ell+\sigma}^\intercal)$ and verifying $\vec{\mathbf{C}}^\intercal = \vec{\mathbf{B}}^\intercal \cdot \vec{\mathbf{A}}^\intercal$.

Lemma 3.1. *Assume that step 5 in Proc. 2 is implemented using a hash-based check that aborts if $\vec{\mathbf{W}} \not\equiv_{2^{\ell+\sigma}} \vec{\mathbf{0}}$ except for probability negligible in κ and accepts otherwise. Then, for triples $(\langle\vec{\mathbf{A}}\rangle_{\ell+\sigma}, \langle\vec{\mathbf{B}}\rangle_{\ell+\sigma}, \langle\vec{\mathbf{C}}\rangle_{\ell+\sigma})$ and $(\langle\vec{\hat{\mathbf{A}}}\rangle_{\ell+\sigma}, \langle\vec{\mathbf{B}}\rangle_{\ell+\sigma}, \langle\vec{\hat{\mathbf{C}}}\rangle_{\ell+\sigma})$, the matrix triple verification in Proc. 2 accepts if $\vec{\mathbf{C}} \equiv_{2^{\ell+\sigma}} \vec{\mathbf{A}}\cdot\vec{\mathbf{B}}$, and $\vec{\hat{\mathbf{C}}} \equiv_{2^{\ell+\sigma}} \vec{\hat{\mathbf{A}}}\cdot\vec{\mathbf{B}}$, i.e., no errors exists, and aborts except for probability negligible in σ, κ if $\vec{\mathbf{C}} \not\equiv_{2^\ell} \vec{\mathbf{A}} \cdot \vec{\mathbf{B}}$, i.e., the triple is incorrect over $\mathbb{Z}_{2^\ell}$.*

Proof. This proof is based on the proof for scalar multiplication in [29]. First, we define the potential errors introduced by a cheating server as $\vec{\boldsymbol{\Delta}}^{\mathbf{1}} = \vec{\mathbf{C}} - (\vec{\mathbf{A}} \cdot \vec{\mathbf{B}}) \mod 2^{\ell+\sigma}$ and $\vec{\boldsymbol{\Delta}}^{\mathbf{2}} = \vec{\hat{\mathbf{C}}} - (\vec{\hat{\mathbf{A}}} \cdot \vec{\mathbf{B}}) \mod 2^{\ell+\sigma}$. It holds that

$$\begin{aligned}\vec{\mathbf{W}} &\equiv_{2^{\ell+\sigma}} \left((r\vec{\mathbf{A}} - \vec{\hat{\mathbf{A}}}) \cdot \vec{\mathbf{B}}\right) - r\vec{\mathbf{C}} + \vec{\hat{\mathbf{C}}} \\ &\equiv_{2^{\ell+\sigma}} r(\vec{\mathbf{A}} \cdot \vec{\mathbf{B}}) - (\vec{\hat{\mathbf{A}}} \cdot \vec{\mathbf{B}}) - r\vec{\mathbf{C}} + \vec{\hat{\mathbf{C}}} \\ &\equiv_{2^{\ell+\sigma}} \vec{\boldsymbol{\Delta}}^{\mathbf{2}} - r\vec{\boldsymbol{\Delta}}^{\mathbf{1}}.\end{aligned}$$

For the case when no errors exist, i.e., $\vec{\boldsymbol{\Delta}}^{\mathbf{1}} = \vec{\boldsymbol{\Delta}}^{\mathbf{2}} = \vec{\mathbf{0}}$, $\vec{\mathbf{W}} \equiv_{2^{\ell+\sigma}} \vec{\mathbf{0}}$ and the hash-based check accepts.

Now, let us assume that $\vec{\mathbf{C}} \not\equiv_{2^\ell} \vec{\mathbf{A}} \cdot \vec{\mathbf{B}}$ and hence $\vec{\boldsymbol{\Delta}}^{\mathbf{1}} \not\equiv_{2^\ell} \vec{\mathbf{0}}$, while the hash-based check accepts. Then, it holds that $\vec{\mathbf{W}} \equiv_{2^{\ell+\sigma}} \vec{\mathbf{0}}$ except for probability negligible in the computational security parameter κ. Let (i,j) be the coordinates where $\mathbf{C}_{i,j} \not\equiv_{2^\ell} (\vec{\mathbf{A}}\cdot\vec{\mathbf{B}})_{i,j}$ and thus, $\boldsymbol{\Delta}^1_{i,j} \not\equiv_{2^\ell} 0$. Let $k \in \mathbb{Z}$ be maximal such that 2^k divides $\boldsymbol{\Delta}^1_{i,j}$. Note that $k < \ell$ and that $\frac{\boldsymbol{\Delta}^1_{i,j}}{2^k}$ can be inverted modulo $2^{\ell+\sigma-k}$. Thus,

$$\begin{aligned}\mathbf{W}_{i,j} &\equiv_{2^{\ell+\sigma}} \vec{\mathbf{0}} \\ \Rightarrow \boldsymbol{\Delta}^2_{i,j} &\equiv_{2^{\ell+\sigma}} r\boldsymbol{\Delta}^1_{i,j} \\ \Rightarrow \frac{\boldsymbol{\Delta}^2_{i,j}}{2^k} &\equiv_{2^{\ell+\sigma-k}} r\frac{\boldsymbol{\Delta}^1_{i,j}}{2^k} \\ \Rightarrow \frac{\boldsymbol{\Delta}^2_{i,j}}{2^k} \cdot \left(\frac{\boldsymbol{\Delta}^1_{i,j}}{2^k}\right)^{-1} &\equiv_{2^{\ell+\sigma-k}} r.\end{aligned}$$

As r is unknown to the cheater until after all triples have been generated, $\boldsymbol{\Delta}^1_{i,j}, \boldsymbol{\Delta}^2_{i,j}$ have to be picked independent of r. Hence, this equivalence holds with probability of at most $2^{-\sigma-(\ell-k)} < 2^{-\sigma}$ rendering the probability of the verification accepting for incorrect $\vec{\mathbf{C}}$ negligible in σ, κ. □

3.1.1. Triple Sacrificing in the Fully Malicious Setting. The triple verification approach in [37] uses replicated secret-sharing (RSS) which works as follows: For a value $v \in \mathbb{Z}_{2^\ell}$, $\langle v\rangle$ is called a RSS-share of v, if S_0 holds $v^0, v^2 \in \mathbb{Z}_{2^\ell}$, S_1 holds $v^1, v^0 \in \mathbb{Z}_{2^\ell}$, and S_2 holds $v^2, v^1 \in \mathbb{Z}_{2^\ell}$, with $v^0 + v^1 + v^2 = v$.[4] With this sharing, the amortized cost for step 1 in Proc. 2 becomes 0 as it only requires each of the three servers to secret-share a fresh random value. Similarly, step 3 incurs an amortized communication of $3uw$ elements from $\mathbb{Z}_{2^{\ell+\sigma}}$ [37]. Finally, [37] implements the hash based check in step 5 as follows (generalized to matrix triples here): For $0 \le i < 3$, S_i broadcasts[5] $h_i = \mathsf{H}(\vec{\mathbf{W}}^{\mathbf{0}}||\vec{\mathbf{W}}^{\mathbf{1}}||\vec{\mathbf{W}}^{\mathbf{2}})$. For this, S_i assumes that $\vec{\mathbf{W}}^{\mathbf{i+1}} = -\vec{\mathbf{W}}^{\mathbf{i}} - \vec{\mathbf{W}}^{\mathbf{i-1}}$ as this is the only share it does not hold directly. Then, if there is no hash collision, $\vec{\mathbf{W}} \not\equiv_{2^{\ell+\sigma}} \vec{\mathbf{0}}$ implies that $h_0 = h_1 = h_2$ does not hold which will be detected by an honest server.

4. ℓ is replaced by $\ell+\sigma$ when operating over a larger ring.
5. An echo-broadcast is sufficient for the case of abort.

3.1.2. Triple Sacrificing in the Fixed Corruption Setting. When only one fixed server can behave maliciously, we can significantly simplify the verification process. W.l.o.g., assuming S_0 and S_1 are semi-honest, we can locally transform a replicated sharing $\langle v \rangle$ into an additive sharing $[u]$ between these servers. S_0 holds $u^0 = v^0 + v^2$, and S_1 holds $u^1 = v^1$, ensuring $u = v$. This allows them to perform the verification in a semi-honest manner, without the third server.

Step 1 in Proc. 2 becomes non-interactive as $\mathsf{S}_0, \mathsf{S}_1$ can locally sample r unknown to malicious S_2. Step 3 simplifies to semi-honest reconstruction among $\mathsf{S}_0, \mathsf{S}_1$, requiring communication of $2uw$ elements from $\mathbb{Z}_{2^{\ell+\sigma}}$. Finally, $\vec{\mathbf{W}} = \vec{\mathbf{W}}^{\mathbf{0}} + \vec{\mathbf{W}}^{\mathbf{1}} = \vec{\mathbf{0}}$ where $\vec{\mathbf{W}}^{\mathbf{0}}, \vec{\mathbf{W}}^{\mathbf{1}}$ are the $[\cdot]$-shares, if and only if $\vec{\mathbf{W}}^{\mathbf{0}} = -\vec{\mathbf{W}}^{\mathbf{1}}$ in $\mathbb{Z}_{2^{\ell+\sigma}}$. This can be verified by S_0 computing $h = \mathsf{H}(\vec{\mathbf{W}}^{\mathbf{0}})$ and sending it to S_1 that then checks if $h = \mathsf{H}(-\vec{\mathbf{W}}^{\mathbf{1}})$.

3.1.3. Computation on Binary Rings. Adapting the triple sacrificing for the binary domain ($\ell = 1$) results in a large overhead by lifting a single bit to a $\sigma + 1$ bit ring. E.g., in a fully malicious setting, this would require 369 bits of communication in the preprocessing for a single AND gate when $\sigma = 40$ (cf. §6). Instead, we switch to a computationally lightweight cut-and-choose approach [3], [40] adapted to our setting, and the details are provided in Appendix A.2.

3.2. Distributed Zero-Knowledge Proofs Approach

Using distributed zero-knowledge proofs (DZKPs) [12], [15], one can verify the correctness of multiplication at zero amortized communication overhead but higher local computation than approaches like triple sacrificing (§3.1) and cut-and-choose (Appendix A.2).

At a high level, verification of a multiplication using DZKPs [15] involves each server proving the correctness of their multiplication messages. In the context of 3PC RSS protocols, [15] shows how to achieve this over the maliciously-private[6] protocol of [4], by utilizing the collective availability of values in the statement among the other two servers. The servers then construct a circuit for the statement to be proved that evaluates to 0 if everything is correct. [15] proposes a 2-round and $\log_2(m)$ variant for verifying a batch of m multiplications with trade-offs in computation and communication.

Our 3PC protocols, AUXILIATOR (§4) and SOCIUM (§5), will seamlessly integrate with the DZKP approach above. We employ an extended version of DZKP [15] for dot products, as proposed in SWIFT [51], [75] and optimized for the known corruption setting. The specifics, along with a detailed cost analysis for both variants of DZKP for SWIFT, are provided in Appendix A.3.

4. AUXILIATOR: Untrusted Helper Case

We introduce AUXILIATOR, a 3PC protocol designed to protect against a malicious helper server. In this protocol, two semi-honest evaluators $\mathsf{S}_1, \mathsf{S}_2$ run an optimized online phase with assistance from a potentially malicious helper server S_0 during the setup phase.

AUXILIATOR is based on the 3PC semi-honest protocol ASTRA [24], [75], where S_0 also acts as a helper during setup phase. Both AUXILIATOR and ASTRA exhibit high performance in arithmetic and binary worlds, with a particularly fast online phase that only involves two servers,[7] further improving practical performance.

Since the malicious server S_0 is only involved in the setup phase, additional verification as discussed in §3 incurs no online overhead over ASTRA. We now review ASTRA briefly and discuss the modifications made to create AUXILIATOR, which protect against the malicious S_0.

4.1. Sharing Semantics

AUXILIATOR utilizes the same secret sharing schemes as ASTRA, including intermediate $[\cdot]$-sharing and $\langle \cdot \rangle$-sharing.

Intermediate $[\cdot]$-sharing. For a value $v \in \mathbb{Z}_{2^\ell}$, $[v]$ is an additive sharing of v between S_1 and S_2, i.e., S_1 holds v^1 and S_2 holds v^2 such that $v^1 + v^2 = v$.

$\langle \cdot \rangle$-sharing. For $v \in \mathbb{Z}_{2^\ell}$, $\langle v \rangle$ is a replicated sharing of v, where a random mask $\lambda_v \in \mathbb{Z}_{2^\ell}$ is $[\cdot]$-shared between $\mathsf{S}_1, \mathsf{S}_2$ with S_0 holding both shares of $[\lambda_v]$ and $\mathsf{S}_1, \mathsf{S}_2$ hold value $\mathsf{m}_v \in \mathbb{Z}_{2^\ell}$ s.t. $\mathsf{m}_v + \lambda_v = v$. Writing down all shares reveals the replicated nature of this scheme:

$$\mathsf{S}_0 : (\lambda_v^1, \lambda_v^2), \quad \mathsf{S}_1 : (\lambda_v^1, \mathsf{m}_v), \quad \mathsf{S}_2 : (\lambda_v^2, \mathsf{m}_v)$$

The input sharing of a $v \in \mathbb{Z}_{2^\ell}$ by S_i follows ASTRA, where servers non-interactively sample λ_v^1 and λ_v^2 s.t. S_i learns λ_v. This is followed by S_i sending $\mathsf{m}_v = v - \lambda_v$ to S_1 and S_2. In AUXILIATOR, a corrupt S_0 may send inconsistent m_v. This can be easily thwarted by a hash-based check over the m_v values among S_1 and S_2. Similarly, reconstruction of a value can be accomplished with S_1 and S_2, as they possess the missing shares. Both secret-sharing schemes are *linear*, meaning they allow computing any linear combination of shared values without interaction. Additionally, any public value or value known only to $\mathsf{S}_1, \mathsf{S}_2$ can be non-interactively added to $[v]$ by adding it to v^1 or to $\langle v \rangle$ by adding it to m_v.

4.2. Multiplication

ASTRA and AUXILIATOR differ mainly in their multiplication protocols. This section explores safeguarding against a malicious S_0 in ASTRA's matrix multiplication.[8] We prove security of AUXILIATOR in Appendix A.4.

Given $\langle \vec{\mathbf{X}} \rangle, \langle \vec{\mathbf{Y}} \rangle$ for some matrices $\vec{\mathbf{X}} \in \mathbb{Z}_{2^\ell}^{u \times w}, \vec{\mathbf{Y}} \in \mathbb{Z}_{2^\ell}^{w \times v}$, it holds that

$$\begin{aligned}\vec{\mathbf{Z}} := \vec{\mathbf{X}} \cdot \vec{\mathbf{Y}} &= (\vec{\mathsf{m}}_{\mathbf{X}} + \vec{\lambda}_{\mathbf{X}}) \cdot (\vec{\mathsf{m}}_{\mathbf{Y}} + \vec{\lambda}_{\mathbf{Y}}) \\ &= \vec{\mathsf{m}}_{\mathbf{X}} \cdot \vec{\mathsf{m}}_{\mathbf{Y}} + \vec{\mathsf{m}}_{\mathbf{X}} \cdot \vec{\lambda}_{\mathbf{Y}} + \vec{\lambda}_{\mathbf{X}} \cdot \vec{\mathsf{m}}_{\mathbf{Y}} + \vec{\lambda}_{\mathbf{X}} \cdot \vec{\lambda}_{\mathbf{Y}}.\end{aligned}$$

6. Privacy is maintained even during malicious corruption.

7. If S_0 provides inputs or receives output, it also participates in the input and output phase at a low cost.

8. [24] does only contain protocols for multiplication and dot products, but the generalization to matrices presented here is straightforward.

ASTRA computes $\langle \vec{\mathbf{Z}} \rangle = \langle \vec{\mathbf{X}} \cdot \vec{\mathbf{Y}} \rangle$ based on the above observation, as shown in Proc. 3.

Procedure 3 Matrix Multiplication in ASTRA [24], [75]

1: S_0 generates $[\cdot]$-sharing of $\vec{\gamma}_{\mathbf{XY}} = \vec{\lambda}_{\mathbf{X}} \cdot \vec{\lambda}_{\mathbf{Y}}$ among $\mathsf{S}_1, \mathsf{S}_2$.
2: Servers locally sample random $[\vec{\lambda}_{\mathbf{Z}}]$ s.t. S_0 learns $\vec{\lambda}^{\mathbf{1}}_{\mathbf{Z}}, \vec{\lambda}^{\mathbf{2}}_{\mathbf{Z}}$.
3: $\mathsf{S}_1, \mathsf{S}_2$ compute $[\vec{\mathbf{Z}}] = \vec{\mathbf{m}}_{\mathbf{X}} \cdot \vec{\mathbf{m}}_{\mathbf{Y}} + \vec{\mathbf{m}}_{\mathbf{X}} \cdot [\vec{\lambda}_{\mathbf{Y}}] + [\vec{\lambda}_{\mathbf{X}}] \cdot \vec{\mathbf{m}}_{\mathbf{Y}} + [\vec{\gamma}_{\mathbf{XY}}]$.
4: $\mathsf{S}_1, \mathsf{S}_2$ reconstruct $\vec{\mathbf{m}}_{\mathbf{Z}}$ by exchanging shares of $[\vec{\mathbf{Z}}] - [\vec{\lambda}_{\mathbf{Z}}]$.

Incorporating truncation. While operating over fixed-point arithmetic, Proc. 3 allows free probabilistic truncation [61]. This is achieved by servers using $[\vec{\mathbf{Z}}/2^d]$ in steps 3 and 4 of Proc. 3, instead of $[\vec{\mathbf{Z}}]$ [24], [75].

Malicious S_0. The only opportunity for S_0 to cheat in Proc. 3 is in step 1, where it can generate $[\cdot]$-shares for an arbitrary $\vec{\gamma}_{\mathbf{XY}} \neq \vec{\lambda}_{\mathbf{X}} \cdot \vec{\lambda}_{\mathbf{Y}}$. By outsourcing this step to an ideal functionality $\mathcal{F}_{\mathsf{MultPre}}$, the remaining steps become secure against both semi-honest and malicious helper settings. AUXILIATOR uses this version and is presented in Fig. 3.

Protocol $\Pi_{\mathsf{Mult}}(\langle \vec{\mathbf{X}} \rangle, \langle \vec{\mathbf{Y}} \rangle)$

Input: $\langle \cdot \rangle$-shares of $\vec{\mathbf{X}} \in \mathbb{Z}_{2^\ell}^{u \times w}$, $\vec{\mathbf{Y}} \in \mathbb{Z}_{2^\ell}^{w \times v}$.

Output: $\langle \cdot \rangle$-shares of $\vec{\mathbf{Z}} = \vec{\mathbf{X}} \cdot \vec{\mathbf{Y}} \in \mathbb{Z}_{2^\ell}^{u \times v}$.

Setup:

1. Invoke $\mathcal{F}_{\mathsf{MultPre}}([\vec{\lambda}_{\mathbf{X}}], [\vec{\lambda}_{\mathbf{Y}}])$ to obtain $[\vec{\gamma}_{\mathbf{XY}} = \vec{\lambda}_{\mathbf{X}} \cdot \vec{\lambda}_{\mathbf{Y}}]$.
2. Non-interactively generate $[\vec{\lambda}_{\mathbf{Z}}]$ by $\mathsf{S}_0, \mathsf{S}_1$ sampling random $\vec{\lambda}^{\mathbf{1}}_{\mathbf{Z}} \in \mathbb{Z}_{2^\ell}^{u \times v}$, $\mathsf{S}_0, \mathsf{S}_2$ sampling random $\vec{\lambda}^{\mathbf{2}}_{\mathbf{Z}} \in \mathbb{Z}_{2^\ell}^{u \times v}$.

Online:

1. For $i \in \{1, 2\}$, locally compute the following:
   - S_i: $\vec{\mathbf{m}}^i_{\mathbf{Z}} = (2-i) \cdot \vec{\mathbf{m}}_{\mathbf{X}} \cdot \vec{\mathbf{m}}_{\mathbf{Y}} + \vec{\mathbf{m}}_{\mathbf{X}} \cdot \vec{\lambda}^{\mathbf{i}}_{\mathbf{Y}} + \vec{\lambda}^{\mathbf{i}}_{\mathbf{X}} \cdot \vec{\mathbf{m}}_{\mathbf{Y}} + \vec{\gamma}^{\mathbf{i}}_{\mathbf{XY}} - \vec{\lambda}^{\mathbf{i}}_{\mathbf{Z}}$.
2. $\mathsf{S}_1, \mathsf{S}_2$ exchange $\vec{\mathbf{m}}^1_{\mathbf{Z}}, \vec{\mathbf{m}}^2_{\mathbf{Z}}$ and reconstruct $\vec{\mathbf{m}}_{\mathbf{Z}} = \vec{\mathbf{m}}^1_{\mathbf{Z}} + \vec{\mathbf{m}}^2_{\mathbf{Z}}$

Figure 3: Matrix multiplication in AUXILIATOR.

4.2.1. Instantiating $\mathcal{F}_{\mathsf{MultPre}}$ in AUXILIATOR. In ASTRA, $\mathcal{F}_{\mathsf{MultPre}}$ consists of S_0 and S_1 sampling random $\vec{\gamma}^{\mathbf{1}}_{\mathbf{XY}} \in \mathbb{Z}_{2^\ell}^{u \times v}$, and S_0 sending $\vec{\gamma}^{\mathbf{2}}_{\mathbf{XY}} = \vec{\lambda}_{\mathbf{X}} \cdot \vec{\lambda}_{\mathbf{Y}} - \vec{\gamma}^{\mathbf{1}}_{\mathbf{XY}}$ to S_2. This semi-honest instantiation of $\mathcal{F}_{\mathsf{MultPre}}$ is denoted as $\Pi^{\mathsf{semi}}_{\mathsf{MultPre}}$. To implement $\mathcal{F}_{\mathsf{MultPre}}$ securely in the presence of a malicious S_0, AUXILIATOR first uses $\Pi^{\mathsf{semi}}_{\mathsf{MultPre}}$ to compute $[\vec{\gamma}_{\mathbf{XY}}]$. This is followed by a verification, using the techniques from §3, and the details are presented next.

Triple Sacrificing and Cut-and-Choose: As discussed in §3.1, a triple $([\vec{\mathbf{A}}], [\vec{\mathbf{B}}], [\vec{\mathbf{C}}])$ is verified using a second triple $([\hat{\vec{\mathbf{A}}}], [\hat{\vec{\mathbf{B}}}], [\hat{\vec{\mathbf{C}}}])$ with both triples being over a larger ring $\mathbb{Z}_{2^{\ell+\sigma}}$. Now, to verify the triple $([\vec{\lambda}_{\mathbf{X}}], [\vec{\lambda}_{\mathbf{y}}], [\vec{\gamma}_{\mathbf{XY}}])$, servers locally map the triple to $([\vec{\mathbf{A}}], [\vec{\mathbf{B}}], [\vec{\mathbf{C}}])$, but over the larger ring $\mathbb{Z}_{2^{\ell+\sigma}}$ by extending each share's entries by σ number of zero bits. Regarding the second triple, random $[\hat{\vec{\mathbf{A}}}]$ can be generated non-interactively, while $[\hat{\vec{\mathbf{C}}}]$ can be obtained by running $\Pi^{\mathsf{semi}}_{\mathsf{MultPre}}$ on $[\hat{\vec{\mathbf{A}}}]$ and $[\vec{\mathbf{B}}]$. The remaining steps follows the verification described in §3.1.2.

Note that the local ring extension mentioned earlier might not maintain correctness over the larger ring. E.g., we may have $\vec{\mathbf{A}} \not\equiv_{2^{\ell+\sigma}} \vec{\lambda}_{\mathbf{X}}$, but we can be certain that $\vec{\mathbf{A}} \equiv_{2^\ell} \vec{\lambda}_{\mathbf{X}}$. This suffices as the final result of $\mathcal{F}_{\mathsf{MultPre}}$ will be converted back to $\mathbb{Z}_{2^\ell}$ for the subsequent online phase to execute on.

Communication. The amortized cost of instantiating $\mathcal{F}_{\mathsf{MultPre}}$ in AUXILIATOR with triple sacrificing for matrices in $\mathbb{Z}_{2^\ell}^{u \times w}, \mathbb{Z}_{2^\ell}^{w \times v}$ is $2u \cdot (v+w)$ elements of $\mathbb{Z}_{2^{\ell+\sigma}}$ ($2 \times \Pi^{\mathsf{semi}}_{\mathsf{MultPre}}$ + verification in §3.1.2). Also, using cut-and-choose for AND incurs a communication of 11 bits (Appendix A.2.2). Table 2 in §6.1 provides an overview of the communication costs.

Distributed Zero-Knowledge Proofs: To verify the correctness of $\Pi^{\mathsf{semi}}_{\mathsf{MultPre}}$, AUXILIATOR can use the extension of DZKPs to dot products, proposed in SWIFT [51], [75]. Note that each entry p of $\vec{\gamma}_{\mathbf{XY}}$ is the dot product of one row $\vec{\mathbf{a}}$ of $\vec{\lambda}_{\mathbf{X}}$ and a column $\vec{\mathbf{b}}$ of $\vec{\lambda}_{\mathbf{Y}}$. Hence, we can proceed by verifying the correctness of each such dot product $\vec{\mathbf{a}} \cdot \vec{\mathbf{b}} = p$ by checking the circuit $c\left(\left\{\vec{\mathbf{a}}^1_k, \vec{\mathbf{a}}^2_k, \vec{\mathbf{b}}^1_k, \vec{\mathbf{b}}^2_k\right\}_{k=1}^{w}, p^1, p^2\right) := \sum_{k=1}^{w}\left(\vec{\mathbf{a}}^1_k \vec{\mathbf{b}}^1_k + \vec{\mathbf{a}}^1_k \vec{\mathbf{b}}^2_k + \vec{\mathbf{a}}^2_k \vec{\mathbf{b}}^1_k + \vec{\mathbf{a}}^2_k \vec{\mathbf{b}}^2_k\right) - p^1 - p^2 = 0$. Here, each variable is known to S_0 as well as to at least S_1 or S_2 and thus the DZKPs approach can be used directly. The case for binary domain is similar except circuit $c()$ being replaced by a binary domain polynomial over 6 variables.

Communication: The amortized cost of instantiating $\mathcal{F}_{\mathsf{MultPre}}$ with DZKPs for matrices in $\mathbb{Z}_{2^\ell}^{u \times w}, \mathbb{Z}_{2^\ell}^{w \times v}$ is uv elements of $\mathbb{Z}_{2^\ell}$ for running $\Pi^{\mathsf{semi}}_{\mathsf{MultPre}}$, while the proof cost amortizes to zero (see Appendix A.3). Furthermore, the amortized cost of $\mathcal{F}_{\mathsf{MultPre}}$ for an AND operation is 1 bit.

5. SOCIUM: Untrusted Evaluator Case

In scenarios where the malicious server takes on the role of an evaluator instead of a helper (as in AUXILIATOR), we utilize our second protocol called SOCIUM. In this setup, the semi-honest S_0 acts as a helper, only participating in a lightweight verification phase during the online execution, while most of the verification occurs in the setup phase. To avoid the expensive extension of the semi-honest ASTRA protocol to safeguard against a potentially cheating evaluator in the online phase (cf. ASTRA [24, §4.2]), we opt for the robust and fully malicious 3PC protocol of SWIFT [51][9] as the basis for SOCIUM. SWIFT has the advantage of moving all expensive verification to the setup phase. In the following, we consider S_2 as the malicious server, with the case for malicious S_1 being symmetrical. Leveraging the semi-honest behavior of S_0 and S_1, we modify and optimize SWIFT to suit our fixed corruption setting, resulting in our novel protocol SOCIUM.

5.1. Sharing Semantics

In SOCIUM, we employ secret-sharing schemes from SWIFT, including intermediate $[\cdot]$-sharings, $\langle \cdot \rangle$-sharings, and $[\![\cdot]\!]$-sharings. It is important to note that although we use the

9. We use the simplified version of SWIFT presented in [75].

same notations as AUXILIATOR, the sharing semantics are not identical, despite some similarities.

Intermediate $[\cdot]$-sharing. $[v]$ is an additive sharing of $v \in \mathbb{Z}_{2^\ell}$, where S_0 holds v^0, S_1 holds v^1 and S_2 holds v^2 s.t. $v = v^0 + v^1 + v^2$.

Intermediate $\langle\cdot\rangle$-sharing. $\langle v \rangle$ is a replicated sharing of $v \in \mathbb{Z}_{2^\ell}$, where S_0 holds (v^0, v^2), S_1 holds (v^1, v^0), and S_2 holds (v^2, v^1) s.t. $v^0 + v^1 + v^2 = v$.

$[\![\cdot]\!]$-sharing. Sharing $[\![v]\!]$ of a value $v \in \mathbb{Z}_{2^\ell}$ consists of a random mask $\lambda_v \in \mathbb{Z}_{2^\ell}$ that is $\langle\cdot\rangle$-shared between $\mathsf{S}_0, \mathsf{S}_1, \mathsf{S}_2$ and $\mathsf{m}_v \in \mathbb{Z}_{2^\ell}$ held by $\mathsf{S}_0, \mathsf{S}_1, \mathsf{S}_2$ s.t. $v = \mathsf{m}_v + \lambda_v$. Written out, the shares are:

$$\mathsf{S}_0 : (\mathsf{m}_v, \lambda_v^0, \lambda_v^2), \quad \mathsf{S}_1 : (\mathsf{m}_v, \lambda_v^1, \lambda_v^0), \quad \mathsf{S}_2 : (\mathsf{m}_v, \lambda_v^2, \lambda_v^1).$$

The input sharing and output reconstruction protocols in SOCIUM closely resemble those in AUXILIATOR (§4.1). However, in SOCIUM, the consistency check[10] is necessary only if a malicious S_2 shares a value. Additionally, all the sharings are linear, enabling the computation of linear combinations of shared values in a non-interactive manner.

5.2. Multiplication

We begin with recalling the matrix multiplication in SWIFT. Similar to AUXILIATOR, SWIFT uses the following relation, as depicted in Proc. 4:

$$\vec{\mathbf{Z}} := \vec{\mathbf{X}} \cdot \vec{\mathbf{Y}} = \vec{\mathsf{m}}_{\mathbf{X}} \cdot \vec{\mathsf{m}}_{\mathbf{Y}} + \vec{\mathsf{m}}_{\mathbf{X}} \cdot \vec{\lambda}_{\mathbf{Y}} + \vec{\lambda}_{\mathbf{X}} \cdot \vec{\mathsf{m}}_{\mathbf{Y}} + \vec{\lambda}_{\mathbf{X}} \cdot \vec{\lambda}_{\mathbf{Y}}.$$

The shares of $\langle \vec{\mathsf{m}}_{\mathbf{Z}} \rangle$ in step 4 of Proc. 4 are distributed among the servers as follows:

$$\begin{aligned}
\mathsf{S}_0, \mathsf{S}_1 &: \vec{\mathsf{m}}^0_{\mathbf{Z}} = \vec{\mathsf{m}}_{\mathbf{X}} \cdot \vec{\lambda}^{\mathbf{0}}_{\mathbf{Y}} + \vec{\lambda}^{\mathbf{0}}_{\mathbf{X}} \cdot \vec{\mathsf{m}}_{\mathbf{Y}} + \vec{\gamma}^{\mathbf{0}}_{\mathbf{XY}} - \vec{\lambda}^{\mathbf{0}}_{\mathbf{Z}} \\
\mathsf{S}_1, \mathsf{S}_2 &: \vec{\mathsf{m}}^1_{\mathbf{Z}} = \vec{\mathsf{m}}_{\mathbf{X}} \cdot \vec{\lambda}^{\mathbf{1}}_{\mathbf{Y}} + \vec{\lambda}^{\mathbf{1}}_{\mathbf{X}} \cdot \vec{\mathsf{m}}_{\mathbf{Y}} + \vec{\gamma}^{\mathbf{1}}_{\mathbf{XY}} - \vec{\lambda}^{\mathbf{1}}_{\mathbf{Z}} \\
\mathsf{S}_2, \mathsf{S}_0 &: \vec{\mathsf{m}}^2_{\mathbf{Z}} = \vec{\mathsf{m}}_{\mathbf{X}} \cdot \vec{\lambda}^{\mathbf{2}}_{\mathbf{Y}} + \vec{\lambda}^{\mathbf{2}}_{\mathbf{X}} \cdot \vec{\mathsf{m}}_{\mathbf{Y}} + \vec{\gamma}^{\mathbf{2}}_{\mathbf{XY}} - \vec{\lambda}^{\mathbf{2}}_{\mathbf{Z}} + \vec{\mathsf{m}}_{\mathbf{X}} \cdot \vec{\mathsf{m}}_{\mathbf{Y}}
\end{aligned}$$

Procedure 4 Matrix Multiplication in SWIFT [51], [75]

1: Generate $\langle\cdot\rangle$-sharing of $\vec{\gamma}_{\mathbf{XY}} = \vec{\lambda}_{\mathbf{X}} \cdot \vec{\lambda}_{\mathbf{Y}}$ using $\mathcal{F}_{\mathsf{MultPre}}$.
2: Locally sample random $\langle \vec{\lambda}_{\mathbf{Z}} \rangle$.
3: Compute $\langle \vec{\mathbf{Z}} \rangle = \vec{\mathsf{m}}_{\mathbf{X}} \cdot \vec{\mathsf{m}}_{\mathbf{Y}} + \vec{\mathsf{m}}_{\mathbf{X}} \cdot \langle \vec{\lambda}_{\mathbf{Y}} \rangle + \langle \vec{\lambda}_{\mathbf{X}} \rangle \cdot \vec{\mathsf{m}}_{\mathbf{Y}} + \langle \vec{\gamma}_{\mathbf{XY}} \rangle$.
4: Reconstruct $\vec{\mathsf{m}}_{\mathbf{Z}}$ by exchanging shares of $\langle \vec{\mathsf{m}}_{\mathbf{Z}} \rangle = \langle \vec{\mathbf{Z}} \rangle - \langle \vec{\lambda}_{\mathbf{Z}} \rangle$.

In SWIFT, all the messages sent to or received from helper S_0 are delayed to a final verification phase, allowing S_0 to remain inactive for most of the online phase of the protocol. The specific sequence of sent messages during the online phase, using hash-based consistency checks, is illustrated in Fig. 4 for a small example circuit.

Each layer of multiplications in Fig. 4, corresponding to $\vec{\mathbf{Z}}$ and $\vec{\mathbf{W}}$, requires one round where $\mathsf{S}_1, \mathsf{S}_2$ interact. S_0 receives $\vec{\mathsf{m}}^1_{\mathbf{U}}$ for each multiplication output $\vec{\mathbf{U}} \in \{\vec{\mathbf{Z}}, \vec{\mathbf{W}}\}$ only during the final verification in a single round. This is followed by sending of hash values by S_0 to $\mathsf{S}_1, \mathsf{S}_2$. Thus, the verification phase requires only constant 2 rounds for arbitrarily many previous online rounds introduced by the depth of the computed circuit.

10. SWIFT ensures consistency through its jsend primitive, which also identifies an honest party when detecting inconsistency. Although this is necessary for robustness, the hash-based check is enough to achieve security with abort in SWIFT [75]. In our fixed-corruption setting, abort security is sufficient, making robustness straightforward.

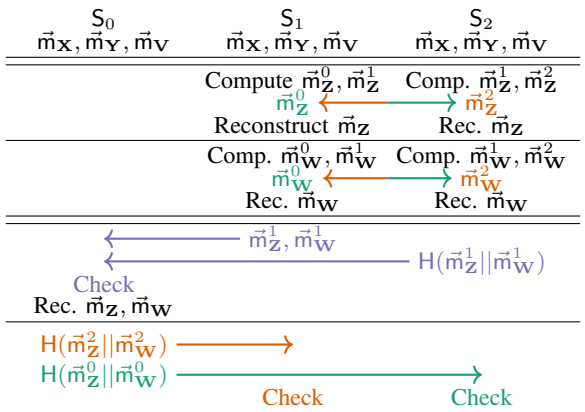

Figure 4: Messages sent during the online phase in SWIFT while computing $[\![\vec{\mathbf{W}}]\!] = \left([\![\vec{\mathbf{X}}]\!] \cdot [\![\vec{\mathbf{Y}}]\!]\right) \cdot [\![\vec{\mathbf{V}}]\!] = [\![\vec{\mathbf{Z}}]\!] \cdot [\![\vec{\mathbf{V}}]\!]$ (excludes input sharing and output reconstruction). The sent values are colored to match their respective consistency checks.

Malicious S_2. We now explain how to tackle a malicious S_2 in SOCIUM. Note that any message from S_0 or S_1 doesn't need a consistency check, reducing required checks by $3\times$. Also, we let semi-honest S_1 to send $\vec{\mathsf{m}}^0_{\mathbf{Z}} + \vec{\mathsf{m}}^1_{\mathbf{Z}}$ directly to S_2. This simplifies the protocol by eliminating the need for S_2 to compute $\vec{\mathsf{m}}^1_{\mathbf{Z}}$ during the online phase, which also removes the requirement for S_2 to obtain $\vec{\gamma}^{\mathbf{1}}_{\mathbf{XY}}$ in the setup phase.

Protocol $\Pi_{\mathsf{Mult}}([\![\vec{\mathbf{X}}]\!], [\![\vec{\mathbf{Y}}]\!])$

Input: $[\![\cdot]\!]$-shares of $\vec{\mathbf{X}} \in \mathbb{Z}_{2^\ell}^{u \times w}$, $\vec{\mathbf{Y}} \in \mathbb{Z}_{2^\ell}^{w \times v}$.

Output: $[\![\cdot]\!]$-shares of $\vec{\mathbf{Z}} = \vec{\mathbf{X}} \cdot \vec{\mathbf{Y}} \in \mathbb{Z}_{2^\ell}^{u \times v}$.

Setup:

1. Invoke $\mathcal{F}'_{\mathsf{MultPre}}$ on $\langle \vec{\lambda}_{\mathbf{X}} \rangle$ and $\langle \vec{\lambda}_{\mathbf{Y}} \rangle$ to obtain $\langle \vec{\gamma}_{\mathbf{XY}} \rangle'$ with $\vec{\gamma}_{\mathbf{XY}} = \vec{\lambda}_{\mathbf{X}} \cdot \vec{\lambda}_{\mathbf{Y}}$ (S_0 has $\vec{\gamma}^{\mathbf{0}}_{\mathbf{XY}}, \vec{\gamma}^{\mathbf{2}}_{\mathbf{XY}}$; S_1 has $\vec{\gamma}^{\mathbf{1}}_{\mathbf{XY}}, \vec{\gamma}^{\mathbf{0}}_{\mathbf{XY}}$; S_2 has $\vec{\gamma}^{\mathbf{2}}_{\mathbf{XY}}$).
2. Non-interactively generate $\langle \vec{\lambda}_{\mathbf{Z}} \rangle$ by $\mathsf{S}_i, \mathsf{S}_{i+1}$ sampling random $\vec{\lambda}^{\mathbf{i}}_{\mathbf{Z}} \in \mathbb{Z}_{2^\ell}$ for $i \in \{0, 1, 2\}$.

Online:

1. Locally compute the following (S_0 during final verification):
   - S_1 : $\vec{\mathsf{m}}^0_{\mathbf{Z}} = \vec{\mathsf{m}}_{\mathbf{X}} \cdot \vec{\lambda}^{\mathbf{0}}_{\mathbf{Y}} + \vec{\lambda}^{\mathbf{0}}_{\mathbf{X}} \cdot \vec{\mathsf{m}}_{\mathbf{Y}} + \vec{\gamma}^{\mathbf{0}}_{\mathbf{XY}} - \vec{\lambda}^{\mathbf{0}}_{\mathbf{Z}}$.
   - S_1 : $\vec{\mathsf{m}}^1_{\mathbf{Z}} = \vec{\mathsf{m}}_{\mathbf{X}} \cdot \vec{\lambda}^{\mathbf{1}}_{\mathbf{Y}} + \vec{\lambda}^{\mathbf{1}}_{\mathbf{X}} \cdot \vec{\mathsf{m}}_{\mathbf{Y}} + \vec{\gamma}^{\mathbf{1}}_{\mathbf{XY}} - \vec{\lambda}^{\mathbf{1}}_{\mathbf{Z}}$.
   - $\mathsf{S}_2, \mathsf{S}_0$: $\vec{\mathsf{m}}^2_{\mathbf{Z}} = \vec{\mathsf{m}}_{\mathbf{X}} \cdot \vec{\lambda}^{\mathbf{2}}_{\mathbf{Y}} + \vec{\lambda}^{\mathbf{2}}_{\mathbf{X}} \cdot \vec{\mathsf{m}}_{\mathbf{Y}} + \vec{\gamma}^{\mathbf{2}}_{\mathbf{XY}} - \vec{\lambda}^{\mathbf{2}}_{\mathbf{Z}} + \vec{\mathsf{m}}_{\mathbf{X}} \cdot \vec{\mathsf{m}}_{\mathbf{Y}}$.
2. S_1 sends $\vec{\mathsf{m}}^0_{\mathbf{Z}} + \vec{\mathsf{m}}^1_{\mathbf{Z}}$ to S_2, while S_2 sends $\vec{\mathsf{m}}^2_{\mathbf{Z}}$ to S_1.
3. $\mathsf{S}_1, \mathsf{S}_2$ reconstruct $\vec{\mathsf{m}}_{\mathbf{Z}} = \vec{\mathsf{m}}^0_{\mathbf{Z}} + \vec{\mathsf{m}}^1_{\mathbf{Z}} + \vec{\mathsf{m}}^2_{\mathbf{Z}}$.

Final Verification (batched over all multiplications):

1. S_1 sends $\vec{\mathsf{m}}^0_{\mathbf{Z}} + \vec{\mathsf{m}}^1_{\mathbf{Z}}$ to S_0.
2. S_0 computes $\vec{\mathsf{m}}^2_{\mathbf{Z}}$ and reconstructs $\vec{\mathsf{m}}_{\mathbf{Z}} = \vec{\mathsf{m}}^0_{\mathbf{Z}} + \vec{\mathsf{m}}^1_{\mathbf{Z}} + \vec{\mathsf{m}}^2_{\mathbf{Z}}$.
3. S_0 sends $\mathsf{H}(\vec{\mathsf{m}}^2_{\mathbf{Z}})$ to S_1 who checks consistency to the $\vec{\mathsf{m}}^2_{\mathbf{Z}}$ it received from S_2.

Figure 5: Matrix multiplication in SOCIUM.

For the simplification above, we replace $\mathcal{F}_{\mathsf{MultPre}}$ in SWIFT with a modified functionality $\mathcal{F}'_{\mathsf{MultPre}}$. This functionality computes $\langle \vec{\gamma}_{\mathbf{XY}} \rangle'$, which is an *incomplete replicated sharing* lacking $\vec{\gamma}^{\mathbf{1}}_{\mathbf{XY}}$ possessed by S_2, unlike $\langle \vec{\gamma}_{\mathbf{XY}} \rangle$. Accordingly, we let S_1 to send $\vec{\mathsf{m}}^0_{\mathbf{Z}} + \vec{\mathsf{m}}^1_{\mathbf{Z}}$ to S_0 during the verification in the online phase as well. The resulting multiplication protocol in the $\mathcal{F}'_{\mathsf{MultPre}}$-hybrid model is given in Fig. 5. Similar to SWIFT, it needs $3uv\ell$ bits of online communication, and one round per multiplication (amortized), but reduces hash-based verification by $3\times$.

Incorporating truncation. In contrast to SWIFT, SOCIUM allows for *free* probabilistic fixed-point truncation [61], similar to AUXILIATOR. To see this, note that the multiplication in SOCIUM allows the servers S_1 and $(\mathsf{S}_2, \mathsf{S}_0)$ to locally compute a 2-out-of-2 additive sharing of $\vec{\mathbf{Z}}$. Specifically,

$$\begin{aligned} \mathsf{S}_1 &: \vec{\mathbf{Z}}_1 = (\vec{\mathsf{m}}^0_{\mathbf{Z}} + \vec{\mathsf{m}}^1_{\mathbf{Z}}) + (\vec{\lambda}^{\mathbf{0}}_{\mathbf{Z}} + \vec{\lambda}^{\mathbf{1}}_{\mathbf{Z}}) \\ \mathsf{S}_2, \mathsf{S}_0 &: \vec{\mathbf{Z}}_2 = \vec{\mathsf{m}}^2_{\mathbf{Z}} + \vec{\lambda}^{\mathbf{2}}_{\mathbf{Z}}. \end{aligned}$$

Then, for $\vec{\mathbf{U}} = \vec{\mathbf{Z}}/2^d$ and $\vec{\lambda}^{\mathbf{j}}_{\mathbf{U}} = \vec{\lambda}^{\mathbf{j}}_{\mathbf{Z}}$ for $j \in \{0,1,2\}$, servers locally set the shares as follows:

$$\begin{aligned} \mathsf{S}_1 &: \vec{\mathsf{m}}^0_{\mathbf{U}} + \vec{\mathsf{m}}^1_{\mathbf{U}} = (\vec{\mathbf{Z}}_1/2^d) - (\vec{\lambda}^{\mathbf{0}}_{\mathbf{U}} + \vec{\lambda}^{\mathbf{1}}_{\mathbf{U}}) \\ \mathsf{S}_2, \mathsf{S}_0 &: \vec{\mathsf{m}}^2_{\mathbf{U}} = (\vec{\mathbf{Z}}_2/2^d) - \vec{\lambda}^{\mathbf{2}}_{\mathbf{U}}. \end{aligned}$$

The remaining steps follow a similar protocol as the multiplication in Fig. 5, but now using $\vec{\mathbf{U}}$ instead of $\vec{\mathbf{Z}}$.

5.2.1. Instantiating $\mathcal{F}'_{\mathsf{MultPre}}$ in SOCIUM. To instantiate $\mathcal{F}_{\mathsf{MultPre}}$, SWIFT employs $\Pi^{\mathsf{semi}}_{\mathsf{MultPre}}$, a semi-honest multiplication protocol [4], [26], followed by verification using DZKPs [15]. The protocol can further be extended to handle matrix multiplication of $\vec{\mathbf{C}} = \vec{\mathbf{A}} \cdot \vec{\mathbf{B}}$, as shown in Proc. 5.

Procedure 5 $\Pi^{\mathsf{semi}}_{\mathsf{MultPre}}$: Semi-honest matrix multiplication [4], [26]

1: Locally sample $[\vec{\alpha}]$ with $\vec{\alpha} = \vec{\mathbf{0}} \in \mathbb{Z}_{2^\ell}^{u \times v}$ as in [4].

2: Locally compute $[\vec{\mathbf{C}}]$ for $\vec{\mathbf{C}} = \vec{\mathbf{A}} \cdot \vec{\mathbf{B}}$:

$$\begin{aligned} \mathsf{S}_0 &: \vec{\mathbf{C}}^{\mathbf{0}} = \vec{\mathbf{A}}^{\mathbf{0}} \cdot \vec{\mathbf{B}}^{\mathbf{0}} + \vec{\mathbf{A}}^{\mathbf{0}} \cdot \vec{\mathbf{B}}^{\mathbf{2}} + \vec{\mathbf{A}}^{\mathbf{2}} \cdot \vec{\mathbf{B}}^{\mathbf{0}} + \vec{\alpha}^{\mathbf{0}} \\ \mathsf{S}_1 &: \vec{\mathbf{C}}^{\mathbf{1}} = \vec{\mathbf{A}}^{\mathbf{1}} \cdot \vec{\mathbf{B}}^{\mathbf{1}} + \vec{\mathbf{A}}^{\mathbf{1}} \cdot \vec{\mathbf{B}}^{\mathbf{0}} + \vec{\mathbf{A}}^{\mathbf{0}} \cdot \vec{\mathbf{B}}^{\mathbf{1}} + \vec{\alpha}^{\mathbf{1}} \\ \mathsf{S}_2 &: \vec{\mathbf{C}}^{\mathbf{2}} = \vec{\mathbf{A}}^{\mathbf{2}} \cdot \vec{\mathbf{B}}^{\mathbf{2}} + \vec{\mathbf{A}}^{\mathbf{2}} \cdot \vec{\mathbf{B}}^{\mathbf{1}} + \vec{\mathbf{A}}^{\mathbf{1}} \cdot \vec{\mathbf{B}}^{\mathbf{2}} + \vec{\alpha}^{\mathbf{2}} \end{aligned}$$

3: $[\vec{\mathbf{C}}]$ to $\langle \vec{\mathbf{C}} \rangle$: S_i sends $\vec{\mathbf{C}}^{\mathbf{i}}$ to S_{i+1} for $0 \le i < 3$.

While computing $\vec{\gamma}_{\mathbf{XY}} = \vec{\lambda}_{\mathbf{X}} \cdot \vec{\lambda}_{\mathbf{Y}}$ using $\Pi^{\mathsf{semi}}_{\mathsf{MultPre}}$, note that SOCIUM only requires $\langle \vec{\gamma}_{\mathbf{XY}} \rangle'$ to be computed instead of $\langle \vec{\gamma}_{\mathbf{XY}} \rangle$. Thus, step 3 in Proc. 5 can be simplified: S_0 sends $\vec{\gamma}^{\mathbf{0}}_{\mathbf{XY}}$ to S_1, and S_2 sends $\vec{\gamma}^{\mathbf{1}}_{\mathbf{XY}}$ to S_0. We use $\Pi^{\mathsf{semi}'}_{\mathsf{MultPre}}$ to denote this simplified version that computes $\langle \vec{\gamma}_{\mathbf{XY}} \rangle'$. Note that $\Pi^{\mathsf{semi}'}_{\mathsf{MultPre}}$ is *maliciously private*, meaning that a malicious adversary can only compromise correctness by introducing an additive error s.t. $\vec{\gamma}_{\mathbf{XY}} = \vec{\lambda}_{\mathbf{X}} \cdot \vec{\lambda}_{\mathbf{Y}} + \vec{\delta}$, but not privacy, like $\Pi^{\mathsf{semi}}_{\mathsf{MultPre}}$ [57]. Next, we'll review how the verification techniques from §3, along with $\Pi^{\mathsf{semi}'}_{\mathsf{MultPre}}$, are applied in SOCIUM to securely instantiate $\mathcal{F}'_{\mathsf{MultPre}}$. We prove security of SOCIUM in Appendix A.4.

Triple Sacrificing and Cut-and-Choose: This is done similar to AUXILIATOR, where the triple to be verified $(\langle \vec{\lambda}_{\mathbf{X}} \rangle, \langle \vec{\lambda}_{\mathbf{Y}} \rangle, \langle \vec{\gamma}_{\mathbf{XY}} \rangle')$ is mapped to $(\langle \vec{\mathbf{A}} \rangle, \langle \vec{\mathbf{B}} \rangle, \langle \vec{\mathbf{C}} \rangle')$ over $\mathbb{Z}_{2^{\ell+\sigma}}$ and the triple $(\langle \vec{\hat{\mathbf{A}}} \rangle, \langle \vec{\mathbf{B}} \rangle, \langle \vec{\hat{\mathbf{C}}} \rangle')$ is generated using $\Pi^{\mathsf{semi}'}_{\mathsf{MultPre}}$. $\mathsf{S}_0, \mathsf{S}_1$ can convert these triples to an additive sharing among them locally and perform verification using the sacrificing approach in §3.1.2.

Since sacrificed triples immediately become additive sharings between $\mathsf{S}_0, \mathsf{S}_1$, $\Pi^{\mathsf{semi}'}_{\mathsf{MultPre}}$ doesn't require S_0 to send $\vec{\hat{\mathbf{C}}}^0$ to S_1. S_2 sending $\vec{\hat{\mathbf{C}}}^2$ is enough for S_0 to participate in verification using share $\vec{\hat{\mathbf{C}}}^0 + \vec{\hat{\mathbf{C}}}^2$ while S_1 uses $\vec{\hat{\mathbf{C}}}^1$.

Communication. The amortized cost of instantiating $\mathcal{F}'_{\mathsf{MultPre}}$ in SOCIUM with triple sacrificing for matrices in $\mathbb{Z}_{2^\ell}^{u \times w}, \mathbb{Z}_{2^\ell}^{w \times v}$ is $3uv + 2uw$ elements of $\mathbb{Z}_{2^{\ell+\sigma}}$ ($2 \times \Pi^{\mathsf{semi}'}_{\mathsf{MultPre}}$ + verification §3.1.2). Similarly, using cut-and-choose for AND incurs a communication of 12 bits (cf. §A.2.2). Table 2 in §6.1 provides an overview of the communication costs.

Distributed Zero-Knowledge Proofs: Similar to AUXILIATOR, checking the correctness of $\Pi^{\mathsf{semi}'}_{\mathsf{MultPre}}$ using DZKPs involves verifying if the value $\vec{\gamma}^{\mathbf{2}}_{\mathbf{XY}}$ sent by S_2 to S_0 is correct (§3.2). For this, given a row $\vec{\mathbf{a}}$ of $\vec{\lambda}_{\mathbf{X}}$ and a column $\vec{\mathbf{b}}$ of $\vec{\lambda}_{\mathbf{Y}}$, we apply the DZKP proof for dot-product from SWIFT [51], [75] that verifies $\vec{\mathbf{a}} \cdot \vec{\mathbf{b}} = p$ as follows: $c\left(\left\{\vec{\mathbf{a}}^1_k, \vec{\mathbf{a}}^2_k, \vec{\mathbf{b}}^1_k, \vec{\mathbf{b}}^2_k\right\}_{k=1}^{w}, \alpha^2, p^2\right) := \sum_{k=1}^{w}\left(\vec{\mathbf{a}}^2_k\vec{\mathbf{b}}^2_k + \vec{\mathbf{a}}^2_k\vec{\mathbf{b}}^1_k + \vec{\mathbf{a}}^1_k\vec{\mathbf{b}}^2_k + \vec{\mathbf{a}}^2_k\vec{\mathbf{b}}^2_k\right) + \alpha^2 - p^2 = 0.$

Communication. The amortized cost of instantiating $\mathcal{F}'_{\mathsf{MultPre}}$ with DZKPs for matrices in $\mathbb{Z}_{2^\ell}^{u \times w}, \mathbb{Z}_{2^\ell}^{w \times v}$ is $2uv$ elements of $\mathbb{Z}_{2^\ell}$ for running $\Pi^{\mathsf{semi}'}_{\mathsf{MultPre}}$, while the proof cost amortizes to zero (see Appendix A.3). Furthermore, the amortized cost of $\mathcal{F}'_{\mathsf{MultPre}}$ for an AND operation is 2 bits.

6. Evaluation

In §6.1, we evaluate AUXILIATOR (§4) and SOCIUM (§5) against semi-honest ASTRA [24] and malicious SWIFT [51], providing a comprehensive comparison among these closely related 3PC protocols. Additionally, we compare AUXILIATOR to the related 2PC semi-honest protocol ABY2.0 [65] in §6.2 to investigate the benefits of using an untrusted but non-colluding helper. Additional benchmarks to investigate the scalability of our protocols' application in machine learning are provided in §A.5. Moreover, we provide a comparison between SOCIUM and SIMC [23] for the *client-malicious* setting in §6.3, and §A.6 compares AUXILIATOR to ELSA [69], a federated learning [59] protocol with asymmetric trust.

Choice of Parameters: In our evaluation, we use ring size $\ell = 64$ and computational security parameter $\kappa = 128$. For triple sacrificing (§3.1), we set the statistical security parameter $\sigma = 64$, running on $\mathbb{Z}_{2^{128}}$ for optimal hardware support. For cut-and-choose and DZKPs, we use $\sigma = 40$ and a batch size of 2^{20}. We also employ the logarithmic round variant of DZKPs (Appendix A.3), with communication of 0.24 bits per multiplication or 0.35 bits per dimension for

a dot product over 1024-length vectors. Note that matrix multiplication naturally batches proofs for dot products, resulting in a smaller actual batch size. Throughout the section, "sacrificing" refers to using sacrificing in the arithmetic and cut-and-choose in the binary domain for brevity.

Implementation and Benchmarking Environment: Besides a theoretical communication analysis, we implement ASTRA, AUXILIATOR, SOCIUM, and SWIFT in the MPC framework MOTION [16] to benchmark runtimes.[11] Due to the complexity of implementing four protocols, we use only triple sacrificing (cf. §3.1) as the verification method, including in the binary domain.[12] We run our benchmarks on three servers, each equipped with a 16-core Intel Core i9-7960X CPU at 2.8GHz and 128GB of RAM at 2666MHz. Furthermore, we consider a LAN setting with 10Gbit/s bandwidth and 1ms round-trip time (RTT), and a WAN setting with 100Mbit/s bandwidth and 100ms RTT.

Evaluation for ML Inference: Parts of our evaluation consider ML inference as a possible application of our protocols. For that, we consider a 2-layer convolutional neural network (CNN) trained on the MNIST dataset and a 7-layer CNN trained on the CIFAR-10 dataset, both CNNs being from MiniONN [58]. In line with MUSE [55] and SIMC [23], we replace max pooling for MNIST by average pooling. Hence, we require gates for (matrix-)multiplication and fixed-point truncation for linear layers as well as rectified linear units (ReLUs) for the non-linear layers. The required gates for ReLUs are provided by ASTRA and SWIFT and a translation to AUXILIATOR and SOCIUM is straightforward. For completeness regarding the multitude of required truncation and ReLU realizations for all protocols, we refer to Appendix B. We implement the different linear and non-linear layers separately and aggregate their individual performances for our runtime evaluations.

6.1. Spectrum: Semi-Honest to Malicious 3PC

In Table 2, we provide the amortized cost for multiplication, AND gates, and matrix multiplication in ASTRA [24], AUXILIATOR(§4), SOCIUM(§5), and SWIFT [51]. Notably, AUXILIATOR is based on ASTRA, and SOCIUM is based on SWIFT, resulting in same online communication for ASTRA and AUXILIATOR, as well as for SOCIUM and SWIFT.

Per-Gate Performance: The setup communication overhead for one multiplication using sacrificing over semi-honest ASTRA is as follows: It is a factor of 4 for AUXILIATOR, 5 for SOCIUM, and 9 for SWIFT, in addition to the overhead of working on an extended ring $\mathbb{Z}_{2^{\ell+\sigma}}$, which adds another $2\times$ for our settings. In the case of matrix multiplication, the overhead factor is split between a uv part representing the output dimensions, and a uw part for the dimensions of the first input matrix. For dot products, i.e., $u = v = 1$ and $w > 0$, the overhead for sacrificing becomes more evident as the communication becomes linear in the vector dimension w, whereas it's independent in ASTRA. A similar trend in communication is observed for the binary domain as well, where the overhead gradually increases from AUXILIATOR over SOCIUM to SWIFT.

When using DZKP for verification, the amortized setup cost for AUXILIATOR matches that of ASTRA, i.e., protection against a malicious helper comes for free regarding communication. SOCIUM has an overhead of $2\times$ over both ASTRA and AUXILIATOR, but outperforms SWIFT by 33%. Also, AUXILIATOR requires $3\times$ fewer DZKPs than SWIFT.

For both verification approaches, we observe a spectrum between fully semi-honest and malicious 3PC. The protocol complexity increases for a *stronger* malicious adversary: a fixed malicious evaluator as in SOCIUM requires more expensive protection than a malicious helper as in AUXILIATOR, but the highest cost comes with a malicious adversary able to corrupt any party.

Performance for ML Inference: The communication for ML inference using CNNs in MiniONN [58] is given in Fig. 6. In the online phase, SOCIUM requires $1.68-1.69\times$ more communication than AUXILIATOR due to S_0's involvement in the final verification phase. The online communication of ASTRA and AUXILIATOR or SWIFT and SOCIUM, respectively, are similar, with small deviations stemming from slightly different truncation and ReLU protocols, affecting how computation is distributed between setup and online phases.

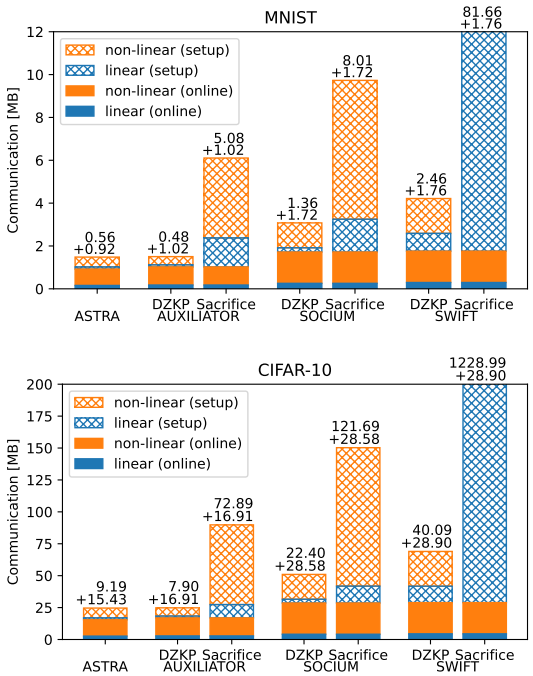

Figure 6: Total ML inference communication for ASTRA [24], AUXILIATOR (§4), SOCIUM (§5), and SWIFT [51] using triple sacrificing (§3.1) with cut-and-choose for binary domain (§A.2) and distributed zero-knowledge proofs (§3.2) otherwise. The costs are split between linear and non-linear layers. Numbers reported on top of bars state setup + online communication.

11. https://encrypto.de/code/MOTION-FD
12. We use $\sigma = 63$ in sacrificing for optimal hardware support on $\mathbb{Z}_{2^{64}}$.

TABLE 2: Comparison of amortized setup and online communication for ASTRA [24], AUXILIATOR (§4), SOCIUM (§5), and SWIFT [51] in ring elements/bits. Dimensions for matrix multiplication are $u \times w, w \times v$. Setup communication is given (except for semi-honest ASTRA) for verification using triple sacrificing §3.1/cut-and-choose §A.2 and distributed zero-knowledge proofs (DZKP) §3.2.

Operation	Multiplication		AND		Matrix Multiplication	
Verification method	Sacrifice	DZKP	Cut-and-Choose	DZKP	Sacrifice	DZKP
Setup phase						
ASTRA [24]	$1 \times \mathbb{Z}_{2^\ell}$		1 bits		$uv \times \mathbb{Z}_{2^\ell}$	
AUXILIATOR (§4)	$4 \times \mathbb{Z}_{2^{\ell+\sigma}}$	$1 \times \mathbb{Z}_{2^\ell}$	11 bits	1 bits	$(2uv + 2uw) \times \mathbb{Z}_{2^{\ell+\sigma}}$	$uv \times \mathbb{Z}_{2^\ell}$
SOCIUM (§5)	$5 \times \mathbb{Z}_{2^{\ell+\sigma}}$	$2 \times \mathbb{Z}_{2^\ell}$	12 bits	2 bits	$(3uv + 2uw) \times \mathbb{Z}_{2^{\ell+\sigma}}$	$2uv \times \mathbb{Z}_{2^\ell}$
SWIFT [51]	$9 \times \mathbb{Z}_{2^{\ell+\sigma}}$	$3 \times \mathbb{Z}_{2^\ell}$	21 bits	3 bits	$(6uv + 3uw) \times \mathbb{Z}_{2^{\ell+\sigma}}$	$3uv \times \mathbb{Z}_{2^\ell}$
Online phase (including final verification if applicable)						
ASTRA [24]/AUXILIATOR (§4)	$2 \times \mathbb{Z}_{2^\ell}$		2 bits		$2uv \times \mathbb{Z}_{2^\ell}$	
SWIFT [51]/SOCIUM (§5)	$3 \times \mathbb{Z}_{2^\ell}$		3 bits		$3uv \times \mathbb{Z}_{2^\ell}$	

The total communication of AUXILIATOR with DZKP has an overhead of only 0.8-1.4% compared to ASTRA. On the other side, the overhead when using sacrificing is 3.6-4.1× while providing cheaper local computation. When using DZKP, the total communication of SOCIUM is 2.05× worse than AUXILIATOR and 1.35-1.37× better than SWIFT. For sacrificing, it is 1.59-1.67× worse than AUXILIATOR and 8.37-8.57× better than SWIFT. Sacrificing instead of DZKP increases the setup by 9.22-10.54× for AUXILIATOR, 5.43-5.90× for SOCIUM, and 30.65-33.23× for SWIFT. We observe that the substantial overhead of SWIFT with sacrificing is mainly due to the linear layers, which have an overhead of over 80× compared to the DZKP variant. This is because sacrificing is inefficient for dot products in SWIFT, that are necessary for truncation.

In general, the main bottleneck is the non-linear layers, accounting for 74.9-88.8% of the overall communication. However, in the sacrificing variant of SWIFT, the linear layers cause over 83% of the overall communication due to aforementioned issues regarding truncation. The protocols require 28-35 online rounds for MNIST and 64-75 rounds for CIFAR-10, mainly due to ReLUs implemented with depth logarithmic in ℓ.

We provide measured runtimes of our implementations in Fig. 7 for both LAN and WAN network settings. It's important to note that we use triple sacrificing not only for arithmetic but also for binary computation. As a result, the non-linear layers significantly bottleneck the protocol execution, more than expected for an optimized implementation using cut-and-choose (§A.2). Online runtimes are generally similar between all protocols, with slight deviations attributed to differences in truncations, ReLUs, compiler-side optimizations, and scheduling. However, it is worth noting that both ASTRA and AUXILIATOR involve only two servers during the online phase, while the rest require the helper's involvement for the verification part.

Efficient batching reduces the previously observed significant communication overhead of SWIFT truncation. For LAN, the setup of AUXILIATOR is 2.50-8.99× slower than ASTRA, while SOCIUM is 1.51-1.73× faster than SWIFT. For WAN, these factors are 3.67-8.68× and 1.61-1.62×, respectively.

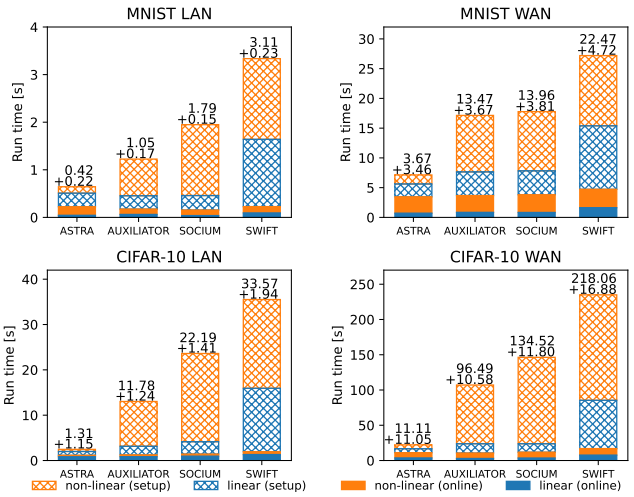

Figure 7: Total ML inference runtimes for our implementations of ASTRA [24], AUXILIATOR (§4), SOCIUM (§5), and SWIFT [51] using triple sacrificing (§3.1). The costs are split between linear and non-linear layers. Numbers reported on top of the bars correspond to setup + online runtimes.

6.2. Why to Keep a Malicious Helper: AUXILIATOR vs ABY2.0

The setting of AUXILIATOR where a helper turns malicious has a straightforward solution: exclude the helper and run a semi-honest 2PC protocol between both evaluators. ABY2.0 [65], the state-of-the-art for this setting, follows precisely this approach. It uses similar secret-sharing semantics as ASTRA [24] and AUXILIATOR, and has a nearly equal online phase while emulating the helper in 2PC, for example, based on oblivious transfer (OT). In this context, we investigate the benefits of using a malicious helper in AUXILIATOR compared to the baseline of ABY2.0.

ABY2.0 uses an OT-based setup using the correlated OT from IKNP-style OT extension [5], [44]. An alternate option for ABY2.0 is to instantiate the OT using the SilentOT approach [14]. SilentOT, similar to DZKP compared to sacrificing for AUXILIATOR, results in less communication

but more computation. In the binary domain, SilentOT was shown in [17] to decrease the setup cost per AND gate from 134 bits[13] to 4 bits, plus the cost of 2 random OTs, which can be decreased below 1 bit. For simplicity, we assume zero communication per random OT. In the arithmetic domain, it is possible to replace the correlated OT in ABY2.0 with a random OT, converted to a correlated OT using the conversion from [16], reducing the setup communication for 64-bit multiplication from 24 576 bits to 8 320 bits.[14]

TABLE 3: Setup communication (in MB) for ML inference in ABY2.0 [65] when using either IKNP-style setup or SilentOT [14] as well as for AUXILIATOR using sacrificing or DKZPs (§4).

Network	Layers	ABY2.0 [65]		AUXILIATOR (§4)	
		IKNP-style	SilentOT	Sacrifice	DZKP
MNIST	Linear	1952.95	661.17	1.35	0.10
	Non-linear	31.32	1.38	3.73	0.38
	Total	1984.27	662.54	5.08	0.48
CIFAR-10	Linear	178974.16	60590.31	10.46	1.48
	Non-linear	524.25	23.04	62.43	6.42
	Total	179498.40	60613.36	72.89	7.90

In Table 3, we compare ML inference setups between ABY2.0 with either IKNP-based OT extension or SilentOT and AUXILIATOR based on either sacrificing or DZKP. It is evident that even sacrifice-based AUXILIATOR significantly outperforms both variants of ABY2.0 by at least two orders of magnitude, confirming the substantial benefits of adding a non-colluding but malicious helper. The improvement is particularly prominent in the linear CNN layers, where sacrifice-based AUXILIATOR outperforms IKNP-style ABY2.0 by 1 442-17 112×, and SilentOT-based ABY2.0 by 488-5 793×, with DZKP-based AUXILIATOR providing roughly another order of magnitude of improvement.

However, this advantage decreases significantly for non-linear layers: AUXILIATOR using sacrifice outperforms IKNP-style ABY2.0 by only 8.40×, and AUXILIATOR using DZKP outperforms SilentOT-based ABY2.0 by 3.59×. Interestingly, the computationally expensive SilentOT-based ABY2.0 outperforms the computationally cheap sacrifice-based AUXILIATOR by 2.71×. Depending on the efficiency of SilentOT and DZKP implementations, it may be an option to use a sacrifice-based malicious helper for linear layers and switch to SilentOT-based ABY2.0 for non-linear layers, as both protocols' secret-sharing semantics are compatible.

The advantage of ABY2.0 with SilentOT for non-linear layers is due to the low cost of one AND gate, which is roughly 4 bits. In contrast, the cut-and-choose based AUXILIATOR requires 11 bits of communication. On the other hand, the significant advantage of AUXILIATOR over ABY2.0 for linear layers can be attributed to two factors. First, the cost of a multiplication is much higher for ABY2.0. While a multiplication triple comes nearly for free in the binary domain using SilentOT, it does not in the arithmetic domain. Second, AUXILIATOR based on sacrifice has a setup cost of $(2uv + 2uw)$ elements of $\mathbb{Z}_{2^{\ell+\sigma}}$ for multiplying matrices of dimensions $u \times w, w \times v$, which is $u \cdot (v+w)/2\times$ the cost of a scalar multiplication. In contrast, ABY2.0 uses naive matrix multiplication in its setup due to the lack of a helper, resulting in a matrix multiplication setup cost of $uvw\times$ the cost of a scalar multiplication. E.g., for the second convolution layer of the CNN for CIFAR-10, these factors are approximately $\approx$ 51k for AUXILIATOR, but $\approx$ 37 749k for ABY2.0.

13. This includes an optimization for binary domain that ABY2.0 uses and which is based on [35].

14. It is possible to generate a multiplication triple using random OT, as described in [47], followed by Beaver's multiplication [6]. However, this approach is slightly more expensive.

6.3. Semi-Honest Helper in the Client-Malicious Setting: SOCIUM vs SIMC

It is predictable that the client-malicious setting, i.e., the 2PC setting with a malicious client and semi-honest server, as introduced by MUSE [55] only allows for less efficient protocols than an honest majority 3PC setting. However, we are interested in investigating the impact of adding a semi-honest non-colluding helper to the setting and comparing it to the state-of-the-art protocol for ML inference in the client-malicious 2PC setting, SIMC [23]. Hence, we perform a comparison between SIMC and our more generic MPC protocol SOCIUM (§5).

TABLE 4: Total communication (in MB) for ML inference in SIMC [23] and SOCIUM with either sacrificing or DKZPs (§5).

Network	Layers	SIMC [23]	SOCIUM (§5)	
			Sacrifice	DZKP
MNIST	Linear	50	1.95	0.45
	Non-linear	133	8.86	2.62
	Total	183	10.81	3.08
CIFAR-10	Linear	140	20.01	7.08
	Non-linear	2240	148.33	43.90
	Total	2380	168.34	50.98

Table 4 provides a comparison regarding the total communication for ML inference tasks. As expected, SOCIUM clearly outperforms SIMC by a factor 14.14-16.93× when using sacrificing and 46.69-59.47× when using DZKPs. For non-linear layers in both CNNs, SOCIUM performs better by $\approx 15\times$ with sacrificing and $\approx 51\times$ with DZKPs. Regarding linear layers, SOCIUM performs better by 6.99-25.69× with sacrificing and 19.78-110.18× with DZKPs.

TABLE 5: Total runtime (in s) for ML inference in SIMC [23] and sacrifice-based SOCIUM (§5) with non-optimized nonlinear layers.

Network	Layers	LAN		WAN	
		SIMC	SOCIUM	SIMC	SOCIUM
MNIST	Linear	4.28	0.34	13.60	4.85
	Non-linear	0.22	1.61	14.17	12.92
	Total	4.50	1.95	27.77	17.77
CIFAR-10	Linear	22.96	3.70	45.39	15.53
	Non-linear	2.41	19.91	191.65	130.80
	Total	25.36	23.61	237.03	146.32

Furthermore, the runtime comparison to SIMC in our network settings, presented in Table 5, demonstrates that SOCIUM outperforms SIMC by 1.07-2.31× in a LAN and 1.56-1.62× in a WAN. It is essential to note that our implementation's non-linear layers are not well optimized, as we use triple sacrificing for binary computation as well. However, the optimized linear layers of SOCIUM outperform SIMC by 6.21-12.51× for LAN and 2.80-2.92× for WAN. The lower improvement in WAN can be explained by SIMC using homomorphic encryption for these layers.

TABLE 6: Online ML inference communication (in MB) for SIMC++ [23] and sacrifice or DZKP based SOCIUM (§5).

Network	SIMC++ [23]	SOCIUM (§5)
MNIST	10	1.72
CIFAR-10	230	28.58

Recall that one of the design goals for SOCIUM was a streamlined online phase. While SIMC doesn't have a setup phase, the authors proposed a variant called SIMC++ [23] that sacrifices total communication for a better online phase. However, the total communication of SIMC++ is worse than SIMC by a factor of 1.61-1.77×, which extends the advantage of using SOCIUM. In the comparison of online communications provided in Table 6, we observe that SOCIUM outperforms SIMC++ by a factor of 5.82-8.05×. Unfortunately, since there is only a publicly available implementation of SIMC and not SIMC++, we are unable to provide a comparison of online runtime in our network settings.

7. Conclusion and Future Work

We introduced two robust protocols, AUXILIATOR and SOCIUM, designed to bridge the gap between fully semi-honest and fully-malicious honest-majority settings in three-party computation. These protocols are well-suited for real-world privacy-preserving machine learning scenarios, accommodating varying levels of trust among participating parties and offering flexible computation-communication trade-offs. Specifically, we have showcased how a malicious helper can enhance the performance of semi-honest two-party computation (2PC) and how a semi-honest helper can assist 2PC with a malicious party.

For future research, we aim to explore scenarios where two out of three servers may exhibit malicious behavior, expanding beyond the consideration of a single party in this work. Additionally, our implementation serves as a foundation for integrating various verification techniques, such as cut-and-choose [3], [40], or distributed zero-knowledge [12], [15], into our code framework. We also anticipate extending our framework with additional primitives for both machine learning and other applications. Furthermore, we anticipate that this setting can significantly enhance more recent notions of robustness, such as Friends-and-Foes security [2], [50], leaving this as a promising direction for future work.

Acknowledgment

This project received funding from the ERC under the European Union's Horizon 2020 research and innovation program (grant agreement No. 850990 PSOTI). It was co-funded by the DFG within SFB 1119 CROSSING/236615297 and GRK 2050 Privacy & Trust/251805230.

References

[1] M. Abspoel, A. P. K. Dalskov, D. Escudero, and A. Nof, "An Efficient Passive-to-Active Compiler for Honest-Majority MPC over Rings," in *ACNS*, 2021.

[2] B. Alon, E. Omri, and A. Paskin-Cherniavsky, "MPC with Friends and Foes," in *CRYPTO*, 2020.

[3] T. Araki, A. Barak, J. Furukawa, T. Lichter, Y. Lindell, A. Nof, K. Ohara, A. Watzman, and O. Weinstein, "Optimized Honest-Majority MPC for Malicious Adversaries - Breaking the 1 Billion-Gate Per Second Barrier," in *IEEE S&P*, 2017.

[4] T. Araki, J. Furukawa, Y. Lindell, A. Nof, and K. Ohara, "High-Throughput Semi-Honest Secure Three-Party Computation with an Honest Majority," in *CCS*, 2016.

[5] G. Asharov, Y. Lindell, T. Schneider, and M. Zohner, "More Efficient Oblivious Transfer and Extensions for Faster Secure Computation," in *CCS*, 2013.

[6] D. Beaver, "Efficient Multiparty Protocols Using Circuit Randomization," in *CRYPTO*, 1992.

[7] D. Beaver, S. Micali, and P. Rogaway, "The Round Complexity of Secure Protocols (Extended Abstract)," in *STOC*, 1990.

[8] A. Ben-Efraim, M. Nielsen, and E. Omri, "Turbospeedz: Double Your Online SPDZ! Improving SPDZ Using Function Dependent Preprocessing," in *ACNS*, 2019.

[9] E. Ben-Sasson, S. Fehr, and R. Ostrovsky, "Near-Linear Unconditionally-Secure Multiparty Computation with a Dishonest Minority," in *CRYPTO*, 2012.

[10] F. Boemer, R. Cammarota, D. Demmler, T. Schneider, and H. Yalame, "MP2ML: A Mixed-Protocol Machine Learning Framework for Private Inference," in *ARES*, 2020.

[11] K. Bonawitz, V. Ivanov, B. Kreuter, A. Marcedone, H. B. McMahan, S. Patel, D. Ramage, A. Segal, and K. Seth, "Practical Secure Aggregation for Privacy-Preserving Machine Learning," in *CCS*, 2017.

[12] D. Boneh, E. Boyle, H. Corrigan-Gibbs, N. Gilboa, and Y. Ishai, "Zero-Knowledge Proofs on Secret-Shared Data via Fully Linear PCPs," in *CRYPTO*, 2019.

[13] E. Boyle, N. Chandran, N. Gilboa, D. Gupta, Y. Ishai, N. Kumar, and M. Rathee, "Function Secret Sharing for Mixed-Mode and Fixed-Point Secure Computation," in *EUROCRYPT*, 2021.

[14] E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, L. Kohl, P. Rindal, and P. Scholl, "Efficient Two-Round OT Extension and Silent Non-Interactive Secure Computation," in *CCS*, 2019.

[15] E. Boyle, N. Gilboa, Y. Ishai, and A. Nof, "Practical Fully Secure Three-Party Computation via Sublinear Distributed Zero-Knowledge Proofs," in *CCS*, 2019.

[16] L. Braun, D. Demmler, T. Schneider, and O. Tkachenko, "MOTION - A Framework for Mixed-Protocol Multi-Party Computation," *ACM Trans. Priv. Secur.*, 2022.

[17] A. Brüggemann, R. Hundt, T. Schneider, A. Suresh, and H. Yalame, "FLUTE: Fast and Secure Lookup Table Evaluations," in *IEEE S&P*, 2023.

[18] A. Brüggemann, O. Schick, T. Schneider, A. Suresh, and H. Yalame, "Don't Eject the Impostor: Fast Three-Party Computation With a Known Cheater," in *IEEE S&P*, 2024.

[19] N. Büscher, D. Demmler, S. Katzenbeisser, D. Kretzmer, and T. Schneider, "HyCC: Compilation of Hybrid Protocols for Practical Secure Computation," in *CCS*, 2018.

[20] M. Byali, H. Chaudhari, A. Patra, and A. Suresh, "FLASH: Fast and Robust Framework for Privacy-preserving Machine Learning," *PETS*, 2020.

[21] O. Catrina and S. de Hoogh, "Improved Primitives for Secure Multiparty Integer Computation," in *SCN*, 2010.

[22] O. Catrina and A. Saxena, "Secure Computation with Fixed-Point Numbers," in *FC*, 2010.

[23] N. Chandran, D. Gupta, S. L. B. Obbattu, and A. Shah, "SIMC: ML Inference Secure Against Malicious Clients at Semi-Honest Cost," in *USENIX Security*, 2022.

[24] H. Chaudhari, A. Choudhury, A. Patra, and A. Suresh, "ASTRA: High Throughput 3PC over Rings with Application to Secure Prediction," in *CCSW@CCS*, 2019.

[25] H. Chaudhari, R. Rachuri, and A. Suresh, "Trident: Efficient 4PC Framework for Privacy Preserving Machine Learning," in *NDSS*, 2020.

[26] K. Chida, D. Genkin, K. Hamada, D. Ikarashi, R. Kikuchi, Y. Lindell, and A. Nof, "Fast Large-Scale Honest-Majority MPC for Malicious Adversaries," in *CRYPTO*, 2018.

[27] A. Choudhury and A. Patra, *Secure Multi-Party Computation Against Passive Adversaries*. Springer Nature, 2022.

[28] G. Couteau, P. Rindal, and S. Raghuraman, "Silver: Silent VOLE and Oblivious Transfer from Hardness of Decoding Structured LDPC Codes," in *CRYPTO*, 2021.

[29] R. Cramer, I. Damgård, D. Escudero, P. Scholl, and C. Xing, "SPD $\mathbb{Z}_{2^k}$: Efficient MPC mod 2^k for Dishonest Majority," in *CRYPTO*, 2018.

[30] A. P. K. Dalskov, D. Escudero, and M. Keller, "Fantastic Four: Honest-Majority Four-Party Secure Computation With Malicious Security," in *USENIX Security*, 2021.

[31] I. Damgård and C. Orlandi, "Multiparty Computation for Dishonest Majority: From Passive to Active Security at Low Cost," in *CRYPTO*, 2010.

[32] I. Damgård, C. Orlandi, and M. Simkin, "Yet Another Compiler for Active Security or: Efficient MPC Over Arbitrary Rings," in *CRYPTO*, 2018.

[33] E. Dauterman, M. Rathee, R. A. Popa, and I. Stoica, "Waldo: A Private Time-Series Database from Function Secret Sharing," in *IEEE S&P*, 2022.

[34] D. Demmler, T. Schneider, and M. Zohner, "ABY - A Framework for Efficient Mixed-Protocol Secure Two-Party Computation," in *NDSS*, 2015.

[35] G. Dessouky, F. Koushanfar, A.-R. Sadeghi, T. Schneider, S. Zeitouni, and M. Zohner, "Pushing the Communication Barrier in Secure Computation using Lookup Tables," in *NDSS*, 2017.

[36] C. Dong, J. Weng, J.-N. Liu, Y. Zhang, Y. Tong, A. Yang, Y. Cheng, and S. Hu, "Fusion: Efficient and Secure Inference Resilient to Malicious Servers," in *NDSS*, 2023.

[37] H. Eerikson, M. Keller, C. Orlandi, P. Pullonen, J. Puura, and M. Simkin, "Use Your Brain! Arithmetic 3PC for Any Modulus with Active Security," in *ITC*, 2020.

[38] D. Escudero, M. Jagielski, R. Rachuri, and P. Scholl, "Adversarial Attacks and Countermeasures on Private Training in MPC," *PPML@NeurIPS*, 2021.

[39] D. Evans, V. Kolesnikov, and M. Rosulek, "A Pragmatic Introduction to Secure Multi-Party Computation," *Found. Trends Priv. Secur.*, 2018.

[40] J. Furukawa, Y. Lindell, A. Nof, and O. Weinstein, "High-Throughput Secure Three-Party Computation for Malicious Adversaries and an Honest Majority," in *EUROCRYPT*, 2017.

[41] D. Genkin, Y. Ishai, M. Prabhakaran, A. Sahai, and E. Tromer, "Circuits Resilient to Additive Attacks With Applications to Secure Computation," in *STOC*, 2014.

[42] O. Goldreich, S. Micali, and A. Wigderson, "How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority," in *STOC*, 1987.

[43] S. D. Gordon, S. Ranellucci, and X. Wang, "Secure Computation with Low Communication from Cross-Checking," in *ASIACRYPT*, 2018.

[44] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank, "Extending Oblivious Transfers Efficiently," in *CRYPTO*, 2003.

[45] S. Kamara, P. Mohassel, and M. Raykova, "Outsourcing Multi-Party Computation," *IACR Cryptol. ePrint Arch.*, 2011. [Online]. Available: http://eprint.iacr.org/2011/272

[46] B. Karmakar, N. Koti, A. Patra, S. Patranabis, P. Paul, and D. Ravi, "Asterisk: Super-fast MPC with a Friend," in *IEEE S&P*, 2024.

[47] M. Keller, E. Orsini, and P. Scholl, "MASCOT: Faster Malicious Arithmetic Secure Computation with Oblivious Transfer," in *CCS*, 2016.

[48] B. Knott, S. Venkataraman, A. Y. Hannun, S. Sengupta, M. Ibrahim, and L. van der Maaten, "CrypTen: Secure Multi-Party Computation Meets Machine Learning," in *NeurIPS*, 2021.

[49] V. Kolesnikov and T. Schneider, "Improved Garbled Circuit: Free XOR Gates and Applications," in *ICALP*, 2008.

[50] N. Koti, V. B. Kukkala, A. Patra, and B. Raj Gopal, "PentaGOD: Stepping beyond Traditional GOD with Five Parties," in *CCS*, 2022.

[51] N. Koti, M. Pancholi, A. Patra, and A. Suresh, "SWIFT: Super-fast and Robust Privacy-Preserving Machine Learning," in *USENIX Security*, 2021.

[52] N. Koti, S. M. Patil, A. Patra, and A. Suresh, "MPClan: Protocol Suite for Privacy-Conscious Computations," *J. Cryptol.*, 2023.

[53] N. Koti, A. Patra, R. Rachuri, and A. Suresh, "Tetrad: Actively Secure 4PC for Secure Training and Inference," in *NDSS*, 2022.

[54] H. Kvamme, N. Sellereite, K. Aas, and S. Sjursen, "Predicting mortgage default using convolutional neural networks," *Expert Systems with Applications*, vol. 102, 2018.

[55] R. Lehmkuhl, P. Mishra, A. Srinivasan, and R. A. Popa, "Muse: Secure Inference Resilient to Malicious Clients," in *USENIX Security*, 2021.

[56] Y. Lindell, "Secure Multiparty Computation," in *Commun. ACM*, 2020.

[57] Y. Lindell and A. Nof, "A Framework for Constructing Fast MPC over Arithmetic Circuits with Malicious Adversaries and an Honest-Majority," in *CCS*, 2017.

[58] J. Liu, M. Juuti, Y. Lu, and N. Asokan, "Oblivious Neural Network Predictions via MiniONN Transformations," in *CCS*, 2017.

[59] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, "Communication-Efficient Learning of Deep Networks from Decentralized Data," in *Artificial Intelligence and Statistics*, 2017.

[60] P. Mohassel and P. Rindal, "ABY^3: A Mixed Protocol Framework for Machine Learning," in *CCS*, 2018.

[61] P. Mohassel and Y. Zhang, "SecureML: A System for Scalable Privacy-Preserving Machine Learning," in *IEEE S&P*, 2017.

[62] V.-E. Neagoe, A.-D. Ciotec, and G.-S. Cucu, "Deep Convolutional Neural Networks Versus Multilayer Perceptron for Financial Prediction," in *International Conference on Communications*, 2018.

[63] L. K. Ng and S. S. Chow, "SoK: Cryptographic Neural-Network Computation," in *IEEE S&P*, 2023.

[64] P. S. Nordholt and M. Veeningen, "Minimising Communication in Honest-Majority MPC by Batchwise Multiplication Verification," in *ACNS*, 2018.

[65] A. Patra, T. Schneider, A. Suresh, and H. Yalame, "ABY2.0: Improved Mixed-Protocol Secure Two-Party Computation," in *USENIX Security*, 2021.

[66] A. Patra and A. Suresh, "BLAZE: Blazing Fast Privacy-Preserving Machine Learning," in *NDSS*, 2020.

[67] H. Qian, P. Ma, S. Gao, and Y. Song, "Soft Reordering One-Dimensional Convolutional Neural Network for Credit Scoring," *Knowledge-Based Systems*, vol. 266, 2023.

[68] D. Rathee, M. Rathee, N. Kumar, N. Chandran, D. Gupta, A. Rastogi, and R. Sharma, "CrypTFlow2: Practical 2-Party Secure Inference," in *CCS*, 2020.

[69] M. Rathee, C. Shen, S. Wagh, and R. A. Popa, "ELSA: Secure Aggregation for Federated Learning with Malicious Actors," in *IEEE S&P*, 2023.

[70] M. Rosulek and L. Roy, "Three Halves Make a Whole? Beating the Half-Gates Lower Bound for Garbled Circuits," in *CRYPTO*, 2021.

[71] D. Rotaru and T. Wood, "MArBled Circuits: Mixing Arithmetic and Boolean Circuits with Active Security," in *INDOCRYPT*, 2019.

[72] J. T. Schwartz, "Fast Probabilistic Algorithms for Verification of Polynomial Identities," *J. ACM*, vol. 27, no. 4, oct 1980.

[73] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," in *ICLR*, 2015.

[74] L. Song, J. Wang, Z. Wang, X. Tu, G. Lin, W. Ruan, H. Wu, and W. Han, "pMPL: A Robust Multi-Party Learning Framework with a Privileged Party," in *CCS*, 2022.

[75] A. Suresh, "MPCLeague: Robust MPC Platform for Privacy-Preserving Machine Learning," Ph.D. dissertation, Indian Institute of Science (IISc), Bangalore, 2021, https://arxiv.org/abs/2112.13338.

[76] G. Xu, X. Han, T. Zhang, H. Li, and R. H. Deng, "SIMC 2.0: Improved Secure ML Inference Against Malicious Clients," abs/2112.13338, 2022, https://arxiv.org/abs/2207.04637.

[77] Y. Yang, X. Huang, X. Liu, H. Cheng, J. Weng, X. Luo, and V. Chang, "A Comprehensive Survey on Secure Outsourced Computation and Its Applications," *IEEE Access*, 2019.

[78] A. C.-C. Yao, "Protocols for Secure Computations (Extended Abstract)," in *FOCS*, 1982.

[79] ——, "How to Generate and Exchange Secrets (Extended Abstract)," in *FOCS*, 1986.

[80] B. Zhu, W. Yang, H. Wang, and Y. Yuan, "A Hybrid Deep Learning Model for Consumer Credit Scoring," in *International Conference on Artificial Intelligence and Big Data*, 2018.

[81] R. Zippel, "Probabilistic Algorithms for Sparse Polynomials," in *Symbolic and Algebraic Computation*, 1979.

Appendix A. Additional Details

The appendix contains details regarding the pre-shared key setup used by our protocols (§A.1), verification using cut-and-choose (§A.2) and distributed zero-knowledge proofs (§A.3), the security of our protocols (§A.4), further investigation of our protocols' scalability for ML inference tasks (§A.5), and a comparison to the protocol ELSA [69] for secure federated learning (§A.6).

A.1. Pre-Shared Key Setup

Our protocols rely on a pre-shared pseudo-random function (PRF) key setup used for the non-interactive generation of shared randomness [4], [60]. Hence, we work in the $\mathcal{F}_{\mathsf{KeySetup}}$-hybrid model with $\mathcal{F}_{\mathsf{KeySetup}}$ being formally defined in Fig. 8.

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

$\mathcal{F}_{\mathsf{KeySetup}}$ samples random keys $k_{01}, k_{02}, k_{12}, k_{012} \in \{0,1\}^\kappa$ and sends $(k_{01}, k_{02}, k_{012})$ to S_0, $(k_{01}, k_{12}, k_{012})$ to S_1, and $(k_{02}, k_{12}, k_{012})$ to S_2.

Figure 8: Shared key setup functionality.

Note that this enables any subset of servers to non-interactively sample a pseudo-random number. $\mathsf{S}_i, \mathsf{S}_{i+1}$ for $0 \leq i < 3$ sampling random $\beta_i \in \mathbb{Z}_{2^\ell}$ and S_i then defining $\alpha^i = \beta_i - \beta_{i-1}$ also enables setting up a random additive sharing $[\alpha]$ of $\alpha = 0$ [4].

A.2. Cut-and-Choose for $\mathbb{Z}_2$

As a basis for computationally lightweight verification for binary domain computation, we use the cut-and-choose approach of [3], [40]. The original method uses multiple binary multiplication triples to verify the correctness of the evaluation of each AND gate in a setting where any of three parties may be malicious. It requires the generation of random permutations not known to a malicious adversary until after the triple generation, a verification protocol $\Pi_{\mathsf{VerifyByOpen}}$ that checks one triple's correctness by revealing its values, and a verification protocol $\Pi_{\mathsf{VerifyByOther}}$ that checks one triple consuming another triple and only accepts if either no or both triples are correct.

The verification uses buckets of B potentially incorrect triples each, where one bucket is used to verify one AND. Verifying one AND consists of an amortized overhead of only generating B triples for the corresponding bucket and running B instances of $\Pi_{\mathsf{VerifyByOther}}$. The number of instances of $\Pi_{\mathsf{VerifyByOpen}}$ per AND amortizes to 0. Now, [3] shows that security is attained for $\sigma = 40$ when verifying $N = 2^{20}$ AND gates for bucket size $B = 2$. Note that all verification is executed in the offline phase so that batching of $N = 2^{20}$ AND gates is feasible.

A.2.1. Cut-and-Choose in the Fully Malicious Setting. The required random permutations can be chosen interactively so that the communication per AND amortizes to 0 [40]. Also, protocol $\Pi_{\mathsf{VerifyByOpen}}$ can easily be implemented by revealing a triple to two servers who then verify its correctness in plain. Thus, an incorrect triple is detected by at least one non-cheating server. Checking if a triple $(\langle x\rangle, \langle y\rangle, \langle z\rangle) \in \mathbb{Z}_2^3$ is correct with a second triple $(\langle a\rangle, \langle b\rangle, \langle c\rangle) \in \mathbb{Z}_2^3$ works as depicted in Proc. 6. The amortized cost of this check is that of reconstructing two bits, i.e., 6 bits.

A.2.2. Cut-and-Choose in the Fixed Corruption Setting. In the fixed corruption setting, random permutations can be

Procedure 6 $\Pi_{\mathsf{VerifyByOther}}$: Verify triple using second triple [40]

1: Locally compute $\langle p \rangle = \langle x \rangle \oplus \langle a \rangle, \langle q \rangle = \langle y \rangle \oplus \langle b \rangle$.
2: Reconstruct $p, q \in \mathbb{Z}_2$.
3: Locally compute $\langle v \rangle = \langle z \rangle \oplus \langle c \rangle \oplus p\langle b \rangle \oplus q\langle a \rangle \oplus pq$.
4: Verify that $v = 0$, otherwise `abort`. (Use batched hash based check as in §3.1.1.)

chosen non-interactively by the semi-honest servers that we w.l.o.g. let be $\mathsf{S}_0, \mathsf{S}_1$. Protocol $\Pi_{\mathsf{VerifyByOpen}}$ only requires to reveal a triple to either S_0 or S_1, cutting the communication in half compared to the fully malicious setting.

As for triple sacrificing (cf. §3.1.2), $\Pi_{\mathsf{VerifyByOther}}$ can be optimized by converting from $\langle\cdot\rangle$-sharings to additive $[\cdot]$-sharings between $\mathsf{S}_0, \mathsf{S}_1$ only and excluding S_2 from the remaining steps. Thus, the reconstruction of two bits among the semi-honest servers only uses 4 bits of communication. The final zero check can be implemented using the simplified hash based check from §3.1.2.

A.3. Distributed Zero-Knowledge Proofs

Using distributed zero-knowledge proofs (DZKPs) [12], [15], one can verify the correctness of multiplication at zero amortized communication overhead but higher local computation than the approaches in §3.1, Appendix A.2.

A.3.1. DZKPs in the Fully Malicious Setting. DZKPs enable to add verification to the RSS based maliciously private multiplications from [4] where each server sends one element in $\mathbb{Z}_{2^\ell}$ by proving those messages' correctness. For the proof, it suffices to show that a degree-2 polynomial with 6 variables evaluates to 0 where each variable is known to the sender as well as being known to one of the other servers or being additively shared among them.

[15] demonstrates that such proof for a batch of m multiplications can be implemented using $(8\sqrt{m}+3)\cdot\delta\ell$ bits of communication for $\delta \in \mathbb{N}$ chosen such that $\frac{2^{(\ell-1)\delta}\cdot 2\sqrt{m}+1}{2^{\ell\delta}-\sqrt{m}} \leq 2^{-\sigma}$ in 2 rounds. This is satisfied for $\delta \gtrsim \sigma + \log_2(2\sqrt{m})$. A second option requires less local computation and has lower communication of $(1 + 4\log_2 m) \cdot \delta\ell$ bits while requiring $\mathcal{O}(\log_2 m)$ rounds. Here, δ is chosen such that $\frac{5\log_2 m+1}{2^\delta-2} \leq 2^{-\sigma}$ which is satisfied for $\delta \gtrsim \sigma + 3 + \log_2\log_2 m$. Batch size m can be set such that through amortization, the communication per multiplication verification is roughly 0 bits.

Both approaches work on extension rings of degree δ or the field $\mathbb{F}_{2^\delta}$ for binary domain computation and require polynomial interpolation which yields high computational overhead. In contrast to [15], SWIFT [51], [75] moves this expensive computation to the offline phase also allowing the use of higher m and hence better amortization by batching the setup of multiple protocol runs.

SWIFT [51], [75] also introduces DZKPs for dot products and hence also matrix multiplication. The proof communication for m dot products of dimension d is $(2\sqrt{2m(4\mathsf{d}+2)}+3)\cdot\delta\ell$ bits in 2 rounds for $\delta \in \mathbb{N}$ chosen such that $\frac{2^{(\ell-1)\delta}\cdot\sqrt{2m(4\mathsf{d}+2)}+1}{2^{\ell\delta}-\sqrt{m(4\mathsf{d}+2)/2}} \leq 2^{-\sigma}$ which is satisfied for $\delta \gtrsim \sigma + \log_2\sqrt{2m(4\mathsf{d}+2)}$. Here, degree-2 polynomials in $4\mathsf{d}+2$ variables are used. Again, this can be used to decrease the verification cost per dot product to roughly 0 bits. Applying the logarithmic round construction of [15] yields a second option with $(1+4\log_2(m\mathsf{d}))\cdot\delta\ell$ bits communication in $\mathcal{O}(\log_2(m\mathsf{d}))$ rounds. Now, δ is chosen such that $\frac{5\log_2(m\mathsf{d})+1}{2^\delta-2} \leq 2^{-\sigma}$ which is satisfied for $\delta \gtrsim \sigma + 3 + \log_2\log_2(m\mathsf{d})$.

A.3.2. DZKPs in the Fixed Corruption Setting. As seen in §4.2.1 and §5.2.1, distributed zero-knowledge proofs can be translated to the fixed corruption setting given that the used secret-sharing scheme still is somewhat replicated. While the amortized communication of one proof already is 0 in the fully malicious setting, the fixed corruption setting only requires the single malicious party to prove that its messages are correct. That improves the verification overhead for both communication and computation by factor $3\times$.

A.4. Security

In this section, we discuss the security of our protocols. We focus on the security of the interactive matrix multiplication protocols, as the security of input and output phases trivially follow the considerations in [24], [51]. Our proofs work in the $\mathcal{F}_{\mathsf{KeySetup}}$-hybrid (cf. Appendix A.1).

A.4.1. AUXILIATOR. Recall that the only difference between the matrix multiplication of AUXILIATOR (§4) and ASTRA [24] lies in the instantiation of $\mathcal{F}_{\mathsf{MultPre}}$ in the setup phase. Hence, the security of Π_{Mult} in the $\mathcal{F}_{\mathsf{MultPre}}$-hybrid model immediately follows the security of ASTRA. We now prove that our instantiation of $\mathcal{F}_{\mathsf{MultPre}}$ using triple sacrificing (cf. §4.2.1) is secure. For brevity, we omit proofs for instantiations using cut-and-choose (cf. §4.2.1) or distributed zero-knowledge proofs (cf. §4.2.1) that can be similarly derived from the respective proofs for the fully malicious setting in [3], [15], [40].

Functionality $\mathcal{F}_{\mathsf{MultPre}}([\vec{\mathbf{A}}], [\vec{\mathbf{B}}])$

- $\mathcal{F}_{\mathsf{MultPre}}$ receives $(\vec{\mathbf{A}}^{\mathbf{1}}, \vec{\mathbf{A}}^{\mathbf{2}}, \vec{\mathbf{B}}^{\mathbf{1}}, \vec{\mathbf{B}}^{\mathbf{2}}), (\vec{\mathbf{A}}^{\mathbf{1}}, \vec{\mathbf{B}}^{\mathbf{1}}), (\vec{\mathbf{A}}^{\mathbf{2}}, \vec{\mathbf{B}}^{\mathbf{2}})$ from $\mathsf{S}_0, \mathsf{S}_1, \mathsf{S}_2$ where $\vec{\mathbf{A}}^{\mathbf{i}} \in \mathbb{Z}_{2^\ell}^{u\times w}, \vec{\mathbf{B}}^{\mathbf{i}} \in \mathbb{Z}_{2^\ell}^{w\times v}$ for $i \in \{1,2\}$. If these values are inconsistent, use those by $\mathsf{S}_1, \mathsf{S}_2$. Reconstruct $\vec{\mathbf{A}} = \vec{\mathbf{A}}^{\mathbf{1}} + \vec{\mathbf{A}}^{\mathbf{2}}, \vec{\mathbf{B}} = \vec{\mathbf{B}}^{\mathbf{1}} + \vec{\mathbf{B}}^{\mathbf{2}}$.
- If $\mathcal{A}$ controls S_0, $\mathcal{F}_{\mathsf{MultPre}}$ receives additional values $\vec{\delta}^{\mathbf{1}} \in \mathbb{Z}_{2^\sigma}^{u\times v}, \vec{\mathbf{\Delta}}^{\mathbf{2}} \in \mathbb{Z}_{2^{\ell+\sigma}}^{u\times v}$ from $\mathcal{A}$. Define $\vec{\mathbf{\Delta}}^{\mathbf{1}} = 2^\ell\vec{\delta}^{\mathbf{1}} \in \mathbb{Z}_{2^{\ell+\sigma}}^{u\times v}$.
- If $\mathcal{A}$ controls S_0 or S_1, $\mathcal{F}_{\mathsf{MultPre}}$ also receives $\vec{\mathbf{C}}^{\mathbf{1}} \in \mathbb{Z}_{2^\ell}^{u\times v}$ from $\mathcal{A}$. Otherwise, let $\vec{\mathbf{C}}^{\mathbf{1}} \in \mathbb{Z}_{2^\ell}^{u\times v}$ be sampled uniformly at random.
- If $\mathcal{A}$ controls S_2, $\mathcal{F}_{\mathsf{MultPre}}$ also receives $k_{02}, k_{12} \in \{0,1\}^\kappa$ from $\mathcal{A}$.
- Define $\vec{\mathbf{C}}^{\mathbf{2}} = (\vec{\mathbf{A}} \cdot \vec{\mathbf{B}}) - \vec{\mathbf{C}}^{\mathbf{1}}$ so that $\vec{\mathbf{C}}^{\mathbf{1}} + \vec{\mathbf{C}}^{\mathbf{2}} = \vec{\mathbf{A}} \cdot \vec{\mathbf{B}}$.
- If $\mathcal{A}$ controls S_0 and sends `abort` or $\vec{\mathbf{\Delta}}^{\mathbf{2}} - r\vec{\mathbf{\Delta}}^{\mathbf{1}} \not\equiv_{2^{\ell+\sigma}} \vec{\mathbf{0}}$ for randomly sampled $r \in \mathbb{Z}_{2^\sigma}$, then $\mathcal{F}_{\mathsf{MultPre}}$ sends `abort` to all servers. Otherwise, send $\vec{\mathbf{C}}^{\mathbf{1}}$ to S_1 and $\vec{\mathbf{C}}^{\mathbf{2}}$ to S_2.

Figure 9: Secure offline matrix multiplication functionality.

We provide a formal definition of $\mathcal{F}_{\mathsf{MultPre}}$ in Fig. 9. Observe that due to the use of shared keys for sampling random values, the functionality has additional inputs by the adversary $\mathcal{A}$ which is similar to, e.g., [4]. In particular, an adversary $\mathcal{A}$ that corrupts S_1 already knows output $\vec{\mathbf{C}}^{\mathbf{1}}$ prior to the instantiation of $\mathcal{F}_{\mathsf{MultPre}}$ and hence inputs it in our functionality. Likewise, an adversary $\mathcal{A}$ corrupting S_2 directly inputs the shared keys known to S_2 that are used to sample multiple values within the protocol.

Simulator $\mathcal{S}^{\text{sacrifice}}_{\mathsf{MultPre}}$

The simulator plays the role of $\mathsf{S}_0, \mathsf{S}_1$ and simulates each step of the sacrifice based instantiation of $\mathcal{F}_{\mathsf{MultPre}}$ to S_2 as follows:

1. Receive inputs $\vec{\mathbf{A}}^{\mathbf{2}} \in \mathbb{Z}_{2^\ell}^{u\times w}, \vec{\mathbf{B}}^{\mathbf{2}} \in \mathbb{Z}_{2^\ell}^{w\times v}$ as well as $k_{02}, k_{12} \in \{0,1\}^\kappa$ from $\mathcal{A}$.
2. Send $\vec{\mathbf{A}}^{\mathbf{2}}, \vec{\mathbf{B}}^{\mathbf{2}}, k_{02}, k_{12}$ to $\mathcal{F}_{\mathsf{MultPre}}$ and receive $\vec{\mathbf{C}}^{\mathbf{2}} \in \mathbb{Z}_{2^\ell}^{u\times v}$.
3. Sample random $\vec{\mathbf{C}'}^{\mathbf{2}} \in \mathbb{Z}_{2^{\ell+\sigma}}^{u\times v}$ under the constraint that $\vec{\mathbf{C}'}^{\mathbf{2}} \equiv_{2^\ell} \vec{\mathbf{C}}^{\mathbf{2}}$.
4. Send $\vec{\mathbf{C}'}^{\mathbf{2}}$ to $\mathcal{A}$ on behalf of S_0.
5. Sample random $\vec{\hat{\mathbf{C}}}^2 \in \mathbb{Z}_{2^{\ell+\sigma}}^{u\times v}$ and send it to $\mathcal{A}$ on behalf of S_0.
6. Sample random $\vec{\mathbf{V}}^{\mathbf{1}} \in \mathbb{Z}_{2^{\ell+\sigma}}^{u\times w}$ and send it to $\mathcal{A}$ on behalf of S_1.
7. Sample random $r \in \mathbb{Z}_{2^\sigma}$ using k_{12} and $\vec{\hat{\mathbf{A}}}^2 \in \mathbb{Z}_{2^{\ell+\sigma}}^{u\times w}$ using k_{02}.
8. Compute $\vec{\mathbf{V}} = r\vec{\mathbf{A}'}^{\mathbf{2}} - \vec{\hat{\mathbf{A}}}^2 + \vec{\mathbf{V}}^{\mathbf{1}}$ where $\vec{\mathbf{A}'}^{\mathbf{2}} \in \mathbb{Z}_{2^{\ell+\sigma}}^{u\times w}$ is $\vec{\mathbf{A}}^{\mathbf{2}}$ with each entry lifted to $\mathbb{Z}_{2^{\ell+\sigma}}$.
9. Compute $\vec{\mathbf{W}}^{\mathbf{2}} = \vec{\mathbf{V}} \cdot \vec{\mathbf{B}'}^{\mathbf{2}} - r\vec{\mathbf{C}'}2 + \vec{\hat{\mathbf{C}}}^2$ where $\vec{\mathbf{B}'}^{\mathbf{2}} \in \mathbb{Z}_{2^{\ell+\sigma}}^{w\times v}$ is $\vec{\mathbf{B}}^{\mathbf{2}}$ with each entry lifted to $\mathbb{Z}_{2^{\ell+\sigma}}$.
10. Compute and send $h_1 = \mathsf{H}(-\mathbf{W}^{\mathbf{2}}{}_{1,1}||...|| - \mathbf{W}^{\mathbf{2}}{}_{1,v}||...|| - \mathbf{W}^{\mathbf{2}}{}_{u,v})$ to $\mathcal{A}$ on behalf of S_1.

Figure 10: Simulation of secure offline matrix multiplication protocol using the sacrifice method for the case of a corrupt S_2.

If $\mathcal{A}$ corrupts S_0, it does not only provide $\vec{\mathbf{C}}^{\mathbf{1}}$ to the functionality, but also inputs additional values $\vec{\delta}^{\mathbf{1}} \in \mathbb{Z}_{2^\sigma}^{u\times v}, \vec{\mathbf{\Delta}}^{\mathbf{2}} \in \mathbb{Z}_{2^{\ell+\sigma}}^{u\times v}$. Then, with $\vec{\mathbf{\Delta}}^{\mathbf{1}} = 2^\ell\vec{\delta}^{\mathbf{1}} \in \mathbb{Z}_{2^{\ell+\sigma}}^{u\times v}$, the functionality outputs `abort` if $\mathcal{A}$ sends `abort` or also if $\vec{\mathbf{\Delta}}^{\mathbf{2}} - r\vec{\mathbf{\Delta}}^{\mathbf{1}} \not\equiv_{2^{\ell+\sigma}} \vec{\mathbf{0}}$ for randomly sampled r. This technicality stems from a gap in the completeness and soundness of the triple sacrifice check from Lemma 3.1. Note that it leaves open the case where S_0 cheats in a way where output $\vec{\mathbf{C}}$ still is correct. Indeed, that occurs if $\vec{\mathbf{\Delta}}^{\mathbf{1}}$ from the proof is chosen such that $\vec{\mathbf{\Delta}}^{\mathbf{1}} \equiv_{2^\ell} \vec{\mathbf{0}}$, but $\vec{\mathbf{\Delta}}^{\mathbf{1}} \not\equiv_{2^{\ell+\sigma}} \vec{\mathbf{0}}$ or $\vec{\mathbf{\Delta}}^{\mathbf{2}} \not\equiv_{2^{\ell+\sigma}} \vec{\mathbf{0}}$. In this case, the probability of aborting may not be negligible as, e.g., for an entry $2^{\ell+\sigma-1}$ of $\vec{\mathbf{\Delta}}^{\mathbf{1}}$, $r \cdot 2^{\ell+\sigma-1} \equiv_{2^{\ell+\sigma}} 0$ for all even r. This may seem irrelevant as the output still is correct, but we have to ensure that no selective failure attack is possible where an `abort` message is sent depending on data that must stay secret. Clearly, the `abort` here depends on the errors introduced by $\mathcal{A}$ and randomly sampled r only. Hence, $\mathcal{A}$ may only gain information about value r, but this is chosen randomly and never used again after the check that may result in an `abort` message. Formally, we let functionality $\mathcal{F}_{\mathsf{MultPre}}$ exactly define this behaviour using a random value r. We argue that this gives an adversary controlling S_0 no meaningful additional power as it now may input $\vec{\delta}^{\mathbf{1}}, \vec{\mathbf{\Delta}}^{\mathbf{2}} \in \mathbb{Z}_{2^{\ell+\sigma}}$ that possibly (but independent of any secret data such as inputs of honest parties) lead to an `abort`, but anyway may also send `abort` by itself with exactly the same probability.

Lemma A.1. *The sacrifice based instantiation of $\mathcal{F}_{\mathsf{MultPre}}$ (cf. §4.2.1) securely implements $\mathcal{F}_{\mathsf{MultPre}}$ functionality in the fixed corruption setting, where the malicious adversary $\mathcal{A}$ is allowed to corrupt S_0.*

Proof. Correctness immediately follows from previous considerations including those in Lemma 3.1. Simulation for corrupted S_0 is easy by simply calling the ideal functionality $\mathcal{F}_{\mathsf{MultPre}}$ and forwarding whether it outputs `abort` to $\mathcal{A}$. Note that besides this potential `abort`, S_0 receives no messages. Next, let us first consider semi-honestly corrupted S_2 for which the simulator is given in Fig. 10. S_2 receives shares of $\vec{\mathbf{C}}', \vec{\hat{\mathbf{C}}}$ from S_0 setting up two triples. Here, $\vec{\mathbf{C}}' \equiv_{2^\ell} \vec{\mathbf{C}}$ but is in the larger domain $\mathbb{Z}_{2^{\ell+\sigma}}$. Then, it receives S_1's share of $\vec{\mathbf{V}}$ and a hash h (cf. §3.1.2).

- Message $\vec{\mathbf{C}'}^{\mathbf{2}}$ clearly consists of S_2's output $\vec{\mathbf{C}}^{\mathbf{2}}$ in the ℓ least significant bits while the σ most significant bits appear uniformly random to S_2 due to random selection of $\vec{\mathbf{C}'}^{\mathbf{1}}$ by $\mathsf{S}_0, \mathsf{S}_1$.
- Message $\vec{\hat{\mathbf{C}}}^2$ appears uniformly random to S_2 due to random selection of $\vec{\hat{\mathbf{C}}}^1$ by $\mathsf{S}_0, \mathsf{S}_1$.
- Message $\vec{\mathbf{V}}^{\mathbf{1}}$ appears uniformly random to S_2 due to random selection of its component $\vec{\hat{\mathbf{A}}}^1$ by $\mathsf{S}_0, \mathsf{S}_1$.
- Message h is equal to the hash computed by S_2 due to the completeness of the verification (cf. Lemma 3.1).

As the used randomness is uncorrelated, it is easy to see that the simulation in Fig. 10 cannot be distinguished from a real protocol run. For corrupt S_1, simply observe that receiving $\vec{\mathbf{V}}^{\mathbf{2}}$ is symmetrical to the simulation of S_2 receiving $\vec{\mathbf{V}}^{\mathbf{1}}$ in Fig. 10 and that it receives no further messages. □

A.4.2. SOCIUM. For SOCIUM (§5), we omit the proof for secure instantiation of $\mathcal{F}'_{\mathsf{MultPre}}$ as security up to an additive error for $\Pi^{\mathsf{semi}}_{\mathsf{MultPre}}$ directly follows from [57] while the remaining construction follows symmetric arguments as in §A.4.1. The central difference is that S_2 does not receive message $\vec{\mathbf{C}}^{\mathbf{2}}$ in contrast to SWIFT [51] which clearly does not violate our security.

While the online phase of SOCIUM also differs from that of SWIFT [51], it is easy to see that the differences do not infringe the protocol's security. First, removing consistency checks for messages sent by $\mathsf{S}_0, \mathsf{S}_1$ clearly is valid as SOCIUM assumes these servers to be semi-honest. Second, S_1 sending $\vec{\mathsf{m}}^0_{\mathbf{Z}} + \vec{\mathsf{m}}^1_{\mathbf{Z}}$ in the online phase instead of $\vec{\mathsf{m}}^0_{\mathbf{Z}}$ as in SWIFT obviously is secure as in our setting, S_1 is semi-honest and in SWIFT, S_2 knows $\vec{\mathsf{m}}^1_{\mathbf{Z}}$ beforehand, i.e., gains no more information than in SOCIUM. Likewise, sending this message to S_0 instead of only $\vec{\mathsf{m}}^1_{\mathbf{Z}}$ is secure as S_0 can compute $\vec{\mathsf{m}}^0_{\mathbf{Z}}$ on its own.

A.5. Scalability for ML Inference

To show the scalability of our protocols for ML inference, we compare the performance on the MiniONN [58] 7-layer CNN for CIFAR-10 (with average pooling) to the performance on the larger VGG16 (Net-D) 13-layer CNN [73] on the same dataset. Compared to the CNN in MiniONN, VGG16 is a much larger network with 16 instead of 8 linear layers and 21 instead of 9 non-linear layers leading to a total of ≈ 38 million parameters. As for MiniONN, we replace max pooling required by VGG16 by average pooling here.

The communication when using VGG16 [73] for ML inference on the CIFAR-10 dataset is given in Fig. 11. Compared to the MiniONN [58] CNN for CIFAR-10, using more than twice the number of layers in VGG16 yields a $1.65\times$ increase in offline and online communication across all protocols. The only exception is the setup for linear layers when using sacrificing where the overhead is at most $1.90\times$. This can be attributed to differently sized matrices in VGG16 compared to MiniONN. Here, we use the optimization of calculating $\vec{\mathbf{C}}^\intercal = \vec{\mathbf{B}}^\intercal \cdot \vec{\mathbf{A}}^\intercal$ instead of $\vec{\mathbf{C}} = \vec{\mathbf{A}} \cdot \vec{\mathbf{B}}$ (cf. §3.1) to avoid an overhead of up to $110.80\times$ in the linear setup that would result from the unoptimized version given the specific matrix dimensions in VGG16.

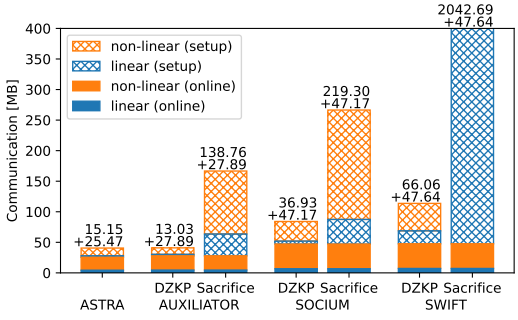

Figure 11: Total ML inference communication on VGG16 [73] for ASTRA [24], AUXILIATOR (§4), SOCIUM (§5), and SWIFT [51] using triple sacrificing (§3.1). The costs are split between linear and non-linear layers. Numbers reported on top of bars state setup + online communication.

As shown in Fig. 12, the runtimes of our implementation that uses sacrificing (except for ASTRA) only increase by 2.58-3.68× for LAN and 2.15-3.03× for WAN, when compared to the MiniONN [58] CNN. The only outlier is ASTRA in the LAN setting with a factor of $5.61\times$.

A.6. Comparison to ELSA for Federated Learning

ELSA [69] is a recent protocol designed for federated learning [59], where it addresses asymmetric trust by assuming clients to be maliciously corrupt. It splits the role of the aggregator between two semi-honest servers[15] and

15. ELSA has extended their protocol to achieve malicious privacy during aggregation. However, this inclusion has introduced additional overhead in their semi-honest variant. To ensure fairness in the comparison, we decided to evaluate AUXILIATOR against their semi-honest variant.

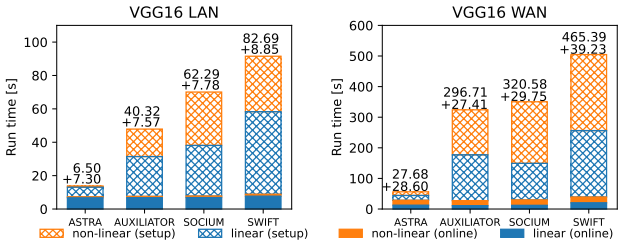

Figure 12: Total ML inference run times on VGG16 [73] for our implementations of ASTRA [24], AUXILIATOR (§4), SOCIUM (§5), and SWIFT [51] using triple sacrificing (§3.1). The costs are split between linear and non-linear layers. Numbers reported on top of the bars correspond to setup + online run times.

incorporates defenses against boosted gradients attacks by malicious clients using ℓ_2 norm and ℓ_∞ norm. ELSA's main innovation lies in having untrusted clients provide correlations to the server, acting as untrusted helpers to enhance efficiency.

Interestingly, AUXILIATOR can model ELSA by allowing malicious clients to take on the role of the malicious helper, and each client provides correlations used only on its own inputs, similar to ELSA. Additionally, in scenarios where bandwidth-constrained clients are present, a dedicated server can act as the helper on their behalf. This adaptation enables AUXILIATOR to cater to use cases similar to those addressed by ELSA.

TABLE 7: Communication (in GB) of secure aggregation using ELSA [69] or AUXILIATOR computing the same functionality. We use a tight analytical analysis of ELSA's communication for splitting it into offline and online communication.

		ELSA [69]	AUXILIATOR (§4)		
			Sacrifice	Sacrifice + ELSA OT	DZKP
200 000 params 50 clients	offline	5.66	19.37	6.87	2.39
	online	2.00	0.23	0.23	0.23
	total	7.66	19.60	7.10	2.62
200 000 params 200 clients	offline	22.65	77.49	27.49	9.57
	online	7.98	0.90	0.90	0.90
	total	30.63	78.38	28.39	10.47

In Table 7, we compare the communication of ELSA (when using a setup phase) with AUXILIATOR. For a fair comparison, we replace ELSA's original linear depth ripple-carry adder with a logarithmic depth parallel prefix adder, which we use in our protocols (Appendix B). This change has a negligible impact on overall communication but reduces the number of online communication rounds from 113 to 17.

The streamlined online phase of AUXILIATOR clearly outperforms ELSA by a factor of $8.81-8.89\times$. Additionally, AUXILIATOR requires only 11 online rounds. While the total communication is $2.92\times$ better when using DZKP, ELSA outperforms the sacrifice-based version by $2.56\times$. Regarding the comparison to DZKP, note that ELSA is based

on computationally lightweight computation rendering this communication comparison unfair.

To achieve good total communication when using the lightweight sacrifice approach, we improve the binary to arithmetic conversion required in ELSA's functionality. This conversion is the main bottleneck responsible for more than 98% of the communication for AUXILIATOR and over 85% for ELSA itself. Note that while the current version of AUXILIATOR is not well optimized for binary to arithmetic conversion, it has $2.89 - 2.90\times$ better communication for all other operations combined. For the conversion, ELSA uses an OT based approach where it profits from a batched verification over the field $\mathbb{F}_{2^\kappa}$, while rest of the protocols work on $\mathbb{Z}_{2^\ell}$.

To further enhance the communication efficiency, we replace the preprocessing step of the conversion used in AUXILIATOR, which is based on the ASTRA approach [24], with the ELSA conversion. By doing this, we enable AUXILIATOR to partially compute on fields during preprocessing only. This results in the sacrifice + ELSA OT variant of AUXILIATOR. This variant not only significantly improves the online phase, but also achieves slightly better total communication than ELSA, surpassing it by $1.08\times$.

Appendix B. Gates for Machine Learning

This section provides details for fixed-point truncation and ReLUs for ASTRA [24], [75] and SWIFT [51], [75], as analyzed and implemented for our evaluation and their modifications for AUXILIATOR (§4) and SOCIUM (§5). Note that among different publications, details for ASTRA and SWIFT may slightly deviate. We use the most current versions provided by [75] with minor tweaks.

Notation: By x^d we denote the truncation of x, i.e., an arithmetic right shift by $d = 16$. By $(b)^{\mathsf{a}}$ for a bit b, we denote its representation as an element of $\mathbb{Z}_{2^\ell}$, i.e., its padding by $\ell - 1 = 63$ 0 bits. To clearly distinguish between shares in the binary and arithmetic domains, we use, e.g., $\langle x \rangle$ for an arithmetic sharing of $x \in \mathbb{Z}_{2^\ell}$ and $\langle x \rangle^{\mathbf{B}}$ for a binary sharing of $b \in \mathbb{Z}_2$. By $x[i]$, we denote bit i of integer x, i.e., x_i where x is decomposed into x_j with $x = \sum_{j=0}^{\ell-1} x_j$.

B.1. Truncation Pair Generation

SWIFT requires a designated protocol to generate truncation pairs that is used in further operations.

Protocol $\Pi_{\mathsf{Tr}}()$

Input: None.

Output: $\langle r \rangle$ for random $r \in \mathbb{Z}_{2^\ell}$, and $[\![r^d]\!]$.

Offline:

1. For $s \in \{1,2\}, 0 \le i < \ell$, $\mathsf{S}_0, \mathsf{S}_s$ non-interactively sample random bit $r_{s,i} \in \{0,1\}$.
2. Compute $\langle r \rangle$:
   a. Compute $\langle r_{s,i} \rangle$ for $s \in \{1,2\}, 0 \le i < \ell$ by setting $r^0_{1,i} = r^2_{1,i} = r^0_{2,i} = r^1_{2,i} = 0 \in \mathbb{Z}_{2^\ell}$ and $r^1_{1,i} = (r_{1,i})^{\mathsf{a}}, r^2_{2,i} = (r_{2,i})^{\mathsf{a}}$ for $0 \le i < \ell$.
   b. Set up vector $\langle \vec{\mathbf{A}} \rangle$ of dimension ℓ where $\langle \vec{\mathbf{A}}_{i+1} \rangle = 2^{i+1} \cdot \langle r_{1,i} \rangle$ for $0 \le i < \ell$.
   c. Set up vector $\langle \vec{\mathbf{B}} \rangle$ of dimension ℓ where $\langle \vec{\mathbf{B}}_{i+1} \rangle = \langle r_{2,i} \rangle$ for $0 \le i < \ell$.
   d. Invoke $\mathcal{F}_{\mathsf{MultPre}}$ on $\langle \vec{\mathbf{A}} \rangle^{\intercal}$ and $\langle \vec{\mathbf{B}} \rangle$ to obtain $\langle x \rangle$ with $x = \vec{\mathbf{A}}^{\intercal} \cdot \vec{\mathbf{B}}$.
   e. Locally compute $\langle r \rangle = \sum_{i=0}^{\ell-1} 2^i (\langle r_{1,i} \rangle + \langle r_{2,i} \rangle) - \langle x \rangle$.
3. Compute $[\![r^d]\!]$:
   a. Compute $\langle r_{s,i} \rangle$ for $s \in \{1,2\}, f \le i < \ell$ by setting $\lambda^0_{r_{1,i}} = \lambda^2_{r_{1,i}} = \mathsf{m}_{r_{1,i}} = \lambda^0_{r_{2,i}} = \lambda^1_{r_{2,i}} = \mathsf{m}_{r_{2,i}} = 0 \in \mathbb{Z}_{2^\ell}$ and $\lambda^1_{r_{1,i}} = (r_{1,i})^{\mathsf{a}}, \lambda^2_{r_{2,i}} = (r_{2,i})^{\mathsf{a}}$ for $f \le i < \ell$.
   b. Set up vector $\langle \vec{\mathbf{C}} \rangle$ of dimension $\ell - f$ where $\langle \vec{\mathbf{C}}_{i+1} \rangle = 2^{i+1} \cdot \langle r_{1,f+i} \rangle$ for $0 \le i < \ell - f - 1$ and $\langle \vec{\mathbf{C}}_{\ell-f} \rangle = (\sum_{i=\ell-f}^{\ell} 2^i) \langle r_{1,\ell-1} \rangle$.
   c. Set up vector $\langle \vec{\mathbf{D}} \rangle$ of dimension $\ell - f$ where $\langle \vec{\mathbf{D}}_{i+1} \rangle = \langle r_{2,f+i} \rangle$ for $0 \le i < \ell - f$.
   d. Invoke $\mathcal{F}_{\mathsf{MultPre}}$ on $\langle \vec{\mathbf{C}} \rangle^{\intercal}$ and $\langle \vec{\mathbf{D}} \rangle$ to obtain $\langle y \rangle$ with $y = \vec{\mathbf{C}}^{\intercal} \cdot \vec{\mathbf{D}}$.
   e. Locally compute $\langle r^d \rangle = \sum_{i=0}^{\ell-f-2} 2^i (\langle r_{1,f+i} \rangle + \langle r_{2,f+i} \rangle) + \sum_{i=\ell-f-1}^{\ell-1} 2^i (\langle r_{1,\ell-1} \rangle + \langle r_{2,\ell-1} \rangle) - \langle y \rangle$ and set $\mathsf{m}_{r^d} = 0$ to obtain $[\![r^d]\!]$.

Figure 13: SWIFT: Truncation pair generation protocol [75].

B.2. Matrix Multiplication With Truncation

Protocol $\Pi^{\mathsf{FPA}}_{\mathsf{Mult}}(\langle \vec{\mathbf{X}} \rangle, \langle \vec{\mathbf{Y}} \rangle)$

Input: $\langle \cdot \rangle$-shares of $\vec{\mathbf{X}} \in \mathbb{Z}_{2^\ell}^{u \times w}$ and $\vec{\mathbf{Y}} \in \mathbb{Z}_{2^\ell}^{w \times v}$.

Output: $\langle \cdot \rangle$-shares of $\vec{\mathbf{Z}} \in \mathbb{Z}_{2^\ell}^{u \times v}$ with $\vec{\mathbf{Z}} = \vec{\mathbf{X}} \cdot \vec{\mathbf{Y}}$.

Setup:

1. Non-interactively generate $[\vec{\mathbf{Q}}]$ by $\mathsf{S}_0, \mathsf{S}_1$ sampling random $\vec{\mathbf{Q}}^{\mathbf{1}} \in \mathbb{Z}_{2^\ell}^{u \times v}$, $\mathsf{S}_0, \mathsf{S}_2$ sampling random $\vec{\mathbf{Q}}^{\mathbf{2}} \in \mathbb{Z}_{2^\ell}^{u \times v}$.
2. $\mathsf{S}_0, \mathsf{S}_1$ sample random $\vec{\lambda}^{\mathbf{1}}_{\mathbf{Z}} \in \mathbb{Z}_{2^\ell}^{u \times v}$.
3. S_0 computes and sends $\vec{\lambda}^{\mathbf{2}}_{\mathbf{Z}} = \left((\vec{\lambda}_{\mathbf{X}} \cdot \vec{\lambda}_{\mathbf{Y}}) - \vec{\mathbf{Q}}^{\mathbf{1}} - \vec{\mathbf{Q}}^{\mathbf{2}} \right)^d - \vec{\lambda}^{\mathbf{1}}_{\mathbf{Z}}$ to S_2.

Online:

1. Locally compute the following:
   - S_1: $\vec{\mathbf{P}}^{\mathbf{1}} = (\vec{\mathsf{m}}_{\mathbf{X}} \cdot \vec{\lambda}^{\mathbf{1}}_{\mathbf{Y}}) + (\vec{\lambda}^{\mathbf{1}}_{\mathbf{X}} \cdot \vec{\mathsf{m}}_{\mathbf{Y}}) + \vec{\mathbf{Q}}^{\mathbf{1}}$.
   - S_2: $\vec{\mathbf{P}}^{\mathbf{2}} = (\vec{\mathsf{m}}_{\mathbf{X}} \cdot \vec{\lambda}^{\mathbf{2}}_{\mathbf{Y}}) + (\vec{\lambda}^{\mathbf{2}}_{\mathbf{X}} \cdot \vec{\mathsf{m}}_{\mathbf{Y}}) + \vec{\mathbf{Q}}^{\mathbf{2}}$.
2. $\mathsf{S}_1, \mathsf{S}_2$ exchange $\vec{\mathbf{P}}^{\mathbf{1}}, \vec{\mathbf{P}}^{\mathbf{2}}$ and reconstruct $\vec{\mathsf{m}}_{\mathbf{Z}} = \left(\vec{\mathbf{P}}^{\mathbf{1}} + \vec{\mathbf{P}}^{\mathbf{2}} + (\vec{\mathsf{m}}_{\mathbf{X}} \cdot \vec{\mathsf{m}}_{\mathbf{Y}}) \right)^d$.

Figure 14: ASTRA: Matrix multiplication protocol for fixed-point arithmetic [75].

Protocol $\Pi^{\mathsf{FPA}}_{\mathsf{Mult}}(\langle \vec{\mathbf{X}} \rangle, \langle \vec{\mathbf{Y}} \rangle)$

Input: $\langle \cdot \rangle$-shares of $\vec{\mathbf{X}} \in \mathbb{Z}_{2^\ell}^{u \times w}, \vec{\mathbf{Y}} \in \mathbb{Z}_{2^\ell}^{w \times v}$.

Output: $\langle \cdot \rangle$-shares of $\vec{\mathbf{Z}} = \vec{\mathbf{X}} \cdot \vec{\mathbf{Y}} \in \mathbb{Z}_{2^\ell}^{u \times v}$.

Setup:

1. Invoke $\mathcal{F}_{\mathsf{MultPre}}([\vec{\lambda}_{\mathbf{X}}], [\vec{\lambda}_{\mathbf{Y}}])$ to obtain $[\vec{\gamma}_{\mathbf{XY}} = \vec{\lambda}_{\mathbf{X}} \cdot \vec{\lambda}_{\mathbf{Y}}]$.

2. Non-interactively generate $[\vec{\lambda}_{\mathbf{Z}}]$ by $\mathsf{S}_0, \mathsf{S}_1$ sampling random $\vec{\lambda}^{\mathbf{1}}_{\mathbf{Z}} \in \mathbb{Z}_{2^\ell}^{u \times v}$, $\mathsf{S}_0, \mathsf{S}_2$ sampling random $\vec{\lambda}^{\mathbf{2}}_{\mathbf{Z}} \in \mathbb{Z}_{2^\ell}^{u \times v}$.

Online:

1. Locally compute the following:
   - S_1: $\vec{\mathbf{P}}^{\mathbf{1}} = \left((\vec{m}_{\mathbf{X}} \cdot \vec{m}_{\mathbf{Y}}) + (\vec{m}_{\mathbf{X}} \cdot \vec{\lambda}^{\mathbf{1}}_{\mathbf{Y}}) + (\vec{\lambda}^{\mathbf{1}}_{\mathbf{X}} \cdot \vec{m}_{\mathbf{Y}}) + \vec{\gamma}^{\mathbf{1}}_{\mathbf{XY}}\right)^d - \vec{\lambda}^{\mathbf{1}}_{\mathbf{Z}}$.
   - S_2: $\vec{\mathbf{P}}^{\mathbf{2}} = \left((\vec{m}_{\mathbf{X}} \cdot \vec{\lambda}^{\mathbf{2}}_{\mathbf{Y}}) + (\vec{\lambda}^{\mathbf{2}}_{\mathbf{X}} \cdot \vec{m}_{\mathbf{Y}}) + \vec{\gamma}^{\mathbf{2}}_{\mathbf{XY}}\right)^d - \vec{\lambda}^{\mathbf{2}}_{\mathbf{Z}}$.
2. $\mathsf{S}_1, \mathsf{S}_2$ exchange $\vec{\mathbf{P}}^{\mathbf{1}}, \vec{\mathbf{P}}^{\mathbf{2}}$ and reconstruct $\vec{m}_{\mathbf{Z}} = \vec{\mathbf{P}}^{\mathbf{1}} + \vec{\mathbf{P}}^{\mathbf{2}}$.

Figure 15: AUXILIATOR: Matrix multiplication protocol for fixed-point arithmetic.

Protocol $\Pi^{\mathsf{FPA}}_{\mathsf{Mult}}([\![\vec{\mathbf{X}}]\!], [\![\vec{\mathbf{Y}}]\!])$

Input: $[\![\cdot]\!]$-shares of $\vec{\mathbf{X}} \in \mathbb{Z}_{2^\ell}^{u \times w}, \vec{\mathbf{Y}} \in \mathbb{Z}_{2^\ell}^{w \times v}$.

Output: $[\![\cdot]\!]$-shares of $\vec{\mathbf{Z}} = \vec{\mathbf{X}} \cdot \vec{\mathbf{Y}} \in \mathbb{Z}_{2^\ell}^{u \times v}$.

Setup:

1. Invoke $\mathcal{F}'_{\mathsf{MultPre}}$ on $\langle \vec{\lambda}_{\mathbf{X}} \rangle$ and $\langle \vec{\lambda}_{\mathbf{Y}} \rangle$ to obtain $\langle \vec{\gamma}_{\mathbf{XY}} \rangle'$ with $\vec{\gamma}_{\mathbf{XY}} = \vec{\lambda}_{\mathbf{X}} \cdot \vec{\lambda}_{\mathbf{Y}}$ (S_0 has $\vec{\gamma}^{\mathbf{0}}_{\mathbf{XY}}, \vec{\gamma}^{\mathbf{2}}_{\mathbf{XY}}$,$\mathsf{S}_1$ has $\vec{\gamma}^{\mathbf{1}}_{\mathbf{XY}}, \vec{\gamma}^{\mathbf{0}}_{\mathbf{XY}}$,$\mathsf{S}_2$ has $\vec{\gamma}^{\mathbf{2}}_{\mathbf{XY}}$).

2. Non-interactively generate $\langle \vec{\lambda}_{\mathbf{Z}} \rangle$ by $\mathsf{S}_i, \mathsf{S}_{i+1}$ sampling random $\vec{\lambda}^{\mathbf{i}}_{\mathbf{Z}} \in \mathbb{Z}_{2^\ell}$ for $i \in \{0, 1, 2\}$.

Online:

1. Locally compute the following (S_0 delays this step until final verification):
   - S_1 : $\vec{\mathbf{P}}^{\mathbf{0}} = \vec{m}_{\mathbf{X}} \cdot \vec{\lambda}^{\mathbf{0}}_{\mathbf{Y}} + \vec{\lambda}^{\mathbf{0}}_{\mathbf{X}} \cdot \vec{m}_{\mathbf{Y}} + \vec{\gamma}^{\mathbf{0}}_{\mathbf{XY}}$.
   - S_1 : $\vec{\mathbf{P}}^{\mathbf{1}} = \vec{m}_{\mathbf{X}} \cdot \vec{\lambda}^{\mathbf{1}}_{\mathbf{Y}} + \vec{\lambda}^{\mathbf{1}}_{\mathbf{X}} \cdot \vec{m}_{\mathbf{Y}} + \vec{\gamma}^{\mathbf{1}}_{\mathbf{XY}}$.
   - $\mathsf{S}_2, \mathsf{S}_0$: $\vec{\mathbf{P}}^{\mathbf{2}} = \vec{m}_{\mathbf{X}} \cdot \vec{\lambda}^{\mathbf{2}}_{\mathbf{Y}} + \vec{\lambda}^{\mathbf{2}}_{\mathbf{X}} \cdot \vec{m}_{\mathbf{Y}} + \vec{\gamma}^{\mathbf{2}}_{\mathbf{XY}} + \vec{m}_{\mathbf{X}} \cdot \vec{m}_{\mathbf{Y}}$.
2. Send the following messages:
   - S_1 sends $\vec{\mathbf{Q}}^{\mathbf{1}} = \left(\vec{\mathbf{P}}^{\mathbf{0}} + \vec{\mathbf{P}}^{\mathbf{1}}\right)^d - \vec{\lambda}^{\mathbf{0}}_{\mathbf{Z}} - \vec{\lambda}^{\mathbf{1}}_{\mathbf{Z}}$ to S_2.
   - S_2 sends $\vec{\mathbf{Q}}^{\mathbf{2}} = \left(\vec{\mathbf{P}}^{\mathbf{2}}\right)^d - \vec{\lambda}^{\mathbf{2}}_{\mathbf{Z}}$ to S_1.
3. $\mathsf{S}_1, \mathsf{S}_2$ reconstruct $\vec{m}_{\mathbf{Z}} = \vec{\mathbf{Q}}^{\mathbf{1}} + \vec{\mathbf{Q}}^{\mathbf{2}}$.

Final Verification (batched over all multiplications):

1. S_1 sends $\vec{\mathbf{Q}}^{\mathbf{1}}$ to S_0.
2. S_0 computes $\vec{\mathbf{Q}}^{\mathbf{2}}$ and reconstructs $\vec{m}_{\mathbf{Z}} = \vec{\mathbf{Q}}^{\mathbf{1}} + \vec{\mathbf{Q}}^{\mathbf{2}}$.
3. S_0 sends $\mathsf{H}(\vec{\mathbf{Q}}^{\mathbf{2}})$ to S_1 that then checks consistency to the $\vec{\mathbf{Q}}^{\mathbf{2}}$ it received from S_2.

Figure 16: SOCIUM: Matrix multiplication protocol for fixed-point arithmetic.

Protocol $\Pi^{\mathsf{FPA}}_{\mathsf{Mult}}([\![\vec{\mathbf{X}}]\!], [\![\vec{\mathbf{Y}}]\!])$

Input: $[\![\cdot]\!]$-shares of $\vec{\mathbf{X}} \in \mathbb{Z}_{2^\ell}^{u \times w}, \vec{\mathbf{Y}} \in \mathbb{Z}_{2^\ell}^{w \times v}$.

Output: $[\![\cdot]\!]$-shares of $\vec{\mathbf{Z}} = \vec{\mathbf{X}} \cdot \vec{\mathbf{Y}} \in \mathbb{Z}_{2^\ell}^{u \times v}$.

Setup:

1. Invoke $\mathcal{F}_{\mathsf{MultPre}}$ on $\langle \vec{\lambda}_{\mathbf{X}} \rangle$ and $\langle \vec{\lambda}_{\mathbf{Y}} \rangle$ to obtain $\langle \vec{\gamma}_{\mathbf{XY}} \rangle$ with $\vec{\gamma}_{\mathbf{XY}} = \vec{\lambda}_{\mathbf{X}} \cdot \vec{\lambda}_{\mathbf{Y}}$.

2. Non-interactively generate $\langle \vec{\lambda}_{\mathbf{W}} \rangle$ where $\mathsf{S}_1, \mathsf{S}_2$ know all shares by $\mathsf{S}_0, \mathsf{S}_1, \mathsf{S}_2$ sampling random $\vec{\lambda}^{\mathbf{0}}_{\mathbf{W}}, \vec{\lambda}^{\mathbf{2}}_{\mathbf{W}} \in \mathbb{Z}_{2^\ell}^{u \times v}$ and $\mathsf{S}_1, \mathsf{S}_2$ sampling random $\vec{\lambda}^{\mathbf{0}}_{\mathbf{W}}, \vec{\lambda}^{\mathbf{2}}_{\mathbf{W}} \in \mathbb{Z}_{2^\ell}^{u \times v}$.

3. Obtain $\langle \vec{\mathbf{R}} \rangle$ and $[\![\vec{\mathbf{R}}^d]\!]$ for random $\vec{\mathbf{R}} \in \mathbb{Z}_{2^\ell}^{u \times v}$ by calling uv instances of Π_{Tr}.

Note: $\vec{\lambda}_{\mathbf{Z}} = \vec{\lambda}_{\mathbf{W}} + \vec{\lambda}_{\mathbf{R^d}}$.

Online:

1. Locally compute the following (S_0 delays this step until final verification):
   - $\mathsf{S}_0, \mathsf{S}_1$: $\vec{\mathbf{P}}^{\mathbf{0}} = \vec{m}_{\mathbf{X}} \cdot \vec{\lambda}^{\mathbf{0}}_{\mathbf{Y}} + \vec{\lambda}^{\mathbf{0}}_{\mathbf{X}} \cdot \vec{m}_{\mathbf{Y}} + \vec{\gamma}^{\mathbf{0}}_{\mathbf{XY}} - \vec{\mathbf{R}}^{\mathbf{0}}$.
   - $\mathsf{S}_1, \mathsf{S}_2$: $\vec{\mathbf{P}}^{\mathbf{1}} = \vec{m}_{\mathbf{X}} \cdot \vec{\lambda}^{\mathbf{1}}_{\mathbf{Y}} + \vec{\lambda}^{\mathbf{1}}_{\mathbf{X}} \cdot \vec{m}_{\mathbf{Y}} + \vec{\gamma}^{\mathbf{1}}_{\mathbf{XY}} - \vec{\mathbf{R}}^{\mathbf{1}}$.
   - $\mathsf{S}_2, \mathsf{S}_0$: $\vec{\mathbf{P}}^{\mathbf{2}} = \vec{m}_{\mathbf{X}} \cdot \vec{\lambda}^{\mathbf{2}}_{\mathbf{Y}} + \vec{\lambda}^{\mathbf{2}}_{\mathbf{X}} \cdot \vec{m}_{\mathbf{Y}} + \vec{\gamma}^{\mathbf{2}}_{\mathbf{XY}} - \vec{\mathbf{R}}^{\mathbf{2}}$.
2. $\mathsf{S}_1, \mathsf{S}_2$ exchange $\vec{\mathbf{P}}^{\mathbf{0}}, \vec{\mathbf{P}}^{\mathbf{2}}$, reconstruct $\vec{\mathbf{P}} = \vec{\mathbf{P}}^{\mathbf{0}} + \vec{\mathbf{P}}^{\mathbf{1}} + \vec{\mathbf{P}}^{\mathbf{2}}$ and set $\vec{\mathbf{W}} = \left(\vec{\mathbf{P}} + \vec{m}_{\mathbf{X}} \cdot \vec{m}_{\mathbf{Y}}\right)^d$.
3. $\mathsf{S}_1, \mathsf{S}_2$ set up $[\![\vec{\mathbf{W}}]\!]$ by setting $\vec{m}_{\mathbf{W}} = \vec{\mathbf{W}} - (\vec{\lambda}^{\mathbf{0}}_{\mathbf{W}} + \vec{\lambda}^{\mathbf{1}}_{\mathbf{W}} + \vec{\lambda}^{\mathbf{2}}_{\mathbf{W}})$
4. $\mathsf{S}_1, \mathsf{S}_2$ compute $[\![\vec{\mathbf{Z}}]\!] = [\![\vec{\mathbf{W}}]\!] + [\![\vec{\mathbf{R}}^d]\!]$.

Final Verification (batched over all multiplications):

1. S_1 sends $\vec{m}_{\mathbf{W}}$ and S_2 sends $\mathsf{H}(\vec{m}_{\mathbf{W}})$ to S_0 that then checks consistency.
2. S_0 computes $\vec{\mathbf{P}}^{\mathbf{0}}, \vec{\mathbf{P}}^{\mathbf{2}}$ and sends $\mathsf{H}(\vec{\mathbf{P}}^{\mathbf{2}})$ to S_1 and $\mathsf{H}(\vec{\mathbf{P}}^{\mathbf{0}})$ to S_2 that then check consistency to the values received in online step 2.
3. S_0 computes its shares of $[\![\vec{\mathbf{Z}}]\!]$.

Figure 17: SWIFT: Matrix multiplication protocol for fixed-point arithmetic [75].

B.3. Multiplication by Constant With Truncation

Protocol $\Pi^{\mathsf{FPA}}_{\mathsf{MultConstant}}(\langle x \rangle, c)$

Input: $\langle \cdot \rangle$-shares of $x \in \mathbb{Z}_{2^\ell}$, and a public constant $c \in \mathbb{Z}_{2^\ell}$.

Output: $\langle \cdot \rangle$-shares of $z \in \mathbb{Z}_{2^\ell}$ with $z = c \cdot x$.

Setup:

1. Generate $[\lambda_z]$ with $\lambda_z = c \cdot \lambda_x$:
   - $\mathsf{S}_0, \mathsf{S}_1$ sample random $\lambda^1_z \in \mathbb{Z}_{2^\ell}$.
   - S_0 computes and sends $\lambda^2_z = (c \cdot \lambda_x)^d - \lambda^1_z$ to S_2.

Online:

2. $\mathsf{S}_1, \mathsf{S}_2$ compute $\mathsf{m}_z = (c \cdot \mathsf{m}_x)^d$.

Figure 18: ASTRA: Multiplication with a constant protocol for fixed-point arithmetic [75].

Protocol $\Pi^{\mathsf{FPA}}_{\mathsf{MultConstant}}(\langle x \rangle, c)$

Input: $\langle \cdot \rangle$-shares of $x \in \mathbb{Z}_{2^\ell}$, and a public constant $c \in \mathbb{Z}_{2^\ell}$.

Output: $\langle \cdot \rangle$-shares of $z \in \mathbb{Z}_{2^\ell}$ with $z = c \cdot x$.

Setup:

1. Non-interactively generate $[\lambda_z]$ by $\mathsf{S}_0, \mathsf{S}_1$ sampling random $\lambda^1_z \in \mathbb{Z}_{2^\ell}$, $\mathsf{S}_0, \mathsf{S}_2$ sampling random $\lambda^2_z \in \mathbb{Z}_{2^\ell}$.
2. S_2 computes and sends $p^2 = \left(c \cdot \lambda^2_x\right)^d - \lambda^2_z$ to S_1.

Online:

1. S_1 computes and sends $p^1 = \left(c \cdot (\mathsf{m}_x + \lambda^1_x)\right)^d - \lambda^1_z$ to S_2.
2. $\mathsf{S}_1, \mathsf{S}_2$ compute $\mathsf{m}_z = p^1 + p^2$.

Figure 19: AUXILIATOR: Multiplication with a constant protocol for fixed-point arithmetic.

Protocol $\Pi^{\mathsf{FPA}}_{\mathsf{MultConstant}}(\llbracket x \rrbracket, c)$

Input: $\llbracket \cdot \rrbracket$-shares of $x \in \mathbb{Z}_{2^\ell}$ and a public constant $c \in \mathbb{Z}_{2^\ell}$.

Output: $\llbracket \cdot \rrbracket$-shares of $z \in \mathbb{Z}_{2^\ell}$ with $z = c \cdot x$.

Setup:

1. Non-interactively generate $\langle \lambda_z \rangle$ by $\mathsf{S}_i, \mathsf{S}_{i+1}$ sampling random $\lambda_z^i \in \mathbb{Z}_{2^\ell}$ for $i \in \{0, 1, 2\}$.
2. S_0 computes and sends $p^0 = \left(c(\lambda_x^0 + \lambda_x^2)\right)^d - \lambda_z^0$ to S_2.
3. S_0 computes and sends $p^2 = \left(c(\lambda_x^0 + \lambda_x^2)\right)^d - \lambda_z^2$ to S_1.

Online:

1. $\mathsf{S}_1, \mathsf{S}_2$ compute $p^1 = \left(c(\mathsf{m}_x + \lambda_x^1)\right)^d - \lambda_z^1$.
2. $\mathsf{S}_1, \mathsf{S}_2$ compute m_z:
   - S_1 computes $\mathsf{m}_z = p^1 + p^2 - \lambda_z^0$.
   - S_2 computes $\mathsf{m}_z = p^1 + p^0 - \lambda_z^2$.

Final Verification (batched over all multiplications):

1. S_1 sends p^1 to S_0.
2. S_0 computes m_z like S_1 did in online step 2.

Figure 20: SOCIUM: Multiplication with a constant protocol for fixed-point arithmetic.

Protocol $\Pi^{\mathsf{FPA}}_{\mathsf{MultConstant}}(\llbracket x \rrbracket, c)$

Input: $\llbracket \cdot \rrbracket$-shares of $x \in \mathbb{Z}_{2^\ell}$ and a public constant $c \in \mathbb{Z}_{2^\ell}$.

Output: $\llbracket \cdot \rrbracket$-shares of $z \in \mathbb{Z}_{2^\ell}$ with $z = c \cdot x$.

Setup:

1. Obtain $\langle r \rangle$ and $\llbracket r^d \rrbracket$ for random $r \in \mathbb{Z}_{2^\ell}$ by calling Π_{Tr}.
2. Reinterpret $\langle r \rangle$ as $\llbracket r \rrbracket$ by setting $\mathsf{m}_r = 0$.

Online:

1. $\mathsf{S}_1, \mathsf{S}_2$ compute $\llbracket p \rrbracket = c \cdot \llbracket x \rrbracket - \llbracket r \rrbracket$.
2. $\mathsf{S}_1, \mathsf{S}_2$ reconstruct p:
   - $\mathsf{S}_1, \mathsf{S}_2$ exchange λ_p^0, λ_p^2 and set $p = \mathsf{m}_p + \lambda_p^0 + \lambda_p^1 + \lambda_p^2$.
3. $\mathsf{S}_1, \mathsf{S}_2$ compute $\llbracket z \rrbracket = p^d + \llbracket r^d \rrbracket$.

Final Verification (batched over all multiplications):

1. S_1 sends λ_p^1 and S_2 sends $\mathsf{H}(\lambda_p^1)$ to S_0 that then checks consistency.
2. S_0 computes p like $\mathsf{S}_1, \mathsf{S}_2$ did in online step 2.
3. S_0 sends $\mathsf{H}(\lambda_p^2)$ to S_1 and $\mathsf{H}(\lambda_p^0)$ to S_2 that then check consistency to the values received in online step 2.
3. S_0 computes its shares of $\llbracket z \rrbracket$.

Figure 21: SWIFT: Multiplication with a constant protocol for fixed-point arithmetic [75].

B.4. Rectified Linear Units (ReLUs)

Note that our ReLUs work for integer and fixed-point arithmetic, but internally only utilize integer arithmetic. This is the case as the functionality is exactly the same for both cases: The most significant bit (MSB) always returns the sign of the number and multiplication by an integer sharing of 0 or 1 without truncation to another integer or fixed-point number is correct. The ReLU protocol is generic and depends on instantiations of Π_{MSB} and Π_{BitA} discussed in subsequent sections.

Protocol $\Pi_{\mathsf{ReLU}}(\langle v \rangle)$

Input: $\langle \cdot \rangle$-shares of $x \in \mathbb{Z}_{2^\ell}$.

Output: $\langle \cdot \rangle$-shares of $z \in \mathbb{Z}_{2^\ell}$ with $z = \max(0, x)$.

Online:

1. $\langle s \rangle^{\mathbf{B}} = \Pi_{\mathsf{MSB}}(\langle x \rangle)$.
2. $\langle t \rangle^{\mathbf{B}} = \langle s \rangle^{\mathbf{B}} \oplus \langle 1 \rangle^{\mathbf{B}}$.
3. $\langle t \rangle = \Pi_{\mathsf{BitA}}(\langle t \rangle^{\mathbf{B}})$.
4. $\langle z \rangle = \langle t \rangle \cdot \langle x \rangle$.

Figure 22: All protocols: Rectified linear unit (ReLU) protocol. For use in a specific protocol, it is sufficient to replace $\langle \cdot \rangle$-shares by the protocol's sharings and to then use the respective gates [75].

B.5. Most Significant Bit (MSB) Extraction

Like ASTRA and SWIFT, we utilize parallel prefix adders (PPAs) for MSB extraction. The utilized PPA (provided alongside our implementation code[16]) uses 128 AND gates and has a multiplicative depth of 6 for $\ell = 64$.

Protocol $\Pi_{\mathsf{MSB}}(\langle x \rangle)$

Input: $\langle \cdot \rangle$-shares of $x \in \mathbb{Z}_{2^\ell}$.

Output: $\langle \cdot \rangle^{\mathbf{B}}$-shares of $s \in \mathbb{Z}_2$ with $s = 1$ iff $x < 0$.

Setup:

1. $\mathsf{S}_0, \mathsf{S}_1$ sample random $\lambda_{a_k}^1 \in \mathbb{Z}_2$ for $0 \leq k < \ell$.
2. S_0 computes and sends $\lambda_{a_k}^2 = (\lambda_x^1 + \lambda_x^2)[k] \oplus \lambda_{a_k}^1$ to S_2 for $0 \leq k < \ell$.
2. Set up $\langle a_k \rangle^{\mathbf{B}} = \langle (\lambda_x^1 + \lambda_x^2)[k] \rangle^{\mathbf{B}}$ by setting $\mathsf{m}_{a_k} = 0$ for $0 \leq k < \ell$.

Online:

1. Set up $\langle b_k \rangle^{\mathbf{B}} = \langle \mathsf{m}_x[k] \rangle^{\mathbf{B}}$ by setting $\mathsf{m}_{b_k} = \mathsf{m}_x[k], \lambda_{b_k}^1 = \lambda_{b_k}^2 = 0$ for $0 \leq k < \ell$.
2. Compute sign $\langle s \rangle^{\mathbf{B}}$ of $\langle x \rangle$ by running a binary PPA circuit on inputs $(\langle a_{\ell-1} \rangle^{\mathbf{B}}, ..., \langle a_0 \rangle^{\mathbf{B}})$ and $(\langle b_{\ell-1} \rangle^{\mathbf{B}}, ..., \langle b_0 \rangle^{\mathbf{B}})$.

Figure 23: ASTRA: Most Significant Bit extraction protocol [75].

Protocol $\Pi_{\mathsf{MSB}}(\langle x \rangle)$

Input: $\langle \cdot \rangle$-shares of $x \in \mathbb{Z}_{2^\ell}$.

Output: $\langle \cdot \rangle^{\mathbf{B}}$-shares of $s \in \mathbb{Z}_2$ with $s = 1$ iff $x < 0$.

Setup:

1. $\mathsf{S}_0, \mathsf{S}_1$ sample random $r_k^1 \in \mathbb{Z}_2$ for $0 \leq k < \ell$.
2. Non-interactively set up $\langle b_k \rangle^{\mathbf{B}} = \langle \lambda_x^2[k] \rangle^{\mathbf{B}}$ for $0 \leq k < \ell$ by setting $\mathsf{m}_{b_k} = \lambda_{b_k}^1 = 0, \lambda_{b_k}^2 = \lambda_x^2[k]$.

Online:

1. S_1 computes and sends $r_k^2 = (\mathsf{m}_x + \lambda_x^1)[k] \oplus r_k^1$ to S_2 for $0 \leq k < \ell$.
2. Non-interactively set up $\langle a_k \rangle^{\mathbf{B}} = \langle (\mathsf{m}_x + \lambda_x^1)[k] \rangle^{\mathbf{B}}$ for $0 \leq k < \ell$ by setting $\mathsf{m}_{b_k} = r_k^2, \lambda_{b_k}^1 = r_k^1, \lambda_{b_k}^2 = 0$.
3. Compute sign $\langle s \rangle^{\mathbf{B}}$ of $\langle x \rangle$ by running a binary PPA circuit on inputs $(\langle a_{\ell-1} \rangle^{\mathbf{B}}, ..., \langle a_0 \rangle^{\mathbf{B}})$ and $(\langle b_{\ell-1} \rangle^{\mathbf{B}}, ..., \langle b_0 \rangle^{\mathbf{B}})$.

Figure 24: AUXILIATOR: Most Significant Bit extraction protocol.

16. https://encrypto.de/code/MOTION-FD

Protocol $\Pi_{\mathsf{MSB}}(\llbracket x \rrbracket)$

Input: $\llbracket \cdot \rrbracket$-shares of $x \in \mathbb{Z}_{2^\ell}$.

Output: $\llbracket \cdot \rrbracket^{\mathbf{B}}$-shares of $s \in \mathbb{Z}_2$ with $s = 1$ iff $x < 0$.

Setup:

1. Non-interactively set up $\llbracket a_k \rrbracket^{\mathbf{B}} = \llbracket \lambda_x^0[k] \rrbracket^{\mathbf{B}}$ for $0 \leq k < \ell$ by setting $\lambda_{a_k}^0 = \lambda_x^0[k], \lambda_{a_k}^1 = \lambda_{a_k}^2 = \mathsf{m}_{a_k} = 0$.

2. Non-interactively set up $\llbracket b_k \rrbracket^{\mathbf{B}} = \llbracket \lambda_x^2[k] \rrbracket^{\mathbf{B}}$ for $0 \leq k < \ell$ by setting $\lambda_{a_k}^2 = \lambda_x^2[k], \lambda_{a_k}^0 = \lambda_{a_k}^1 = \mathsf{m}_{a_k} = 0$.

3. For $0 \leq k < \ell$, $\mathsf{S}_1, \mathsf{S}_2$ sample random $\lambda_{c_k}^1 \in \mathbb{Z}_2$ and $\mathsf{S}_0, \mathsf{S}_1, \mathsf{S}_2$ sample random $\lambda_{c_k}^0, \lambda_{c_k}^2 \in \mathbb{Z}_2$.

Online:

1. Non-interactively (partially) set up $\llbracket c_k \rrbracket^{\mathbf{B}} = \llbracket (\mathsf{m}_x + \lambda_x^1)[k] \rrbracket^{\mathbf{B}}$ for $0 \leq k < \ell$ by $\mathsf{S}_1, \mathsf{S}_2$ setting $\mathsf{m}_{c_k} = (\mathsf{m}_x + \lambda_x^1)[k] \oplus \lambda_{c_k}^0 \oplus \lambda_{c_k}^1 \oplus \lambda_{c_k}^2$.

2. For $0 \leq k < \ell$, compute $\llbracket \cdot \rrbracket^{\mathbf{B}}$-shares of $s_k, t_k \in \mathbb{Z}_2$ that are the outputs of a full-adder with inputs a_k, b_k, c_k as follows:

   – $\llbracket s_k \rrbracket^{\mathbf{B}} = \llbracket a_k \rrbracket^{\mathbf{B}} \oplus \llbracket b_k \rrbracket^{\mathbf{B}} \oplus \llbracket c_k \rrbracket^{\mathbf{B}}$.

   – $\llbracket t_k \rrbracket^{\mathbf{B}} = \llbracket a_k \rrbracket^{\mathbf{B}} \oplus \big((\llbracket a_k \rrbracket^{\mathbf{B}} \oplus \llbracket b_k \rrbracket^{\mathbf{B}}) \wedge (\llbracket a_k \rrbracket^{\mathbf{B}} \oplus \llbracket c_k \rrbracket^{\mathbf{B}}) \big)$.

3. Compute sign $\llbracket s \rrbracket^{\mathbf{B}}$ of $\llbracket x \rrbracket$ by running a binary PPA circuit on inputs $(\llbracket s_{\ell-1} \rrbracket^{\mathbf{B}}, ..., \llbracket s_0 \rrbracket^{\mathbf{B}})$ and $(\llbracket s_{\ell-2} \rrbracket^{\mathbf{B}}, ..., \llbracket s_0 \rrbracket^{\mathbf{B}}, \llbracket 0 \rrbracket^{\mathbf{B}})$.

Final Verification (batched over all multiplications):

1. S_1 sends m_{c_k} to S_0 for $0 \leq k < \ell$.

2. S_0 now has its shares of $\llbracket c_k \rrbracket^{\mathbf{B}}$ available for further verification.

Figure 25: SOCIUM: Most Significant Bit extraction protocol.

Protocol $\Pi_{\mathsf{MSB}}(\llbracket x \rrbracket)$

Input: $\llbracket \cdot \rrbracket$-shares of $x \in \mathbb{Z}_{2^\ell}$.

Output: $\llbracket \cdot \rrbracket^{\mathbf{B}}$-shares of $s \in \mathbb{Z}_2$ with $s = 1$ iff $x < 0$.

Setup:

1. Non-interactively set up $\llbracket a_k \rrbracket^{\mathbf{B}} = \llbracket \lambda_x^0[k] \rrbracket^{\mathbf{B}}$ for $0 \leq k < \ell$ by setting $\lambda_{a_k}^0 = \lambda_x^0[k], \lambda_{a_k}^1 = \lambda_{a_k}^2 = \mathsf{m}_{a_k} = 0$.

2. Non-interactively set up $\llbracket b_k \rrbracket^{\mathbf{B}} = \llbracket \lambda_x^2[k] \rrbracket^{\mathbf{B}}$ for $0 \leq k < \ell$ by setting $\lambda_{a_k}^2 = \lambda_x^2[k], \lambda_{a_k}^0 = \lambda_{a_k}^1 = \mathsf{m}_{a_k} = 0$.

3. For $0 \leq k < \ell$, $\mathsf{S}_1, \mathsf{S}_2$ sample random $\lambda_{c_k}^1 \in \mathbb{Z}_2$ and $\mathsf{S}_0, \mathsf{S}_1, \mathsf{S}_2$ sample random $\lambda_{c_k}^0, \lambda_{c_k}^2 \in \mathbb{Z}_2$.

Online:

1. Non-interactively (partially) set up $\llbracket c_k \rrbracket^{\mathbf{B}} = \llbracket (\mathsf{m}_x + \lambda_x^1)[k] \rrbracket^{\mathbf{B}}$ for $0 \leq k < \ell$ by $\mathsf{S}_1, \mathsf{S}_2$ setting $\mathsf{m}_{c_k} = (\mathsf{m}_x + \lambda_x^1)[k] \oplus \lambda_{c_k}^0 \oplus \lambda_{c_k}^1 \oplus \lambda_{c_k}^2$.

2. For $0 \leq k < \ell$, compute $\llbracket \cdot \rrbracket^{\mathbf{B}}$-shares of $s_k, t_k \in \mathbb{Z}_2$ that are the outputs of a full-adder with inputs a_k, b_k, c_k as follows:

   – $\llbracket s_k \rrbracket^{\mathbf{B}} = \llbracket a_k \rrbracket^{\mathbf{B}} \oplus \llbracket b_k \rrbracket^{\mathbf{B}} \oplus \llbracket c_k \rrbracket^{\mathbf{B}}$.

   – $\llbracket t_k \rrbracket^{\mathbf{B}} = \llbracket a_k \rrbracket^{\mathbf{B}} \oplus \big((\llbracket a_k \rrbracket^{\mathbf{B}} \oplus \llbracket b_k \rrbracket^{\mathbf{B}}) \wedge (\llbracket a_k \rrbracket^{\mathbf{B}} \oplus \llbracket c_k \rrbracket^{\mathbf{B}}) \big)$.

3. Compute sign $\llbracket s \rrbracket^{\mathbf{B}}$ of $\llbracket x \rrbracket$ by running a binary PPA circuit on inputs $(\llbracket s_{\ell-1} \rrbracket^{\mathbf{B}}, ..., \llbracket s_0 \rrbracket^{\mathbf{B}})$ and $(\llbracket s_{\ell-2} \rrbracket^{\mathbf{B}}, ..., \llbracket s_0 \rrbracket^{\mathbf{B}}, \llbracket 0 \rrbracket^{\mathbf{B}})$.

Final Verification (batched over all multiplications):

1. S_1 sends m_{c_k} and S_2 sends $\mathsf{H}(\mathsf{m}_{c_k})$ to S_0 that then checks consistency for $0 \leq k < \ell$.

2. S_0 now has its shares of $\llbracket c_k \rrbracket^{\mathbf{B}}$ available for further verification.

Figure 26: SWIFT: Most Significant Bit extraction protocol [75].

B.6. Bit to Arithmetic Conversion

Protocol $\Pi_{\mathsf{BitA}}(\langle x \rangle^{\mathbf{B}})$

Input: $\langle \cdot \rangle^{\mathbf{B}}$-shares of $x \in \mathbb{Z}_2$.

Output: $\langle \cdot \rangle$-shares of $z = (x)^{\mathsf{a}} \in \mathbb{Z}_{2^\ell}$, i.e., z is x padded with $\ell - 1$ 0 bits.

Setup:

1. Generate $[p] = [(\lambda_x^1 + \lambda_x^2)^{\mathsf{a}}]$:

   – $\mathsf{S}_0, \mathsf{S}_1$ sample random $p^1 \in \mathbb{Z}_{2^\ell}$.

   – S_0 computes and sends $p^2 = (\lambda_x^1 + \lambda_x^2)^{\mathsf{a}} - p^1$ to S_2.

2. Non-interactively generate $[\lambda_z]$ by $\mathsf{S}_0, \mathsf{S}_1$ sampling random $\lambda_z^1 \in \mathbb{Z}_{2^\ell}$, $\mathsf{S}_0, \mathsf{S}_2$ sampling random $\lambda_z^2 \in \mathbb{Z}_{2^\ell}$.

Online:

1. Locally compute the following:

   – S_1: $q^1 = (1 - 2(\mathsf{m}_x)^{\mathsf{a}}) \cdot p^1 - \lambda_z^1$.

   – S_2: $q^2 = (1 - 2(\mathsf{m}_x)^{\mathsf{a}}) \cdot p^2 - \lambda_z^2$.

2. S_1 sends q^1 to S_2 and S_2 sends q^2 to S_1.

3. Compute $\mathsf{m}_z = q^1 + q^2 + (\mathsf{m}_x)^{\mathsf{a}}$.

Figure 27: ASTRA: Bit to arithmetic conversion [75].

Protocol $\Pi_{\mathsf{BitA}}(\langle x \rangle^{\mathbf{B}})$

Input: $\langle \cdot \rangle^{\mathbf{B}}$-shares of $x \in \mathbb{Z}_2$.

Output: $\langle \cdot \rangle$-shares of $z = (x)^{\mathsf{a}} \in \mathbb{Z}_{2^\ell}$, i.e., z is x padded with $\ell - 1$ 0 bits.

Setup:

1. Non-interactively set up $[p] = [(\lambda_x^1)^{\mathsf{a}}]$ by setting $p^1 = (\lambda_x^1)^{\mathsf{a}}, p^2 = 0$.

2. Non-interactively set up $[q] = [(\lambda_x^2)^{\mathsf{a}}]$ by setting $p^1 = 0, p^2 = (\lambda_x^2)^{\mathsf{a}}$.

3. Invoke $\mathcal{F}_{\mathsf{MultPre}}([p], [q])$ to obtain $[pq]$.

4. Compute $[r] = [(\lambda_x^1 \oplus \lambda_x^2)^{\mathsf{a}}] = [p] + [q] - 2[pq]$.

5. Non-interactively generate $[\lambda_z]$ by $\mathsf{S}_0, \mathsf{S}_1$ sampling random $\lambda_z^1 \in \mathbb{Z}_{2^\ell}$, $\mathsf{S}_0, \mathsf{S}_2$ sampling random $\lambda_z^2 \in \mathbb{Z}_{2^\ell}$.

Online:

1. Locally compute the following:

   – S_1: $s^1 = (1 - 2(\mathsf{m}_x)^{\mathsf{a}}) \cdot r^1 - \lambda_z^1$.

   – S_2: $s^2 = (1 - 2(\mathsf{m}_x)^{\mathsf{a}}) \cdot r^2 - \lambda_z^2$.

2. S_1 sends s^1 to S_2 and S_2 sends s^2 to S_1.

3. Compute $\mathsf{m}_z = s^1 + s^2 + (\mathsf{m}_x)^{\mathsf{a}}$.

Figure 28: AUXILIATOR: Bit to arithmetic conversion.

Protocol $\Pi_{\mathsf{BitA}}(\llbracket x \rrbracket^{\mathbf{B}})$

Input: $\llbracket \cdot \rrbracket^{\mathbf{B}}$-shares of $x \in \mathbb{Z}_2$.

Output: $\llbracket \cdot \rrbracket$-shares of $z = (x)^{\mathsf{a}} \in \mathbb{Z}_{2^\ell}$, i.e., z is x padded with $\ell - 1$ 0 bits.

Setup:

1. Non-interactively set up $\langle p \rangle = \langle (\lambda_x^0)^{\mathsf{a}} \rangle$ by setting $p^0 = (\lambda_x^0)^{\mathsf{a}}, p^1 = p^2 = 0$.

2. Non-interactively set up $\langle q \rangle = \langle (\lambda_x^1)^{\mathsf{a}} \rangle$ by setting $q^1 = (\lambda_x^1)^{\mathsf{a}}, q^0 = q^2 = 0$.

3. Non-interactively set up $\langle r \rangle = \langle (\lambda_x^0)^{\mathsf{a}} \rangle$ by setting $r^2 = (\lambda_x^2)^{\mathsf{a}}, r^0 = r^1 = 0$.

4. Invoke $\mathcal{F}_{\mathsf{MultPre}}(\langle p \rangle, \langle q \rangle)$ to obtain $\langle pq \rangle$. **Note that SWIFT func-**

tionality $\mathcal{F}_{\mathsf{MultPre}}$ is used instead of SOCIUM functionality $\mathcal{F}'_{\mathsf{MultPre}}$ here to obtain a full replicated sharing. Still, the simplified verification of SOCIUM can be used in its instantiation.

5. Compute $\langle s\rangle = \langle(\lambda_x^0 \oplus \lambda_x^1)^{\mathsf{a}}\rangle = \langle p\rangle + \langle q\rangle - 2\langle pq\rangle$.

6. Invoke $\mathcal{F}'_{\mathsf{MultPre}}(\langle s\rangle, \langle c\rangle)$ to obtain $\langle sc\rangle'$.

7. Compute $\langle t\rangle' = \langle(\lambda_x^0 \oplus \lambda_x^1 \oplus \lambda_x^2)^{\mathsf{a}}\rangle' = \langle s\rangle' + \langle c\rangle' - 2\langle sc\rangle'$.

8. Non-interactively generate $[\lambda_z]$ by $\mathsf{S}_0, \mathsf{S}_1$ sampling random $\lambda_z^1 \in \mathbb{Z}_{2^\ell}$, $\mathsf{S}_0, \mathsf{S}_2$ sampling random $\lambda_z^2 \in \mathbb{Z}_{2^\ell}$.

Online:

1. Locally compute the following (S_0 delays the step until the final verification):

- S_1 : $u^0 = (1 - 2(\mathsf{m}_x)^{\mathsf{a}}) \cdot t^0 - \lambda_z^0$
- S_1 : $u^1 = (1 - 2(\mathsf{m}_x)^{\mathsf{a}}) \cdot t^1 - \lambda_z^1$
- $\mathsf{S}_2, \mathsf{S}_0$: $u^2 = (1 - 2(\mathsf{m}_x)^{\mathsf{a}}) \cdot t^2 - \lambda_z^2$

2. S_2 sends u^2 to S_1 and S_1 sends $u^0 + u^1$ to S_2.

3. $\mathsf{S}_1, \mathsf{S}_2$ set $\mathsf{m}_z = (\mathsf{m}_x)^{\mathsf{a}} + u^0 + u^1 + u^2$.

Final Verification (batched over all multiplications):

1. S_1 sends $u^0 + u^1$ to S_0.

2. S_0 computes u^2 and reconstructs $\mathsf{m}_z = (\mathsf{m}_x)^{\mathsf{a}} + u^0 + u^1 + u^2$.

3. S_0 sends $\mathcal{H}(u^2)$ to S_1 that then checks consistency to the u^2 it received from S_2.

Figure 29: SOCIUM: Bit to arithmetic conversion.

Protocol $\Pi_{\mathsf{BitA}}(\llbracket x \rrbracket^{\mathbf{B}})$

Input: $\llbracket\cdot\rrbracket^{\mathbf{B}}$-shares of $x \in \mathbb{Z}_2$.

Output: $\llbracket\cdot\rrbracket$-shares of $z = (x)^{\mathsf{a}} \in \mathbb{Z}_{2^\ell}$, i.e., z is x padded with $\ell - 1$ 0 bits.

Setup:

1. Non-interactively set up $\langle p\rangle = \langle(\lambda_x^0)^{\mathsf{a}}\rangle$ by setting $p^0 = (\lambda_x^0)^{\mathsf{a}}, p^1 = p^2 = 0$.

2. Non-interactively set up $\langle q\rangle = \langle(\lambda_x^1)^{\mathsf{a}}\rangle$ by setting $q^1 = (\lambda_x^1)^{\mathsf{a}}, q^0 = q^2 = 0$.

3. Non-interactively set up $\langle r\rangle = \langle(\lambda_x^0)^{\mathsf{a}}\rangle$ by setting $r^2 = (\lambda_x^2)^{\mathsf{a}}, r^0 = r^1 = 0$.

4. Invoke $\mathcal{F}_{\mathsf{MultPre}}(\langle p\rangle, \langle q\rangle)$ to obtain $\langle pq\rangle$.

5. Compute $\langle s\rangle = \langle(\lambda_x^0 \oplus \lambda_x^1)^{\mathsf{a}}\rangle = \langle p\rangle + \langle q\rangle - 2\langle pq\rangle$.

6. Invoke $\mathcal{F}_{\mathsf{MultPre}}(\langle s\rangle, \langle c\rangle)$ to obtain $\langle sc\rangle$.

7. Compute $\langle t\rangle = \langle(\lambda_x^0 \oplus \lambda_x^1 \oplus \lambda_x^2)^{\mathsf{a}}\rangle = \langle s\rangle + \langle c\rangle - 2\langle sc\rangle$.

8. Non-interactively generate $[\lambda_z]$ by $\mathsf{S}_0, \mathsf{S}_1$ sampling random $\lambda_z^1 \in \mathbb{Z}_{2^\ell}$, $\mathsf{S}_0, \mathsf{S}_2$ sampling random $\lambda_z^2 \in \mathbb{Z}_{2^\ell}$.

Online:

1. Locally compute the following (S_0 delays the step until the final verification):

- $\mathsf{S}_0, \mathsf{S}_1$: $u^0 = (1 - 2(\mathsf{m}_x)^{\mathsf{a}}) \cdot t^0 - \lambda_z^0$
- $\mathsf{S}_1, \mathsf{S}_2$: $u^1 = (1 - 2(\mathsf{m}_x)^{\mathsf{a}}) \cdot t^1 - \lambda_z^1$
- $\mathsf{S}_2, \mathsf{S}_0$: $u^2 = (1 - 2(\mathsf{m}_x)^{\mathsf{a}}) \cdot t^2 - \lambda_z^2$

2. S_2 sends u^2 to S_1 and S_1 sends u^0 to S_2.

3. $\mathsf{S}_1, \mathsf{S}_2$ set $\mathsf{m}_z = (\mathsf{m}_x)^{\mathsf{a}} + u^0 + u^1 + u^2$.

Final Verification (batched over all multiplications):

1. S_1 sends u^1 and S_2 sends $\mathsf{H}(u^1)$ to S_0 that then checks for consistency.

2. S_0 computes u^0, u^2 and reconstructs $\mathsf{m}_z = (\mathsf{m}_x)^{\mathsf{a}} + u^0 + u^1 + u^2$.

3. S_0 sends $\mathcal{H}(u^2)$ to S_1 and $\mathcal{H}(u^0)$ to S_2 that then check consistency to the values received in online step 2.

Figure 30: SWIFT: Bit to arithmetic conversion [75].

B.7. Binary to Arithmetic Conversion

For our comparison to ELSA [69] we require a conversion of an integer decomposed into its bits into an arithmetic sharing. This is only needed for AUXILIATOR.

Protocol $\Pi_{\mathsf{B2A}}(\langle x_0\rangle^{\mathbf{B}}, ..., \langle x_{k-1}\rangle^{\mathbf{B}})$

Input: $\langle\cdot\rangle^{\mathbf{B}}$-shares of $x_{k-1}, ..., x_0 \in \mathbb{Z}_2$ for some $k \in \mathbb{N}$.

Output: $\langle\cdot\rangle$-shares of $z \in \mathbb{Z}_{2^\ell}$ where z as binary encoding corresponds to $x_{k-1}, ..., x_0$, i.e., $z = \sum_{i=0}^{k-1} 2^i x_i$.

Setup:

1. Non-interactively set up $[p_i] = [(\lambda_{x_i}^1)^{\mathsf{a}}]$ by setting $p_i^1 = (\lambda_{x_i}^1)^{\mathsf{a}}, p_i^2 = 0$ for $0 \le i < k$.

2. Non-interactively set up $[q_i] = [(\lambda_{x_i}^2)^{\mathsf{a}}]$ by setting $p_i^1 = 0, p_i^2 = (\lambda_{x_i}^2)^{\mathsf{a}}$ for $0 \le i < k$.

3. Invoke $\mathcal{F}_{\mathsf{MultPre}}([p_i], [q_i])$ to obtain $[p_i q_i]$ for $0 \le i < k$.

4. Compute $[r_i] = [(\lambda_{x_i}^1 \oplus \lambda_{x_i}^2)^{\mathsf{a}}] = [p_i] + [q_i] - 2[p_i q_i]$ for $0 \le i < k$.

5. Non-interactively generate $[\lambda_z]$ by $\mathsf{S}_0, \mathsf{S}_1$ sampling random $\lambda_z^1 \in \mathbb{Z}_{2^\ell}$, $\mathsf{S}_0, \mathsf{S}_2$ sampling random $\lambda_z^2 \in \mathbb{Z}_{2^\ell}$.

Online:

1. Locally compute the following:

- S_1: $s^1 = \sum_{i=0}^{k-1} \left(2^i \cdot (1 - 2(\mathsf{m}_{x_i})^{\mathsf{a}}) \cdot r_i^1\right) - \lambda_z^1$.
- S_2: $s^2 = \sum_{i=0}^{k-1} \left(2^i \cdot (1 - 2(\mathsf{m}_{x_i})^{\mathsf{a}}) \cdot r_i^2\right) - \lambda_z^2$.

2. S_1 sends s^1 to S_2 and S_2 sends s^2 to S_1.

3. Compute $\mathsf{m}_z = s^1 + s^2 + \sum_{i=0}^{k-1} \left(2^i \cdot (\mathsf{m}_{x_i})^{\mathsf{a}}\right)$.

Figure 31: AUXILIATOR: Binary to arithmetic conversion.