LightShark: Actively Secure Machine-Learning Inference Based on Lightweight Authenticated Distributed Comparison Function

Chenkai Zeng
Wuhan University
Wuhan, Hubei, China
zckwhu@whu.edu.cn

Qi Feng*
Wuhan University
Wuhan, Hubei, China
fengqi.whu@whu.edu.cn

Debiao He*
Wuhan University
Wuhan, Hubei, China
hedebiao@whu.edu.cn

Min Luo
Wuhan University
Wuhan, Hubei, China
mluo@whu.edu.cn

Abstract

Recently, Shark (S&P'25) considered the problem of actively two-party secure machine learning inference using an authenticated distributed comparison function (DCF). This is the state-of-the-art work in this setting. On the other hand, Grotto (CCS'23) built a variant DCF with the key size half that of classic DCF. Unfortunately, as Shark states, *it is not known how to extend Grotto to the malicious setting*. In this paper, we present the first actively secure Grotto-style DCF scheme. Our authenticated DCF is deliberately designed on the correlated GGM tree and maintains the key-size advantage of semi-honest Grotto. We further implement an actively secure ML inference framework, named LightShark, which supports efficient primitives (e.g., ReLU, spline, and truncation) and ML models (e.g., VGG-16, GPT, BERT). Compared with Shark, our LightShark outperforms by 1.49× ∼ 2.69× and reduces communication costs by 66.7% for Bert-base inference. Surprisingly, for larger LLM models, the experimental evaluation demonstrates that our framework works vastly well.

CCS Concepts

• **Theory of computation → Cryptographic protocols**; • **Security and privacy → Privacy-preserving protocols**.

Keywords

Multi-Party Computation, Secure Machine-Learning Inference, Active Security, Function Secret Sharing, Half-Tree

ACM Reference Format:
Chenkai Zeng, Qi Feng, Debiao He, and Min Luo. 2026. LightShark: Actively Secure Machine-Learning Inference Based on Lightweight Authenticated Distributed Comparison Function. In *Proceedings of the 2026 ACM SIGSAC Conference on Computer and Communications Security (CCS '26), November 15–19, 2026, The Hague, Netherlands.* ACM, New York, NY, USA, 26 pages. https://doi.org/10.1145/3830454.3832717

*Corresponding author



This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.
CCS '26, The Hague, Netherlands.
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2871-6/2026/11
https://doi.org/10.1145/3830454.3832717

1 Introduction

Secure two-party computation (2PC) is a significant cryptographic paradigm for privacy preservation in machine learning (ML). It enables two parties, typically P_0 and P_1-to jointly perform ML inference while maintaining the privacy of sensitive data and proprietary ML model. Over the last decade, there have been many solutions proposed to realize secure ML inference in 2PC setting [20, 27, 35, 42, 43]. However, most of these works achieve semi-honest security – assuming the parties follow the protocol honestly. This assumption would be catastrophic on data confidentiality if either party acts maliciously (e.g., modifying, replaying, injecting a poisoned input, etc, which frequently occur in real-world systems). This motivates the need for *actively* secure 2PC ML inference.

Distributed comparison function (DCF) [6] exhibits strong performance in 2PC ML inference [24–26, 28, 47], due to its *lower communication and computation costs* for comparison operation (the backbone of activation function, e.g., ReLU and being heavily used in ML inference). Specifically, two parties P_0 and P_1 securely perform comparison operation, i.e., $\mathbf{1}\{x < \beta\}$ for given secret input x and secret variant β, with the cost of one communication round and $O(n)$ AES calls of computation costs, where n is the bit length of x and β. Prior to DCF-approach, garbled circuits (GC)[51] requires $O(n)$ AND evaluation (each of them takes four AES calls), while oblivious transfer [43] requires $O(n)$ communication rounds.

However, a significant challenge arises from the *actively security for the DCF evaluation*, i.e., authenticating the computation with input x and maintaining the authenticated state on the comparison result **1** or **0**. One straightforward solution is Boyle et al. [6]' authentication paradigm for DCF, which relies on MAC batching of authenticated secret sharing [16]. Particularly in their protocol, for every DCF protocol that realizes a comparison function, two DCFs are created: the original DCF and an associated DCF with the fixed authentication key, doubling the computational and communication costs. Shark [24] embedded information-theoretic MAC (IT-MACs) in the result of original DCF (denoted as $\{y, y \times \Delta\}$, $y \in \{0, 1\}$ with authentication key Δ fixed throughout the protocol). Thus, parties can perform an IT-MAC batch checking for values [40] and eliminate the need for a second DCF. The main concern of Shark [24] is its key size, which affects communication costs. For instance, given an input x of bit-length n, the key size per party is roughly $n \times (\lambda + \kappa)$ bits, where κ and λ are security parameters.

Grotto [46] built a line of research using the distributed point function (DPF) for performing efficient comparisons in the semi-honest security. Theoretically, DPF is an adjacent primitive of DCF that is specialized for performing point function (i.e., $\mathbf{1}\{x = \alpha\}$) and has a key size half that of DCF. They construct a novel *prefix-parity tree* and leverage this data structure to do with a DPF what others require a DCF to do. Therefore, their DCF-variant offers a key size per party of roughly $n \times \lambda$ bits with the security parameter λ. Storrier et al. [46] have shown that replacing DCF with DPF can lead to vastly improved performance. However, they leave open the problem of *achieving actively secure against malicious adversaries for their DCF-variant and enabling actively secure ML inference*. The core challenge is that the authentication methods proposed in Boyle-DCF [6] and Shark-DCF [24] add an authentication key at the final output, whereas Grotto's output is a linear combination of on-path random values, so deriving a fixed authentication key is negligible in probability.

1.1 Our Contribution

In this work, we address the above challenges and build an efficient authenticated DCF for actively secure computation of ML inference. Our new efficient protocols enable actively secure implementations of Grotto-style DCF construction that complete an LLM inference under a minute! We make three main contributions:

- Firstly, we construct the first authenticated Grotto-style DCF scheme, which ensures the correctness of both the path derivation and all values along the path. Our malicious protocol's key size retains the advantage of semi-honest Grotto, and approximately 66.15% reduction compared with Shark [24] (see Table 1). We prove the security under the simulation-based model.
- Secondly, we provide new optimized actively secure protocols for DReLU and spline. Together with deliberate design that embeds a correlated GGM tree, we simplify the batch checking on boolean shares when authenticating non-linear layers, such as ReLU, Sigmoid, and Reciprocal.
- Finally, we implement a framework- LightShark -that supports secure evaluation of many ML models, e.g., AlexNet [34], VGG-16 [45], GPT [9], BERT [17], etc. Compared to the closely related Shark [24], our protocols achieve 1.49× ∼ 2.69× speedup and reduce communication costs by 66.7% for LLMs, thereby enabling actively secure inference for larger models (e.g., BERT-large) under limited sources. To our surprise, the malicious security just incurs an average increase of 21% in communication costs compared with SOTA semi-honest secure LLMs inference.

1.2 Technique Overview

Our key insight is that Grotto's [46] non-linear protocols (e.g., comparison, spline, etc) can be compatibly authenticated by the correlated structure of Half-Tree [23] with a few costs. Our techniques are summarized as follows.

Step 1: New perspective of Grotto's [46] method. Grotto [46] embeds parity-segment trees in DCF-variant to efficiently perform

Table 1: The Key Size of Authenticated DCFs (*bits*)

Schemes	Ours	Shark-DCF [24]	Boyle-DCF [6]	Grotto-DCF [25]
Key size	$(n+1)\cdot\lambda$	$(n+1)(\lambda+\kappa+1)+2n$	$(n+1)(\lambda+n+s)+2n$	$(n+1)\cdot\lambda+2\lambda-2$
AES calls	n	$2n$	$4n$	$2n$

⋆ $\lambda = 128$, $\kappa = 40$, $s = 64$, $n = 64$ denote the computational security parameter for DCF, statistical security parameters for SPDZ-like IT-MACs [11, 32], and the input length, respectively.
⋆ Grotto's [46] DCF achieves semi-honest security.

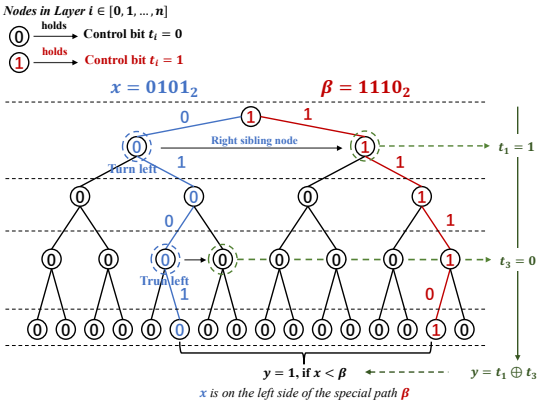

Figure 1: An Example of Two Binary Numbers Comparison

"compare-then-select" operations, which are backbone of most non-linear functions in ML (e.g., $\mathbf{ReLU}(x) = \mathbf{max}(x, 0)$). Specifically, in Grotto, each key defines a GGM-style binary tree with 2^n leaves, where the leaves are labeled by input $x \in \{0,1\}^n$. The evaluation path of x refers to a path from the root to a leaf labeled by x, and special evaluation path refers to the evaluation path of special input β. They observe that if the prefix-parity of leaves from x-th position does not include the β-th position, then it can be considered that $x < \beta$. Our new perspective is that, when evaluating with input x, if the path of x is on the left side of the special path β, then there will definitely exist a right sibling node of x that lies on the special path. Take $x = 0101_2$ and $\beta = 1110_2$ as an example, see Figure 1, their evaluation paths branches at the first bit (i.e., the first layer in the GGM-style binary tree), then x_1's right sibling node must be β_1. In all other cases, when x proceeds to the right, it has no right sibling node; when x proceeds to the left, it will have a non-special right sibling node. Back to the construction of DPF, each node will be labeled by a string of length $\lambda+1$, consisting of a seed $s \in \{0,1\}^\lambda$ and a control bit t that maintain the invariant of $t = 1$ for nodes on the special path and $t = 0$ for other nodes. Therefore, if we XORing all the right sibling nodes where x's evaluation path turns left (two green nodes at the 1st and the 3rd layers in Figure 1), i.e., $t_1 \oplus t_3 = 1$, the result equals to the comparison result between x and β. This DCF-variant has a key size of nearly 50% reduction than Shark [24].

Table 2: Comparison of Secure ML Inference Frameworks

	Works	Active Security	Compact Keys	Nonlinear Layer Basic Methods				Supported Models			
				SS+OT	HE	DCF	DPF	GPT-2 (Base)	BERT (Base)	BERT (Large)	Restricted
2PC	BumbleBee [29]	✗	✗	●	●	❍	❍	●	●	●	❍
	NEXUS [53]	✗	✗	❍	●	❍	❍	❍	●	❍	❍
	SHAFT [30]	✗	✗	●	❍	❍	❍	●	●	●	❍
	BLB [50]	✗	✗	●	●	❍	❍	●	●	●	❍
3PC	Privformer [3]	✓	✗	●	❍	❍	❍	❍	❍	❍	●
	Mosformer [10]	✓	✗	❍	❍	❍	●	●	●	❍	❍
MPC	MD-ML [52]	✓	✗	●	❍	❍	❍	❍	❍	❍	●
(2+1)-PC	Pika [47]	✓	✗	❍	❍	❍	●	❍	❍	❍	●
	Orca [28]	✗	✗	❍	❍	●	❍	❍	❍	❍	●
	SIGMA [25]	✗	✓	❍	❍	❍	●	●	●	●	❍
	Grotto [46]	✗	✓	❍	❍	❍	●	●	●	●	❍
	Shark [24]	✓	✗	❍	❍	●	●	❍	●	❍	❍
	LightShark (Ours)	✓	✓	❍	❍	❍	●	●	●	●	❍

⋆ ✓ indicates that the framework satisfies the corresponding property, whereas ✗ indicates that it does not.
⋆ ● indicates support for a technique or functionality.
⋆ **HE**: Homomorphic encryption, **SS+OT**: Secret sharing with oblivious transfer, **DCF**: distributed comparison functions, **DPF**: distributed point functions, **Restricted** indicates that this framework only supports specific models (like DNN, CNN) or requires fine-tuning.
⋆ Grotto appears to support building blocks typically required by LLMs, although no concrete models are specified.

Step 2: Authenticated DCF without additional keysize. The question is how to authenticate the correct evaluation of x for Grotto-style DCF. We need to answer two aspects: 1) two parties evaluate the identical x's path; 2) the result is correct within $\{0, 1\}$.

Firstly, we try Shark's authentication method. Gupta et al. [24] concatenate an authentication key Δ of κ-bit length in the leaf nodes, i.e., $\{y, y \times \Delta\}$ for $y \in \{0, 1\}$. This method works as Shark's DCF has n additional values $V_{CW} \in \{0, 1\}^{\kappa+1}$ in each key to ensure that all nodes located on the left side of β have $y = 1$ (See Definition 1 for details). However, in the Grotto-style DCF, this method guarantees that the result will be $\{1, 1 \times \Delta\}$ (if $x = \beta$) or $\{0, 0 \times \Delta\}$ (if $x \neq \beta$), which just addresses the "identical x" question, yet cannot resolve the correctness. The same issue arises when applying Boyle-style DCF's [6] authentication method. The core challenge is that Grotto-style DCF's output is aggregated by all values along the evaluation path, so verifying one special path is insufficient. Therefore, we attempt to embed the authentication key in each layer. Concatenating a κ-bit Δ on each right sibling node is not applicable, since it would increase the control bit t at each layer from 1 bit to $1 + \kappa$ bits. Another part of each node's label is the seed s, which is randomly derived by a pseudorandom generator and is negligible in probability to be a fixed key. Fortunately, Half-Tree [23] introduces a correlated GGM (cGGM) tree that ensures all same-layer nodes sum up to a fixed random offset. We prove that it is still actively secure when setting the authentication key Δ as the fixed offset. Leveraging this observation, we can authenticate Grotto-style DCF nodes using IT-MACs, without any additional increase in key size. More specifically, at each level with seed s and control bit t, we define all label values on the special path as $\{s, t\} = \Delta$ and $\{s, t\} = 0$ otherwise, with the global authenticated key $\Delta \in \{0, 1\}^{\lambda}$. At this time, we can construct authenticated secret shares from cGGM nodes freely by unifying the global key of IT-MACs [32] with the cGGM offset. The final key size of our authenticated DCF is shown in Table 1, reducing key size from $(n + 1)(\lambda + \kappa + 1) + 2n$ bits to $(n + 1) \cdot \lambda$ bits compared with Shark's [24], achieving up to 66.15% reduction and halving the AES calls in our implementations.

Step 3: Actively secure ML inference. Given the proposed authenticated DCF, we can construct an actively secure ML inference framework using IT-MACs. The classic method for secure ML is the mixed-mode framework, i.e., splitting the computation of ML into several blocks and each block is executed in one of the two worlds: i) arithmetic (e.g., matrix multiplication), ii) boolean (e.g., activation function). This implementation is a trade-off between communication and computation costs. In this paper, we follow the mixed-mode framework, but apply secret sharing (SS) within the arithmetic world and Grotto-style DCF within the boolean world. The direct observation is that computations in the arithmetic world are authenticated by SPD$\mathbb{Z}_{2^k}$ [11] while boolean operations are verified by our authenticated DCF. Kindly, our authenticated DCF has unified the global key of IT-MACs with the cGGM offset, thus we can directly define the boolean shares authenticated by this global key, and efficiently perform batch checking [32] on boolean computation blocks. We have proven that the probability that a malicious adversary evades the verification is identical to that in [32]. The final step is to check consistency when converting secret values from two worlds. The core challenge lies in the fact that values in boolean world are authenticated over $\mathbb{F}_{2^\kappa}$, which cannot be directly converted into authenticated values over $\mathbb{Z}_{2^{n+s}}$. We adopt the concept of daBits [1, 44] for authenticated conversion, which provides a random bit that is authenticated secretly-shared in both domains and keeps the secret bit identical. This is a classic conversion method, but we slightly modify it to match our authenticated secret sharing definition. The experimental evaluation demonstrates that our method works vastly well in ML inference tasks.

1.3 Related Works.

A summary of related works is shown in Table 2. Boyle et al. [7] proposed a concept of function secret sharing (FSS) for mixed-mode MPC in the preprocessing model. The FSS-based approach to MPC can support many kinds of non-arithmetic operations with the same online communication and round complexity as pure arithmetic computations, and while only making use of symmetric cryptography. Therefore, FSS-based 2PC has been widely used to construct several secure ML protocols [24–26, 28, 47]. There are two typical instantiations of FSS, including distributed point functions (DPF) [8] and distributed comparison functions (DCF) [6]. Pika [47] combined DPF with full-domain evaluation to support arbitrary functions. But Pika's method only support "short" inputs due to its exponential computation cost. LLAMA [26] and Ocra [28] use DCF to eliminate the costly full-domain evaluations for ML tasks. However, the classic DCF proposed by Boyle et al. [6] embedded the output (payload) into an additional correction word at each layer, whereas DPF does not. As a result, the key size of DCF is nearly 2× compared to that of DPF. Consequently, the key size becomes one of the bottlenecks for DCF-based schemes, particularly when applying parallelism on GPUs. Grotto [46] provides a new perspective on DCF's structure, making it possible to support nonlinear functions without additional key size costs. Grotto [46] represents a **milestone** in FSS-based schemes with a smaller key size and lower computation costs. SIGMA [25] leverages Grotto [46]'s structure and achieves highly efficient GPT inference using GPU optimizations.

However, most of these FSS-based 2PC schemes consider semi-honest security, where the adversary follows the prescribed protocol and tries to break privacy by analyzing the transcripts. Active security (a.k.a, parties are corrupted maliciously and adversary can deviate arbitrarily) is the most desirable, but also the most challenging to achieve. The recent work Shark [24] introduces interactive FSS, which combines DPF [8] and DCF [6] with information-theoretic message authentication codes (IT-MACs), to enable actively secure inference in $\mathcal{F}_{\text{dealer}}$-model with fewer rounds and lower communication. However, Shark [24] relies on the heavyweight DCF [6] to achieve actively secure comparison, resulting in a large size of the prepared materials, making it difficult to adapt to larger models.

As stated in Shark [24], *"However, it is not known how to extend Grotto to the malicious setting, and in particular the technique of Boyle et al. [6] does not apply"*. Therefore, the "Grotto moment" for actively secure FSS-based schemes has not yet arrived. This work presents the first actively secure Grotto-style FSS construction based on Half-Tree's [23] DPF.

2 Preliminaries

The notations used in this paper are summarized in Table 3.

Table 3: Summary of Notations

Notation	Description
$[n]$	Set $\{1, 2, \ldots, n\}$ for $n \in \mathbb{N}$.
s, κ	Security parameters satisfing $s - \log s \geq \kappa$.
$\mathbb{Z}_{2^n}, \mathbb{Z}_{2^{n+s}}$	Integer ring modulo 2^n and 2^{n+s}.
$\mathbb{F}_2, \mathbb{F}_{2^\kappa}$	Binary field and its extension field.
x_i	The i-th bit of an integer x.
$\vec{e}, e[i]$	An array $\vec{e}$ and its i-th element.
$\mathsf{lsb}(x)$	Least significant bit of x.
$\mathsf{msb}(x)$	Most significant bit of x.
$\mathsf{hb}(x, t)$	The high-order t bits of x
$\|$	Bit-string concatenation operator.
$P_0 \& P_1$	Two servers.
$[[x]]^{\mathsf{A}}$	Additive shares of x over $\mathbb{Z}_{2^n}$
$[[x]]^{\mathsf{B}}$	Additive shares of x over $\mathbb{F}_2$ or $\mathbb{F}_{2^\kappa}$.
$\langle x \rangle^{\mathsf{A}}$	Arithmetic authenticated sharing of $x \in \mathbb{Z}_{2^n}$.
$\langle x \rangle^{\mathsf{B}}$	Boolean authenticated sharing of $x \in \mathbb{F}_2$ or $\mathbb{F}_{2^\kappa}$.
α	Arithmetic authenticated key.
Δ	Boolean authenticated key.
δ_x	Mask value for x.

2.1 System Overview

2.1.1 Design Goals. We consider the malicious security and aim to achieve the following design goals: **(1) Actively Secure.** The proposed protocols can defend against adversaries that arbitrarily deviate from the protocol; **(2) Lightweight.** Key size, communication, and computation overhead remain manageable, with modest malicious-security cost; **(3) ML-Compatible.** Designed to be compatible with ML across secret-sharing, authentication, and building blocks, and to support multiple large-scale models.

2.1.2 Architecture and Security Model. We consider the special secure two-party computation with a trusted dealer, as in Shark [24]. This architecture is valuable in outsourced computation applications. Generally, it involves two roles:

- **Servers.** Let P_0 and P_1 denote two servers that hold the secret-shared inputs of users and model weights, and jointly perform secure ML inference. Assume two parties communicate over a synchronous network with authenticated channels.
- **Dealer.** To simplify the presentation, we assume a trusted dealer to generate input-independent prepared materials for both servers. This assumption is reasonable and has been adopted in prior works [24, 25, 28, 46, 49]. Formally, the dealer is modeled as an ideal functionality $\mathcal{F}_{\text{dealer}}$ and could be instantiated using an external trusted hardware or secure computation protocols [6, 11, 19]. We also provide a 2PC instantiation for $\mathcal{F}_{\text{dealer}}$ in Appendix E.

This paper considers the security model for security with abort in the presence of a static, malicious adversary. Let $\mathcal{A}$ be the probabilistic polynomial time (PPT) malicious adversary that corrupts one of the parties P_0 and P_1, and can arbitrarily deviate from the protocol. We prove our proposed protocols under simulation-based security in the $\mathcal{F}_{\text{dealer}}$-hybrid model. The formal proofs are presented in Appendix D.

2.2 Authenticated Secret Sharings

In this section, we define our authenticated secret sharing over $\{\mathbb{Z}_{2^{n+s}}, \mathbb{F}_2, \mathbb{F}_{2^\kappa}\}$. We denote $\langle x \rangle^{\mathsf{A}}$ as the *Arithmetic Authenticated*

Sharing from SPD$\mathbb{Z}_{2^k}$ [11] over ring $\mathbb{Z}_{2^{n+s}}$, and $\langle x\rangle^{\mathsf{B}}$ as the *Boolean Authenticated Sharing* from MASCOT [32] over fields $\mathbb{F}_2$ and $\mathbb{F}_{2^\kappa}$.

2.2.1 Arithmetic Authenticated Sharing. An integer $x \in \mathbb{Z}_{2^n}$ is authenticated by a SPD$\mathbb{Z}_{2^k}$-like IT-MACs [11], defined as

$$\langle x\rangle^{\mathsf{A}} := \{[[x]]^{\mathsf{A}}, [[m_x]]^{\mathsf{A}}, [[\alpha]]^{\mathsf{A}}\} \in \mathbb{Z}_{2^{n+s}} \times \mathbb{Z}_{2^{n+s}} \times \mathbb{Z}_{2^s}$$

with security parameters s and κ, s.t., $s - \log s \geq \kappa$. $[[x]]^{\mathsf{A}}$ denotes additive shares of x such that $x = [[x]]_0^{\mathsf{A}} + [[x]]_1^{\mathsf{A}} \mod 2^n$ (So as to the authentication tag $[[m_x]]^{\mathsf{A}}$ and global key $[[\alpha]]^{\mathsf{A}}$). Each server $P_b, b \in \{0,1\}$ holds one share of them, i.e., $[[x]]_b^{\mathsf{A}}$, $[[m_x]]_b^{\mathsf{A}}$ and $[[\alpha]]_b^{\mathsf{A}}$. A key point is $\sum_{b=0}^{1}[[m_x]]_b^{\mathsf{A}} = (\sum_{b=0}^{1}[[x]]_b^{\mathsf{A}}) \cdot (\sum_{b=0}^{1}[[\alpha]]_b^{\mathsf{A}}) \mod 2^{n+s}$. All arithmetic authenticated sharing values exhibit additive homomorphism. For instance, given coefficients $c, c_1, \ldots, c_t$ and authenticated values $\langle x_1\rangle^{\mathsf{A}}, \ldots, \langle x_t\rangle^{\mathsf{A}}$, the parties can locally compute $\langle y\rangle^{\mathsf{A}} = \sum_{i=1}^{t} c_i \cdot \langle x_i\rangle^{\mathsf{A}} + c$. Below, we define two macros that will be used throughout this paper.

ShareA. Let $\langle r\rangle^{\mathsf{A}} \leftarrow \mathsf{ShareA}(r)$ be the generation of an authenticated random value $r \in \mathbb{Z}_{2^n}$. The protocol simply invokes $\mathcal{F}_{\mathsf{dealer}}$ and each party P_b receives $\langle r\rangle_b^{\mathsf{A}}$ from $\mathcal{F}_{\mathsf{dealer}}$.

OpenA. To reconstruct and check $\langle r\rangle^{\mathsf{A}}$, the servers exchange $[[r]]^{\mathsf{A}}$ to compute $r = [[r]]_0^{\mathsf{A}} + [[r]]_1^{\mathsf{A}} \mod 2^n$. Each server $P_b, b \in \{0,1\}$ commits to $z_b = [[m_r]]_b^{\mathsf{A}} - ([[r]]_0^{\mathsf{A}} + [[r]]_1^{\mathsf{A}}) \cdot [[\alpha]]_b^{\mathsf{A}} \mod 2^{n+s}$, and then decommits to check if $\sum_{b=0}^{1} z_b = 0 \mod 2^{n+s}$ holds.

We simplify the SingleCheck procedure of SPD$\mathbb{Z}_{2^k}$ [11], as the authenticated values that need to be opened in our protocol have been randomized. With a random linear combination, the SingleCheck is easily extended to BatchCheck as in SPD$\mathbb{Z}_{2^k}$ [11]. In practice, the protocol executes only the first step of OpenA, requiring $n+s$ bits of communication per party, and performs a BatchCheck of all arithmetic computation blocks. This has no impact on security and is a recommended practice in [11].

2.2.2 Boolean Authenticated Sharing. A bit $x \in \mathbb{F}_2$ will be authenticated by the SPDZ-like IT-MACs [32], defined as

$$\langle x\rangle^{\mathsf{B}} := \{[[x]]^{\mathsf{B}}, [[m_x]]^{\mathsf{B}}, [[\Delta]]^{\mathsf{B}}\} \in \mathbb{F}_2 \times \mathbb{F}_{2^\kappa} \times \mathbb{F}_{2^\kappa}$$

with security parameter κ, satisfying $m_x = x \cdot \Delta \in \mathbb{F}_{2^\kappa}$. Similarly, each server holds the additive shares of bit value x, authentication tag m_x, global key Δ, s.t., $[[x]]_0^{\mathsf{B}} \oplus [[x]]_1^{\mathsf{B}} = x$, $[[m_x]]_0^{\mathsf{B}} \oplus [[m_x]]_1^{\mathsf{B}} = m_x$ and $[[\Delta]]_0^{\mathsf{B}} \oplus [[\Delta]]_1^{\mathsf{B}} = \Delta$. The additive homomorphism property also holds over boolean authenticated sharing, instead replacing addition/multiplication with XOR/AND operations, respectively. We also define two macros that are used in the boolean world.

ShareB. Let $\langle r\rangle^{\mathsf{B}} \leftarrow \mathsf{ShareB}(r)$ be the generation of an authenticated random bit $r \in \mathbb{F}_2$. The protocol simply invokes $\mathcal{F}_{\mathsf{dealer}}$ and each party P_b receives $\langle r\rangle_b^{\mathsf{B}}$ from $\mathcal{F}_{\mathsf{dealer}}$.

OpenB. We omit the details of the reconstruction and checking procedures, and instead replace all operations with those over $\mathbb{F}_{2^\kappa}$. Similarly, we execute the reconstruction step of OpenB and perform a BatchCheck of all boolean computation blocks in practice, as proven in MASCOT [32].

2.3 Classic DPF and Non-Linear Evaluation

The point function with 1-bit output is defined as $PF(x) = 1$ if $x = \beta$ and 0 otherwise over the input domain size $N = 2^n$, and can be viewed as a selection vector $\vec{e}$ with $\vec{e}[\beta] = 1$ and $\vec{e}[i] = 0, i \neq \beta$. Distributed point function (DPF) is a secret shared point function among multiple parties that enables local evaluation on the shares of $PF(x)$ for any given input x. Boyle et al. [8] present a classic two-party DPF construction that encodes the selection vector as a binary tree. Suppose PRG : $\{0,1\}^\lambda \rightarrow \{0,1\}^{2\lambda+2}$ is a pseudorandom generator. Each node in a tree is labeled by a string $(s,t) \in \{0,1\}^{\lambda+1}$, consisting of a PRG seed s and a control bit t, where the label of each node is fully determined by the label of its parent. Generally, the two DPF keys evaluate two GGM-style binary trees: labels on the special path (the β path) are random and independent, whereas labels outside the special path are identical. Overall, DPF achieves keys size of $(n+1)\cdot\lambda$, and consumes $O(n \cdot |\vec{e}|)$ PRG (instantiated by AES in practice) to generate a selection vector.

$$f(x) \approx F(x) := \begin{cases} a_{\{d,1\}}x^d + \cdots + a_{\{1,1\}}x + a_{\{0,1\}} \ if\ x \in S_1, \\ a_{\{d,2\}}x^d + \cdots + a_{\{1,2\}}x + a_{\{0,2\}} \ if\ x \in S_2, \\ \vdots \\ a_{\{d,T\}}x^d + \cdots + a_{\{1,T\}}x + a_{\{0,T\}} \ if\ x \in S_T. \end{cases}$$

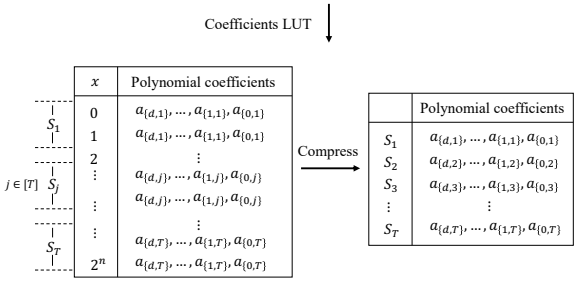

Figure 2: Coefficient LUT for Piecewise-Polynomial Function

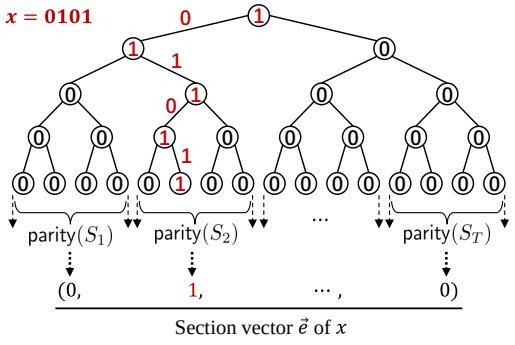

Figure 3: Selection Vector Generation in Grotto [46]

Grotto [46] leverages DPF to achieve efficient non-linear functions using spline evaluation. For example, let piecewise-polynomial function $F(x)$ approximates a non-linear function $f(x)$. The coefficients vectors $\{a_{d,j}, \ldots, a_{1,j}, a_{0,j}\}$ for all $j \in [T]$ define T segments in $F(x)$, and S_j is the j-th continuous interval of $F(x)$ and d is the polynomial degree. $F(x)$ is well-defined in Figure 2 based on a coefficient look-up table (LUT). Grotto [46] adopts DPF to determine the interval to which input x belongs. They observed that Boyle et

al.'s [8] DPF implicitly embeds a *parity-segment tree*—a data structure that supports efficient parity queries over bitstring intervals. Specifically, given a bit decomposition of the input $x = x_1 x_2 \dots x_n$, its subinterval $x[a, b)$ can be interpreted as a segment index. The parity of this interval, computed as $\text{parity}(x[a, b)) = \bigoplus_{i=a}^{b-1} x_i$, determines whether the input falls into a given segment S_i. Finally, as shown in Figure 3, parties can locally evaluate to a secret share of selection vector $\vec{e}$, identifying which polynomial segment x lies in. Finally, parties can use $\vec{e}$ to fetch the appropriate coefficient vector, perform oblivious polynomial evaluation, and aggregate to the result of $F(x)$. The key size is on par with classic DPF [8], yet the computation costs to generate a selection vector are $O(n)$.

2.4 Classic DCF Construction

Distributed Comparison Function (DCF), an adjacent primitive of DPF, specializes in performing a comparison function among multiple parties. This section reviews the definition of classic DCF originally presented by Boyle et al. [6].

Definition 1 (Distributed Comparison Function (DCF)). *Given the security parameter $\lambda \in \mathbb{N}$, the upper bound range $N \in \mathbb{N}$ of the comparison function input, and a distinguished input $\beta \in [N]$ and Abelian group of output $\mathbb{G}$, consider the family of comparison functions*

$$\mathcal{G}^{<} = \left\{ f^{<}_{\beta,1} : \mathbb{Z}_N \to \mathbb{G} \right\}_{\beta \in \mathbb{Z}_N}$$

*where $f^{<}_{\beta,1}(x) = \mathbf{1}\{x < \beta\}$. A (2,2)-DCF scheme for the function family $\mathcal{G}^{<}$ includes two PPT algorithms (*Gen, Eval*), satisfying*

- Correctness. *If the DCF key is $(k_0, k_1) \leftarrow \text{Gen}(1^\lambda, \beta)$, for all $j \in [0, N)$, then*

$$\text{Eval}(0, k_0, j) \oplus \text{Eval}(1, k_1, j) := \begin{cases} 1 & \text{if } j < \beta, \\ 0 & \text{otherwise.} \end{cases}$$

- Privacy. *There exists a probabilistic polynomial-time (PPT) simulator $\mathcal{S}$ such that, for any given N, $\beta \in [0, N)$, and bit $b \in \{0, 1\}$, the distribution ensembles*

$$k_b \leftarrow \mathcal{S}\{1^\lambda, b\}_{\lambda \in \mathbb{N}}$$

and

$$\{k_b | (k_0, k_1) \leftarrow \text{Gen}(1^\lambda, \beta), b \in \{0, 1\}\}_{\lambda \in \mathbb{N}}$$

are computationally indistinguishable.

Unlike DPF, each node of DCF'tree is labeled by a 2λ+1-bits string (s, v, t), consisting of the PRG seed s, control bit t, and an output group element v of κ-bit length. The construction ensures that an aggregate of v over all nodes leading to an input x is exactly equal to $f^{<}_{\beta,1}(x)$. Therefore, the key size of DCF, which is $(n+1)(\lambda+\kappa+1)+2n$ bits, is roughly twice that of DPF. Moreover, DCF uses a PRG to generate $4\lambda + 2$-bits strings as labels of two child nodes, and thus incurs more AES calls when instantiated than DPF.

2.5 DPF from Correlated GGM Construction

The correlated GGM (cGGM) tree, introduced by Half-Tree [23], has an invariant that all same-level nodes sum up to the same global offset. We briefly review the cGGM-style DPF in Half-Tree [23]. Here, two parties share an n-level GGM-style binary tree where (i) the root and the first $n-1$ special-path nodes are labeled by a global offset $R \in \{0,1\}^\lambda$ with $\text{lsb}(R) = 1$ (i.e., $\{s||t\} = R$ with $\lambda - 1$-bit PRG seed s), (ii) the last special-path node (the β-th leaf) is pseudorandom with lsb one, and (iii) other nodes are zero.

Definition 2 (Circular correlation robustness (CCR) hash function [22]). *Let $H_{ccr} : \{0,1\}^\lambda \to \{0,1\}^\lambda$, χ be a distribution on $\{0,1\}^\lambda$, and $\mathcal{F} : \{0,1\}^{\lambda+1} \to \{0,1\}^\lambda$ be a family of functions. Define the oracle $O^{\text{ccr}}_{H_{ccr},R}(x, b) := H_{ccr}(x \oplus R) \oplus b \cdot R$ for $x, R \in \{0,1\}^\lambda, b \in \{0,1\}$. H_{ccr} is a $(t, q, \rho, \varepsilon)$-CCR hash function if, for any distinguisher D running in time at most t and making at most q queries to $O^{\text{ccr}}_{\mathcal{H},R}(\cdot,\cdot)$, and any distribution χ with min-entropy at least ρ, it holds that*

$$\left| \Pr_{R \leftarrow \chi}\left[D^{O^{\text{ccr}}_{H_{ccr},R}(\cdot,\cdot)}(1^\lambda) = 1\right] - \Pr_{f \leftarrow \mathcal{F}}\left[D^{f(\cdot,\cdot)}(1^\lambda) = 1\right] \right| \le \varepsilon,$$

where D is not allowed to query both $(x, 0)$ and $(x, 1)$ for any $x \in \{0,1\}^\lambda$.

According to CCR property, there exists a global key R offsetting the inputs and outputs of a keyed hash function $H_S(x) := H_{ccr}(S \oplus x)$ with some key $S \leftarrow \{0,1\}^\lambda$. Therefore, Guo et al. [23] utilized the CCR hash function (instantiated via fixed-key AES) to construct a lightweight DPF in Half-Tree style. It reduces both the number of AES calls and the communication by half than the classic DPF [8].

3 Lightweight Authenticated DCF

In this section, we explain how to construct secure comparison based on DPF from our new perspective and achieve authentication for active security as shown in Figure 4. We provide a detailed description of our lightweight authenticated DCF scheme with 1-*bit* output in Figure 5. Assume that our DCF preserves a binary tree structure, where each node is labeled by a string $(s, t) \in \{0,1\}^\lambda$, consisting of a PRG seed s and a control bit t the same as DPF. Two DCF keys define the control bits on the special path (the *red* β path) in additive sharing over $\mathbb{F}_2$, satisfying $[\![t]\!]^{\text{B}}_0 \oplus [\![t]\!]^{\text{B}}_1 = 1$, while others are $[\![t]\!]^{\text{B}}_0 \oplus [\![t]\!]^{\text{B}}_1 = 0$.

Definition 3 (Authenticated (2,2)-DCF). *Given the security parameter $\lambda \in \mathbb{N}$, the upper bound range $N \in \mathbb{N}$ of the comparison function input, and a distinguished input $\beta \in [N]$ and Abelian group of output $\mathbb{G}$, consider the family of comparison functions*

$$\mathcal{G}^{<} = \left\{ f^{<}_{\beta,1} : \mathbb{Z}_N \to \mathbb{G} \right\}_{\beta \in \mathbb{Z}_N}$$

where $f^{<}_{\beta,1}(x) = \mathbf{1}\{x < \beta\}$. A secure authenticated (2,2)-DCF scheme for the function family $\mathcal{G}^{<}$ extends the properties of classic DCF (Definition 1) by providing active security against a malicious server. Even if the server deviates from the protocol, the correctness of evaluation is guaranteed except with negligible probability.

Comparison Evaluation from DPF. Let's start off by Grotto's [46] construction as described in Section 2.3. Instead of the complex spline evaluation, we try to explain it in a simple scenario to compare two unsigned integer x and β over $\mathbb{Z}_{2^n}$. Grotto's prefix parity algorithm enables the computation of prefix parity query over a DPF tree by partially traversing the tree from the root to the x-th leaf. Since DPF encodes a single 1 at position β, a prefix parity query

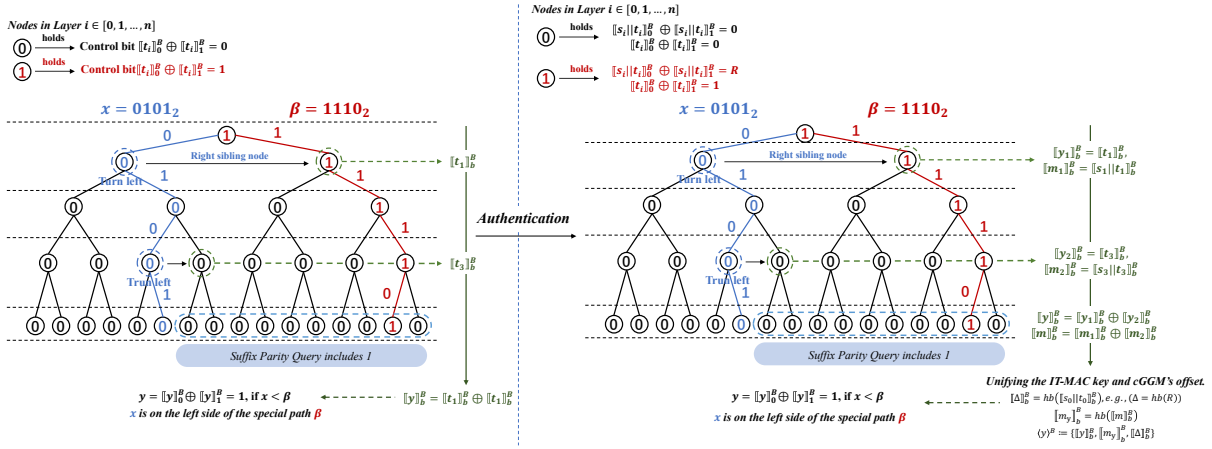

Figure 4: Lightweight Authenticated Distributed Comparison Function Construction

that returns 1 indicates that the target index β lies in the range $[0, x]$, i.e., $x \geq \beta$ holds.

Below, we formally describe the above method from a new perspective. Considering the ring $\mathbb{Z}_{2^n}$ with a size of n bits, we define a domain size $N = 2^n$ for $n \in \mathbb{N}$. Given two unsigned integers β and x, the binary representations are $\beta := (\beta_1, \beta_2, \cdots, \beta_n)$ and $x := (x_1, x_2, \cdots, x_n)$, where each $\beta_i, x_i \in \{0,1\}$, β_i, x_i represents the binary value of the i-th position. The most significant bit (MSB) is β_1 and x_1 respectively. We have the following lemma:

Lemma 1. *Given a domain size $N = 2^n$ for $n \in \mathbb{N}$ and two integers β and x. In a full binary tree, the nodes along the special β path have a value of 1 while others are 0. When $x < \beta$, there is exactly one right sibling node with a value of 1 on the path corresponding to x, which corresponds to the first branching position where $x_j = 0$ and $\beta_j = 1$.*

We provide the proof of Lemma 1 in Apendix D.4.5. According to Lemma 1, we can easily construct the comparison evaluation as shown in the left side of Figure 4. While evaluating our DCF key using public input $x = x_1 \cdots x_n$ from root to the x-th leaf, if the next layer moves to the left child node (which means $x_i = 0$), then the result will be XORed with the control bit $[\![t_i^R]\!]_b^{\mathsf{B}}$ of the right sibling node of that child. Noting that, this method can be viewed as the *suffix parity query* from x-th position and if a suffix parity query that returns 1 indicates that $x < \beta$ holds.

Achieving Active Security. The intuition is to authenticate all control bits of DCF, achieving active security for comparison evaluation. However, Boyle et al.'s DPF [8], which underlies prior works like Grotto [46], SIGMA [25] and Shark [24], lacks a straightforward mechanism for authentication. Fortunately, we make a novel observation, that the control bits of DCF can be implicitly authenticated by the cGGM tree [23], which has an invariant that all same-level nodes sum up to the same global offset. We unify the global offset and the boolean authenticated key, enabling DCF's control bits to be authenticated by IT-MACs with almost no additional overhead.

Building on the above observation, we first consider how to authenticate the control bits. We can construct the seed sequences $\{s_i\}$ and control bit sequences $\{t_i\}$ for the nodes along the path of β in the cGGM tree, ensuring the specific correlation $[\![s_i||t_i]\!]_0^{\mathsf{B}} \oplus [\![s_i||t_i]\!]_1^{\mathsf{B}} = R$. In order to maintain this correlation at each layer, we correct the s_i and t_i at each layer by correction words $CW_i := H_S([\![s_{i-1}\|t_{i-1}]\!]_0^{\mathsf{B}}) \oplus H_S([\![s_{i-1}\|t_{i-1}]\!]_1^{\mathsf{B}}) \oplus \overline{\beta}_i \cdot R$ where $H_S : \{0,1\}^\lambda \to \{0,1\}^\lambda$ is a keyed hash function with circular correlation robustness and β_i is the i-th bit of the input β. Different from Half-Tree [23]'s cGGM construction, we need to maintain the correlation of the last layer and there is no need to construct the CW for output layer due to our single-bit output requirement. The detailed construction is illustrated in DCF.Gen (Figure 5). The cGGM-based DCF key involves Keysize $= (n+1) \cdot \lambda$ bits which requires one seed $[\![s_0\|t_0]\!]_b^{\mathsf{B}} \in \{0,1\}^\lambda$ and n correction words $\{CW_i \in \{0,1\}^\lambda\}_{i \in [1,n]}$.

We utilize the correlation $[\![s_i||t_i]\!]_0^{\mathsf{B}} \oplus [\![s_i||t_i]\!]_1^{\mathsf{B}} = R$ along the β path to construct the result's boolean authenticated sharing by XORing the PRG seed $[\![s_i^R\|t_i^R]\!]_b^{\mathsf{B}}$ of the right sibling node of that child (as shown in Fig. 5, line 7 and line 8 of DCF.Eval). It is worth noting that we define the global offset of the cGGM tree as $\mathsf{hb}(R, \kappa) = \Delta$ in order to adapt our boolean authenticated sharing, thereby unifying the global offset with the boolean authenticated key. We will discuss the security of this construction in Appendix D.1.

These node label $[\![s_i||t_i]\!]^{\mathsf{B}}$ on the β path in cGGM tree can be regarded as $\langle 1 \rangle^{\mathsf{B}}$, the boolean authenticated sharing of 1-value bit, which satisfy $\mathsf{hb}([\![s_i||t_i]\!]_0^{\mathsf{B}} \oplus [\![s_i||t_i]\!]_1^{\mathsf{B}}, \kappa) = 1 \cdot \Delta$ and $[\![t_i]\!]_0^{\mathsf{B}} \oplus [\![t_i]\!]_1^{\mathsf{B}} = 1$. While, the label of all other nodes can be regarded as $\langle 0 \rangle^{\mathsf{B}}$, the 0-value bit boolean authenticated sharing. DCF.Eval computes the boolean authenticated sharing $\langle y \rangle^{\mathsf{B}} = \bigoplus_{i=1}^{n} \{\langle 0 \rangle^{\mathsf{B}}, \langle 1 \rangle^{\mathsf{B}}\}$ according to each $\overline{x}_i$ with public input $x = x_1 \ldots x_n$. If one of the servers is corrupted by malicious adversary, any malicious behaviors will lead the IT-MAC of $\langle y \rangle^{\mathsf{B}}$ invalid, resulting in MAC checking failure.

Theorem 1. *Given the CCR hash function $H_{ccr} : \{0,1\}^\lambda \to \{0,1\}^\lambda$ and keyed hash function $H_S(x) := H_{ccr}(S \oplus x)$ with some key $S \leftarrow \{0,1\}^\lambda$, Figure 5 is a secure authenticated DCF scheme with key size* Keysize $= (n+1) \cdot \lambda$ *bits.*

We provide the detailed proof in Appendix D.1.

Parameters: Domain size $N = 2^n$ for $n \in \mathbb{N}$. Integer ring $\mathbb{Z}_{2^n}$. Hash function $H_{ccr} : \{0,1\}^\lambda \to \{0,1\}^\lambda$ with circular correlation robustness. Keyed hash function $H_S(x) := H_{ccr}(S \oplus x)$ with some key $S \leftarrow \{0,1\}^\lambda$. The output field $\mathbb{F}_{2^\kappa}$ and $\kappa < \lambda$.

DCF.Gen$(1^\lambda, \Delta, \beta, n)$

1: Parse $\beta = \beta_1 \cdots \beta_n \in \{0,1\}^n$.
2: Sample $R \leftarrow \{0,1\}^\lambda$ such that $\mathsf{lsb}(R) = 1$ and $\mathsf{hb}(R, \kappa) = \Delta$.
3: Sample $[\![s_0 \| t_0]\!]^{\mathsf{B}}_0, [\![s_0 \| t_0]\!]^{\mathsf{B}}_1 \leftarrow \{0,1\}^\lambda$ such that $[\![s_0 \| t_0]\!]^{\mathsf{B}}_0 \oplus [\![s_0 \| t_0]\!]^{\mathsf{B}}_1 = R$.
4: **for** $i \in [1, n]$ **do**
5: $\quad CW_i := H_S([\![s_{i-1} \| t_{i-1}]\!]^{\mathsf{B}}_0) \oplus H_S([\![s_{i-1} \| t_{i-1}]\!]^{\mathsf{B}}_1) \oplus \overline{\beta}_i \cdot R$.
6: $\quad [\![s_i \| t_i]\!]^{\mathsf{B}}_0 := H_S([\![s_{i-1} \| t_{i-1}]\!]^{\mathsf{B}}_0) \oplus \beta_i \cdot [\![s_{i-1} \| t_{i-1}]\!]^{\mathsf{B}}_0 \oplus [\![t_{i-1}]\!]^{\mathsf{B}}_0 \cdot CW_i$.
7: $\quad [\![s_i \| t_i]\!]^{\mathsf{B}}_1 := H_S([\![s_{i-1} \| t_{i-1}]\!]^{\mathsf{B}}_1) \oplus \beta_i \cdot [\![s_{i-1} \| t_{i-1}]\!]^{\mathsf{B}}_1 \oplus [\![t_{i-1}]\!]^{\mathsf{B}}_1 \cdot CW_i$.
8: $k_b := ([\![s_0 \| t_0]\!]^{\mathsf{B}}_b, \{CW_i\}_{i \in [1,n]})$.
9: **return** (k_0, k_1)

DCF.Eval(b, k_b, x)

1: Parse $k_b = ([\![s_0 \| t_0]\!]^{\mathsf{B}}_b, \{CW_i\}_{i \in [1,n]})$ and $x = x_1 \ldots x_n$.
2: Define $[\![y]\!]^{\mathsf{B}}_b := 0$ and $[\![m]\!]^{\mathsf{B}}_b := 0$.
3: **for** $i \in [1, n]$ **do**
4: $\quad [\![s^L_i \| t^L_i]\!]^{\mathsf{B}}_b := H_S([\![s_{i-1} \| t_{i-1}]\!]^{\mathsf{B}}_b) \oplus [\![t_{i-1}]\!]^{\mathsf{B}}_b \cdot CW_i$.
5: $\quad [\![s^R_i \| t^R_i]\!]^{\mathsf{B}}_b := [\![s^L_i \| t^L_i]\!]^{\mathsf{B}}_b \oplus [\![s_{i-1} \| t_{i-1}]\!]^{\mathsf{B}}_b$.
6: $\quad [\![s_i \| t_i]\!]^{\mathsf{B}}_b := \overline{x}_i \cdot [\![s^L_i \| t^L_i]\!]^{\mathsf{B}}_b \oplus x_i \cdot [\![s^R_i \| t^R_i]\!]^{\mathsf{B}}_b$.
7: $\quad [\![y]\!]^{\mathsf{B}}_b = [\![y]\!]^{\mathsf{B}}_b \oplus \overline{x}_i \cdot [\![t^R_i]\!]^{\mathsf{B}}_b$.
8: $\quad [\![m]\!]^{\mathsf{B}}_b = [\![m]\!]^{\mathsf{B}}_b \oplus \overline{x}_i \cdot [\![s^R_i \| t^R_i]\!]^{\mathsf{B}}_b$.
9: Define $[\![\Delta]\!]^{\mathsf{B}}_b := \mathsf{hb}([\![s_0 \| t_0]\!]^{\mathsf{B}}_b, \kappa)$ and $[\![m_y]\!]^{\mathsf{B}}_b := \mathsf{hb}([\![m]\!]^{\mathsf{B}}_b, \kappa)$.
10: Let $\langle y \rangle^{\mathsf{B}}_b := \{[\![y]\!]^{\mathsf{B}}_b, [\![m_y]\!]^{\mathsf{B}}_b, [\![\Delta]\!]^{\mathsf{B}}_b\}$.
11: **return** $\langle y \rangle^{\mathsf{B}}_b$

Consistency Checking: In subsequent protocols, if values derived from $\langle y \rangle^{\mathsf{B}}_b$ are opened, the servers can perform BatchCheck for DCF outputs in the end of overall protocol.

Figure 5: Lightweight Authenticated DCF Scheme

4 Actively Secure ML Inference Framework

In this section, we provide an actively-secure inference framework, leveraging our lightweight authenticated DCF scheme presented in section 3. Secure machine-learning inference can be divided into linear layers (matrix multiplications with truncation in Fixed-Point Arithmetic) and non-linear layers (such as ReLU, Maxpool, etc. can be directly performed by a comparison function, while GeLU, Sqrt root and Tanh, etc. can be performed by spline functions). More details for ML inference framework can be found in Appendix C.

4.1 Two-Party Computation with $\mathcal{F}_{dealer}$

We denote two-party computation (2PC) with $\mathcal{F}_{dealer}$ as the $\mathcal{F}_{\mathsf{dealer}}$-2PC setting. Our $\mathcal{F}_{\mathsf{dealer}}$-2PC protocol is adapted from TurboSPDZ [4] and ABY 2.0 [41], with wire masks replaced by authenticated sharings (as described in Section 2.2). We present $\mathcal{F}_{\mathsf{dealer}}$, the ideal functionality of the dealer, in Appendix B.1. The overall $\mathcal{F}_{\mathsf{dealer}}$-2PC protocol proceeds through the following stages.

Initialize. Servers invoke $\mathcal{F}_{\mathsf{dealer}}$ with (Init) to obtain the arithmetic and boolean authenticated key $[\![\alpha]\!]^{\mathsf{A}}$ and $[\![\Delta]\!]^{\mathsf{B}}$.

Input Phase. For arithmetic input $x \in \mathbb{Z}_{2^n}$ of P_b, $b \in \{0,1\}$, servers invoke $\mathcal{F}_{\mathsf{dealer}}$ with (Input, b, A). P_b obtains δ_x and computes $\Lambda_x = x + \delta_x \mod 2^n$. Then P_b sends Λ_x to P_{1-b}. As for boolean input $x \in \mathbb{F}_2$ of P_b, servers invoke $\mathcal{F}_{\mathsf{dealer}}$ with (Input, b, B). P_b obtains $\delta_x \in \mathbb{F}_2$ from $\mathcal{F}_{\mathsf{dealer}}$, computes $\Lambda_x = x \oplus \delta_x \in \mathbb{F}_2$ and opens Λ_x.

Computation Phase. Since $\mathcal{F}_{\mathsf{dealer}}$ generates the uniform masks for all inputs, it can perform input-independent preprocessing based on these masks. Servers obtain the prepared materials from $\mathcal{F}_{\mathsf{dealer}}$ and perform online computations using the masked inputs Λ_x. The detailed designs of DReLU and Spline are provided in the following section 4.2 and section 4.3. We further provide the 2PC instantiation in Appendix E.

For example, consider a function $y = f(x)$ with input $x \in \mathbb{Z}_{2^n}$ and output $y \in \mathbb{F}_2$. Servers invoke $\mathcal{F}_{\mathsf{dealer}}$ with (Func, $f(x)$) and obtain the prepared materials (including $\langle \delta_x \rangle^{\mathsf{A}}$, $\langle \delta_y \rangle^{\mathsf{B}}$ and other materials for online computation). Servers take the masked input $\Lambda_x = x + \delta_x \bmod 2^n$ (obtained from the input phase) and compute $\Lambda_y = f(\Lambda_x - \delta_x) \oplus \delta_y$ with prepared materials.

Output Phase. For each arithmetic output $y \in \mathbb{Z}_{2^n}$, servers invoke $\mathcal{F}_{\mathsf{dealer}}$ with (Output, A) to obtain $\langle -\hat{\delta}_y \rangle^{\mathsf{A}}$, and locally compute $\langle y \rangle^{\mathsf{A}} = \Lambda_y + \langle -\hat{\delta}_y \rangle^{\mathsf{A}}$. For each boolean output $y \in \mathbb{F}_2$, servers ervers invoke $\mathcal{F}_{\mathsf{dealer}}$ with (Output, B) to obtain $\langle \delta_y \rangle^{\mathsf{B}}$, and locally compute $\langle y \rangle^{\mathsf{B}} = \Lambda_y \oplus \langle \delta_y \rangle^{\mathsf{B}}$. Before output reconstruction, the servers perform BatchCheck on all previously opened values in both the arithmetic and boolean worlds. The probability that a malicious adversary evades the verification is $2^{-s+\log(s+1)}$ and $2^{-\kappa}$, respectively. Finally, reconstruct the outputs with the corresponding MAC checking.

4.2 Derivative of ReLU Function

ReLU introduces essential non-linearity, allowing neural networks to capture complex features with efficient inference. ReLU function can be expressed as $\mathsf{ReLU}(x) = \max(x, 0) = x \cdot \mathsf{DReLU}(x)$, where DReLU denotes the derivative of ReLU. We present here our design for DReLU protocol in the $\mathcal{F}_{\mathsf{dealer}}$-2PC setting.

Observing that $\mathsf{DReLU}(x) = 1 \cdot \{x \geq 0\}$, the main challenge is to determine the sign bit of x encoded in a signed integer within $\mathbb{Z}_{2^n}$. Therefore, $\mathsf{DReLU}(x) = 1 \oplus \mathsf{msb}(x)$. In our $\mathcal{F}_{\mathsf{dealer}}$-2PC setting, servers hold masked $\Lambda_x = x + \delta_x \mod 2^n$. We have the following equation,

$$\begin{aligned}\mathsf{msb}(x) &= \mathsf{msb}(\Lambda_x - \delta_x) \\ &= \mathsf{msb}(\Lambda_x) \oplus \mathsf{msb}(-\delta_x) \oplus 1 \cdot \{\Lambda'_x + \delta'_x \geq 2^{n-1}\} \\ &= \mathsf{msb}(\Lambda_x) \oplus \mathsf{msb}(-\delta_x) \oplus 1 \cdot \{2^{n-1} - \Lambda'_x - 1 < \delta'_x\}\end{aligned}$$

where $\Lambda'_x = \Lambda_x \mod 2^{n-1}$ and $\delta'_x = -\delta_x \mod 2^{n-1}$.

Prior works [13, 24] have followed the above approach using DCF or boolean circuit to achieve the comparison $1 \cdot \{2^{n-1} - \Lambda'_x - 1 < \delta'_x\}$ with heavyweight FSS key or non-constant communication rounds against malicious adversary. Benefiting from our DCF scheme (Figure 5), we can achieve the same goal with more compact key and lightweight computation to complete this comparison. Π_{DReLU} is detailed presented in Figure 6, which requires prepared materials of $(1+\kappa)$ + Keysize bits in the preprocessing phase including one boolean authenticated share and one DCF key. While the online phase requires communication overhead of 1 bit per party in one round for one boolean authenticated share reconstruction and $n-1$ AES calls for DCF.Eval.

Theorem 2. *(Security) Figure 6 shows a actively secure derivative of ReLU protocol against the malicious adversary who can corrupt one of two servers P_0 and P_1 in $\mathcal{F}_{\mathsf{PrepDReLU}}$-hybrid model.*

We provide the detailed proof in Appendix D.2.

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

The preprocessing functionality has all the same features as $\mathcal{F}_{\mathsf{dealer}}$, with the additional commands:

DReLU: On input $(\mathsf{Func}, \mathsf{DReLU})$ from P_b, $b \in \{0,1\}$, $\mathcal{F}_{\mathsf{PrepDReLU}}$ executes as follows:

(1) Restore the input mask $\hat{\delta}_x \in \mathbb{Z}_{2^{n+s}}$. Sample the uniform output mask $\delta_y \in \mathbb{F}_2$ and store it.
(2) Compute $\delta_x = \hat{\delta}_x \mod 2^n$ and $\delta'_x = -\delta_x \mod 2^{n-1}$.
(3) Execute $(k_0, k_1) \leftarrow \mathsf{DCF.Gen}(1^\lambda, \Delta, \delta'_x, n-1)$ and send k_b to P_b, $b \in \{0,1\}$.
(4) Compute $\delta_t = 1 \oplus \delta_y \oplus \mathsf{msb}(\delta'_x)$.
(5) Execute $\langle \delta_t \rangle^{\mathsf{B}} \leftarrow \mathsf{ShareB}(\delta_t)$.

Protocol Π_{DReLU}

Preprocessing Phase: Server P_b, $b \in \{0,1\}$ invokes $\mathcal{F}_{\mathsf{PrepDReLU}}$ with $(\mathsf{Func}, \mathsf{DReLU})$ to obtain the prepared materials $\langle \delta_t \rangle^{\mathsf{B}}_b$ and k_b for DReLU.

Online Phase: P_b computes as follows with inputs Λ_x and prepared materials from $\mathcal{F}_{\mathsf{PrepDReLU}}$:

(1) Compute $x_{in} = (2^{n-1} - (\Lambda_x \mod 2^{n-1}) - 1) \mod 2^n$.
(2) Execute $\langle t \rangle^{\mathsf{B}} \leftarrow \mathsf{DCF.Eval}(b, x_{in}, k_b)$.
(3) Locally compute $\langle \Lambda_y \rangle^{\mathsf{B}} = \langle \delta_t \rangle^{\mathsf{B}} \oplus \langle t \rangle^{\mathsf{B}} \oplus \mathsf{msb}(\Lambda_x)$.
(4) Execute $\mathsf{OpenB}(\langle \Lambda_y \rangle^{\mathsf{B}})$ to obtain Λ_y.

Figure 6: Actively-secure Derivative of ReLU

4.3 Spline Function

Spline functions are usually used for approximation of complex nonlinear functions. We can construct a piecewise-polynomial function $F(x)$ to approximate $f(x)$ and define the coefficients vectors $\{a_{d,j}, \ldots, a_{1,j}, a_{0,j}\}$ for all $j \in [T]$, where T is the number of intervals in the piecewise function and d is the polynomial degree.

Actively secure construction. Our DCF.Eval construction in Figure 5 can achieve the comparison $x < \beta$ with input x and a special random value β from DCF key. We can say that if the path of x is on the left side of β, the output is 1, otherwise the output is 0. The above result can also be interpreted as *suffix parity query* $\mathsf{parity}(v[x+1, 2^n+1)) = \bigoplus_{i=x+1}^{2^n} v_i$, where $v = v_1 v_2 \ldots v_{2^n}$ is represented as the output string of the full-domain evaluation.

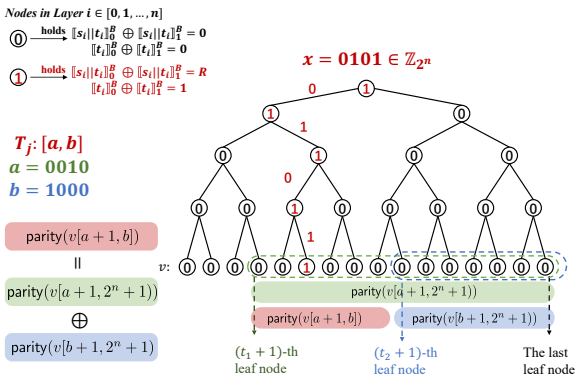

Figure 7: Our Selection Vector Generation

As illustrated in Figure 7, supposing that we have a cGGM tree with a special path corresponding our input x, we can obtain the suffix parity query of interval S_j by computing $\mathsf{parity}(v[a+1, b]) = \mathsf{parity}(v[a+1, 2^n+1]) \oplus \mathsf{parity}(v[b+1, 2^n+1])$ utilizing two invocations of our DCF.Eval with inputs a and b. Utilizing this approach, we can easily compute the selection vector $\vec{e}$ by computing the parity queries of all interval S_j for $j \in [T]$ in the piecewise-polynomial function. And the selection vector $\vec{e}$ is defined as $\vec{e} = (\mathsf{parity}(S_1), \mathsf{parity}(S_2), \ldots, \mathsf{parity}(S_T))$. Similarly, we can directly apply the Grotto's method to reduce the calls to AES by reusing the nodes that have already been evaluated.

However, we can not directly generate a DCF key using the secret input x. In our $\mathcal{F}_{\mathsf{dealer}}$-2PC setting, servers hold public $\Lambda_x = x + \delta_x \mod 2^n$, and we can pre-generate a DCF key for the function $f^{<}_{\delta'_x, 1} : \{0,1\}^n \to \{0,1\}$ with the input mask $\delta'_x = -\delta_x \mod 2^n$. Following Grotto [46], we first cyclically shift each endpoint of interval $S_j, j \in [T]$ to the left according to Λ_x. This method can make the leaf node with 1 output corresponding to δ'_x's path fall within the interval containing input x. This step can be locally computed by servers due to each interval S_j is publicly known.

Polynomial evaluation. Since our DCF scheme is designed for boolean outputs, let vector $\langle \vec{e} \rangle^{\mathsf{B}} := \{\langle e[1] \rangle^{\mathsf{B}}, \langle e[2] \rangle^{\mathsf{B}}, \ldots, \langle e[T] \rangle^{\mathsf{B}}\}$ denote the boolean authenticated shares of selection vector. First, we need to convert them into arithmetic domain. Grotto [46] can perform this conversion locally: since additive shares over $\mathbb{F}_2$ take values in $\{0,1\}$, both servers locally set $[\![e[j]]\!]^{\mathsf{A}} := (-1)^b \cdot [\![e[j]]\!]^{\mathsf{B}} \mod 2^n$ for $b \in \{0,1\}$ and $j \in [T]$, and then apply a sign correction for the case $([\![e[j]]\!]^{\mathsf{B}}_0 \neq [\![e[j]]\!]^{\mathsf{B}}_1)$ to complete the conversion. Grotto hides this sign correction within the polynomial evaluation. However, this approach cannot be directly applied in the actively secure setting. The core challenge lies in converting the authentication from $\mathbb{F}_{2^\kappa}$ to $\mathbb{Z}_{2^{n+s}}$.

Here, we introduce the concept of *daBits* [1, 44] to perform Boolean-Arithmetic conversion. A daBit provides authentication of the same bit in two worlds. In our setting, random daBits can be defined as $\{\langle \vec{r} \rangle^{\mathsf{A}}, \langle \vec{r} \rangle^{\mathsf{B}}\}$ for $\vec{r} = \{r[1], r[2], \ldots, r[T]\} \in \{0,1\}^T$. We can compute $\langle e[j] \rangle^{\mathsf{B}} \oplus \langle r[j] \rangle^{\mathsf{B}}$ and reveal $s[j] = e[j] \oplus r[j]$ for $j \in [T]$. And then compute the arithmetic authenticated share $\langle e[j] \rangle^{\mathsf{A}} = s[j] + (1 - 2 \cdot s[j]) \cdot \langle r[j] \rangle^{\mathsf{A}}$ for $j \in [T]$. However, the above method introduces an additional communication round. We further optimize this process by fusing the conversion with the selection of correct polynomial coefficients, thereby completing the spline evaluation without introducing additional communication rounds compared to Grotto [46].

After revealing $s[j] = e[j] \oplus r[j]$, we can compute the coefficients $\langle a[i] \rangle^{\mathsf{A}} = \sum_{j=1}^{T} \left[s[j] + (1 - 2 \cdot s[j]) \cdot \langle r[j] \rangle^{\mathsf{A}} \right] \cdot a_{i,j}$ for $i \in \{0, 1, \ldots, d\}$ and $j \in [T]$ over arithmetic domain. To evaluate the corresponding polynomial on masked input $\Lambda_x = x + \delta_x \mod 2^n$ and $\delta'_x = -\delta_x \mod 2^n$, we have

$$
\begin{aligned}
&\sum_{i=0}^{d} a[i] \cdot (\Lambda_x + \delta'_x)^i \\
&= \sum_{i=0}^{d} \Big(\sum_{j=1}^{T} (s[j] + (1 - 2 \cdot s[j]) \cdot r[j]) \cdot a_{i,j} \Big) \cdot (\Lambda_x + \delta'_x)^i \\
&= \sum_{k=0}^{d} \Lambda_x^k \sum_{i=k}^{d} \binom{i}{k} (\delta'_x)^{i-k} \sum_{j=1}^{T} s[j] \cdot a_{i,j} \\
&\quad - \sum_{k=0}^{d} \Lambda_x^k \sum_{i=k}^{d} \binom{i}{k} \sum_{j=1}^{T} (2 \cdot s[j] - 1) \cdot a_{i,j} \cdot r[j] \cdot (\delta'_x)^{i-k}.
\end{aligned}
$$

During the online phase, only δ'_x and $\vec{r}$ in the above equation are secret-shared. Accordingly, we only need to prepare authenticated sharings of $(\delta'_x)^i \mod 2^n$ and $r[j] \cdot (\delta'_x)^i \mod 2^n$, for $j \in \{1, \ldots, T\}$ and $i \in \{0, 1, \ldots, d\}$. The detailed protocol is presented in Figure 8, which requires prepared materials of $T \cdot (1 + \kappa) + 2 \cdot (1 + (d+1) \cdot (T+1)) \cdot (n+s) +$ Keysize bits in the preprocessing phase including T boolean authenticated shares, $(1 + d + 1 + (d+1) \cdot T)$ arithmetic authenticated shares and one DCF key. While the online phase requires communication overhead of $T + n + s$ bits per party in two rounds for reconstruction of T boolean authenticated shares and one arithmetic authenticated share. Moreover, $T \cdot n$ AES calls are required for DCF.Eval.

Theorem 3. *Figure 8 shows a actively secure spline protocol against the malicious adversary who can corrupt one of two servers P_0 and P_1 in $\mathcal{F}_{\mathsf{PrepSpline}}$-hybrid model.*

We provide the detailed proof in Appendix D.3.

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

The preprocessing functionality has all the same features as $\mathcal{F}_{\mathsf{dealer}}$, with the additional commands:

Spline: On input (Func, Spline) from $P_b, b \in \{0,1\}$, $\mathcal{F}_{\mathsf{PrepSpline}}$ executes as follows:

1. Restore the input mask $\hat{\delta}_x \in \mathbb{Z}_{2^{n+s}}$. Sample the uniform output mask $\hat{\delta}_z \in \mathbb{Z}_{2^{n+s}}$ and store it.
2. Compute $\delta_x = \hat{\delta}_x \mod 2^n, \delta'_x = -\delta_x \mod 2^n$. Sample uniform bits $\{r[1], \ldots, r[T]\} \in \{0,1\}^T$.
3. Execute $(k_0, k_1) \leftarrow \mathsf{DCF.Gen}(1^\lambda, \Delta, \delta'_x, n)$ and send k_b to $P_b, b \in \{0,1\}$.
4. For $j \in \{1, \ldots, T\}$ and $i \in \{0, 1, \ldots, d\}$, compute $A[i] = (\delta'_x)^i \mod 2^n$ and $B[i \cdot T + j] = r[j] \cdot (\delta'_x)^i \mod 2^n$.
5. Execute $\langle \vec{r} \rangle^{\mathsf{B}} \leftarrow \mathsf{ShareB}(\vec{r})$, $\langle \hat{\delta}_z \rangle^{\mathsf{A}} \leftarrow \mathsf{ShareA}(\hat{\delta}_z)$, $\langle \vec{A} \rangle^{\mathsf{A}} \leftarrow \mathsf{ShareA}(\vec{A})$ and $\langle \vec{B} \rangle^{\mathsf{A}} \leftarrow \mathsf{ShareA}(\vec{B})$ where $\vec{A} := \{A[0], \ldots, A[d]\}$ and $\vec{B} := \{B[1], \ldots, B[(d+1) \cdot T]\}$.

Protocol Π_{Spline}

Preprocessing Phase: Server P_b, $b \in \{0,1\}$ invokes $\mathcal{F}_{\mathsf{PrepSpline}}$ with (Func, Spline) to obtain the prepared materials $\langle \vec{r} \rangle^{\mathsf{B}}$, $\langle \hat{\delta}_z \rangle^{\mathsf{A}}$, $\langle \vec{A} \rangle^{\mathsf{A}}$, $\langle \vec{B} \rangle^{\mathsf{A}}$ and k_b for Spline function.

Online Phase: P_b computes as follows with inputs Λ_x and prepared materials from $\mathcal{F}_{\mathsf{PrepSpline}}$:

1. For $j \in \{1, 2, \ldots, T\}$ and interval $S_j := [a, b]$, circular left shift to obtain interval $S'_j := [a', b']$ where $a' = a - \Lambda_x \mod 2^n$ and $b' = b - \Lambda_x \mod 2^n$.
2. Execute DCF.Eval according to the endpoints of S'_j for $j \in \{1, 2, \ldots, T\}$ to obtain $\langle \vec{e} \rangle^{\mathsf{B}} := \{\langle e[1] \rangle^{\mathsf{B}}, \ldots, \langle e[T] \rangle^{\mathsf{B}}\}$.
3. For $j \in \{1, 2, \ldots, T\}$, compute $\langle s[j] \rangle^{\mathsf{B}} = \langle e[j] \rangle^{\mathsf{B}} \oplus \langle r[j] \rangle^{\mathsf{B}}$ and execute $\mathsf{OpenB}(\langle s[j] \rangle^{\mathsf{B}})$ to obtain $s[j]$.
4. Locally compute as follows: $\langle \Lambda_z \rangle^{\mathsf{A}} = (\sum_{k=0}^{d} \Lambda_x^k \sum_{i=k}^{d} \binom{i}{k} \langle A[i-k] \rangle^{\mathsf{A}} \sum_{j=1}^{T} s[j] \cdot a_{i,j} - \sum_{k=0}^{d} \Lambda_x^k \sum_{i=k}^{d} \binom{i}{k} \sum_{j=1}^{T} (2 \cdot s[j] - 1) \cdot a_{i,j} \cdot \langle B[(i-k) \cdot T + j] \rangle^{\mathsf{A}}) + \langle \hat{\delta}_z \rangle^{\mathsf{A}}$.
5. Execute $\mathsf{OpenA}(\langle \Lambda_z \rangle^{\mathsf{A}})$ to obtain Λ_z.

Figure 8: Actively-secure Spline Function.

5 Evaluation and Implementation

To showcase the efficiency of our protocols, we implement several privacy-preserving inference tasks such as Convolutional Neural Networks (CNNs) and Large Language Models (LLMs).

5.1 Implementation and Experimental Setup

Testbed Environment. Our implementations are built on the primitives provided by Shark [24]. Additionally, we use EMP-toolkit [48] for CCR hash and Eigen [21] for matrix operations. All evaluations and implementations are run on a Ubuntu PC equipped with an Intel(R) Xeon(R) Silver 4210 and 128GB of RAM. We use the tc tool to control the bandwidth between servers. The LAN network bandwidth is up to 1 Gbps with an RTT of 0.5 ms, while the WAN network bandwidth is up to 200 Mbps with an RTT of 40 ms.

Concrete Parameters. Our benchmarks are conducted under parameters $n = s = 64$ and $\kappa = 40$. We set $\lambda = 128$ for DCF scheme. We use Fixed-Point Arithmetic (FPA) to represent decimal values, with f denoting the precision of the fixed-point arithmetic. We choose

$f = 16, 12$ respectively for CNNs and LLMs. The parameter settings ensure the statistical security and that the gap always exists in our truncation protocol (placed in Appendix B.4) for ML inference.

ML Models. We implement secure inference of CNNs and LLMs. Specifically, for CIFAR-10 [33], we evaluate a CNN consisting of three convolutional layers and three max-pooling layers. We also implement secure inference for AlexNet [34] and VGG-16 [45], two groundbreaking CNNs for image classification and recognition. To demonstrate the efficiency and scalability of our implementations in LLMs, we evaluate the performance of transformer inference for different parameter scales of Bert [17] and GPT2 [9]. Nonlinear functions in transformer (e.g., softmax, GELU, square root, reciprocal) are primarily used in Multi-head Attention and Feed-Forward Network. We follow the approach of MPCFormer [36], SIGMA [25] and Shark [24] to approximate these expensive operations with operations that are easier to compute.

Table 4: Online Performance of Nonlinear Protocols

DReLU Test					
Inputs	Scheme	LAN (*Sec*)	WAN (*Sec*)	Prep. (MB)	Comm. (MB)
10^5	Mosformer [10]	0.162	0.422	–	11.444
	MD-ML [52]	0.799	350	**19.027**	199.85
	Shark [24]	0.101	0.544	155.07	1.14
	Ours	**0.079**	**0.42**	101.6	1.14
10^6	Mosformer [10]	1.992	4.949	–	114.44
	MD-ML [52]	7.421	448.4	**190.25**	1998.5
	Shark [24]	0.473	1.876	1550.7	11.44
	Ours	**0.252**	**1.436**	1016.6	11.44
Other Nonlinear Function Test					
Func.	Scheme	LAN (*Sec*)	WAN (*Sec*)	Prep. (MB)	Comm. (MB)
Sigmoid	Grotto [46]	0.136	0.61	6.86	0.687
	Shark [24]	0.188	0.907	35.67	1.87
	Ours	0.148	0.69	35.06	0.99
Reciprocal	Shark [24]	0.393	1.46	85.42	2.54
	Ours	0.144	0.71	43.29	0.992
RSqrt	Grotto [46]	0.133	0.62	7.2	0.687
	Ours	0.144	0.66	27.18	0.801
Tanh	Grotto [46]	0.135	0.63	6.86	0.687
	Ours	0.144	0.82	43.9	1.125

⋆ **Prep.** denotes the size of prepared materials; **Comm.** denotes the online communication cost.
⋆ All benchmarks in other nonlinear function test are conducted with 10^4 inputs.

5.2 Benchmarking for Protocols

Baselines. DReLU, together with Sigmoid, Tanh, Reciprocal, and Reciprocal of square root (RSqrt), captures the core nonlinear building blocks required by most machine learning inference workloads. These nonlinear functions can be approximated by our DReLU and spline protocols and we use the ULP error [1] to measure the accuracy of the approximated nonlinear function. We compare our DReLU and Spline protocols with state-of-the-art (SOTA) actively secure works: Shark [24], Mosformer [10], and MD-ML [52]. Shark utilizes Boyle's DCF [6] to achieve 2PC actively secure comparison, which is the most closely related to our work. Mosformer presents a malicious 3PC DReLU protocol, reducing DCF to DPF with small DPF domain by using the DGK comparison [14]. MD-ML achieves the SOTA malicious comparison with addition circuit for multi-party setting. However, MD-ML's implementations only support NN with ReLU and convolution, excluding sigmoid, reciprocal and RSqrt. Therefore, we only compare ours with MD-ML in DReLU Test. We also consider the closely related semi-honest work Grotto [46]. However, Grotto only provides basic primitives and does not implement ML tasks, we compare with Grotto only in Other Nonlinear Function Test. Results for Shark and MD-ML are obtained by re-running their open-source implementations in our environment, while the results of Mosformer and Grotto are taken directly from their papers.

Benchmarking results. Table 4 shows the online performance of nonlinear protocols. Our benchmark for sigmoid uses a degree-2 spline with 19 intervals with ULPs error ≤ 4. Reciprocal function uses a degree-3 spline with 19 intervals with ULPs error ≤ 8. RSqrt function uses a degree-3 spline with 9 intervals with ULPs error ≤ 5. Tanh function uses a degree-2 spline with 20 intervals with ULPs error ≤ 6. Compared to MD-ML [52] in 2PC setting, our protocol achieves speedups of up to 29.45× and 831× in LAN and WAN, respectively, due to the fewer communication rounds. Compared to Mosformer [10], even our communication latency is higher (Mosformer uses 0.5 ms RTT in LAN and 4 ms RTT in WAN, whereas our WAN RTT is 40 ms), DReLU protocol still obtains speedups of 7.9× in LAN and 3.44× in WAN. Compared to Shark [24], our more compact DPF keys yield increasing advantages as the input size grows as shown in DReLU test. Furthermore, our reciprocal protocol achieves speedups of up to 2.73× in LAN and 2.04× in WAN. Compared to Grotto [46], we try to ensure a fair comparison by keeping the intervals and degrees as consistent as possible. Similar to Grotto, our approach requires one additional round to perform Boolean-to-Arithmetic conversion compared to the DCF approach [6, 26]. We further hide this round within the polynomial evaluation, which leads to higher preprocessing overhead. Nevertheless, due to our lightweight authenticated DCF based on the cGGM offset, the online computation overhead increases by only about 16% compared to Grotto, even under active security.

5.3 Benchmarking for Inference Tasks

Baselines. We compare with the SOTA actively secure ML inference works such as Shark [24], Mosformer [10] and MD-ML [52]. However, MD-ML's implementation only includes AlexNet and relies on a trusted third party for preprocessing. We exclude the LLMs comparisons with MD-ML. We also include comparisons with semi-honest ML inference frameworks: BumbleBee [29], SHAFT [30], and SIGMA [25]. These frameworks represent the SOTA approaches

[1] ULP (unit of least precision) is the number of representable values between our result and output over reals [26].

Table 5: Performance Comparison of CNNs inference

Models	(Input dimension)	Scheme	LAN (Online, *Sec*)	WAN (Online, *Sec*)	Prep. time (*Sec*)	Comm. (Online, GB)	Comm. (Offline, GB)
CNN	$16 * (3, 32, 32)$	Shark [24]	1.954 (1.43×)	14.36 (1.34×)	20.3	0.098	3.49
		Ours	**1.37**	**10.74**	11.9	0.098	2.39
AlexNet	$1 * (3, 224, 224)$	MD-ML [52]	22.73 (6.33×)	428.1 (45.8×)	539	3.24	24.75
		Shark [24]	4.7 (1.31×)	9.34 (1.23×)	21.04	0.045	6.31
		Ours	**3.6**	**7.6**	17.6	0.045	5.73
VGG-16	$1 * (3, 224, 224)$	Shark [24]	44.9 (1.49×)	117 (1.27×)	241.5	0.86	33
		Ours	**30.06**	**91.87**	172	0.86	24.7

⋆ **Comm.** denotes online and offline communication costs; the offline communication accounts for the size of prepared materials received from dealer.
⋆ **Prep.time** denotes the time for the dealer to generate prepared materials.

Table 6: Performance Comparison of LLMs inference

GPT2-base Test with 64 tokens							
Architecture	Security Model	Scheme	LAN (Online, *Sec*)	WAN (Online, *Sec*)	Prep. time (*Sec*)	Comm. (Online, GB)	Comm. (Offline, GB)
2-PC	Passive	BumbleBee [29]	77.17 (7.07×)	1118.8 (15.3×)	0	2.77	0
2-PC	Passive	SHAFT [30]	59.3 (5.4×)	672.9 (9.2×)	–	6.16	–
3-PC	Active	Mosformer [10]	39.26 (3.6×)	98.89 (1.35×)	–	3.07	–
(2+1)-PC	Passive	SIGMA (GPU) [25]	2.05	9.9	1.78	0.37	6.8
(2+1)-PC	Active	**Ours**	**10.9**	**73.2**	29.5	0.6	8.11
Bert-base Test with 128 tokens							
Architecture	Security Model	Scheme	LAN (Online, *Sec*)	WAN (Online, *Sec*)	Prep. time (*Sec*)	Comm. (Online, GB)	Comm. (Offline, GB)
2-PC	Passive	BumbleBee [29]	111.84 (5.95×)	1849.4 (17.1×)	0	6.4	0
2-PC	Passive	SHAFT [30]	97.97 (5.2×)	1019.97 (9.4×)	–	13.41	–
3-PC	Active	Mosformer [10]	59.47 (3.16×)	143.02 (1.33×)	–	4.6	–
(2+1)-PC	Passive	SIGMA (GPU) [25]	3.08	11.8	2.25	0.99	12.83
(2+1)-PC	Active	Shark [24]	50.58 (2.69×)	160.7 (1.49×)	477.3	1.43	50.5
(2+1)-PC	Active	**Ours**	**18.8**	**107.9**	75.1	1.38	15.88
Bert-large Test with 128 tokens							
Architecture	Security Model	Scheme	LAN (Online, *Sec*)	WAN (Online, *Sec*)	Prep. time (*Sec*)	Comm. (Online, GB)	Comm. (Offline, GB)
2-PC	Passive	BumbleBee [29]	298.65 (5.05×)	2692.6 (9.08×)	0	16.37	0
2-PC	Passive	SHAFT [30]	265.2 (4.48×)	2525.8 (8.5×)	–	36.26	–
(2+1)-PC	Passive	SIGMA (GPU) [25]	7.4	40.1	7.46	2.64	40.44
(2+1)-PC	Active	**Ours**	**59.1**	**296.6**	385.8	3.7	46.8

⋆ **Comm.** denotes the online and offline communication cost; the offline communication accounts for the size of preprocessing materials received from dealer.
⋆ **Prep.time** denotes the time for the dealer to generate prepared materials.
⋆ BumbleBee [29] does not involve Prep.size and Prep.time and does not explicitly distinguish a preprocessing phase.
⋆ – indicates that we cannot distinguish between the online and preprocessing phases and cannot measure the dealer's preprocessing time.

along different technical lines, including homomorphic encryption (HE), secret sharing (SS), and function secret sharing (FSS), respectively. Moreover, SIGMA's implementation only provides a GPU-based version. To enable a comprehensive comparison, we evaluate SIGMA on GPUs, while ours is evaluated on CPUs. Results for Mosformer are taken from their paper, while others are obtained by re-running their open-source implementations.

Results for CNNs inference. Our implementation, similar to Shark [24] and MD-ML [52], relies on a dealer to generate the materials required for the online phase. Therefore, we report both the dealer's preprocessing time and the offline communication (also the size of prepared materials) to demonstrate our optimization of communication bandwidth. Table 5 compares the online performance of CNNs inference with Shark [24] and MD-ML [52]. Compared to

MD-ML [52], our AlexNet inference achieves speedups of 6.33× in LAN and 45.8× in WAN. Since our online communication cost is the same as Shark [24] and CNNs have only a few nonlinear operations (ReLU and truncation) compared with LLMs, the runtime is mainly dominated by the cost of matrix multiplication, leading to a limited speedup of up to 1.49×. The size of the prepared materials generated by the dealer is reduced up to 31%, compared to Shark [24].

Results for LLMs inference. Table 6 shows the performance of LLMs inference. Let's first consider the comparison with actively-secure works. Compared to Mosformer [10], despite being evaluated under higher communication latency, our scheme still achieves speedups ranging from 1.33× to 3.6× in both LAN and WAN. And the online communication overhead is reduced by up to 80% compared to Mosformer. Compared to Shark [24], the advantages of our compact DPF keys and reduced AES calls become increasingly apparent with larger and more complex models. We achieve speedups of 2.69× in LAN and 1.49× in WAN for BERT-base inference. By reducing the size of prepared materials by up to 68% compared to Shark [24], our framework enables actively secure inference for larger models (e.g., Bert-large) on resource-constrained platforms with limited storage and memory, and is also GPU-friendly by reducing PCIe overhead [28].

Compared with semi-honest works, our scheme still achieves significant improvement over the HE-based BumbleBee [29] and the SS-based SHAFT [30]. For GPT2-base, we achieve speedups of 5.7×–7.07× in LAN and 9.2×–15.3× in WAN; for BERT-base, up to 5.95× in LAN and 17.1× in WAN; and for BERT-large, up to 5.05× in LAN and 9.08× in WAN. Even when accounting for the communication cost of transmitting the prepared materials (For BERT-base, transmitting 15.88 GB of prepared materials in WAN theoretically takes about 635 s, resulting in a total running time of 742.9 s.), our approach still outperforms BumbleBee and SHAFT in WAN. However, compared to the closely related semi-honest work SIGMA [25], our performance is slightly inferior, mainly due to the absence of GPU acceleration and higher communication overhead for active security. Notably, despite the stronger requirement of active security, we achieve comparable or superior online performance, indicating ours practicality for real-world deployment.

6 Conclusion

This paper present LightShark, the first Grotto-style FSS-based machine-learning inference framework. We typically design a lightweight authenticated DCF scheme based on more compact DPF keys, which builds a bridge between Grotto-style DCF and SPDZ-like IT-MACs. Then, we present a series building blocks for machine-learning inference to achieve active security. We demonstrate our efficiency for practical applications by implementing machine-learning inference tasks. Benchmarking results indicate that our implementation benefits from our framework, resulting in increased efficiency for application deployment.

Acknowledgments

We thank the anonymous reviewers for their insightful comments and suggestions. This work is supported by the National Natural Science Foundation of China (Nos. 62325209, 62572355, U23A20302) and the National Key Research and Development Program of China (No. 2022YFB4400700, 2021YFA1000600). This paper was edited for grammar using ChatGPT.

Ethical Considerations

We have made every effort to address the ethical considerations necessary to ensure our work enhances data privacy while minimizing potential risks, during cloud-based machine-learning inference. Our framework provides active security against malicious adversary who may arbitrarily deviate from the protocol, while ensuring that servers' outputs are generated correctly without compromising the privacy of either the model parameters or inference data. We strictly follow ethical guidelines, avoiding deceptive practices, unauthorized disclosures, live system experiments, or any actions that could jeopardize the well-being of us. All evaluations of our work are conducted in a controlled, simulated environment to prevent any unintended impact on actual users or deployed services. And we aim to further improve the data privacy in machine-learning inference services without introducing new ethical concerns beyond those already associated with machine-learning.

Open Science

Our source-code is packaged as a single self-contained open-source library and can be found from the Github repository https://github.com/Crypto-Lab-WHU/LightShark-Opensource.

References

[1] Amit Agarwal, Carsten Baum, Lennart Braun, and Peter Scholl. 2025. Low-Bandwidth Mixed Arithmetic in VOLE-Based ZK from Low-Degree PRGs. In *EUROCRYPT 2025, Part IV (LNCS, Vol. 15604)*, Serge Fehr and Pierre-Alain Fouque (Eds.). Springer, Cham, Switzerland, Madrid, Spain, 396–426. https://doi.org/10.1007/978-3-031-91134-7_14

[2] Amit Agarwal, Elette Boyle, Nishanth Chandran, Niv Gilboa, Divya Gupta, Yuval Ishai, Mahimna Kelkar, and Yiping Ma. 2024. Secure Sorting and Selection via Function Secret Sharing. In *ACM CCS 2024*, Bo Luo, Xiaojing Liao, Jun Xu, Engin Kirda, and David Lie (Eds.). ACM Press, Salt Lake City, UT, USA, 3023–3037. https://doi.org/10.1145/3658644.3690359

[3] Yoshimasa Akimoto, Kazuto Fukuchi, Youhei Akimoto, and Jun Sakuma. 2023. Privformer: Privacy-preserving Transformer with MPC. In *2023 IEEE European Symposium on Security and Privacy*. IEEE Computer Society Press, Delft, The Netherlands, 392–410. https://doi.org/10.1109/EuroSP57164.2023.00031

[4] Aner Ben-Efraim, Michael Nielsen, and Eran Omri. 2019. Turbospeedz: Double Your Online SPDZ! Improving SPDZ Using Function Dependent Preprocessing. In *ACNS 2019 (LNCS, Vol. 11464)*, Robert H. Deng, Valérie Gauthier-Umaña, Martín Ochoa, and Moti Yung (Eds.). Springer, Cham, Switzerland, Bogota, Colombia, 530–549. https://doi.org/10.1007/978-3-030-21568-2_26

[5] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. 2011. Semi-homomorphic Encryption and Multiparty Computation. In *EUROCRYPT 2011 (LNCS, Vol. 6632)*, Kenneth G. Paterson (Ed.). Springer Berlin Heidelberg, Germany, Tallinn, Estonia, 169–188. https://doi.org/10.1007/978-3-642-20465-4_11

[6] Elette Boyle, Nishanth Chandran, Niv Gilboa, Divya Gupta, Yuval Ishai, Nishant Kumar, and Mayank Rathee. 2021. Function Secret Sharing for Mixed-Mode and Fixed-Point Secure Computation. In *EUROCRYPT 2021, Part II (LNCS, Vol. 12697)*, Anne Canteaut and François-Xavier Standaert (Eds.). Springer, Cham, Switzerland, Zagreb, Croatia, 871–900. https://doi.org/10.1007/978-3-030-77886-6_30

[7] Elette Boyle, Niv Gilboa, and Yuval Ishai. 2015. Function Secret Sharing. In *EUROCRYPT 2015, Part II (LNCS, Vol. 9057)*, Elisabeth Oswald and Marc Fischlin (Eds.). Springer Berlin Heidelberg, Germany, Sofia, Bulgaria, 337–367. https://doi.org/10.1007/978-3-662-46803-6_12

[8] Elette Boyle, Niv Gilboa, and Yuval Ishai. 2016. Function Secret Sharing: Improvements and Extensions. In *ACM CCS 2016*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM Press, Vienna, Austria, 1292–1303. https://doi.org/10.1145/2976749.2978429

[9] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[10] Ke Cheng, Yuheng Xia, Anxiao Song, Jiaxuan Fu, Wenjie Qu, Yulong Shen, and Jiaheng Zhang. 2025. Mosformer: Maliciously Secure Three-Party Inference Framework for Large Transformers. In *ACM CCS 2025*, Chun-Ying Huang, Jyh-Cheng Chen, Shiuh-Pyng Shieh, David Lie, and Véronique Cortier (Eds.). ACM Press, Taipei, Taiwan, 4124–4138. https://doi.org/10.1145/3719027.3765028

[11] Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. 2018. SPD $\mathbb{Z}_{2^k}$: Efficient MPC mod 2^k for Dishonest Majority. In *CRYPTO 2018, Part II (LNCS, Vol. 10992)*, Hovav Shacham and Alexandra Boldyreva (Eds.). Springer, Cham, Switzerland, Santa Barbara, CA, USA, 769–798. https://doi.org/10.1007/978-3-319-96881-0_26

[12] Hongrui Cui, Xiao Wang, Kang Yang, and Yu Yu. 2025. Actively Secure Half-Gates with Minimum Overhead under Duplex Networks. *Journal of Cryptology* 38, 2 (April 2025), 19. https://doi.org/10.1007/s00145-025-09539-4

[13] Ivan Damgård, Daniel Escudero, Tore Kasper Frederiksen, Marcel Keller, Peter Scholl, and Nikolaj Volgushev. 2019. New Primitives for Actively-Secure MPC over Rings with Applications to Private Machine Learning. In *2019 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, San Francisco, CA, USA, 1102–1120. https://doi.org/10.1109/SP.2019.00078

[14] Ivan Damgård, Martin Geisler, and Mikkel Krøigaard. 2007. Efficient and Secure Comparison for On-Line Auctions. In *ACISP 07 (LNCS, Vol. 4586)*, Josef Pieprzyk, Hossein Ghodosi, and Ed Dawson (Eds.). Springer Berlin Heidelberg, Germany, Townsville, Australia, 416–430. https://doi.org/10.1007/978-3-540-73458-1_30

[15] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. 2013. Practical Covertly Secure MPC for Dishonest Majority - Or: Breaking the SPDZ Limits. In *ESORICS 2013 (LNCS, Vol. 8134)*, Jason Crampton, Sushil Jajodia, and Keith Mayes (Eds.). Springer Berlin Heidelberg, Germany, Egham, UK, 1–18. https://doi.org/10.1007/978-3-642-40203-6_1

[16] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. 2012. Multiparty Computation from Somewhat Homomorphic Encryption. In *CRYPTO 2012 (LNCS, Vol. 7417)*, Reihaneh Safavi-Naini and Ran Canetti (Eds.). Springer Berlin Heidelberg, Germany, Santa Barbara, CA, USA, 643–662. https://doi.org/10.1007/978-3-642-32009-5_38

[17] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*. 4171–4186.

[18] Daniel Escudero, Satrajit Ghosh, Marcel Keller, Rahul Rachuri, and Peter Scholl. 2020. Improved Primitives for MPC over Mixed Arithmetic-Binary Circuits. In *CRYPTO 2020, Part II (LNCS, Vol. 12171)*, Daniele Micciancio and Thomas Ristenpart (Eds.). Springer, Cham, Switzerland, Santa Barbara, CA, USA, 823–852. https://doi.org/10.1007/978-3-030-56880-1_29

[19] Tore Kasper Frederiksen, Marcel Keller, Emmanuela Orsini, and Peter Scholl. 2015. A Unified Approach to MPC with Preprocessing Using OT. In *ASIACRYPT 2015, Part I (LNCS, Vol. 9452)*, Tetsu Iwata and Jung Hee Cheon (Eds.). Springer Berlin Heidelberg, Germany, Auckland, New Zealand, 711–735. https://doi.org/10.1007/978-3-662-48797-6_29

[20] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. 2016. CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy. In *Proceedings of The 33rd International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 48)*, Maria Florina Balcan and Kilian Q. Weinberger (Eds.). PMLR, New York, New York, USA, 201–210. https://proceedings.mlr.press/v48/gilad-bachrach16.html

[21] Gaël Guennebaud, Benoît Jacob, et al. 2010. Eigen v3. http://eigen.tuxfamily.org.

[22] Chun Guo, Jonathan Katz, Xiao Wang, and Yu Yu. 2020. Efficient and Secure Multiparty Computation from Fixed-Key Block Ciphers. In *2020 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, San Francisco, CA, USA, 825–841. https://doi.org/10.1109/SP40000.2020.00016

[23] Xiaojie Guo, Kang Yang, Xiao Wang, Wenhao Zhang, Xiang Xie, Jiang Zhang, and Zheli Liu. 2023. Half-Tree: Halving the Cost of Tree Expansion in COT and DPF. In *EUROCRYPT 2023, Part I (LNCS, Vol. 14004)*, Carmit Hazay and Martijn Stam (Eds.). Springer, Cham, Switzerland, Lyon, France, 330–362. https://doi.org/10.1007/978-3-031-30545-0_12

[24] Kanav Gupta, Nishanth Chandran, Divya Gupta, Jonathan Katz, and Rahul Sharma. 2025. SHARK: Actively Secure Inference Using Function Secret Sharing. In *2025 IEEE Symposium on Security and Privacy*, Marina Blanton, William Enck, and Cristina Nita-Rotaru (Eds.). IEEE Computer Society Press, San Francisco, CA, USA, 2472–2490. https://doi.org/10.1109/SP61157.2025.00175

[25] Kanav Gupta, Neha Jawalkar, Ananta Mukherjee, Nishanth Chandran, Divya Gupta, Ashish Panwar, and Rahul Sharma. 2024. SIGMA: Secure GPT Inference with Function Secret Sharing. *PoPETs* 2024, 4 (Oct. 2024), 61–79. https://doi.org/10.56553/popets-2024-0107

[26] Kanav Gupta, Deepak Kumaraswamy, Nishanth Chandran, and Divya Gupta. 2022. LLAMA: A Low Latency Math Library for Secure Inference. *PoPETs* 2022, 4 (Oct. 2022), 274–294. https://doi.org/10.56553/popets-2022-0109

[27] Zhicong Huang, Wen jie Lu, Cheng Hong, and Jiansheng Ding. 2022. Cheetah: Lean and Fast Secure Two-Party Deep Neural Network Inference. In *USENIX Security 2022*, Kevin R. B. Butler and Kurt Thomas (Eds.). USENIX Association, Boston, MA, USA, 809–826. https://www.usenix.org/conference/usenixsecurity22/presentation/huang-zhicong

[28] Neha Jawalkar, Kanav Gupta, Arkaprava Basu, Nishanth Chandran, Divya Gupta, and Rahul Sharma. 2024. Orca: FSS-based Secure Training and Inference with GPUs. In *2024 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, San Francisco, CA, USA, 597–616. https://doi.org/10.1109/SP54263.2024.00063

[29] Wen jie Lu, Zhicong Huang, Zhen Gu, Jingyu Li, Jian Liu, Cheng Hong, Kui Ren, Tao Wei, and Wenguang Chen. 2025. BumbleBee: Secure Two-party Inference Framework for Large Transformers. In *NDSS 2025*. The Internet Society, San Diego, CA, USA.

[30] Andes Y. L. Kei and Sherman S. M. Chow. 2025. SHAFT: Secure, Handy, Accurate and Fast Transformer Inference. In *NDSS 2025*. The Internet Society, San Diego, CA, USA.

[31] Marcel Keller. 2020. MP-SPDZ: A Versatile Framework for Multi-Party Computation. In *ACM CCS 2020*, Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna (Eds.). ACM Press, Virtual Event, USA, 1575–1590. https://doi.org/10.1145/3372297.3417872

[32] Marcel Keller, Emmanuela Orsini, and Peter Scholl. 2016. MASCOT: Faster Malicious Arithmetic Secure Computation with Oblivious Transfer. In *ACM CCS 2016*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM Press, Vienna, Austria, 830–842. https://doi.org/10.1145/2976749.2978357

[33] Alex Krizhevsky, Geoffrey Hinton, et al. 2009. Learning multiple layers of features from tiny images. (2009).

[34] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems* 25 (2012).

[35] Nishant Kumar, Mayank Rathee, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. 2020. CrypTFlow: Secure TensorFlow Inference. In *2020 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, San Francisco, CA, USA, 336–353. https://doi.org/10.1109/SP40000.2020.00092

[36] Dacheng Li, Rulin Shao, Hongyi Wang, Han Guo, Eric P. Xing, and Hao Zhang. 2023. MPCFormer: fast, performant and private Transformer inference with MPC. arXiv:2211.01452 [cs.LG] https://arxiv.org/abs/2211.01452

[37] Yun Li, Yufei Duan, Zhicong Huang, Cheng Hong, Chao Zhang, and Yifan Song. 2023. Efficient 3PC for Binary Circuits with Application to Maliciously-Secure DNN Inference. In *USENIX Security 2023*, Joseph A. Calandrino and Carmela Troncoso (Eds.). USENIX Association, Anaheim, CA, USA, 5377–5394. https://www.usenix.org/conference/usenixsecurity23/presentation/li-yun

[38] Jinglong Luo, Yehong Zhang, Zhuo Zhang, Jiaqi Zhang, Xin Mu, Hui Wang, Yue Yu, and Zenglin Xu. 2024. Secformer: fast and accurate privacy-preserving inference for transformer models via SMPC. In *Findings of the Association for Computational Linguistics ACL 2024*. 13333–13348.

[39] Payman Mohassel and Yupeng Zhang. 2017. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *2017 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, San Jose, CA, USA, 19–38. https://doi.org/10.1109/SP.2017.12

[40] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. 2012. A New Approach to Practical Active-Secure Two-Party Computation. In *CRYPTO 2012 (LNCS, Vol. 7417)*, Reihaneh Safavi-Naini and Ran Canetti (Eds.). Springer Berlin Heidelberg, Germany, Santa Barbara, CA, USA, 681–700. https://doi.org/10.1007/978-3-642-32009-5_40

[41] Arpita Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. 2021. ABY2.0: Improved Mixed-Protocol Secure Two-Party Computation. In *USENIX Security 2021*, Michael Bailey and Rachel Greenstadt (Eds.). USENIX Association, 2165–2182. https://www.usenix.org/conference/usenixsecurity21/presentation/patra

[42] Deevashwer Rathee, Mayank Rathee, Rahul Kranti Kiran Goli, Divya Gupta, Rahul Sharma, Nishanth Chandran, and Aseem Rastogi. 2021. SiRnn: A Math Library for Secure RNN Inference. In *2021 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, San Francisco, CA, USA, 1003–1020. https://doi.org/10.1109/SP40001.2021.00086

[43] Deevashwer Rathee, Mayank Rathee, Nishant Kumar, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. 2020. CrypTFlow2: Practical 2-Party Secure Inference. In *ACM CCS 2020*, Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna (Eds.). ACM Press, Virtual Event, USA, 325–342. https://doi.org/10.1145/3372297.3417274

[44] Dragos Rotaru and Tim Wood. 2019. MArBled Circuits: Mixing Arithmetic and Boolean Circuits with Active Security. In *INDOCRYPT 2019 (LNCS, Vol. 11898)*, Feng Hao, Sushmita Ruj, and Sourav Sen Gupta (Eds.). Springer, Cham, Switzerland, Hyderabad, India, 227–249. https://doi.org/10.1007/978-3-030-35423-7_12

[45] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[46] Kyle Storrier, Adithya Vadapalli, Allan Lyons, and Ryan Henry. 2023. Grotto: Screaming fast (2+1)-PC or $\mathbb{Z}_{2^n}$ via (2, 2)-DPFs. In *ACM CCS 2023*, Weizhi Meng,

Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda (Eds.). ACM Press, Copenhagen, Denmark, 2143–2157. https://doi.org/10.1145/3576915.3623147
[47] Sameer Wagh. 2022. Pika: Secure Computation using Function Secret Sharing over Rings. *PoPETs* 2022, 4 (Oct. 2022), 351–377. https://doi.org/10.56553/popets-2022-0113
[48] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. 2016. EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit.
[49] Jean-Luc Watson, Sameer Wagh, and Raluca Ada Popa. 2022. Piranha: A GPU Platform for Secure Computation. In *USENIX Security 2022*, Kevin R. B. Butler and Kurt Thomas (Eds.). USENIX Association, Boston, MA, USA, 827–844. https://www.usenix.org/conference/usenixsecurity22/presentation/watson
[50] Tianshi Xu, Wen jie Lu, Jiangrui Yu, Yi Chen, Chenqi Lin, Runsheng Wang, and Meng Li. 2025. Breaking the Layer Barrier: Remodeling Private Transformer Inference with Hybrid CKKS and MPC. In *USENIX Security 2025*, Lujo Bauer and Giancarlo Pellegrino (Eds.). USENIX Association, Seattle, WA, USA, 2653–2672. https://www.usenix.org/conference/usenixsecurity25/presentation/xu-tianshi
[51] Andrew Chi-Chih Yao. 1986. How to Generate and Exchange Secrets (Extended Abstract). In *27th FOCS*. IEEE Computer Society Press, Toronto, Ontario, Canada, 162–167. https://doi.org/10.1109/SFCS.1986.25
[52] Boshi Yuan, Shixuan Yang, Yongxiang Zhang, Ning Ding, Dawu Gu, and Shi-Feng Sun. 2024. MD-ML: Super Fast Privacy-Preserving Machine Learning for Malicious Security with a Dishonest Majority. In *USENIX Security 2024*, Davide Balzarotti and Wenyuan Xu (Eds.). USENIX Association, Philadelphia, PA, USA. https://www.usenix.org/conference/usenixsecurity24/presentation/yuan
[53] Jiawen Zhang, Xinpeng Yang, Lipeng He, Kejia Chen, Wen jie Lu, Yinghao Wang, Xiaoyang Hou, Jian Liu, Kui Ren, and Xiaohu Yang. 2025. Secure Transformer Inference Made Non-interactive. In *NDSS 2025*. The Internet Society, San Diego, CA, USA.
[54] Wenhao Zhang, Xiaojie Guo, Kang Yang, Ruiyu Zhu, Yu Yu, and Xiao Wang. 2024. Efficient Actively Secure DPF and RAM-based 2PC with One-Bit Leakage. In *2024 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, San Francisco, CA, USA, 561–577. https://doi.org/10.1109/SP54263.2024.00205

A IT-MAC Checking for Authenticated Secret Sharings

In this section, we present the detailed open and check procedures, denoted as OpenA and OpenB, as described in Section 2.2. The IT-MAC checking schemes are taken from SPD$\mathbb{Z}_{2^k}$ [11], MASCOT [32] and Shark [24].

A.1 OpenA Procedure

In our protocols, since all values to be reconstructed have already been properly randomized, we simplify the open phase of SPD$\mathbb{Z}_{2^k}$ [11]: the two parties only need to exchange their shares without re-randomization. The details are presneted in Figure 9. As for the correctness checking, let $\{\langle x_i\rangle^{\mathsf{A}}\}_{i\in[\ell]}$ be Arithmetic authenticated sharings over $\mathbb{Z}_{2^{n+s}}$, and let $\{\hat{x}_i\}_{i\in[\ell]}$ be the corresponding reconstructed values over $\mathbb{Z}_{2^{n+s}}$. To verify the correctness of all opened values in a batch, the parties jointly sample public random values $\chi_1,\ldots,\chi_\ell \leftarrow \mathbb{Z}_{2^s}$. For each $i \in [\ell]$, party P_b, where $b \in \{0,1\}$, locally computes $z_b^{(i)} := [[m_{x_i}]]_b^{\mathsf{A}} - \tilde{x}_i \cdot [[\alpha]]_b^{\mathsf{A}} \bmod 2^{n+s}$. Then P_b computes $z_b := \sum_{i\in[\ell]} \chi_i \cdot z_b^{(i)} \bmod 2^{n+s}$. The parties commit to z_b, open the commitments, and accept if $z_0 + z_1 = 0 \bmod 2^{n+s}$; otherwise, they abort. The details of OpenA procedure are presented in Figure 9

A.2 OpenB Procedure

The OpenB procedure is presneted in Figure 10. As for the correctness checking, let $\{\langle y_i\rangle^{\mathsf{B}}\}_{i\in[\ell]}$ be Boolean authenticated sharings, and let $\{y_i\}_{i\in[\ell]}$ be the corresponding opened bits. To check their correctness in a batch, the parties jointly sample public random values $\chi_1,\ldots,\chi_\ell \leftarrow \mathbb{F}_{2^\kappa}$. For each $i \in [\ell]$, party P_b, where $b \in \{0,1\}$, locally computes $z_b^{(i)} := [[m_{y_i}]]_b^{\mathsf{B}} \oplus y_i \cdot [[\Delta]]_b^{\mathsf{B}} \in \mathbb{F}_{2^\kappa}$. Then P_b computes $z_b := \sum_{i\in[\ell]} \chi_i \cdot z_b^{(i)} \in \mathbb{F}_{2^\kappa}$. The parties commit to z_b, open the commitments, and accept if $z_0 = z_1$; otherwise, abort.

Procedure OpenA

P_b, for $b \in \{0,1\}$, opens and checks a batch of arithmetic authenticated sharings $\{\langle x_i\rangle^{\mathsf{A}}\}_{i\in[\ell]}$, where $\langle x_i\rangle^{\mathsf{A}} = \{[[x_i]]^{\mathsf{A}}, [[m_{x_i}]]^{\mathsf{A}}, [[\alpha]]^{\mathsf{A}}\}$:

Open phase

(1) For each $i \in [\ell]$, P_b sends $[[x_i]]_b^{\mathsf{A}}$ to P_{1-b}.
(2) Upon receiving $[[x_i]]_{1-b}^{\mathsf{A}}$, P_b reconstructs $\hat{x}_i := [[x_i]]_0^{\mathsf{A}} + [[x_i]]_1^{\mathsf{A}} \bmod 2^{n+s}$ and outputs $x_i := \hat{x}_i \bmod 2^n$.
(3) For each $i \in [\ell]$, P_b locally computes and stores $z_b^{(i)} := [[m_{x_i}]]_b^{\mathsf{A}} - \hat{x}_i \cdot [[\alpha]]_b^{\mathsf{A}} \bmod 2^{n+s}$.

Batch Checking

(1) The parties jointly sample public random values $\chi_1,\ldots,\chi_\ell \leftarrow \mathbb{Z}_{2^s}$.
(2) P_b locally computes $z_b := \sum_{i\in[\ell]} \chi_i \cdot z_b^{(i)} \bmod 2^{n+s}$.
(3) P_b commits to z_b, and then both parties open the commitments.
(4) The parties accept if $z_0 + z_1 = 0 \bmod 2^{n+s}$; otherwise, abort.

Figure 9: Open and Check Procedure for Arithmetic Authenticated Sharings

Procedure OpenB

P_b, for $b \in \{0,1\}$, opens and checks a batch of boolean authenticated sharings $\{\langle y_i\rangle^{\mathsf{B}}\}_{i\in[\ell]}$, where $\langle y_i\rangle^{\mathsf{B}} = \{[[y_i]]^{\mathsf{B}}, [[m_{y_i}]]^{\mathsf{B}}, [[\Delta]]^{\mathsf{B}}\}$:

Open phase

(1) For each $i \in [\ell]$, P_b sends $[[y_i]]_b^{\mathsf{B}}$ to P_{1-b}.
(2) Upon receiving $[[y_i]]_{1-b}^{\mathsf{B}}$, P_b reconstructs $y_i := [[y_i]]_0^{\mathsf{B}} \oplus [[y_i]]_1^{\mathsf{B}}$.
(3) For each $i \in [\ell]$, P_b locally computes and stores $z_b^{(i)} := [[m_{y_i}]]_b^{\mathsf{B}} \oplus y_i \cdot [[\Delta]]_b^{\mathsf{B}} \in \mathbb{F}_{2^\kappa}$.

Batch Chacking

(1) The parties jointly sample public random values $\chi_1,\ldots,\chi_\ell \leftarrow \mathbb{F}_{2^\kappa}$.
(2) P_b locally computes $z_b := \sum_{i\in[\ell]} \chi_i \cdot z_b^{(i)} \in \mathbb{F}_{2^\kappa}$.
(3) P_b commits to z_b, and then both parties decommit.
(4) The parties accept if $z_0 = z_1$; otherwise, abort.

Figure 10: Open and Check Procedure for Boolean Authenticated Sharings

B $\mathcal{F}_{\text{dealer}}$-based Actively Secure 2PC Protocols

B.1 Ideal Functionality for Dealer

We provide the ideal functionality for $\mathcal{F}_{\text{dealer}}$ as shown in Figure 11. Dealer generates the prepared materials, including input-independent DCF keys and authenticated random values for different computation phase. Figure 11 presents the ideal functionality for *Initialize*, *Input Phase*, *Computation Phase*, and *Output Phase*. The details of the *Computation Phase* are provided in the protocol design as a supplement to $\mathcal{F}_{\text{dealer}}$.

Functionality $\mathcal{F}_{\text{dealer}}$

$\mathcal{F}_{\text{dealer}}$ interacts with the servers P_0, P_1 to generate prepared materials as follows:

Initialize: On input (Init) from P_b, $b \in \{0,1\}$.

- Sample uniform arithmetic and boolean authenticated key $\alpha \in \mathbb{Z}_{2^s}$ and $\Delta \in \mathbb{F}_{2^\kappa}$.
- Sample uniform $[\![\alpha]\!]_0^{\mathsf{A}}$, $[\![\alpha]\!]_1^{\mathsf{A}} \in \mathbb{Z}_{2^s}$ and $[\![\Delta]\!]_0^{\mathsf{B}}$, $[\![\Delta]\!]_1^{\mathsf{B}} \in \mathbb{F}_{2^\kappa}$, satisfying $\alpha = [\![\alpha]\!]_0^{\mathsf{A}} + [\![\alpha]\!]_1^{\mathsf{A}} \mod 2^{n+s}$,$\Delta = [\![\Delta]\!]_0^{\mathsf{B}} \oplus [\![\Delta]\!]_1^{\mathsf{B}}$.
- Send $[\![\alpha]\!]_b^{\mathsf{A}}$ and $[\![\Delta]\!]_b^{\mathsf{B}}$ to P_b.

Input: On input $(\mathsf{Input}, b, \mathcal{D} \in \{\mathsf{A}, \mathsf{B}\})$ from P_b, $b \in \{0,1\}$..

- If $\mathcal{D} = \mathsf{A}$, sample uniform mask $\hat{\delta}_x \in \mathbb{Z}_{2^{n+s}}$ and store it. Send $\delta_x = \hat{\delta}_x \mod 2^n$ to P_b.
- If $\mathcal{D} = \mathsf{B}$, sample uniform mask $\delta_x \in \mathbb{F}_2$ and store it. Send it to P_b.

Computation: On input $(\mathsf{Func}, \mathcal{F})$ from P_b, $b \in \{0,1\}$, where $\mathcal{F} \in \{\mathsf{DReLU}, \mathsf{Spline}, \dots\}$ corresponds to a specific sub-protocol such as DReLU and Spline. As an example of a single-input/output sub-protocol, assume that the input wire is x and the output wire is y, and the sub-protocol computes $y = f(x)$.

- Restore the input masks $\hat{\delta}_x \in \mathbb{Z}_{2^{n+s}}$ (or $\delta_x \in \mathbb{F}_2$) of sub-protocol $\mathcal{F}$.
- Sample uniform output masks $\hat{\delta}_y \in \mathbb{Z}_{2^{n+s}}$ (or $\delta_y \in \mathbb{F}_2$) of sub-protocol $\mathcal{F}$ and store them.
- Compute the prepared materials required for online protocols and send them to P_b.

Output: On input $(\mathsf{Output}, \mathcal{D} \in \{\mathsf{A}, \mathsf{B}\})$ from P_b, $b \in \{0,1\}$.

- If $\mathcal{D} = \mathsf{A}$, restore the output mask $\hat{\delta}_y \in \mathbb{Z}_{2^{n+s}}$. Execute $\langle -\hat{\delta}_y \rangle^{\mathsf{A}} \leftarrow \mathsf{ShareA}(\hat{\delta}_y)$ and send it to P_b.
- If $\mathcal{D} = \mathsf{B}$, restore the output mask $\delta_y \in \mathbb{F}_2$. Execute $\langle \delta_y \rangle^{\mathsf{B}} \leftarrow \mathsf{ShareB}(\delta_y)$ and send it to P_b.

Figure 11: Ideal Functionality for the Dealer

B.2 Multiplication Protocol

For the arithmetic multiplication with masked inputs $\Lambda_x = x + \delta_x$ and $\Lambda_y = y + \delta_y$ over $\mathbb{Z}_{2^n}$, we can observe that

$$\begin{aligned} z = \mathsf{Mul}(x,y) &= (\Lambda_x - \delta_x)(\Lambda_y - \delta_y) \\ &= \Lambda_x \cdot \Lambda_y - \Lambda_x \cdot \delta_y - \delta_x \cdot \Lambda_y + \delta_x \cdot \delta_y \mod 2^n. \end{aligned}$$

Also, $\Lambda_z = z + \delta_z \mod 2^n$. Therefore, we can prepare the arithmetic authenticated sharings of $\delta_x \cdot \delta_y$ and locally compute the output shares during online phase. The detailed protocol for arithmetic multiplication is shown in Figure 12, which requires 3 arithmetic authenticated shares for prepared materials. While the online phase requires one round communication and $n + s$ bits sent by per party.

Theorem 4. *Figure 12 shows a actively secure multiplication protocol against the malicious adversary who can corrupt one of two servers P_0 and P_1 in $\mathcal{F}_{\mathsf{PrepMul}}$-hybrid model.*

We provide the detailed proof in Appendix D.4.2.

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

The preprocessing functionality has all the same features as $\mathcal{F}_{\text{dealer}}$, with the additional commands:

ReLU: On input $(\mathsf{Func}, \mathsf{Mul})$ from P_b, $b \in \{0,1\}$, $\mathcal{F}_{\mathsf{PrepMul}}$ executes as follows:

(1) Restore the input mask $\hat{\delta}_x \in \mathbb{Z}_{2^{n+s}}$ and $\hat{\delta}_y \in \mathbb{Z}_{2^{n+s}}$. Sample uniform output mask $\hat{\delta}_z \in \mathbb{Z}_{2^{n+s}}$ and store it.
(2) Compute $\delta_{xy} = \hat{\delta}_x \cdot \hat{\delta}_y + \hat{\delta}_z \mod 2^{n+s}$.
(3) Execute $\langle \hat{\delta}_x \rangle^{\mathsf{A}} \leftarrow \mathsf{ShareA}(\hat{\delta}_x)$, $\langle \hat{\delta}_y \rangle^{\mathsf{A}} \leftarrow \mathsf{ShareA}(\hat{\delta}_y)$ and $\langle \hat{\delta}_{xy} \rangle^{\mathsf{A}} \leftarrow \mathsf{ShareA}(\hat{\delta}_{xy})$.

Protocol Π_{Mul}

Preprocessing Phase: Server P_b, $b \in \{0,1\}$ invokes $\mathcal{F}_{\mathsf{PrepMul}}$ with $(\mathsf{Func}, \mathsf{Mul})$ to obtain the prepared materials $\langle \hat{\delta}_{xy} \rangle^{\mathsf{A}}$, $\langle \hat{\delta}_y \rangle^{\mathsf{A}}$, and $\langle \hat{\delta}_x \rangle^{\mathsf{A}}$ for multiplication.

Online Phase: P_b computes as follows with inputs Λ_x and prepared materials from $\mathcal{F}_{\mathsf{PrepMul}}$:

1. Locally compute $\langle \Lambda_z \rangle^{\mathsf{A}} = \Lambda_x \cdot \Lambda_y - \Lambda_x \cdot \langle \hat{\delta}_x \rangle^{\mathsf{A}} - \Lambda_y \cdot \langle \hat{\delta}_y \rangle^{\mathsf{A}} + \langle \hat{\delta}_{xy} \rangle^{\mathsf{A}}$.
2. Execute $\mathsf{OpenA}(\langle \Lambda_z \rangle^{\mathsf{A}})$ to obtain Λ_z.

Figure 12: Actively-secure arithmetic multiplication

B.3 ReLU Protocol

As for ReLU function, we have $\mathsf{ReLU}(x) = x \cdot \mathsf{DReLU}(x)$. However, we cannot directly multiply x with $\mathsf{DReLU}(x)$ because the output of Π_{DReLU} is a masked bit $\Lambda_y = y \oplus \delta_y \in \{0,1\}$. We need convert this bit to the arithmetic domain by computing $y = \mathsf{DReLU}(x) = \Lambda_y \oplus \delta_y = \Lambda_y + (1 - 2 \cdot \Lambda_y) \cdot \delta_y$ over $\mathbb{Z}_{2^n}$. Let $\Lambda_x = x + \delta_x \mod 2^n$.

We have

$$\begin{aligned}\mathsf{ReLU}(x) &= (\Lambda_x - \delta_x)\cdot \mathsf{DReLU}(x) \mod 2^n\\ &= (\Lambda_x - \delta_x)\cdot(\Lambda_y + (1-2\cdot\Lambda_y)\cdot\delta_y) \mod 2^n\\ &= \Lambda_x\cdot\Lambda_y + (1-2\cdot\Lambda_y)\cdot\Lambda_x\cdot\delta_y - \Lambda_y\cdot\delta_x\\ &\quad - (1-2\cdot\Lambda_y)\cdot\delta_x\cdot\delta_y.\end{aligned}$$

The detailed description of ReLU protocol is shown in Figure 13, which requires 4 additional arithmetic authenticated shares for prepared materials compared with Π_{DReLU}. While the online phase requires communication overhead of $1+(n+s)$ bits per party in two rounds for the reconstruction of one boolean authenticated share and one arithmetic authenticated share and $n-1$ AES calls for DCF.Eval.

Theorem 5. *Figure 13 shows a actively secure ReLU protocol against the malicious adversary who can corrupt one of two servers P_0 and P_1 in $\mathcal{F}_{\mathsf{PrepReLU}}$-hybrid model.*

We provide the detailed proof in Appendix D.4.3.

B.3.1 ReLU Variant. Considering that $\mathsf{ReLU}(x) = x\cdot\mathsf{DReLU}(x)$, we can construct a ReLU variant that computes $z\cdot\mathsf{DReLU}(x)$, taking x and z as inputs, named $\mathsf{Select}(x,z)$. This variant outputs z conditioned on the sign of the input x. We also have $\mathsf{Select}(x,x) = \mathsf{ReLU}(x)$. As in the Π_{ReLU}, we still need to convert the output of DReLU into the arithmetic domain by computing $y = \mathsf{DReLU}(x) = \Lambda_y \oplus \delta_y = \Lambda_y + (1-2\cdot\Lambda_y)\cdot\delta_y$ over $\mathbb{Z}_{2^n}$. Let $\Lambda_x = x+\delta_x \mod 2^n$, $\Lambda_z = z+\delta_z \mod 2^n$. We have

$$\begin{aligned}\mathsf{Select}(x,z) &= z\cdot\mathsf{DReLU}(x)\\ &= (\Lambda_z-\delta_z)\cdot(\Lambda_y+(1-2\cdot\Lambda_y)\cdot\delta_y) \mod 2^n\\ &= \Lambda_z\cdot\Lambda_y + (1-2\cdot\Lambda_y)\cdot\Lambda_z\cdot\delta_y - \Lambda_y\cdot\delta_z\\ &\quad - (1-2\cdot\Lambda_y)\cdot\delta_z\cdot\delta_y.\end{aligned}$$

We denote the ReLU variant protocol as Π_{Select}. Since Π_{Select} follows the same protocol as Π_{ReLU}, with the only change being the replacement of x by z in the multiplication step, we omit the formal description of the ReLU variant protocol.

B.4 Truncation Protocol

We utilize Fixed-Point Arithmetic (FPA) in our actively-secure inference, encoding rational numbers into $\mathbb{Z}_{2^n}$ using 2's complement. We define the last f bits of a fixed-point number as the fractional part, therefore, in order to ensure consistent precision for FPA, we need to perform truncation operations (also known as arithmetic right shifts) to adjust the precision.

B.4.1 Our truncation with 1-bit gap. Previous work [39, 41] using probability truncation has a large gap between the secret value and the actual modulus, requiring the plaintext value $x \in [0, 2^{\ell_x}) \cup [2^n - 2^{\ell_x}, 2^n)$ where $\ell_x \ll n$. We aim to reduce this gap to 1-bit to achieve faithful truncation with only one DCF keys and one additional communication round except for the final reconstruction.

Following the approach in [18], we directly adapt it in our $\mathcal{F}_{\mathsf{dealer}}$-2PC setting. When we truncate $x \in [0, 2^{n-2}) \cup [2^n - 2^{n-2}, 2^n)$ with f bits, we have $\mathsf{ARS}(x,f) = \mathsf{LRS}(x+2^{n-2}, f) - 2^{n-2-f}$ over $\mathbb{Z}_{2^n}$, where ARS and LRS denote arithmetic right shift and logical right shift, respectively. In our $\mathcal{F}_{\mathsf{dealer}}$-2PC setting, servers hold masked value $\Lambda_x = x+\delta_x \mod 2^n$. Then, we have $\mathsf{LRS}(x+2^{n-2}, f) = (\mathsf{LRS}(\Lambda_x + 2^{n-2}, f) - \mathsf{LRS}(\delta_x, f) + 2^{n-f}\cdot 1\{\Lambda_x + 2^{n-2} < \delta_x\} - 1\{(\Lambda_x + 2^{n-2} \mod 2^f) < (\delta_x \mod 2^f)\}) \mod 2^n$. Note that all LRS operations are performed over $\mathbb{Z}_{2^n}$. To determine if $1\{(\Lambda_x + 2^{n-2} \mod 2^f) < (\delta_x \mod 2^f)\}$ holds, we can easily achieve it utilizing our authenticated DCF with *daBits* $\{\langle r\rangle^{\mathsf{A}}, \langle r\rangle^{\mathsf{B}}\}$(to convert the DCF output to arithmetic domain). While as for $1\{\Lambda_x + 2^{n-2} < \delta_x\}$ term, we can complete it without DCF keys by the following Lemma 2.

Lemma 2. *For ring $\mathbb{Z}_{2^n}$, let $z \in [0, 2^{n-1})$. Given $z = c - r$,*

$$\begin{aligned}1\{c<r\} &= 1\{\mathsf{msb}(c)=0\} \cap 1\{\mathsf{msb}(r)=1\}\\ &= (1-\mathsf{msb}(c))\cdot\mathsf{msb}(r).\end{aligned}$$

Proof. To compute $1\{c<r\}$, consider the following four cases:

(1) $\mathsf{msb}(c)=1$ and $\mathsf{msb}(r)=0$.
Since $c \geq 2^{n-1} > r$, it follows that $1\{c<r\}=0$.
(2) $\mathsf{msb}(c)=0$ and $\mathsf{msb}(r)=1$.
Since $c < 2^{n-1} \leq r$, we have $1\{c<r\}=1$.
(3) $\mathsf{msb}(c)=\mathsf{msb}(r)=0$.
Because $z < 2^{n-1}$ and $r < 2^{n-1}$, and $z+r < 2^n$, we have $c = z+r \mod 2^n = z+r$, which implies $1\{c<r\}=0$.

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

The preprocessing functionality has all the same features as $\mathcal{F}_{\mathsf{dealer}}$, with the additional commands:

ReLU: On input (Func, ReLU) from P_b, $b\in\{0,1\}$, $\mathcal{F}_{\mathsf{PrepReLU}}$ executes as follows:

(1) Restore the input mask $\hat{\delta}_x \in \mathbb{Z}_{2^{n+s}}$. Sample the uniform output mask $\hat{\delta}_z \in \mathbb{Z}_{2^{n+s}}$ and store it.
(2) Execute the same procedure as in $\mathcal{F}_{\mathsf{PrepDReLU}}$ to obtain $\delta_y \in \mathbb{F}_2$, and send the prepared materials of Π_{DReLU} to P_b.
(3) Compute $\delta_{xy} = \hat{\delta}_x\cdot\delta_y \mod 2^n$.
(4) Execute $\langle\delta_{xy}\rangle^{\mathsf{A}} \leftarrow \mathsf{ShareA}(\delta_{xy})$, $\langle\hat{\delta}_z\rangle^{\mathsf{A}} \leftarrow \mathsf{ShareA}(\hat{\delta}_z)$, $\langle\delta_y\rangle^{\mathsf{A}} \leftarrow \mathsf{ShareA}(\delta_y)$ and $\langle\hat{\delta}_x\rangle^{\mathsf{A}} \leftarrow \mathsf{ShareA}(\hat{\delta}_x)$.

Protocol Π_{ReLU}

Preprocessing Phase: Server P_b, $b \in \{0,1\}$ invokes $\mathcal{F}_{\mathsf{PrepReLU}}$ with (Func, ReLU) to obtain the prepared materials of Π_{DReLU}, $\langle\delta_{xy}\rangle^{\mathsf{A}}$, $\langle\hat{\delta}_z\rangle^{\mathsf{A}}$, $\langle\delta_y\rangle^{\mathsf{A}}$, and $\langle\hat{\delta}_x\rangle^{\mathsf{A}}$ for ReLU.

Online Phase: P_b computes as follows with inputs Λ_x and prepared materials from $\mathcal{F}_{\mathsf{PrepReLU}}$:

(1) Execute the online phase of Π_{DReLU} to obtain Λ_y where $\Lambda_y = \mathsf{DReLU}(x)\oplus\delta_y \in \{0,1\}$.
(2) Locally compute $\langle\Lambda_z\rangle^{\mathsf{A}} = \Lambda_x\cdot\Lambda_y + (1-2\cdot\Lambda_y)\cdot\Lambda_x\cdot\langle\delta_y\rangle^{\mathsf{A}} - \Lambda_y\cdot\langle\hat{\delta}_x\rangle^{\mathsf{A}} - (1-2\cdot\Lambda_y)\cdot\langle\delta_{xy}\rangle^{\mathsf{A}} + \langle\hat{\delta}_z\rangle^{\mathsf{A}}$.
(3) Execute $\mathsf{OpenA}(\langle\Lambda_z\rangle^{\mathsf{A}})$ to obtain Λ_z.

Figure 13: Actively-secure ReLU Function

(4) $\mathsf{msb(c)} = \mathsf{msb(r)} = 1$.
Since $z < 2^{n-1}$ and $2^{n-1} \le r < 2^n$, we have $z + r \in [2^{n-1}, 2^n + 2^{n-1})$. Noting that $c \ge 2^{n-1}$ and $z + r < 2^n$, it follows that$c = z + r \bmod 2^n = z + r \ge r$, and hence $1\{c < r\} = 0$.

Therefore, $1\{c < r\} = 1\{\mathsf{msb(c)} = 0 \text{ and } \mathsf{msb(r)} = 1\} = \mathsf{msb}(r) \cdot (1 - \mathsf{msb}(c))$. □

We instantiate using $c = \Lambda_x$ and $r = \delta_x$, and observe that the condition $x + 2^{n-2} \in [0, 2^{n-1})$ always holds due to $x \in [0, 2^{n-2}) \cup [2^n - 2^{n-2}, 2^n)$. Therefore, we only consider $\mathsf{msb}(\Lambda_x + 2^{n-2})$ and $\mathsf{msb}(\delta_x)$. The detailed description is illustrated in Figure 14, which requires prepared materials of $6 \cdot (n + s) + (\kappa + 1) + \text{Keysize}$ bits for three arithmetic authenticated shares, one boolean authenticated share and one DCF keys. While the online phase requires communication overhead of $(n + s + 1)$ bits per party in two rounds for the reconstruction of one arithmetic authenticated share and one boolean authenticated share.

Theorem 6. *Figure 14 shows a actively secure truncation protocol against the malicious adversary who can corrupt one of two servers P_0 and P_1 in $\mathcal{F}_{\mathsf{PrepTrunc}}$-hybrid model.*

We provide the detailed proof in Appendix D.4.4.

Truncation after ReLU without 1-bit gap. As stated in Lemma 2, we find that the range requirement for the truncation value x only needs to be restricted to $[0, 2^{n-1})$. Meanwhile, $\mathsf{ReLU}(x) = \max(x, 0) \in [0, 2^{n-1})$, there is no need to limit the range of x in $\mathbb{Z}_{2^n}$, thus reducing the 1-bit gap mentioned above. Therefore, in machine learning inference tasks, we can appropriately exchange the positions of ReLU and truncation for this optimization. We denote the combination of ReLU and truncation as TruncReLU.

B.4.2 Probabilistic truncation with 1-bit error. We observe that when the 1-bit truncation error does not need to be considered, the truncation protocol no longer requires DCF keys and just involves 1 communication round. Although Li et al. [37] point out that some probabilistic truncation protocols are not secure in the simulation-based sense. We can prove the security of probabilistic truncation by slightly modifying the functionality of the truncation protocol. This functionality was introduced in BLB [50] and allows additional leakage of random f bits to the adversary. This leakage is reasonable in our protocol, since servers hold the public masked value $\Lambda_x = x + \delta_x \bmod 2^n$ which adversary can also see the uniform $\Lambda_x \bmod 2^f$.

We conduct additional experiments and find that, when probabilistic truncation is employed, our implementation for LLMs reduces communication overhead (including online communication and prepared materials) by up to 87% compared to Shark [24].

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

The preprocessing functionality has all the same features as $\mathcal{F}_{\mathsf{dealer}}$, with the additional commands:

ReLU: On input (Func, Trunc) from P_b, $b \in \{0, 1\}$, $\mathcal{F}_{\mathsf{PrepTrunc}}$ executes as follows:

(1) Restore the input mask $\hat{\delta}_x \in \mathbb{Z}_{2^{n+s}}$. Sample uniform output mask $\hat{\delta}_{\widetilde{x}} \in \mathbb{Z}_{2^{n+s}}$ and store it.
(2) Compute $\delta_x = \hat{\delta}_x \bmod 2^n$.
(3) Execute $(k_0, k_1) \leftarrow \mathsf{DCF.Gen}(1^\lambda, \Delta, (\delta_x \bmod 2^f), f)$ and send k_b to P_b, $b \in \{0, 1\}$.
(4) Compute $\widetilde{\delta_x} = \mathsf{LRS}(\delta_x, f)$ and $\delta_{msb} = \mathsf{msb}(\delta_x)$ over $\mathbb{Z}_{2^n}$.
(5) Compute $\widetilde{\delta_x} = \widetilde{\delta_x} - \hat{\delta}_{\widetilde{x}} \bmod 2^{n+s}$.
(6) Sample uniform bit $r \in \mathbb{F}_2$.
(7) Execute $\langle r\rangle^{\mathsf{B}} \leftarrow \mathsf{ShareB}(r)$, $\langle r\rangle^{\mathsf{A}} \leftarrow \mathsf{ShareA}(r)$, $\langle\widetilde{\delta_x}\rangle^{\mathsf{A}} \leftarrow \mathsf{ShareA}(\widetilde{\delta_x})$, $\langle\delta_{msb}\rangle^{\mathsf{A}} \leftarrow \mathsf{ShareA}(\delta_{msb})$.

Protocol Π_{Trunc}

Preprocessing Phase: Server P_b, $b \in \{0, 1\}$ invokes $\mathcal{F}_{\mathsf{PrepTrunc}}$ with (Func, Trunc) to obtain the prepared materials $\langle r\rangle^{\mathsf{B}}$,$\langle r\rangle^{\mathsf{A}}$, $\langle\widetilde{\delta_x}\rangle^{\mathsf{A}}$, $\langle\delta_{msb}\rangle^{\mathsf{A}}$ and k_b.

Online Phase: P_b computes as follows with inputs Λ_x, truncation bits f and prepared materials from $\mathcal{F}_{\mathsf{PrepTrunc}}$:

(1) Compute $t_{in} = \Lambda_x + 2^{n-2} \bmod 2^f$ and execute $\langle t_{out}\rangle^{\mathsf{B}} \leftarrow \mathsf{DCF.Eval}(b, t_{in}, k_b)$.
(2) Execute $\mathsf{OpenB}(\langle t_{out}\rangle^{\mathsf{B}} \oplus \langle r\rangle^{\mathsf{B}})$ to obtain t, where $t = t_{out} \oplus r$.
(3) Locally compute $\langle e\rangle^{\mathsf{A}} = t + \langle r\rangle^{\mathsf{A}} - 2 \cdot t \cdot \langle r\rangle^{\mathsf{A}}$.
(4) Compute $\widetilde{\Lambda_x} = \mathsf{LRS}(\Lambda_x + 2^{n-2}, f)$ and $\Lambda_{msb} = 1 - \mathsf{msb}(\Lambda_x + 2^{n-2})$ over $\mathbb{Z}_{2^n}$.
(5) Locally compute $\langle\Lambda_{\widetilde{x}}\rangle^{\mathsf{A}} = \widetilde{\Lambda_x} - \langle\widetilde{\delta_x}\rangle^{\mathsf{A}} + 2^{n-f} \cdot \Lambda_{msb} \cdot \langle\delta_{msb}\rangle^{\mathsf{A}} - 2^{n-2-f} - \langle e\rangle^{\mathsf{A}}$.
(6) Execute $\mathsf{OpenA}(\langle\Lambda_{\widetilde{x}}\rangle^{\mathsf{A}})$ to obtain $\Lambda_{\widetilde{x}}$.

Figure 14: Actively-secure Truncation

C Privacy-Preserving ML Inference

Given the $\mathcal{F}_{\mathsf{dealer}}$-based 2PC protocols (placed in section 4 and Appendix B), we can easily construct the building blocks for machine learning inference. It is worth noting that, benefiting from our lightweight authenticated DCF scheme, the protocol construction techniques commonly adopted in semi-honest machine learning inference works [10, 25, 28, 46, 49] can still be leveraged to design actively secure protocols. We provide a brief technical overview of how to construct machine-learning building blocks from our $\mathcal{F}_{\mathsf{dealer}}$-based 2PC protocols. In this section, we denote $\mathbf{x} \in \mathbb{R}^d$ as the vector, $\mathbf{W} \in \mathbb{R}^{m \times d}$ as the matrix, $\mathbf{Wx}$ as matrix multiplication with $\mathbf{W}$ and $\mathbf{x}$, and $\odot$ as the Hadamard product operator.

C.1 Neural networks.

C.1.1 Architecture. A neural network is a parameterized function composed of multiple layers that alternately apply linear transformations and nonlinear activation functions. Given an input vector $\mathbf{x} \in \mathbb{R}^d$, a neural network layer computes $\mathbf{h} = \phi(\mathbf{Wx} + \mathbf{b})$, where $\mathbf{W} \in \mathbb{R}^{m \times d}$ and $\mathbf{b} \in \mathbb{R}^m$ denote learnable parameters, and $\phi(\cdot)$ is a nonlinear activation function. Stacking such layers yields a deep model capable of representing complex functions. Common nonlinear components include activation functions such as ReLU,

Sigmoid and Tanh functions. In addition, architectures may incorporate nonlinear pooling operators (e.g., max-pooling) to improve invariance and feature abstraction.

C.1.2 Building Blocks. The required building blocks are relatively simple for neural networks, consisting mainly of matrix multiplication with truncation protocols and corresponding activation function protocols (e.g., ReLU, Sigmoid, and Tanh). We can easily achieve these building blocks utilizing Π_{Trunc} (Appendix B.4) Π_{ReLU} (Appendix B.3) and Π_{Spline} (Section 4.3). In our implementations, the Sigmoid function is approximated using a degree-2 spline with 19 intervals, achieving a ULP error of at most 4, while the Tanh function is approximated using a degree-2 spline with 20 intervals and a ULP error of at most 6. Although these spline approximations may not fully cover the entire input domain, they can be adjusted according to the input range of a specific model to achieve acceptable accuracy. Therefore, our protocols can cover almost all the building blocks required for neural networks.

C.2 Large Language Models.

C.2.1 Architecture. Large language models (LLMs) are predominantly based on the Transformer architecture, which is composed of a stack of blocks integrating Self-Attention and Feed-Forward Network (FFN) layers. Given a sequence of token embeddings $\mathbf{X} \in \mathbb{R}^{n\times d}$, each Transformer block first applies multi-head self-attention,

$$\text{Attention}(\mathbf{Q},\mathbf{K},\mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right)\mathbf{V},$$

where the query, key, and value matrices $\mathbf{Q}$, $\mathbf{K}$, and $\mathbf{V}$ are obtained via linear projections of $\mathbf{X}$.

The softmax operation introduces essential nonlinear components such as exponentiation, normalization, and division. For numerical stability, the softmax function is commonly expressed in the shifted form. For a vector $\mathbf{x} \in \mathbb{R}^k$ and $x_{\max} = \max(x[0], x[1], \ldots, x[k-1])$, the softmax function maps $\mathbf{x}$ to a vector $\mathbf{y} \in \mathbb{R}^k$ such that

$$y[i] = \frac{\exp(x[i] - x_{\max})}{\sum_{j=0}^{k-1}\exp(x[j] - x_{\max})}, \quad i = 0, \ldots, k-1.$$

Additionally, each block further contains a feed-forward layer, $\text{FFN}(\mathbf{x}) = \phi(\mathbf{x}\mathbf{W}^{(1)} + \mathbf{b}^{(1)})\mathbf{W}^{(2)} + \mathbf{b}^{(2)}$, where $\phi(\cdot)$ denotes an activation function (e.g., ReLU or GELU).

To stabilize training and facilitate deep stacking, layer normalization is applied around both the Attention and FFN layers. For an input vector $\mathbf{x} \in \mathbb{R}^d$, layer normalization is defined as

$$\text{LN}(\mathbf{x}) = \boldsymbol{\gamma} \odot \frac{\mathbf{x} - \mu(\mathbf{x})}{\sqrt{\sigma^2(\mathbf{x}) + \epsilon}} + \boldsymbol{\beta},$$

where $\mu(\mathbf{x})$ and $\sigma^2(\mathbf{x})$ denote the mean and variance computed across feature dimensions, and $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$ are learnable parameters.

C.2.2 Building Blocks. According to the above description, there are more complex nonlinear building blocks required for LLMs. Here, we briefly introduce the construction of the GeLU function, negative exponential function, and reciprocal square root. The dependencies among our protocols are illustrated in Figure 15. Π_{LUT} and $\Pi_{\mathsf{IntervalLUT}}$ are simplified and extended variants of our spline protocol and thus we omit the detailed descriptions of these protocols. Noting that both Π_{LUT} and $\Pi_{\mathsf{IntervalLUT}}$ require daBits to convert the outputs into the arithmetic domain, and thus incur additional online communication.

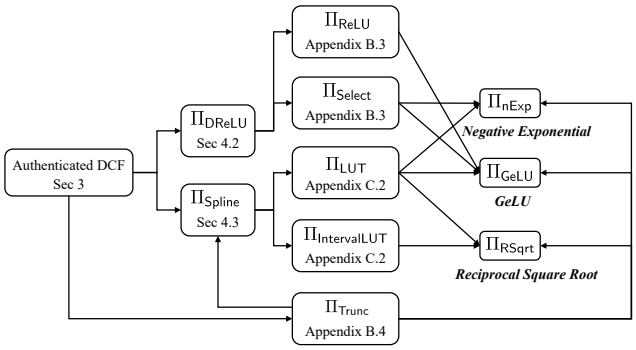

Figure 15: Protocols Dependencies

We model the lookup table (LUT) protocol Π_{LUT} as a zero-degree spline by partitioning the input domain into 2^n disjoint intervals of the form $[i, i+1)$ for $i \in [0, 2^n - 1)$, where n denotes the input bit-length of the LUT. Each interval is associated with a constant output value. Since each interval contains only one single leaf of the DCF, the selection vector for LUT can be obtained by performing a full-domain evaluation of the DCF with 2^n AES calls. Therefore, the input bit-length n of LUT determines both the number of required daBits and the computational complexity of the full-domain evaluation. We should minimize the input bit-length as much as possible for protocol design.

We use $\Pi_{\mathsf{IntervalLUT}}$ to extract both the exponent index m and the corresponding power-of-two scale 2^m such that $2^m \le x < 2^{m+1}$. We model the interval lookup table as a zero-degree spline over a coarser partition of the input domain. Specifically, we partition the domain $[0, 2^n)$ into $n+1$ disjoint intervals $I_0 = [0, 1)$, and $I_i = [2^{i-1}, 2^i)$ for $i \in [1, n]$. We can associate each interval I_i with several constant payloads.

GeLU Function. The GELU function is defined as $\text{GELU}(x) = x \cdot \Phi(x)$, where $\Phi(x) = \int_{-\infty}^{x} \frac{1}{\sqrt{2\pi}} e^{-\frac{t^2}{2}}\, dt$ denotes the cumulative distribution function. In the semi-honest setting, there exist many approaches for evaluating GeLU, including LUTs [25], spline-based evaluation [29], exact computation, Fourier-transform-based methods [38], and iterative ODE-based approximations [30]. Our DCF provides a fundamental secure comparison primitive that supports most of these GeLU evaluation techniques. To minimize online communication cost, we adopt a range-restricted LUT construction for evaluating GeLU, inspired by SIGMA [25].

Considering that $\text{GeLU}(x)$ can be expressed as $\text{ReLU}(x) - \delta(x)$, where the values of $\delta(x)$ are concentrated in the range $[-4, 4]$, become negligible outside this interval, and $\delta(x)$ is an even function, we set the input precision of $\delta(x)$ to 6 bits and implement $\delta(x)$ using an 8-bit input Π_{LUT} with $T_{\mathsf{GeLU}}[i] = \left\lfloor \delta(i/2^6) \cdot 2^f \right\rfloor$ for $i \in [0, 256)$. Furthermore, we should apply Π_{Trunc} to adjust the precision of the

input x to 6 bits, and then use Π_{Select} to clip x into the range $[0, 256)$. The detailed description of GeLU protocol is shown in Figure 16.

Protocol Π_{GeLU}

P_b computes as follows with inputs Λ_x and lookup table T_{GeLU}:

(1) $\Lambda_y \leftarrow \Pi_{\mathsf{Trunc}}(\Lambda_x, f-6)$. // $y = x >> f-6$ to adjust the precision of the input x to 6 bits
(2) $\Lambda_{p_y} \leftarrow \Pi_{\mathsf{ReLU}}(\Lambda_y)$.
(3) $\Lambda_{p_x} \leftarrow \Pi_{\mathsf{Select}}(\Lambda_y, \Lambda_x)$.
(4) $\Lambda_a = 2 \cdot \Lambda_{p_y} - \Lambda_y$. // $a = \mathrm{abs}(y)$
(5) $\Lambda_i \leftarrow \Pi_{\mathsf{Select}}(256 - \Lambda_a, \Lambda_a)$. // clip x into the range $[0, 256)$
(6) $\Lambda_z \leftarrow \Lambda_{p_x} - \Pi_{\mathsf{LUT}}(\Lambda_i, T_{\mathsf{GeLU}})$. // $z = \mathrm{ReLU}(x) - \delta(x)$

Figure 16: Actively-secure GeLU Protocol

Negative Exponential Function. When evaluating Softmax function, a common technique to avoid overflow is to reformulate it using negative exponential. However, this introduces additional overhead due to the need to compute the Max function. Agarwal et al. [2] provide a constant-round ranking method based on FSS, which can be instantiated with our DCF scheme to achieve actively secure $\max(\vec{x})$. Back to negative exponential function, SIGMA [25] also provide the range-restricted LUT-based approach. We adopt the LUT-based method to achieve negative exponential function.

We consider the negative exponential function $\mathsf{nExp}(x) = e^{-x}$ for $x \in \mathbb{R}^+$. Inspired by SIGMA [25], the fixed-point representation satisfies $\lfloor e^{-x} \cdot 2^{12} \rfloor = 0$ for all $x \geq 16$ under fixed-point precision with $f = 12$. Therefore, it suffices to restrict the input domain to the interval $[0, 16)$ which requires 16-bits LUT. To enhance the effiency, we can further divide the 16-bits LUT into two 8-bits LUTs. Let $x^{(1)}$ and $x^{(0)}$ correspond to the most significant and least significant 8 bits, respectively. The fixed-point evaluation of nExp can be approximated as $\lfloor \mathsf{nExp}(x/2^{12}) \cdot 2^f \rfloor \approx \mathsf{ARS}(T_{\mathsf{nExp1}}[x^{(1)}] \cdot T_{\mathsf{nExp0}}[x^{(0)}], f)$, where $T_{\mathsf{nExp1}}[i] = \lfloor \mathsf{nExp}(i/2^4) \cdot 2^f \rfloor$ and $T_{\mathsf{nExp0}}[i] = \lfloor \mathsf{nExp}(i/2^{12}) \cdot 2^f \rfloor$ for $i \in [0, 256)$ are two 8-bit LUTs with n-bit outputs. Therefore, we first apply Π_{Select} to clip x into the range $[0, 2^{16})$ and divide it into two 8-bits parts utilizing our Π_{Trunc}. By fusing Π_{LUT} and the multiplication, we can remove the additional communication round required by our Π_{LUT}. The detailed description of negative exponential protocol is shown in Figure 17.

Reciprocal Square Root. We denote reciprocal square root as $\mathsf{RSqrt}(x) = 1/\sqrt{x}$ where $x \in \mathbb{R}^+$. RSqrt is the most complex nonlinear function in layer normalization. Mosformer [10] introduces a general domain reduction strategy to reduce the input-length of LUT used in RSqrt. Given a positive input x, we first determine an integer index k such that $2^{k-1} \leq x < 2^k$. The input can be normalized as $x' = x \cdot 2^{-k} \in [1/2, 1)$, which restricts the evaluation domain to a compact interval. With f-bits fixed-point precision, this normalization reduces the required lookup table to $f-1$-bits. We can adapt this strategy to our framework in a more lightweight and efficient manner.

Specifically, we can use $\Pi_{\mathsf{IntervalLUT}}$ to retrieval the scaling factor. We partition the input domain into disjoint power-of-two intervals $I_0 = [0, 1)$, $I_i = [2^{i-1}, 2^i)$ for $i \in [1.n]$, such that for any positive input $x \in [2^{i-1}, 2^i)$. Each interval I_i is associated with a constant payload consisting of two values: the interval index and the corresponding scaling factor. Formally, the payload is defined as $v[i] = \{\lfloor \sqrt{(2^i/2^f)^{-1}} \cdot 2^f \rfloor, \lfloor (2^i/2^f)^{-1} \cdot 2^f \rfloor\}$. Evaluating $\Pi_{\mathsf{IntervalLUT}}$ on input fixed-point x directly outputs both $\lfloor \sqrt{(2^k/2^f)^{-1}} \cdot 2^f \rfloor$ and scaling factor $\lfloor (2^k/2^f)^{-1} \cdot 2^f \rfloor$, where $2^k > x \geq 2^{k-1}$. Then we apply the scaling factor to obtain x', reducing the range of x to $[1/2, 1)$ with f-bits precision utilizing the multiplication and Π_{Trunc}. We can prepare the lookup table $T_{\mathsf{RSqrt}}[i] = \lfloor \sqrt{(i/2^f + 1/2)^{-1}} \cdot 2^f \rfloor$ for $i \in [0, (1-1/2) \cdot 2^f]$ and evaluate with input $x' - (1/2) \cdot 2^f$ using Π_{LUT}. Finally, apply multiplication with truncation to obtain the result of reciprocal square root. The detailed description is shown in Figure 18.

Protocol Π_{nExp}

P_b computes as follows with inputs Λ_x and lookup tables T_{nExp0} and T_{nExp1} :

(1) $\Lambda_y \leftarrow \Pi_{\mathsf{Select}}(\Lambda_x - 2^{16}, 2^{16} - \Lambda_x - 1) + \Lambda_x$. // clip x into the range $[0, 2^{16})$
(2) $\Lambda_{x^{(1)}} \leftarrow \Pi_{\mathsf{Trunc}}(\Lambda_y, 8)$ and $\Lambda_{x^{(0)}} = \Lambda_y \mod 2^8$.
(3) $\Lambda_{p^{(1)}} \leftarrow \Pi_{\mathsf{LUT}}(\Lambda_{x^{(1)}}, T_{\mathsf{nExp1}})$.
(4) $\Lambda_{p^{(0)}} \leftarrow \Pi_{\mathsf{LUT}}(\Lambda_{x^{(0)}}, T_{\mathsf{nExp0}})$.
(5) $\Lambda_z \leftarrow \Pi_{\mathsf{Mul}}(\Lambda_{p^{(1)}}, \Lambda_{p^{(0)}})$.
(6) $\Lambda_{\tilde{z}} \leftarrow \Pi_{\mathsf{Trunc}}(\Lambda_z, f)$.

Figure 17: Actively-secure Negative Exponential Protocol

Protocol Π_{RSqrt}

P_b computes as follows with inputs Λ_x and lookup table T_{RSqrt}:

(1) $(\Lambda_{v^{(0)}}, \Lambda_{v^{(1)}}) \leftarrow \Pi_{\mathsf{IntervalLUT}}(\Lambda_x)$. // retrieve $v^0 = \lfloor \sqrt{(2^k/2^f)^{-1}} \cdot 2^f \rfloor$ and $v^1 = \lfloor (2^k/2^f)^{-1} \cdot 2^f \rfloor$, where $2^k > x \geq 2^{k-1}$
(2) $\Lambda_{x'} \leftarrow \Pi_{\mathsf{Mul}}(\Lambda_{v^{(1)}}, \Lambda_x)$.
(3) $\Lambda_{\tilde{x'}} \leftarrow \Pi_{\mathsf{Trunc}}(\Lambda_{x'}, f)$. // reduce x into $[1/2, 1)$ with f-bits precision
(4) $\Lambda_t \leftarrow \Pi_{\mathsf{LUT}}(\Lambda_{\tilde{x'}} - (1/2) \cdot 2^f, T_{\mathsf{RSqrt}})$. // make the input of LUT in range $[0, 2^{f-1})$
(5) $\Lambda_z \leftarrow \Pi_{\mathsf{Mul}}(\Lambda_{v^{(0)}}, \Lambda_t)$.
(6) $\Lambda_{\tilde{z}} \leftarrow \Pi_{\mathsf{Trunc}}(\Lambda_z, f)$.

Figure 18: Actively-secure Reciprocal Square Root Protocol

D Security Proof

This section presents security proofs for our DCF scheme and actively secure protocols. We first present the definitions required for the proofs.

Definition 4 (Natural and Non-Trivial Operations from Half-Tree [23]). *Let $H : \{0,1\}^\lambda \to \{0,1\}^\lambda$ be a function, O :*

$\{0,1\}^\lambda \times \{0,1\} \to \{0,1\}^\lambda$ *be an oracle, and let* $T \subseteq \{0,1\}^\lambda \setminus \{0^\lambda\}$ *be a set of linearly independent strings. An operation* Q_i *is called a* natural and non-trivial operation (NNO) *if it defines a result* $x_i \in \{0,1\}^\lambda$ *such that:*

(1) $x_i \leftarrow \{0,1\}^\lambda$.
(2) $x_i := x_j \oplus x_k$, *where* x_j, x_k *were defined by two distinct NNOs* Q_j, Q_k.
(3) $x_i := x_j \oplus \tau$, *where* x_j *was defined by an NNO* Q_j, *and* $\tau \in T$.
(4) $x_i := H(x_j)$, *where* x_j *was defined by an NNO* Q_j.
(5) $x_i := O(x_j, b_j)$, *where* x_j *was defined by an NNO* Q_j, *and* $b_j \in \{0,1\}$.

Moreover, x_i *must syntactically differ from any* x_j *that was defined by an NNO* Q_j.

Definition 5 (CCR hash function). *Let* $H_{ccr} : \{0,1\}^\lambda \to \{0,1\}^\lambda$, *let* χ *be a distribution on* $\{0,1\}^\lambda$, *and let* $\mathcal{F} : \{0,1\}^{\lambda+1} \to \{0,1\}^\lambda$ *be a family of functions. Define the oracle* $O^{\mathrm{ccr}}_{H_{ccr},R}(x,b) := H_{ccr}(x \oplus R) \oplus b \cdot R$ *for* $x, R \in \{0,1\}^\lambda, b \in \{0,1\}$.

We say that H_{ccr} *is* $(t, q, \rho, \varepsilon)$*-CCR hash if, for any distinguisher* D *running in time at most* t *and making at most* q *queries to* $O^{\mathrm{ccr}}_{\mathcal{H},R}(\cdot,\cdot)$, *and any distribution* χ *with min-entropy at least* ρ,

$$\left| \Pr_{R \leftarrow \chi}\left[D^{O^{\mathrm{ccr}}_{H_{ccr},R}(\cdot,\cdot)}(1^\lambda) = 1\right] - \Pr_{f \leftarrow \mathcal{F}}\left[D^{f(\cdot,\cdot)}(1^\lambda) = 1\right] \right| \le \varepsilon,$$

where D *is not allowed to query both* $(x,0)$ *and* $(x,1)$ *for any* $x \in \{0,1\}^\lambda$.

D.1 Proof of Theorem 1

Theorem 7. *(Restated of Theorem 1) Given the CCR hash function* $H_{ccr} : \{0,1\}^\lambda \to \{0,1\}^\lambda$ *and keyed hash function* $H_S(x) := H_{ccr}(S \oplus x)$ *with some key* $S \leftarrow \{0,1\}^\lambda$, *Figure 5 is a secure authenticated DCF scheme with key size* $(n+1) \cdot \lambda$ *bits.*

Proof. **Correctness.** The correctness follows from Lemmea 1. if $x < \beta$, we have

$$[[y]]^{\mathsf{B}}_0 \oplus [[y]]^{\mathsf{B}}_1 = 1 \text{ and } [[m]]^{\mathsf{B}}_0 \oplus [[m]]^{\mathsf{B}}_1 = R,$$

accordingly, we have

$$\mathsf{hb}([[m]]^{\mathsf{B}}_0, \kappa) \oplus \mathsf{hb}([[m]]^{\mathsf{B}}_1, \kappa) = \mathsf{hb}(R, \kappa) = \Delta \in \mathbb{F}_{2^\kappa},$$

otherwise,

$$[[y]]^{\mathsf{B}}_0 \oplus [[y]]^{\mathsf{B}}_1 = 0 \text{ and } [[m]]^{\mathsf{B}}_0 \oplus [[m]]^{\mathsf{B}}_1 = 0.$$

Privacy. Our lightweight authenticated DCF scheme is inspired by Half-Tree [23]. Similarly, we first consider the hybrids $\{\mathsf{Hyb}_0, \mathsf{Hyb}_1\}$ as shown in Figure 19, where Hyb_0 corresponds to our original scheme, and Hyb_1 is defined by replacing the correction words CW_i with uniform values.

According to the NNOs defined in Definition 4, let $\mathsf{Real}_{H_S,R}(\cdot)$ be a real-world oracle which sets $H(\cdot) = H_S(\cdot)$ and $O(\cdot,\cdot) = O^{\mathrm{ccr}}_{H_S,R}(\cdot,\cdot)$ (which can be found in Definition 5), executes the operation as per Definition 4, and returns the defined result. The ideal-world oracle $\mathsf{Ideal}_{H_S}(\cdot)$ is identical to $\mathsf{Real}_{H_S,R}(\cdot)$, except that it sets $O(\cdot,\cdot)$ to an oracle that returns a fresh uniform string upon every invocation. We have the following lemma to complete our proof, which is proven in Half-Tree's Appendix A.

$\mathsf{Hyb}_d(1^\lambda, \Delta, \beta, n)$
1: Parse $\beta = \beta_1 \cdots \beta_n \in \{0,1\}^n$.
2: Sample $R \leftarrow \Delta \| \{0,1\}^{\lambda-\kappa-1} \| 1$.
3: Sample $[[s_0 \| t_0]]^{\mathsf{B}}_b \leftarrow \{0,1\}^\lambda$ for $b \in \{0,1\}$.
4: **for** $i \in [1,n]$ **do**
5: **if** $d = 1$ **then** Sample $CW_i \leftarrow \{0,1\}^\lambda$.
6: **else**
7: $CW_i := H_S([[s_{i-1} \| t_{i-1}]]^{\mathsf{B}}_b) \oplus H_S([[s_{i-1} \| t_{i-1}]]^{\mathsf{B}}_b \oplus R) \oplus \overline{\beta}_i \cdot R$.
8: $[[s_i \| t_i]]^{\mathsf{B}}_b := H_S([[s_{i-1} \| t_{i-1}]]^{\mathsf{B}}_b) \oplus \beta_i \cdot [[s_{i-1} \| t_{i-1}]]^{\mathsf{B}}_b \oplus [[t_{i-1}]]^{\mathsf{B}}_b \cdot CW_i$ for $b \in \{0,1\}$.
9: $k_b := ([[s_0 \| t_0]]^{\mathsf{B}}_b, \{CW_i\}_{i \in [1,n]})$.
10: **return** (k_0, k_1)

Figure 19: The hybrids for our DCF Scheme

Lemma 3 (Circular Correlation Robustness for Restricted Queries). *Let* $H_{ccr} : \{0,1\}^\lambda \to \{0,1\}^\lambda$ *be a* $(t, q, \rho, \varepsilon)$*-CCR hash function, and let* χ *be a distribution on* $\{0,1\}^\lambda$ *with min-entropy at least* ρ. *Define* $H_S(x) := H_{ccr}(S \oplus x)$ *for* $S, x \in \{0,1\}^\lambda$.

There exists a polynomial $\mathsf{poly}(\cdot)$ *such that for any PPT distinguisher* D *running in time at most* $t - \mathsf{poly}(\lambda)$ *and querying the oracle* $\mathsf{Real}_{H_S,\Delta}(\cdot)$ *with at most* q *natural and non-trivial operations, it holds*

$$\left| \Pr_{\substack{R \leftarrow \chi, \\ S \leftarrow \chi, \\ \mathsf{coins} \leftarrow \mathcal{U}}} \left[D^{\mathsf{Real}_{H_S,R}}(1^\lambda) = 1\right] - \Pr_{\substack{\mathsf{coins}' \leftarrow \mathcal{U}, \\ S \leftarrow \chi, \\ \mathsf{coins} \leftarrow \mathcal{U}}} \left[D^{\mathsf{Ideal}_{H_S}}(1^\lambda) = 1\right] \right| \le \mathsf{negl}(\lambda).$$

Here, negl *denotes the negligible function,* $\mathcal{U}$ *denotes the uniform distribution over bit-strings of an unspecified polynomial length.* coins *denotes the random coins used to define the results of type (1) operations, while* coins′ *denotes the random coins used to define the results of type (5) operations in* $\mathsf{Ideal}_{H_S}(\cdot)$.

Suppose, for contradiction, that there exists a PPT adversary $\mathcal{A}$ that distinguishes Hyb_0 and Hyb_1 with non-negligible advantage. We can construct a PPT simulator $\mathcal{S}$, that is given oracle $O' \in \{\mathsf{Real}_{H_S,R}, \mathsf{Ideal}_{H_S}\}$ and can use $\mathcal{A}$ to break Lemma 3.

$\mathcal{S}$ simulates an execution of there hybrids as follows. All randomness not involving H_S is sampled honestly. Whenever the hybrid requires a value derived from $H_S(\cdot)$, $\mathcal{S}$ expresses the computation as a sequence of allowed NNOs and forwards them to the oracle O'. There exists the following types of NNOs for $i \in [1,n]$:

- Query O' with Q_1 : $[[s_0 \| t_0]]^{\mathsf{B}}_b \leftarrow \{0,1\}^\lambda$.
- Query O' with $Q_{i,1}$: $temp_1 := H_S([[s_{i-1} \| t_{i-1}]]^{\mathsf{B}}_b)$.
- Query O' with $Q_{i,2}$: $temp_2 := O([[s_{i-1} \| t_{i-1}]]^{\mathsf{B}}_b, \overline{\beta_i})$.
- Query O' with $Q_{i,3}$: $CW_i := temp_1 \oplus temp_2$.
- If $\beta_i = 0$ and $[[t_{i-1}]]^{\mathsf{B}}_b = 0$, regard $temp_1$ as $[[s_i \| t_i]]^{\mathsf{B}}_b$ (without a new query).

- If $\beta_i = 0$ and $[\![t_{i-1}]\!]_b^{\mathsf{B}} = 1$, regard $temp_2$ as $[\![s_i \| t_i]\!]_b^{\mathsf{B}}$ (without a new query).
- If $\beta_i = 1$ and $[\![t_{i-1}]\!]_b^{\mathsf{B}} = 0$, query O' with $Q_{i,4}$: $[\![s_i \| t_i]\!]_b^{\mathsf{B}} := \mathsf{temp}_1 \oplus [\![s_{i-1} \| t_{i-1}]\!]_b^{\mathsf{B}}$.
- If $\beta_i = 1$ and $[\![t_{i-1}]\!]_b^{\mathsf{B}} = 1$, query O' with $Q_{i,4}$: $[\![s_i \| t_i]\!]_b^{\mathsf{B}} := \mathsf{temp}_2 \oplus [\![s_{i-1} \| t_{i-1}]\!]_b^{\mathsf{B}}$.

Above all, if $O' = \mathsf{Real}_{H_S,\Delta}$, then all oracle responses are computed using the real function H_S, and the simulation is distributed identically to Hyb_0. And if $O' = \mathsf{Ideal}_{H_S}$, then all oracle responses are uniform and independent, which results in the same distribution as in Hyb_1 where $\{CW_i\}_{i\in[1,n]}$ are sampled uniformly.

As for the security of $R = \Delta \| \{0,1\}^{\lambda-\kappa-1} \| 1$. We can explain it from two aspects:

- 1) We can regard R as $R = R' \oplus f(\Delta)$ where $R' \leftarrow \chi$ with min-entropy $\lambda - 1$ and $f(\Delta)$ is independent of R'. Then $O^{\mathsf{ccr}}_{H_S,R}(x,b) = H_S(x \oplus R) \oplus b \cdot R = H_S(x \oplus R' \oplus f(\Delta)) \oplus b \cdot R' \oplus b \cdot f(\Delta) = O^{\mathsf{ccr}}_{H_S,R'}(x \oplus f(\Delta), b) \oplus b \cdot f(\Delta)$. Due to $\Delta \neq 0$ is fixed as the boolean authenticated key and $f(\Delta)$ is also fixed and independent from R', it still satisfies the type (3) operations of NNOs.
- 2) Since Δ is sampled uniformly from $\{0,1\}^\kappa$ and the remaining $\lambda - \kappa - 1$ bits of R are uniformly random, the resulting distribution of R has min-entropy at least $\lambda - 1$.

Therefore, the advantage of $\mathcal{S}$ in distinguishing $\mathsf{Real}_{H_S,R}$ from Ideal_{H_S} is exactly the advantage of $\mathcal{A}$ in distinguishing Hyb_0 from Hyb_1. Since $\mathcal{A}$ can distinguish Hyb_0 and Hyb_1 with non-negligible advantage, we have

$$\left| \Pr_{R\leftarrow\chi}\left[\mathcal{S}^{\mathsf{Real}_{H_S,R}(\cdot)}(1^\lambda) = 1 \right] - \Pr\left[\mathcal{S}^{\mathsf{Ideal}_{H_S}(\cdot)}(1^\lambda) = 1 \right] \right| > \mathsf{negl}(\lambda),$$

which contradicts with Lemma 3.

In summary, we can construct a PPT simulator $\mathcal{S}$ that outputs a key of $(n+1)\cdot\lambda$ random bits which are computationally indistinguishable from $\{k_b | k_0, k_1 \leftarrow \mathsf{DCF.Gen}(1^\lambda, \Delta, \beta, n)\}$.

Active Security. DCF.Eval outputs $\langle y \rangle^B$ with $m_y = y \cdot \Delta$. If a malicious adversary changes y to $y' \neq y$ and still passes verification in subsequent protocols, it must produce a valid authentication tag such that $m_{y'} - m_y = (y' - y) \cdot \Delta$. Since the global key Δ is hidden and uniformly random over $\mathbb{F}_2^\kappa$, implying a MAC forgery, which succeeds with probability at most $2^{-\kappa}$ [32]. Hence the correctness of DCF evaluation holds with probability at least $1 - 2^{-\kappa}$. □

D.2 Proof of Theorem 2

Theorem 8. *(Restated of Theorem 2) Figure 6 shows a actively secure derivative of ReLU protocol against the malicious adversary who can corrupt one of two servers P_0 and P_1 in $\mathcal{F}_{\mathsf{PrepDReLU}}$-hybrid model.*

Proof. Considering the $\mathcal{F}_{\mathsf{PrepDReLU}}$-hybrid model, **Preprocessing Phase** in Π_{DReLU} can be achieved by $\mathcal{F}_{\mathsf{PrepDReLU}}$. Meanwhile, as the two parties are symmetric in **Online Phase** of Π_{DReLU}, we w.l.o.g. assume that P_b is corrupted by adversary $\mathcal{A}$ and P_{1-b} is honest for some fixed $b \in \{0,1\}$. Simulator $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{PrepDReLU}}$ and samples uniform $[\![\Delta]\!]_0^{\mathsf{B}}, [\![\Delta]\!]_1^{\mathsf{B}} \in \mathbb{F}_{2^\kappa}$. $\mathcal{S}$ stores $\Delta = [\![\Delta]\!]_0^{\mathsf{B}} \oplus [\![\Delta]\!]_1^{\mathsf{B}}$ as the global key and sends $[\![\Delta]\!]_b^{\mathsf{B}}$ to $\mathcal{A}$. The simulation is constructed as follows:

(1) $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{PrepDReLU}}$ and samples uniform $\langle \delta_y \rangle_b^{\mathsf{B}}$ and random $k_b \leftarrow \{0,1\}^{(n+1)\cdot\lambda}$. $\mathcal{S}$ sends them to $\mathcal{A}$.
(2) $\mathcal{S}$ executes the protocol honestly and obtains $\langle \Lambda_y \rangle_b^{\mathsf{B}} := \{[\![\Lambda_y]\!]^{\mathsf{B}}, [\![m_{\Lambda_y}]\!]^{\mathsf{B}}\} \in \mathbb{F}_2 \times \mathbb{F}_{2^\kappa}$.
(3) $\mathcal{S}$ samples uniform bit Λ_y and computes $[\![\Lambda_y]\!]_{1-b}^{\mathsf{B}} = \Lambda_y \oplus [\![\Lambda_y]\!]_b^{\mathsf{B}}$, $[\![m_{\Lambda_y}]\!]_{1-b}^{\mathsf{B}} = \Lambda_y \cdot \Delta \oplus [\![m_{\Lambda_y}]\!]_b^{\mathsf{B}}$.
(4) $\mathcal{S}$ emulates OpenB by sending $[\![\Lambda_y]\!]_{1-b}^{\mathsf{B}}$ to $\mathcal{A}$.

It's clear that the real and ideal executions are indistinguishable unless the adversary $\mathcal{A}$ breaks the CCR property of H_{ccr} according to theorem 1. □

D.3 Proof of Theorem 3

Theorem 9. *(Restated of Theorem 3) Figure 8 shows a actively secure spline protocol against the malicious adversary who can corrupt one of two servers P_0 and P_1 in $\mathcal{F}_{\mathsf{PrepSpline}}$-hybrid model.*

Proof. Considering the $\mathcal{F}_{\mathsf{PrepSpline}}$-hybrid model, **Preprocessing Phase** in Π_{Spline} can be achieved by $\mathcal{F}_{\mathsf{PrepSpline}}$. Meanwhile, as the two parties are symmetric in **Online Phase** of Π_{Spline}, we w.l.o.g. assume that P_b is corrupted by adversary $\mathcal{A}$ and P_{1-b} is honest for some fixed $b \in \{0,1\}$. Simulator $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{PrepSpline}}$ and samples uniform $[\![\alpha]\!]_0^{\mathsf{A}}, [\![\alpha]\!]_1^{\mathsf{A}} \in \mathbb{Z}_{2^s}$ and $[\![\Delta]\!]_0^{\mathsf{B}}, [\![\Delta]\!]_1^{\mathsf{B}} \in \mathbb{F}_{2^\kappa}$. $\mathcal{S}$ stores $\alpha = [\![\alpha]\!]_0^{\mathsf{A}} + [\![\alpha]\!]_1^{\mathsf{A}} \mod 2^{n+s}$ and $\Delta = [\![\Delta]\!]_0^{\mathsf{B}} \oplus [\![\Delta]\!]_1^{\mathsf{B}}$ as the global keys and sends $[\![\alpha]\!]_b^{\mathsf{A}}$ and $[\![\Delta]\!]_b^{\mathsf{B}}$ to $\mathcal{A}$. The simulation is constructed as follows:

(1) $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{PrepSpline}}$ and samples uniform $\langle \vec{r} \rangle_b^{\mathsf{B}}$, $\langle \hat{\delta}_z \rangle_b^{\mathsf{A}}$, $\langle \vec{a} \rangle_b^{\mathsf{A}}$, $\langle \vec{b} \rangle_b^{\mathsf{A}}$ and random $k_b \leftarrow \{0,1\}^{(n+1)\cdot\lambda}$. $\mathcal{S}$ sends them to $\mathcal{A}$.
(2) $\mathcal{S}$ executes the protocol honestly and obtains $\langle \vec{s} \rangle_b^{\mathsf{B}} := \{\langle s[1] \rangle_b^{\mathsf{B}}, \ldots, \langle s[T] \rangle_b^{\mathsf{B}}\}$.
(3) For $j \in \{1, 2, \ldots, T\}$, $\mathcal{S}$ samples uniform bit $s[j]$ and computes $[\![s[j]]\!]_{1-b}^{\mathsf{B}} = s[j] \oplus [\![s[j]]\!]_b^{\mathsf{B}}$, $[\![m_{s[j]}]\!]_{1-b}^{\mathsf{B}} = s[j] \cdot \Delta \oplus [\![m_{s[j]}]\!]_b^{\mathsf{B}}$.
(4) $\mathcal{S}$ emulates OpenB procedure by sending $[\![s[1]]\!]_{1-b}^{\mathsf{B}}, [\![s[2]]\!]_{1-b}^{\mathsf{B}}, \ldots, [\![s[T]]\!]_{1-b}^{\mathsf{B}}$ to $\mathcal{A}$.
(5) $\mathcal{S}$ executes the remaining protocol honestly and obtains $\langle \Lambda_z \rangle_b^{\mathsf{A}} := \{[\![\Lambda_z]\!]_b^{\mathsf{A}}, [\![m_{\Lambda_z}]\!]_b^{\mathsf{A}}\} \in \mathbb{Z}_{2^{n+s}} \times \mathbb{Z}_{2^{n+s}}$.
(6) $\mathcal{S}$ samples uniform Λ_z and computes $[\![\Lambda_z]\!]_{1-b}^{\mathsf{A}} = \Lambda_z - [\![\Lambda_z]\!]_b^{\mathsf{A}}$, $[\![m_{\Lambda_z}]\!]_{1-b}^{\mathsf{A}} = \Lambda_z \cdot \alpha - [\![m_{\Lambda_z}]\!]_b^{\mathsf{A}}$.
(7) $\mathcal{S}$ emulates OpenA by sending $[\![\Lambda_z]\!]_{1-b}^{\mathsf{A}}$ to $\mathcal{A}$.

All the values sent to $\mathcal{A}$ are uniformly sampled in both real and ideal executions. As for the DCF key k_b, the DCF key in real execution and the uniformly sampled key in simulation are indistinguishable unless the adversary $\mathcal{A}$ breaks the CCR property of H_{ccr} according to theorem 1. Summing up, the real and ideal executions are indistinguishable. □

D.4 Other Proofs

D.4.1 Proof of overall $\mathcal{F}_{\mathsf{dealer}}$-2PC protocol.

Theorem 10. *Section 4.1 shows the overall of $\mathcal{F}_{\text{dealer}}$ 2PC protocol against the malicious adversary who can corrupt one of two servers P_0 and P_1 in $\mathcal{F}_{\text{dealer}}$-hybrid model.*

Proof. Considering the $\mathcal{F}_{\text{dealer}}$-hybrid model, we w.l.o.g. assume that P_b is corrupted by adversary $\mathcal{A}$ and P_{1-b} is honest for some fixed $b \in \{0,1\}$. Simulator $\mathcal{S}$ emulates $\mathcal{F}_{\text{dealer}}$ and samples uniform $[\![\alpha]\!]^{\mathsf{A}}_0, [\![\alpha]\!]^{\mathsf{A}}_1 \in \mathbb{Z}_{2^s}$, $[\![\Delta]\!]^{\mathsf{B}}_0, [\![\Delta]\!]^{\mathsf{B}}_1 \in \mathbb{F}_{2^\kappa}$. $\mathcal{S}$ stores $\alpha = [\![\alpha]\!]^{\mathsf{A}}_0 + [\![\alpha]\!]^{\mathsf{A}}_1 \mod 2^{n+s}$ and $\Delta = [\![\Delta]\!]^{\mathsf{B}}_0 \oplus [\![\Delta]\!]^{\mathsf{B}}_1$ as the global keys and sends $[\![\alpha]\!]^{\mathsf{A}}_b$, $[\![\Delta]\!]^{\mathsf{B}}_b$ to $\mathcal{A}$. The simulation is constructed as follows:

Input Phase, $\mathcal{S}$ samples uniform δ_x for P_b and sends δ_x to $\mathcal{A}$. Receiving Λ_x from $\mathcal{A}$.

Computation Phase, the simulation follows the security proofs of the corresponding computation protocols (e.g., Π_{DReLU}, Π_{Spline}, etc.).

Output Phase, $\mathcal{S}$ emulates as follows:

(1) Execute BatchCheck with $\mathcal{A}$ for both the arithmetic and boolean worlds, on behalf of the virtual honest P_{1-b}. If the check fails then $\mathcal{S}$ sends abort to $\mathcal{A}$.
(2) If the above check passed and the output is defined over arithmetic world:
   - sample uniform $\langle -\hat{\delta}_y \rangle^{\mathsf{A}}_b$ and compute $\langle y \rangle^{\mathsf{A}} = \Lambda_y + \langle -\hat{\delta}_y \rangle^{\mathsf{A}}_b := \{[\![y]\!]^{\mathsf{A}}_b, [\![m_y]\!]^{\mathsf{A}}_b\}$.
   - Sample uniform y and compute $[\![y]\!]^{\mathsf{A}}_{1-b} = y - [\![y]\!]^{\mathsf{A}}_b$, $[\![m_y]\!]^{\mathsf{A}}_{1-b} = y \cdot \alpha - [\![m_y]\!]^{\mathsf{A}}_b$.
   - Execute MAC checking procedure for arithmetic world with the adversary, on behalf of the honest P_{1-b}.
(3) If the above check passed and the output is defined over boolean world:
   - sample uniform $\langle \delta_y \rangle^{\mathsf{B}}_b$ and compute $\langle y \rangle^{\mathsf{B}} = \Lambda_y \oplus \langle \delta_y \rangle^{\mathsf{B}}_b := \{[\![y]\!]^{\mathsf{B}}_b, [\![m_y]\!]^{\mathsf{B}}_b\}$.
   - Sample uniform y and compute $[\![y]\!]^{\mathsf{B}}_{1-b} = y \oplus [\![y]\!]^{\mathsf{B}}_b$, $[\![m_y]\!]^{\mathsf{B}}_{1-b} = y \cdot \Delta \oplus [\![m_y]\!]^{\mathsf{B}}_b$.
   - Execute MAC checking procedure with the adversary for boolean world, on behalf of the honest P_{1-b}.

We argue the indistinguishability between the real and ideal executions. This is clear for **Input Phase** and **Computation Phase**, where the adversary $\mathcal{A}$ only see the uniform values in both real and ideal executions. It is only in the **Output Phase** where cares need to be taken. According to SPD$\mathbb{Z}_{2^k}$ [11] and MASCOT [32], if the MAC checking passes, the distinguishing advantage of $\mathcal{A}$ can be bounded by $2^{-s+\log_2(s+1)} + 2^{-\kappa} + \max\{2^{-s}, 2^{-\kappa}\}$. □

D.4.2 Proof of Theorem 4.

Theorem 11. *(Restated of Theorem 4) Figure 12 shows a actively secure multiplication protocol against the malicious adversary who can corrupt one of two servers P_0 and P_1 in $\mathcal{F}_{\text{PrepMul}}$-hybrid model.*

Proof. Considering the $\mathcal{F}_{\text{PrepMul}}$-hybrid model, **Preprocessing Phase** in Π_{Mul} can be achieved by $\mathcal{F}_{\text{PrepMul}}$. Meanwhile, as the two parties are symmetric in **Online Phase** of Π_{Mul}, we w.l.o.g. assume that P_b is corrupted by adversary $\mathcal{A}$ and P_{1-b} is honest for some fixed $b \in \{0,1\}$. Simulator $\mathcal{S}$ emulates $\mathcal{F}_{\text{dealer}}$ and samples uniform $[\![\alpha]\!]^{\mathsf{A}}_0, [\![\alpha]\!]^{\mathsf{A}}_1 \in \mathbb{Z}_{2^s}$. $\mathcal{S}$ stores $\alpha = [\![\alpha]\!]^{\mathsf{A}}_0 + [\![\alpha]\!]^{\mathsf{A}}_1 \mod 2^{n+s}$ as the global key and sends $[\![\alpha]\!]^{\mathsf{A}}_b$ to $\mathcal{A}$. The simulation is constructed as follows:

(1) $\mathcal{S}$ emulates $\mathcal{F}_{\text{PrepMul}}$ and samples uniform $\langle \hat{\delta}_x \rangle^{\mathsf{A}}_b$, $\langle \hat{\delta}_y \rangle^{\mathsf{A}}_b$ and $\langle \hat{\delta}_{xy} \rangle^{\mathsf{A}}_b$. $\mathcal{S}$ sends them to $\mathcal{A}$.
(2) $\mathcal{S}$ emulates the protocol honestly and obtain $\langle \Lambda_z \rangle^{\mathsf{A}}_b := \{[\![\Lambda_z]\!]^{\mathsf{A}}_b, [\![m_{\Lambda_z}]\!]^{\mathsf{A}}_b\}$.
(3) $\mathcal{S}$ samples uniform Λ_z and compute $[\![\Lambda_z]\!]^{\mathsf{A}}_{1-b} = \Lambda_z - [\![\Lambda_z]\!]^{\mathsf{A}}_b$, $[\![m_{\Lambda_z}]\!]^{\mathsf{A}}_{1-b} = \Lambda_z \cdot \alpha - [\![m_{\Lambda_z}]\!]^{\mathsf{A}}_b$.
(4) $\mathcal{S}$ emulates OpenA by sending $[\![\Lambda_z]\!]^{\mathsf{A}}_{1-b}$ to $\mathcal{A}$.

The simulation of Π_{Mul} is perfect and the real and ideal executions are indistinguishable. □

D.4.3 Proof of Theorem 5.

Theorem 12. *(Restated of Theorem 5) Figure 13 shows a actively secure ReLU protocol against the malicious adversary who can corrupt one of two servers P_0 and P_1 in $\mathcal{F}_{\text{PrepReLU}}$-hybrid model.*

Proof. Π_{ReLU} heavily relies on the Π_{DReLU}, which proved to be perfectly simulatable in Theorem 2. In the remaining step of the protocol, simulator $\mathcal{S}$ emulates $\mathcal{F}_{\text{PrepReLU}}$ and samples uniform $[\![\alpha]\!]^{\mathsf{A}}_0, [\![\alpha]\!]^{\mathsf{A}}_1 \in \mathbb{Z}_{2^s}$. $\mathcal{S}$ stores $\alpha = [\![\alpha]\!]^{\mathsf{A}}_0 + [\![\alpha]\!]^{\mathsf{A}}_1 \mod 2^{n+s}$ as the global key and sends $[\![\alpha]\!]^{\mathsf{A}}_b$ to $\mathcal{A}$. The simulation is constructed as follows:

(1) $\mathcal{S}$ emulates $\mathcal{F}_{\text{PrepReLU}}$ and samples uniform $\langle \delta_{xy} \rangle^{\mathsf{A}}_b$, $\langle \hat{\delta}_z \rangle^{\mathsf{A}}_b$, $\langle \delta_y \rangle^{\mathsf{A}}_b$ and $\langle \hat{\delta}_x \rangle^{\mathsf{A}}_b$. $\mathcal{S}$ sends them to $\mathcal{A}$.
(2) $\mathcal{S}$ executes the protocol honestly and obtains $\langle \Lambda_z \rangle^{\mathsf{A}}_b := \{[\![\Lambda_z]\!]^{\mathsf{A}}_b, [\![m_{\Lambda_z}]\!]^{\mathsf{A}}_b\} \in \mathbb{Z}_{2^{n+s}} \times \mathbb{Z}_{2^{n+s}}$.
(3) $\mathcal{S}$ samples uniform Λ_z and computes $[\![\Lambda_z]\!]^{\mathsf{A}}_{1-b} = \Lambda_z - [\![\Lambda_z]\!]^{\mathsf{A}}_b$, $[\![m_{\Lambda_z}]\!]^{\mathsf{A}}_{1-b} = \Lambda_z \cdot \alpha - [\![m_{\Lambda_z}]\!]^{\mathsf{A}}_b$.
(4) $\mathcal{S}$ emulates OpenA by sending $[\![\Lambda_z]\!]^{\mathsf{A}}_{1-b}$ to $\mathcal{A}$.

The simulation of Π_{ReLU} is the same as that of Π_{DReLU}. The real and ideal executions are indistinguishable unless the adversary $\mathcal{A}$ breaks the CCR property of H_{ccr} according to theorem 1. □

D.4.4 Proof of Theorem 6.

Theorem 13. *(Restated of Theorem 6) Figure 14 shows a actively secure truncation protocol against the malicious adversary who can corrupt one of two servers P_0 and P_1 in $\mathcal{F}_{\text{PrepTrunc}}$-hybrid model.*

Proof. Considering the $\mathcal{F}_{\text{PrepTrunc}}$-hybrid model, **Preprocessing Phase** in Π_{Trunc} can be achieved by $\mathcal{F}_{\text{PrepTrunc}}$. Meanwhile, as the two parties are symmetric in **Online Phase** of Π_{Trunc}, we w.l.o.g. assume that P_b is corrupted by adversary $\mathcal{A}$ and P_{1-b} is honest for some fixed $b \in \{0,1\}$. Simulator $\mathcal{S}$ emulates $\mathcal{F}_{\text{PrepTrunc}}$ and samples uniform $[\![\alpha]\!]^{\mathsf{A}}_0, [\![\alpha]\!]^{\mathsf{A}}_1 \in \mathbb{Z}_{2^s}$. $\mathcal{S}$ stores $\alpha = [\![\alpha]\!]^{\mathsf{A}}_0 + [\![\alpha]\!]^{\mathsf{A}}_1 \mod 2^{n+s}$ as the global key and sends $[\![\alpha]\!]^{\mathsf{A}}_b$ to $\mathcal{A}$. The simulation is constructed as follows:

(1) $\mathcal{S}$ emulates $\mathcal{F}_{\text{PrepTrunc}}$ and samples uniform $\langle r \rangle^{\mathsf{A}}$, $\langle r \rangle^{\mathsf{B}}$ $\langle \widetilde{\delta}_x \rangle^{\mathsf{A}}_b$, $\langle \delta_{msb} \rangle^{\mathsf{A}}_b$ and k_b, and sends them to $\mathcal{A}$.
(2) $\mathcal{S}$ executes the protocol honestly and obtains $\langle t \rangle^{\mathsf{B}} = \langle t_{out} \rangle^{\mathsf{B}} \oplus \langle r \rangle^{\mathsf{B}}$.
(3) $\mathcal{S}$ samples uniform bits t and computes $[\![t]\!]^{\mathsf{B}}_{1-b} = t \oplus [\![t]\!]^{\mathsf{B}}_b$, $[\![m_t]\!]^{\mathsf{B}}_{1-b} = t \cdot \Delta \oplus [\![m_t]\!]^{\mathsf{B}}_b$.

(4) $\mathcal{S}$ emulates OpenB procedure by sending t to $\mathcal{A}$.
(5) $\mathcal{S}$ executes the protocol honestly and obtains $\langle \Lambda_{\widetilde{x}} \rangle_b^{\mathsf{A}} := \{[[\Lambda_{\widetilde{x}}]]_b^{\mathsf{A}}, [[m_{\Lambda_{\widetilde{x}}}]]_b^{\mathsf{A}}\} \in \mathbb{Z}_{2^{n+s}} \times \mathbb{Z}_{2^{n+s}}$.
(6) $\mathcal{S}$ samples uniform $\Lambda_{\widetilde{x}}$ and computes $[[\Lambda_{\widetilde{x}}]]_{1-b}^{\mathsf{A}} = \Lambda_{\widetilde{x}} - [[\Lambda_{\widetilde{x}}]]_b^{\mathsf{A}}$, $[[m_{\Lambda_{\widetilde{x}}}]]_{1-b}^{\mathsf{A}} = \Lambda_{\widetilde{x}} \cdot \alpha - [[m_{\Lambda_{\widetilde{x}}}]]_b^{\mathsf{A}}$.
(7) $\mathcal{S}$ emulates OpenA by sending $[[\Lambda_{\widetilde{x}}]]_{1-b}^{\mathsf{A}}$ to $\mathcal{A}$.

The simulation is perfect and $\mathcal{A}$'s view are uniformly sampled values in both real and ideal executions. Two worlds executions are indistinguishable. □

D.4.5 Proof of Lemma 1.

Lemma 4. *(Restated of Lemma 1) Given a domain size $N = 2^n$ for $n \in \mathbb{N}$ and two integers β and x. In a full binary tree, the nodes along the special β path have a value of 1 while others are 0. When $x < \beta$, there is exactly one right sibling node with a value of 1 on the path corresponding to x, which corresponds to the first branching position where $x_j = 0$ and $\beta_j = 1$.*

Proof. Let $\mathcal{P}(\beta) = (v_0, \ldots, v_n)$ and $\mathcal{P}(x) = (w_0, \ldots, w_n)$ denote the root-to-leaf paths for β and x in the DPF tree, respectively. Each of v_i and w_i is a node on the path.

Assuming $x < \beta$, it implies that there exists a position j such that $x_1 = \beta_1, x_1 = \beta_1, \cdots, x_j = \beta_j$ and $x_{j+1} \neq \beta_{j+1}$. We can find that $x_{j+1} = 0$ and $\beta_{j+1} = 1$ which indicates that on the path, the first j layers of $\mathcal{P}(\beta)$ and $\mathcal{P}(x)$ are the same, but they start to diverge from the *j-th* layer. In the *j-th* layer, $\mathcal{P}(x)$ selected the left child node, while $\mathcal{P}(\beta)$ selected the right child node. Therefore, the value of the right sibling node of the *j-th* layer of $\mathcal{P}(x)$ is 1. In addition, in other layers of the path $\mathcal{P}(x)$, there are two cases as follows:

1) For the position $i < j$, the path of $\mathcal{P}(x)$ and $\mathcal{P}(\beta)$ is the same, and there is no right sibling node or the value of the right sibling node is 0.
2) For the position $i > j$, $\mathcal{P}(x)$ and $\mathcal{P}(\beta)$ have completely separated paths, with all right sibling node values being 0.

Therefore, The *j-th* layer necessarily contains a right sibling node with a value of 1.

Assuming there exist two or more right sibling nodes with a value of 1 on the path of $\mathcal{P}(x)$. Due to the fact that each node in each layer has only one right sibling node, and only the *j-th* layer causes the right sibling node to fall on $\mathcal{P}(\beta)$, there are two cases for other layers:

1) For the position $i < j$, the path of $\mathcal{P}(x)$ and $\mathcal{P}(\beta)$ is the same without any branching.
2) For the position $i > j$, $\mathcal{P}(x)$ has been separated from $\mathcal{P}(\beta)$, and the values of all right sibling nodes are 0.

The assumption contradicts with the above cases. The *j-th* layer is the only possible position to have a right sibling value of 1. □

E Instantiation of $\mathcal{F}_{\text{dealer}}$ via Two-Party Computation

This section presents a 2PC approach for instantiating $\mathcal{F}_{\text{dealer}}$. We provide the corresponding 2PC protocols for DCF generation and for the preprocessing phases of Π_{DReLU} and Π_{Spline}. Our instantiation follows the technology adopted in Turbospeedz [4] and ABY 2.0 [41] replacing the wire masks with the authenticated secret sharing. The purpose of the dealer is to preprocess wire masks and to generate the randomness and correlated materials required for evaluating the computation circuit. Therefore, the preprocessing protocols take as input the masks of the input wires of the computation circuit (i.e., $\langle \delta_x \rangle^{\mathsf{A}}$ or $\langle \delta_x \rangle^{\mathsf{B}}$, where the actual input x is masked as $\Lambda_x = x + \delta_x$ over $\mathbb{Z}_{2^n}$ or $\mathbb{F}_2$), and generate the circuit-related prepared materials as well as the output masks (i.e., $\langle \delta_y \rangle^{\mathsf{A}}$ or $\langle \delta_y \rangle^{\mathsf{B}}$), with the overall workflow following SPD$\mathbb{Z}_{2^k}$ [11] and MASCOT [32].

E.1 Cryptographic Tools

The construction of the 2PC protocols for instantiating $\mathcal{F}_{\text{dealer}}$ is based on the following cryptographic tools and ideal functionalities to achieve active security.

E.1.1 BDOZ-style authenticated sharing.

For a bit $x \in \mathbb{F}_2$, we use $\langle\langle x \rangle\rangle$ to denote a BDOZ-style [5] authenticated secret sharing. Concretely, the sharing is given by $\langle\langle x \rangle\rangle := (\langle\langle x \rangle\rangle_0, \langle\langle x \rangle\rangle_1)$, where for each $b \in \{0, 1\}$, server P_b holds

$$\langle\langle x \rangle\rangle_b := \{[[x]]_b^{\mathsf{B}},\ K_b[[[x]]_{1-b}^{\mathsf{B}}],\ M_b[[[x]]_b^{\mathsf{B}}]\} \in \mathbb{F}_2 \times \mathbb{F}_{2^\lambda} \times \mathbb{F}_{2^\lambda}.$$

The corresponding IT-MAC is defined as $M_b[[[x]]_b^{\mathsf{B}}] = K_{1-b}[[[x]]_b^{\mathsf{B}}] \oplus [[x]]_b^{\mathsf{B}} \cdot [[R]]_{1-b}^{\mathsf{B}} \in \mathbb{F}_{2^\lambda}$, where the global offset $R = [[R]]_0^{\mathsf{B}} \oplus [[R]]_1^{\mathsf{B}} \in \mathbb{F}_{2^\lambda}$. To ensure the condition $\mathsf{lsb}([[R]]_0^{\mathsf{B}} \oplus [[R]]_1^{\mathsf{B}}) = 1$, an additional consistency check [12] can be applied at the cost of sacrificing a small number of authenticated sharings.

Note that we can easily convert a BDOZ-style authenticated sharing to the boolean authenticated sharing locally. Specifically, given $\langle\langle x \rangle\rangle_b := \{[[x]]_b^{\mathsf{B}},\ K_b[[[x]]_{1-b}^{\mathsf{B}}],\ M_b[[[x]]_b^{\mathsf{B}}]\}$, each server P_b can locally compute as follows.

$$\langle x \rangle_b^{\mathsf{B}} := \{[[x]]_b^{\mathsf{B}}, [[m_x]]_b^{\mathsf{B}}, [[R]]_b^{\mathsf{B}}\}$$
$$\text{where } [[m_x]]_b^{\mathsf{B}} = K_b[[[x]]_{1-b}^{\mathsf{B}}] \oplus M_b[[[x]]_b^{\mathsf{B}}] \oplus [[x]]_b^{\mathsf{B}} \cdot [[R]]_b^{\mathsf{B}}.$$

We denote the above procedure as $\langle x \rangle^{\mathsf{B}} \leftarrow \mathsf{convert}(\langle\langle x \rangle\rangle)$ to achieve the conversion from BDOZ-style authenticated sharing to the boolean authenticated sharing.

E.1.2 Ideal Functionality.

- The authenticated bit functionality $\mathcal{F}_{\text{aBit}}$ [54]: Sample random bit r and share $\langle\langle r \rangle\rangle$ between servers.
- The daBits functionality $\mathcal{F}_{\text{daBits}}$ [1, 44]: Sample random bit r and share $\langle r \rangle^{\mathsf{A}}$ and $\langle r \rangle^{\mathsf{B}}$ between servers.
- The coin-tossing functionality $\mathcal{F}_{\text{coin}}$ [15]: Sample common random value $r \in \mathbb{F}_{2^\lambda}$ and send it to both servers.
- The randomness generation functionality $\mathcal{F}_{\text{rand}}$ [11, 32]: Sample random value $r \in \{\mathbb{Z}_{2^n}, \mathbb{F}_2\}$ and share $\langle r \rangle^{\mathsf{A}}$ (or $\langle r \rangle^{\mathsf{B}}$) between servers.
- The exponentiation functionality $\mathcal{F}_{\text{exp}}$ [4, 31]: Given $\langle x \rangle^{\mathsf{A}}$ and integer i, share $\langle x^i \rangle^{\mathsf{A}}$ between servers.
- The multiplication functionality $\mathcal{F}_{\text{mult}}$ [11]: Given $\langle x \rangle^{\mathsf{A}}$ and $\langle y \rangle^{\mathsf{A}}$, share $\langle x \cdot y \rangle^{\mathsf{A}}$ between servers.

E.1.3 Boolean to Arithmetic Sharing Conversion.

As described in section 4.3, we introduce the concept of *daBits* [1, 44] to perform Boolean-Arithmetic conversion. Both servers can invoke $\{\langle r \rangle^{\mathsf{A}},$

Protocol $\Pi_{\mathsf{DCF-Gen}}$

Protocol inputs: Two servers hold n BDOZ-style authenticated sharings $\langle\langle\vec{\beta}\rangle\rangle = \{\langle\langle\beta[1]\rangle\rangle, \ldots, \langle\langle\beta[n]\rangle\rangle\}$. Domain size $N = 2^n$ for $n \in \mathbb{N}$. Integer ring $\mathbb{Z}_{2^n}$. Hash function $H_{ccr} : \{0,1\}^\lambda \to \{0,1\}^\lambda$ with circular correlation robustness. Keyed hash function $H_S(x) := H_{ccr}(S \oplus x)$ with some key $S \leftarrow \{0,1\}^\lambda$.

Generate DCF key: Let $\langle\langle\beta[i]\rangle\rangle_b = ([[\beta[i]]]_b^{\mathsf{B}}, K_b[[[\beta[i]]]_{1-b}^{\mathsf{B}}], M_b[[[\beta]]_b^{\mathsf{B}}])$ for $b \in \{0,1\}$ and $i \in [1,n]$, two servers execute as follows.

(1) Both servers invoke $W \leftarrow \mathcal{F}_{\mathsf{coin}}$ to sample a public value $W \in \mathbb{F}_{2^\lambda}$ and set $[[s_{(0,0)} \| t_{(0,0)}]]_b^{\mathsf{B}} := [[R]]_b^{\mathsf{B}} \oplus W \in \{0,1\}^\lambda$.

(2) For each $b \in \{0,1\}$ and each $i \in [1,n]$,

- P_b computes,

$$[[CW_i]]_b^{\mathsf{B}} = \bigoplus_{j \in [0,2^{i-1})} H_S([[s_{(i-1,j)} \| t_{(i-1,j)}]]_b^{\mathsf{B}}) \oplus [[R]]_b^{\mathsf{B}} \oplus \left([[\beta[i]]]_b^{\mathsf{B}} \cdot [[R]]_b^{\mathsf{B}} \oplus K_b[[[\beta[i]]]_{1-b}^{\mathsf{B}}] \oplus M_b[[[\beta[i]]]_b^{\mathsf{B}}]\right) \in \{0,1\}^\lambda$$

- P_b sends $[[CW_i]]_b^{\mathsf{B}}$ to P_{1-b}.
- Both servers compute $CW_i = [[CW_i]]_0^{\mathsf{B}} \oplus [[CW_i]]_1^{\mathsf{B}} \in \{0,1\}^\lambda$.
- For all $j \in [0, 2^{i-1})$, Each server computes,

$$[[s_{(i,2j)} \| t_{(i,2j)}]]_b^{\mathsf{B}} = H_S([[s_{(i-1,j)} \| t_{(i-1,j)}]]_b^{\mathsf{B}}) \oplus [[t_{(i-1,j)}]]_b^{\mathsf{B}} \cdot CW_i,$$
$$[[s_{(i,2j+1)} \| t_{(i,2j+1)}]]_b^{\mathsf{B}} = H_S([[s_{(i-1,j)} \| t_{(i-1,j)}]]_b^{\mathsf{B}}) \oplus [[s_{(i-1,j)} \| t_{(i-1,j)}]]_b^{\mathsf{B}} \oplus [[t_{(i-1,j)}]]_b^{\mathsf{B}} \cdot CW_i.$$

(3) Both servers define the DCF key $k_b := ([[s_{0,0} \| t_{0,0}]]_b^{\mathsf{B}}, \{CW_i\}_{i \in [1,n]})$

Check consistency:

(4) For each $b \in \{0,1\}$ and $j \in [0,N)$, P_b sets

$$\langle u_j \rangle_b^{\mathsf{B}} := \{[[u_j]]^{\mathsf{B}} = [[t_{(n,j)}]]_b^{\mathsf{B}},\ [[m_{u_j}]]_b^{\mathsf{B}} = [[s_{(n,j)} \| t_{(n,j)}]]_b^{\mathsf{B}}\}$$

(5) Both servers invoke $\langle r \rangle^{\mathsf{B}} \leftarrow \mathcal{F}_{\mathsf{rand}}$ to generate $\langle r \rangle^{\mathsf{B}}$ for random $r \in \mathbb{F}_{2^\lambda}$.

(6) Servers invoke $\chi \leftarrow \mathcal{F}_{\mathsf{coin}}$ to sample a common random value $\chi \in \mathbb{F}_{2^\lambda}$ and locally compute

$$\langle a \rangle^{\mathsf{B}} := \sum_{j \in [0,N)} \chi^j \cdot \langle u_j \rangle^{\mathsf{B}} + \langle r \rangle^{\mathsf{B}} \in \mathbb{F}_{2^\lambda}.$$

(7) Servers execute $\mathsf{OpenB}(\langle a \rangle^{\mathsf{B}})$ with MAC checking mentioned in section 2.2.

Figure 20: Actively-secure Two-Party DCF Key Generation Protocol.

$\langle r \rangle^{\mathsf{B}}\} \leftarrow \mathcal{F}_{\mathsf{daBits}}$ to generate the daBit of the random bit $r \in \mathbb{F}_2$. Given the boolean authenticated share $\langle x \rangle^{\mathsf{B}}$ and $x \in \mathbb{F}_2$, servers compute $\langle t \rangle^{\mathsf{B}} = \langle x \rangle^{\mathsf{B}} \oplus \langle r \rangle^{\mathsf{B}}$ and execute $t \leftarrow \mathsf{OpenB}(\langle t \rangle^{\mathsf{B}})$. Then each server can locally compute the arithmetic authenticated sharing $\langle x \rangle^{\mathsf{A}} = t + \langle r \rangle^{\mathsf{A}} - 2 \cdot t \cdot \langle r \rangle^{\mathsf{A}}$. For simplicity, we define the above procedure as a macro $\langle x \rangle^{\mathsf{A}} \leftarrow \mathsf{Bit2A}(\langle x \rangle^{\mathsf{B}})$.

E.2 DCF Generation Protocol

Half-tree [23] has provided the semi-honest two-party DPF correlation generation protocol and Zhang et al. [54] extend it to active security. Here we present the modified protocol of Zhang et al.'s DPF in Figure 20 to adapt to our DCF scheme. Since we only require 1-bit output to construct boolean authenticated sharing, we eliminate the group conversion at the leaf nodes mentioned in et al. [54], thereby preserving the global offset correlation across all levels. Zhang et al. [54] have proved the security of the two-party DPF generation relying on CCR hash. We can also move the consistency check to the end of the overall protocol.

E.3 Two-Party Protocol for Π_{DReLU}

The detailed description is shown in Figure 21. When instantiating the preprocessing phase of DReLU via two-party computation, each server only holds $\langle \delta_x \rangle^{\mathsf{A}}$ the authenticated sharing of δ_x. Moreover, DCF key generation requires BDOZ-style authenticated sharings of the bit decomposition of δ_x. A straightforward approach is to perform an arithmetic-to-boolean conversion on δ_x; however, this approach is typically complex and incurs substantial overhead. Here, we introduce a trick to avoid arithmetic-to-boolean conversion, at the cost of only a slight increase in local computation during online phase.

Specifically, we aim to replace the original input mask δ_x with a freshly generated random value β. During the generation of the random value, we can obtain its bit decomposition in the form of BDOZ-style authenticated sharings, as well as its arithmetic authenticated sharing. We start by generating n random bits with BDOZ-style authenticated sharings $\{\langle\langle\beta[1]\rangle\rangle, \ldots, \langle\langle\beta[n]\rangle\rangle\}$. By applying $\langle\vec{\beta}\rangle^{\mathsf{B}} \leftarrow \mathsf{convert}(\langle\langle\vec{\beta}\rangle\rangle)$ followed by $\langle\vec{\beta}\rangle^{\mathsf{A}} \leftarrow \mathsf{Bit2A}(\langle\vec{\beta}\rangle^{\mathsf{B}})$, we obtain n arithmetic authenticated sharings $\{\langle\beta[1]\rangle^{\mathsf{A}}, \ldots, \langle\beta[n]\rangle^{\mathsf{A}}\}$, which can then be aggregated to form a new random mask $\langle\beta\rangle^{\mathsf{A}} =$

$\sum_{i=1}^{n} \langle \beta[i] \rangle^{\mathsf{A}} \cdot 2^{n-i}$. In this way, we obtain the tuple $\{\langle \beta \rangle^{\mathsf{A}}, \langle \vec{\beta} \rangle^{\mathsf{B}}, \langle\langle \vec{\beta} \rangle\rangle\}$, which suffices to carry out the desired computation. Then, we should execute $t = \mathsf{OpenA}(\langle \delta_x \rangle^{\mathsf{A}} + \langle \beta \rangle^{\mathsf{A}})$. By simply computing $\Lambda'_x = \Lambda_x - t$ during the pre-online phase, we can obtain the masked input $\Lambda'_x = x - \beta$ with the new random mask β.

Protocol $\Pi_{\mathsf{DReLU_2PC}}$

Protocol Inputs: Two servers hold the arithmetic authenticated sharing of input mask $\langle \delta_x \rangle^{\mathsf{A}}$, where $\delta_x \in \mathbb{Z}_{2^n}$.

Preprocessing Phase: Let $\langle \delta_x \rangle_b^{\mathsf{A}} := \{[[\delta_x]]_b^{\mathsf{A}}, [[m_{\delta_x}]]_b^{\mathsf{A}}, [[\alpha]]_b^{\mathsf{A}}\}$ for $b \in \{0,1\}$. Two servers execute as follows.

1. Both servers P_0 and P_1 invoke $\langle\langle \vec{\beta} \rangle\rangle \leftarrow \mathcal{F}_{\mathsf{aBit}}$ to generate the BDOZ-style authenticated sharing vector $\langle\langle \vec{\beta} \rangle\rangle := \{\langle\langle \beta[1] \rangle\rangle, \ldots, \langle\langle \beta[n] \rangle\rangle\}$, where $\vec{\beta} \in \mathbb{F}_2^n$.
2. P_0 and P_1 locally compute $\langle \vec{\beta} \rangle^{\mathsf{B}} \leftarrow \mathsf{convert}(\langle\langle \vec{\beta} \rangle\rangle)$ and then execute micro $\langle \vec{\beta} \rangle^{\mathsf{A}} \leftarrow \mathsf{Bit2A}(\langle \vec{\beta} \rangle^{\mathsf{B}})$ to convert them into arithmetic domain.
3. For each $b \in \{0,1\}$, P_b executes $k_b \leftarrow \Pi_{\mathsf{Gen}}(\langle\langle \beta[2] \rangle\rangle, \ldots, \langle\langle \beta[n] \rangle\rangle)$ to obtain the DCF keys.
4. For each $b \in \{0,1\}$, P_b computes $\langle \beta \rangle^{\mathsf{A}} = \sum_{i=1}^{n} \langle \beta[i] \rangle^{\mathsf{A}} \cdot 2^{n-i}$ and $\langle t \rangle^{\mathsf{A}} = \langle \delta_x \rangle^{\mathsf{A}} + \langle \beta \rangle^{\mathsf{A}}$.
5. Both servers execute $t \leftarrow \mathsf{OpenA}(\langle t \rangle^{\mathsf{A}})$ to obtain $t = \delta_x + \beta \mod 2^n$.
6. Both servers invoke $\langle \delta_y \rangle^{\mathsf{B}} \leftarrow \mathcal{F}_{\mathsf{rand}}$ to sample $\delta_y \in \mathbb{F}_2$.
7. For each $b \in \{0,1\}$, P_b computes $\langle \delta_t \rangle^{\mathsf{B}} = 1 \oplus \langle \delta_y \rangle^{\mathsf{B}} \oplus \langle \beta[1] \rangle^{\mathsf{B}}$.

Online Phase: Upon obtaining the masked input Λ_x, both servers locally compute $\Lambda_x^* := \Lambda_x - t \mod 2^n$, where $\Lambda_x^* = x - \beta \mod 2^n$. Then servers execute the same as Π_{DReLU} (placed in section 4.2, Figure 6).

Figure 21: Actively-secure Two-Party Protocol for Π_{DReLU}

E.4 Preprocessing Protocol for Π_{Spline}

The preprocessing protocol for Π_{Spline} follows the same approach as that of Π_{DReLU}. δ_x is defined over arithmetic domain, yet it is still required to generate the corresponding DCF keys. We apply the same trick in the preprocessing protocol of Π_{Spline} as well. The detailed description is shown in Figure 22.

Protocol $\Pi_{\mathsf{Spline_2PC}}$

Protocol Inputs: Two servers hold the arithmetic authenticated sharing of input mask $\langle \delta_x \rangle^{\mathsf{A}}$, where $\delta_x \in \mathbb{Z}_{2^n}$.

Preprocessing Phase: Two servers P_0 and P_1 execute as follows.

1. Both servers P_0 and P_1 invoke $\langle\langle \vec{\beta} \rangle\rangle \leftarrow \mathcal{F}_{\mathsf{aBit}}$ to generate the BDOZ-style authenticated sharing vector $\langle\langle \vec{\beta} \rangle\rangle := \{\langle\langle \beta[1] \rangle\rangle, \ldots, \langle\langle \beta[n] \rangle\rangle\}$, where $\vec{\beta} \in \mathbb{F}_2^n$.
2. P_0 and P_1 locally compute $\langle \vec{\beta} \rangle^{\mathsf{B}} \leftarrow \mathsf{convert}(\langle\langle \vec{\beta} \rangle\rangle)$ and then execute micro $\langle \vec{\beta} \rangle^{\mathsf{A}} \leftarrow \mathsf{Bit2A}(\langle \vec{\beta} \rangle^{\mathsf{B}})$ to convert them into arithmetic domain.
3. For each $b \in \{0,1\}$, P_b executes $k_b \leftarrow \Pi_{\mathsf{Gen}}(\langle\langle \vec{\beta} \rangle\rangle)$ to obtain the DCF keys.
4. For each $b \in \{0,1\}$, P_b computes $\langle \beta \rangle^{\mathsf{A}} = \sum_{i=1}^{n} \langle \beta[i] \rangle^{\mathsf{A}} \cdot 2^{n-i}$ and $\langle t \rangle^{\mathsf{A}} = \langle \delta_x \rangle^{\mathsf{A}} + \langle \beta \rangle^{\mathsf{A}}$.
5. Both servers execute $t \leftarrow \mathsf{OpenA}(\langle t \rangle^{\mathsf{A}})$ to obtain $t = \delta_x + \beta$.
6. Both servers invoke $\langle \delta_z \rangle^{\mathsf{A}} \leftarrow \mathcal{F}_{\mathsf{rand}}$ to sample $\delta_z \in \mathbb{Z}_{2^n}$.
7. Both servers invoke $\{\langle \vec{r} \rangle^{\mathsf{A}}, \langle \vec{r} \rangle^{\mathsf{B}}\} \leftarrow \mathcal{F}_{\mathsf{daBits}}$ to sample dabits of ramdom bit vector $\vec{r} := \{r[1], \ldots, r[T]\}$.
8. Let $\langle \delta'_x \rangle^{\mathsf{A}} = \langle \beta \rangle^{\mathsf{A}}$. For $i \in \{0, 1, \ldots, d\}$, both servers invoke $\langle \vec{A}[i] \rangle^{\mathsf{A}} \leftarrow \mathcal{F}_{\mathsf{exp}}(\langle \langle \delta'_x \rangle^{\mathsf{A}}, i \rangle^{\mathsf{A}})$ in parallel.
9. For $j \in \{1, \ldots, T\}$ and $i \in \{0, 1, \ldots, d\}$, both servers invoke $\langle \vec{B}[i \cdot T + j] \rangle^{\mathsf{A}} \leftarrow \mathcal{F}_{\mathsf{mult}}(\langle r[j] \rangle^{\mathsf{A}}, \langle (\delta'_x)^i \rangle^{\mathsf{A}})$ in parallel.

Online Phase: Upon obtaining the masked input Λ_x, both servers locally compute $\Lambda_x^* := \Lambda_x - t \mod 2^n$, where $\Lambda_x^* = x - \beta \mod 2^n$. Then servers execute the same as Π_{Spline} (placed in section 4.3, Figure 8).

Figure 22: Actively-secure Two-Party Protocol for Π_{Spline}