Distributed Function Secret Sharing and Applications

Pengzhi Xing*, Hongwei Li*, Meng Hao†,✉, Hanxiao Chen*, Jia Hu*, and Dongxiao Liu*
*University of Electronic Science and Technology of China
†Singapore Management University
{*p.xing, jiahu*}*@std.uestc.edu.cn*, {*hongweili, hanxiao.chen, dongxiao.liu*}*@uestc.edu.cn, menghao303@gmail.com*

***Abstract*—Function Secret Sharing (FSS) has emerged as a pivotal cryptographic tool for secure computation, delivering exceptional online efficiency with constant interaction rounds. However, the reliance on a trusted third party for key generation in existing FSS works compromises both security and practical deployment. In this paper, we introduce efficient distributed key generation schemes for FSS-based distributed point function and distributed comparison function, supporting both input and output to be arithmetic-shared. We further design crucial FSS-based components optimized for online efficiency, serving as the building blocks for advanced protocols. Finally, we propose an efficient framework for evaluating complex trigonometric functions, ubiquitous in scientific computations. Our framework leverages the periodic property of trigonometric functions, which reduces the bit length of input during FSS evaluation. This mitigates the potential performance bottleneck for FSS-based protocols incurred by bit length. Extensive empirical evaluations on real-world applications demonstrate a latency reduction of up to $14.73\times$ and a communication cost decrease ranging from $27.67 \sim 184.42\times$ over the state-of-the-art work.**

I. Introduction

Secure two-party computation (2PC) in the preprocessing setting has shown tremendous success in real-world scenarios [1]. This setting involves two stages, i.e., *offline stage* and *online stage*. During the offline stage, extensive cryptographic operations are performed to generate input-independent correlated randomness. The randomness is then consumed in the online stage, which only involves lightweight operations. Among various preprocessing-based 2PC protocols, Function Secret Sharing (FSS) [2] stands out as a promising and fundamental protocol, introducing constant interaction rounds and minimal communication overhead during the online stage. This efficiency advantage makes it a powerful tool for developing privacy-preserving applications involving complex non-linear functions [3], [4], [5].

Unfortunately, most advanced FSS-based protocols [6], [4], [3] rely on a trusted third party (also called dealer) to generate the FSS key at the offline stage, which is unrealistic in real-world applications and weakens security guarantees. On the one hand, identifying a universally trusted dealer is challenging across various scenarios. For instance, in the satellite collision prediction task [7] where two military entities seek to collaboratively and privately conduct proximity testing without disclosing the actual trajectories, it is difficult to find a dealer that both parties trust simultaneously. On the other hand, the dealer introduces additional security risks [8]. Specifically, if a compromised party colludes with the dealer, the private information held by the honest party would be fully exposed.

✉Corresponding author.

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.242233
www.ndss-symposium.org

To address the above problem, a few works [9], [10] explored using 2PC protocols to generate FSS keys instead of relying on a dealer. For example, Floram [9] presents the first dealer-less generation of FSS key for distributed point function (DPF). More recently, Half-tree [10] optimizes the DPF protocol of Floram and further proposes a 2PC-based key generation protocol for distributed comparison function (DCF). However, these protocols can not support arithmetic-shared inputs or outputs, which limits the applicability for many practical scenarios, such as secure machine learning training and inference [11], [12]. On the other hand, to facilitate real-world applications, several works propose optimized FSS protocols [13] and customized FSS-based schemes [4], [3] in the dealer setting. Unfortunately, these works heavily rely on the knowledge of the pre-generated randomness possessed by the dealer, thus restricting their extension to the dealer-less 2-party setting. Therefore, the above discovery urges us to propose efficient dealer-less FSS protocols with applications to real-world scenarios.

In this paper, we tackle the above challenges by designing efficient and practical dealer-less key generation schemes for DPF and DCF. Specifically, at the core of our DPF scheme is a lightweight *constrained comparison* protocol to support arithmetic-shared results, which utilizes merely a single invocation to AND protocol. Furthermore, we construct an efficient 2PC-based DCF scheme based on a new *correlated correction words generation* protocol. This optimizes communication overhead by reducing the invocations of underlying multiplexer operations. We additionally resolve the overflow problem that has hindered previous works, in order to support DCF-based comparisons on arithmetic-shared inputs.

As depicted in Figure 1, building on our efficient and practical 2PC-based DPF and DCF schemes, we develop several essential building blocks including *equality test*, *comparison*, *truncation*, *digit decomposition*, and *interval containment*. These blocks incorporate multiple optimizations to achieve constant round evaluation and small communication. Upon this, we carefully design a framework to evaluate complex mathematical functions in scientific computation, implementing efficient lookup table and spline polynomial approximation methods. This framework exploits periodic properties to reduce the bit length required for protocol evaluation, significantly alleviating performance bottlenecks in the FSS-based schemes

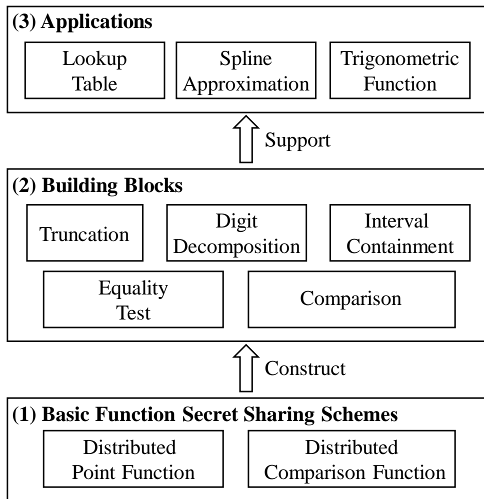

Fig. 1: The high-level overview of our framework.

and yielding substantial performance improvements in real-world applications. Our extensive experiments demonstrate that our approach significantly outperforms prior work. Our contribution can be summarized as follows.

- We propose the first efficient 2PC protocol to generate FSS keys for DPF and DCF without a dealer, supporting both arithmetic-shared inputs and outputs.
- We design several constant-round FSS-based building blocks within our efficient 2PC-based FSS paradigm.
- We introduce novel lookup table and spline polynomial approximation methods for complex mathematical functions in scientific computation.
- We fully implement our DPF and DCF protocols along with a series of FSS-based building blocks in the 2PC setting. When applied in the privacy-preserving proximity tests and biometric authentication, our protocols achieve $27.67 \sim 184.42\times$ improvements in communication performance and $1.16 \sim 14.73\times$ less latency over the prior art.

II. Related Works

We summarize the related works on function secret sharing and compare them with our work in Table I.

Function secret sharing. FSS provides an efficient way for additively secret-sharing a function within a function family. The FSS scheme for DPF was first proposed in the initial work [2]. Later, Boyle et al. [14] proposed an optimized DPF construction with reduced key size while supporting the evaluation results as elements in arbitrary groups. They also proposed FSS-based DCF to support comparison function family. Then, incremental DPF (iDPF) [6] was introduced, which generalizes DPF to output different payloads in each layer. More recently, Boyle et al. [13] further proposed an optimized DCF, with reduced DCF share size and arithmetic DCF output. However, the aforementioned works assume the existence of a dealer to handle the FSS key generation.

FSS-based protocols. In the following, we discuss existing works that utilize FSS to construct 2PC protocols. FSS serves as the primitives that efficiently evaluate a private function on public inputs [2]. In contrast, most 2PC protocols are designed to protect the sensitive inputs, i.e., they handle the private inputs with a publicly known function. To fill this gap, Boyle et al. [15] proposed a new concept known as *offset gate* (refer to Section III-E), making it possible to take advantage of both FSS online efficiency and 2PC compatibility. Later, several optimized building blocks based on FSS, such as interval containment and bit decomposition, were proposed [13]. Ryffel et al. [5] leverages DCF to build the ReLU function in machine learning models with probabilistic error. LLAMA [4] and Agarwal et al. [16] employed the FSS technique to devise a series of tailored protocols specifically for machine learning tasks, where the general spline approximation method [17] is utilized to evaluate math functions. Wagh [3] also proposed an FSS-based secure evaluation method for complex functions with a trusted dealer, by employing the help of the DPF-based lookup table technique [18]. Unfortunately, the above works do not consider the potential overhead bottleneck introduced by large bit-length in the FSS solution.

Emulating the FSS dealer. In the above-mentioned FSS-based protocols and applications, the offline stage is entrusted to the dealer, who possesses knowledge of the function intended for secret sharing and the random masks employed in subsequent operations. Recently, Floram [19] proposed a 2PC-based distributed key generation approach for DPF [2], outputting payload in the binary field. Their construction effectively obviates the requirement for the dealer. Another 2PC implementation for DPF and DCF, Half-Tree [10], utilizes a new GGM-tree construction, leading to simpler key generation and evaluation with reduced PRG invocations. Unfortunately, the above methods cannot support both arithmetic input and output while constructing further building blocks due to a potential overflow problem, limiting their practicality. (Refer to Section V-A).

III. Preliminaries

A. Notations

Let λ be the computational security parameter. We use $\text{negl}(\cdot)$ to present a negligible function. $[n]$ denotes the set $\{0, 1, \ldots, n-1\}$. $\mathbb{1}\{\mathsf{event}\}$ represents the indicator function, which equals 1 if event is true, and 0 otherwise. We use bold lower-case letters (e.g., $\mathbf{x}$) to represent vectors, and $\mathbf{x}[i]$ denotes the i-th element in this vector. Similarly, we use $x[i]$ to denote the i-th bit of a bit string x. Let $\mathbb{R}$ be the set of real numbers and $\mathbb{Z}_{2^\ell}$ be an ℓ-bit finite ring. We embed a real number $x \in \mathbb{R}$ into $\mathbb{Z}_{2^\ell}$ via the fixed-point representation $\lfloor 2^s \cdot x \rfloor \bmod 2^\ell$, where s is the fractional scale. To differentiate the decimal number x from its binary representation, we use x_2 to denote the binary form. We use $\mathsf{Convert}_{\mathbb{G}} : \{0,1\}^* \to \mathbb{G}$, abbreviated as $C_{\mathbb{G}}(\cdot)$, to denote a function that maps random strings to pseudo-random elements in $\mathbb{G}$ [13] and $G(\cdot)$ to denote a pseudo-random generator (PRG). $x := y$ denotes that x is assigned by y.

B. Threat model

The security of our 2PC protocols is provably provided against a static semi-honest probabilistic polynomial-time

TABLE I: Comparison with related works on function secret sharing

Protocols	Dealer-lesss DPF	Dealer-less DCF	Arithmetic-shared Inputs	Arithmetic-shared Outputs	Applications
BGI15 [2]	×	×	×	×	×
Floram [9]	✓	×	×	×	✓
BCG+21 [13]	×	×	✓	✓	×
LLAMA [4]	×	×	✓	✓	✓
Pika [3]	×	×	✓	✓	✓
Half-Tree [10]	✓	✓	×	✓	×
This work	✓	✓	✓	✓	✓

(PPT) adversary $\mathcal{A}$ that corrupts one of the parties but not both. In this setting, $\mathcal{A}$ strictly follows the protocol specification, but tries to learn additional information about the honest party's input from the received messages. We model security using the simulation paradigm [20], which defines a *real* interaction and an *ideal* interaction. In the real execution, both parties execute protocols in the presence of $\mathcal{A}$. In the ideal execution, both parties send their inputs to a trusted functionality $\mathcal{F}$ that faithfully executes the operation. Security requires that no efficient distinguisher can distinguish between real and ideal interactions. Our protocols invoke several sub-protocols, and for ease of exposition, we describe them using the *hybrid model*, which is the same as a real interaction except that the sub-protocols are replaced with the corresponding functionalities. In this work, a protocol invoking $\mathcal{F}$ is referred to as the *$\mathcal{F}$-hybrid model*.

C. Secret sharing

In this work, we use 2-out-of-2 additive secret sharing schemes [21], [22], including both arithmetic and Boolean sharing, denoted as $\langle\cdot\rangle$. Specifically, $\langle x\rangle^\ell$ indicates that x is arithmetic shared over $\mathbb{Z}_{2^\ell}$, and $\langle x\rangle^B$ denotes the Boolean shares over $\mathbb{Z}_2$. We use the subscription of party id $b \in \{0,1\}$ to distinguish the share held by parties, namely $\langle\cdot\rangle_0$ and $\langle\cdot\rangle_1$. We invoke existing secret sharing-based 2PC functionalities, including $\mathcal{F}^{\ell,B}_{\mathsf{MUX}}$, $\mathcal{F}^{\ell,A}_{\mathsf{MUX}}$, $\mathcal{F}_{\mathsf{AND}}$, $\mathcal{F}_{\mathsf{OR}}$, $\mathcal{F}^{\ell}_{\mathsf{MILL}}$ and $\mathcal{F}^{\ell}_{\mathsf{B2A}}$. We present the detailed description in Appendix A.

D. Function secret sharing

A two-party function secret sharing (FSS) scheme for a function family $\mathcal{F}$ is an efficient algorithm that splits a function $f \in \mathcal{F}$ into two additive shares f_0, f_1, such that (1) for $b \in \{0,1\}$, each share f_b hides the function f and (2) for any x in the domain of f, it holds $f_0(x) + f_1(x) = f(x)$. The formal definition of FSS is described as follows.

Definition 1 (Function Secret Sharing [14]). For a function family $\mathcal{F}_{\mathbb{G}^{\mathsf{in}},\mathbb{G}^{\mathsf{out}}}$ with domain $\mathbb{G}^{\mathsf{in}}$ and range $\mathbb{G}^{\mathsf{out}}$, a two-party FSS scheme with key space $\mathcal{K}_0 \times \mathcal{K}_1$ has a pair of algorithms $(\mathsf{Gen}, \mathsf{Eval})$, both taking λ as an implicit input:

- $(k_0, k_1) \leftarrow \mathsf{Gen}(\hat{f})$. On input the description $\hat{f} \in \{0,1\}^*$ of a function $f \in \mathcal{F}_{\mathbb{G}^{\mathsf{in}},\mathbb{G}^{\mathsf{out}}}$, the key generation algorithm outputs a key pair $(k_0, k_1) \in \mathcal{K}_0 \times \mathcal{K}_1$. Here, $\hat{f}$ contains the domain $\mathbb{G}^{\mathsf{in}}$ and range $\mathbb{G}^{\mathsf{out}}$.
- $f_b(x) \leftarrow \mathsf{Eval}(b, k_b, x)$. On input $b \in \{0,1\}$, the party's key $k_b \in \mathcal{K}_b$, and a function input $x \in \mathbb{G}^{\mathsf{in}}$, the evaluation algorithm outputs $f_b(x) \in \mathbb{G}^{\mathsf{out}}$.

An FSS scheme $(\mathsf{Gen}, \mathsf{Eval})$ is secure for the function family $\mathcal{F}_{\mathbb{G}^{\mathsf{in}},\mathbb{G}^{\mathsf{out}}}$ with respect to a leakage function $\mathsf{Leak} : \{0,1\}^* \to \{0,1\}^*$ if it satisfies the following requirements. In this work, Leak only includes the function domain and range.

- **Correctness.** For any function $f \in \mathcal{F}_{\mathbb{G}^{\mathsf{in}},\mathbb{G}^{\mathsf{out}}}$ with description $\hat{f}$ and any function input $x \in \mathbb{G}^{\mathsf{in}}$, if $(k_0, k_1) \leftarrow \mathsf{Gen}(\hat{f})$ then $\Pr[\mathsf{Eval}(0, k_0, x) + \mathsf{Eval}(1, k_1, x) = f(x)] = 1$.
- **Security.** For any function $f \in \mathcal{F}_{\mathbb{G}^{\mathsf{in}},\mathbb{G}^{\mathsf{out}}}$ with description $\hat{f}$, any party id $b \in \{0,1\}$, and any PPT adversary $\mathcal{A}$, there exists a PPT simulator Sim, such that

$$\begin{aligned}&\Pr[\mathcal{A}(k_b) = 1 \mid (k_0, k_1) \leftarrow \mathsf{Gen}(\hat{f})] - \\ &\Pr[\mathcal{A}(k_b) = 1 \mid k_b \leftarrow \mathsf{Sim}(b, \mathsf{Leak}(\hat{f}))] \leq \mathrm{negl}(\lambda).\end{aligned} \tag{1}$$

Distributed point function (DPF) [2]. A point function $f_{\alpha,\beta}$, for $\alpha \in \mathbb{G}^{\mathsf{in}}$ and $\beta \in \mathbb{G}^{\mathsf{out}}$, is defined as $f_{\alpha,\beta}(\alpha) = \beta$ and $f_{\alpha,\beta}(x) = 0$ for $x \neq \alpha$. A distributed point function (DPF) with domain $\mathbb{G}^{\mathsf{in}}$ and range $\mathbb{G}^{\mathsf{out}}$ is a two-party FSS scheme $(\mathsf{Gen}_{\mathsf{DPF}}, \mathsf{Eval}_{\mathsf{DPF}})$ for the point function family $\mathcal{F}_{\mathbb{G}^{\mathsf{in}},\mathbb{G}^{\mathsf{out}}} = \{f_{\alpha,\beta}\}_{\alpha\in\mathbb{G}^{\mathsf{in}},\beta\in\mathbb{G}^{\mathsf{out}}}$.

DPF invariant [14]. The FSS key generation and evaluation are built upon the GGM-style construction [23]. A node on the binary tree consists of a label and its control bit. For DPF with an ℓ-bit value α, a *special path* is defined from the root node to the α-th leaf node, corresponding to the binary representation of α from the most significant bit (MSB) to the least significant bit (LSB), where $\alpha[i] = 0$ to the left child and 1 to the right child at i-th level. The DPF invariant property defines that control bits of all the on-path nodes are 1, and 0 for off-path nodes. Accordingly, given an input x, the Boolean FSS output should be the control bit of the x-th leaf node. We demonstrate the DPF invariant property in Figure 2.

Maintaining DPF invariant via Correction Words [14]. As illustrated, labels and control bits in the GGM-style binary tree are pseudo-random. During key generation, the dealer's task is to walk through the special path and generate *correction words* for each layer to convert a random GGM-style tree to one following the DPF invariant. Specifically, the correction words will correct the control bit of the node on the off-path side and the consecutive labels to zero.

Supporting arithmetic output [14]. The original DPF scheme defines the control bit of the leaf node as the output, which can be seen as the Boolean-shared result. To output the payload β as the element in $\mathbb{G}^{\mathsf{out}}$, it suffices to compute the multiplication with payload β and the one-bit final control bit. For simplicity, we use t_0 and t_1 to denote the control

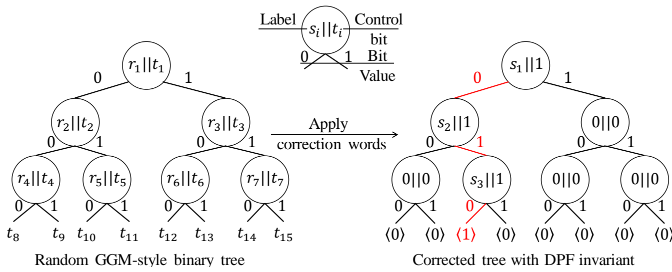

Fig. 2: The transformation from a random GGM-style binary tree to a tree with DPF invariant property on DPF target $\alpha = 2$. We use r_i and t_i to represent the random value and red lines to denote the special path.

bit held by the two parties. Given the random value in $\mathbb{G}^{\mathsf{out}}$ determined by the node label, namely $r_0 := C_{\mathbb{G}}(s_0)$ held by P_0 and $r_1 := C_{\mathbb{G}}(s_1)$ held by P_1, Boyle et al. [14] present the following transformation to avoid revealing β to parties.

$$\begin{aligned}&(t_0 \oplus t_1) \cdot \beta \\ =&(t_0 - t_1) \cdot \underbrace{(-1)^{t_1} \cdot (\beta - r_0 + r_1)}_{\text{Final Correction Word}} + r_0 - r_1\end{aligned} \tag{2}$$

As shown, the dealer sets the final correction term as $(-1)^{t_1} \cdot (\beta - r_0 + r_1)$, using two random values to conceal β. Upon reaching the leaf node, both parties multiply the correction term by their control bits and then add (or subtract) their respective random values to obtain the arithmetic output. If the control bits differ, the output is β. If the control bits are identical, by the DPF invariant, they share the same label and random value, i.e. $r_0 = r_1$, resulting in an output of 0.

Therefore, to generate the key, which can be seen as distributing the tree to two parties in a shared manner, a DPF key consists of the root node label for determining the full binary tree, correction words for each layer for maintaining DPF invariant, and a final correction word for supporting arithmetic output. We present this process in Figure 3.

Distributed comparison function (DCF) [13]. A comparison function $f_{\alpha,\beta}$, for $\alpha \in \mathbb{G}^{\mathsf{in}}$ and $\beta \in \mathbb{G}^{\mathsf{out}}$, is defined as $f_{\alpha,\beta}(x) = \beta$ for $x < \alpha$ and $f_{\alpha,\beta}(x) = 0$ for $x \geq \alpha$. A distributed comparison function (DCF) is an FSS scheme for the family of all comparison functions, with the leakage function $\mathsf{Leak}(\hat{f}) = (\mathbb{G}^{\mathsf{in}}, \mathbb{G}^{\mathsf{out}})$.

DCF invariant [13]. Nodes on the GGM tree for DCF consist of three parts: a control bit t, a node label s (the same as in DPF), and an additional *comparison label* v. While the comparison label v is not corrected by CW or passed to the next level, it is derived solely from its parent node label. Nevertheless, the labels and control bits still adhere to the DPF invariant property, as all node labels s and control bits t are corrected by CW. Additionally, the DCF invariant defines an intermediate value $V \in \mathbb{G}^{\mathsf{out}}$ for each node. Given an ℓ-bit target value α for comparison and input value x, the cumulative sum of V along the path to any x-th leaf node will be zero if $x \geq \alpha$, and $\beta \in \mathbb{G}^{\mathsf{out}}$ if $x < \alpha$.

E. FSS with secret-shared inputs

The original FSS scheme cannot be directly integrated with MPC frameworks due to its inherent assumption that a secret parameterized function is to be evaluated with a public input, which restricts its applicability.

To address this problem, Boyle et al. [15] proposed *FSS offset gate*. It requires the dealer to generate keys for the function $f(x - r^{\mathsf{in}})$ instead of $f(x)$, where r^{in} is chosen randomly by the dealer. Upon receiving the key k_b and shares of random mask $\langle r^{\mathsf{in}} \rangle_b$, two parties reconstruct $x_{\mathsf{real}} + r^{\mathsf{in}}$ as the public input and evaluate the function to get the correct output of $f(x_{\mathsf{real}})$. The offset function family or an *offset FSS gate* is defined as follows. Let $\mathcal{G} = \{g : \mathbb{G}^{\mathsf{in}} \to \mathbb{G}^{\mathsf{out}}\}$ be a computation gate for a parameterized function by input and output groups $\mathbb{G}^{\mathsf{in}}, \mathbb{G}^{\mathsf{out}}$. For $g : \mathbb{G}^{\mathsf{in}} \to \mathbb{G}^{\mathsf{out}} \in \mathcal{G}, r^{\mathsf{in}} \in \mathbb{G}^{\mathsf{in}}, r^{\mathsf{out}} \in \mathbb{G}^{\mathsf{out}}$, the family of offset functions $\hat{\mathcal{G}}$ of $\mathcal{G}$ is given by

$$\begin{aligned}\hat{\mathcal{G}} &:= \{g^{[r^{\mathsf{in}}, r^{\mathsf{out}}]} : \mathbb{G}^{\mathsf{in}} \to \mathbb{G}^{\mathsf{out}}\}, \text{where} \\ & g^{[r^{\mathsf{in}}, r^{\mathsf{out}}]}(x) := g(x - r^{\mathsf{in}}) + r^{\mathsf{out}}\end{aligned} \tag{3}$$

and $g^{[r^{\mathsf{in}}, r^{\mathsf{out}}]}$ contains an explicit description of $r^{\mathsf{in}}, r^{\mathsf{out}}$ with an FSS scheme.

IV. Distributed Function Secret Sharing

In this section, we present efficient 2PC FSS key generation for both DPF and DCF, supporting arithmetic-shared inputs and outputs.

A. Dealer-less FSS key generation for DPF

DPF is an important underlying building block itself and is wildly used in ORAM [19] and PIR [24]. We propose a two-party DPF key generation protocol, enabling arithmetic-shared inputs and outputs. This is particularly useful in real-world applications and addresses the limitation from Floram [9] that only supports Boolean shares. As illustrated in Algorithm 2, our protocol mainly consists of two phases: (1) traversing the GGM tree without knowing the exact special path, to generate layer-wise correction words, and (2) computing the correction word for arithmetic output. Before the first phase, we convert the ℓ_{in}-bit arithmetic share of the target index $\langle \alpha \rangle_b^{\ell_{\mathsf{in}}}$ into Boolean shares with Π_{BitDec} detailed in Appendix B1.

During the first phase, as shown in Figure 4, we follow the observation from Floram [9] to generate correction words without knowing the actual special path, as the off-path labels are identical and can be canceled after sum-up based on the *DPF invariant*. Therefore, for each party, it suffices to add all their left (right resp.) child nodes together as the shares of left (right resp.) label and use the multiplexer to decide the correct correction word based on $\alpha[i]$.

During the second phase, to support arithmetic output, we have to generate an extra correction word $(-1)^{t_1} \cdot (\beta - r_0 + r_1)$ following Equation 2. Without the dealer, the parties are unaware of their α-th control bits, which results in t_1 not being determined. Our observation is that according to the invariant of the DPF's GGM tree, the parties have access to the $2^{\ell_{\mathsf{in}}}$ control bits at the last level leaf nodes, instead of their α-th control bits, namely t_1 in Equation 2. Given DPF invariant $t_0 \oplus t_1 = 1$, the problem simplifies to determine which of t_0 or

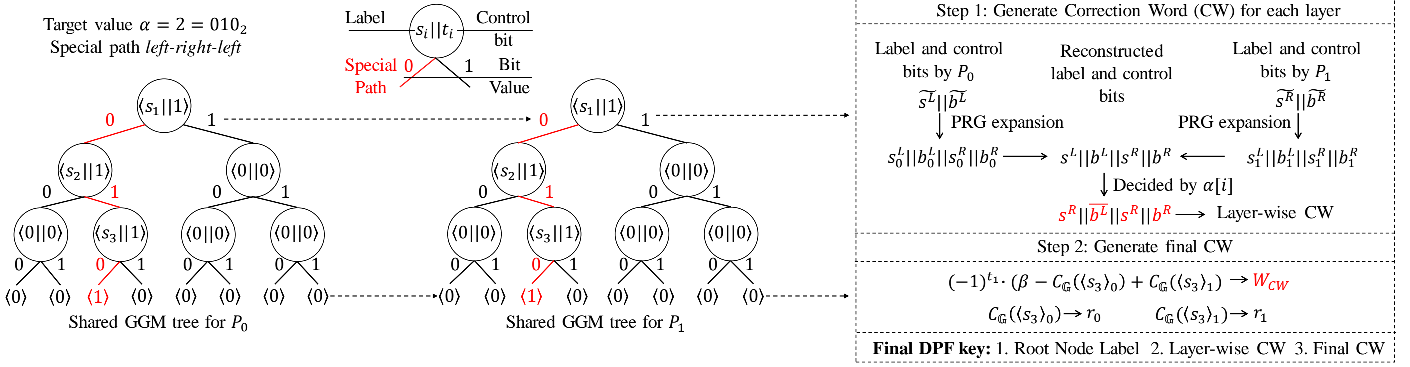

Fig. 3: An example for the dealer-based key generation with target value $\alpha = 2$.

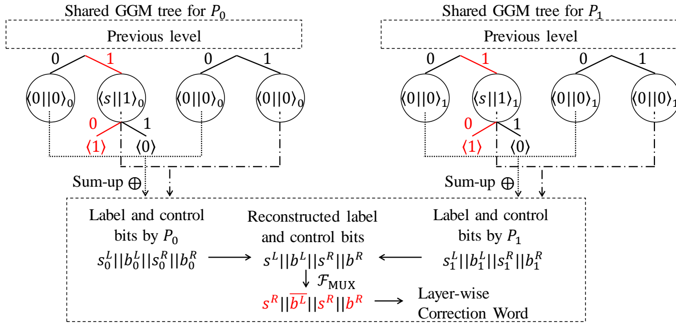

Fig. 4: Dealer-less layer correction word generation (Step 5-8, Algorithm 2).

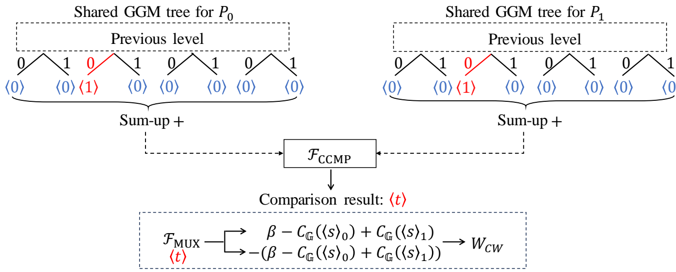

Fig. 5: Dealer-less generation of correction word for arithmetic output (Step 17-21, Algorithm 2).

t_1 is 1. To achieve this, parties sum their respective control bits from the last level and determine the party holding the bigger value differentiating by 1. We depict this phase in Figure 5.

We propose an efficient constrained integer comparison using only one AND gate and present our protocol Π_{CCMP} in Algorithm 1 based on the following theorem.

Lemma 1. *Given two positive integers $x_0, x_1 \in \mathbb{N}$ that differ by 1, $\mathbb{1}\{x_0 < x_1\}$ can be computed as follows:*

$$\mathbb{1}\{x_0 < x_1\} = \begin{cases} l_1 & h_0 = h_1 \\ \neg h_1 & h_0 \neq h_1, h_0 = l_0 \\ h_1 & h_0 \neq h_1, h_0 \neq l_0 \end{cases}$$

where for $b \in \{0,1\}$, l_b and h_b are the last bit and second to last bit of x_b, respectively.

Algorithm 1 Constrained Comparison, Π_{CCMP}

Input: P_b holds $x_b \in \mathbb{N}$ with $|x_0 - x_1| = 1$.
Output: P_b obtains $\langle y \rangle_b^B$ where $y = \mathbb{1}\{x_0 < x_1\}$.

1: P_b extracts last two bits of x_b, sets $h_b || l_b := \mathsf{lsb2}(x_b)$.
2: P_b sets $\langle z_0 \rangle_b^B := h_b$ and $\langle z_1 \rangle_b^B := h_b \oplus l_b \oplus b$.
3: P_b computes $\langle t \rangle_b^B := \mathcal{F}_{\mathsf{AND}}(\langle z_0 \rangle_b^B, \langle z_1 \rangle_b^B)$.
4: P_b outputs $\langle y \rangle_b^B := \langle t \rangle_b^B \oplus (l_b \wedge b)$.

Proof: Without the loss of generality, we assume t_0 is the smaller value between x_0 and x_1 and hence $t_1 = t_0 + 1$. We analyze it through four cases: (1) $\mathsf{lsb2}(t_0) = 00_2$, $\mathsf{lsb2}(t_1) = 01_2$; (2) $\mathsf{lsb2}(t_0) = 01_2$ and $\mathsf{lsb2}(t_1) = 10_2$; (3) $\mathsf{lsb2}(t_0) = 10_2$ and $\mathsf{lsb2}(t_1) = 11_2$; (4) $\mathsf{lsb2}(t_0) = 11_2$ and $\mathsf{lsb2}(t_1) = 00_2$. The four cases are mutually exclusive.

Further, the comparison result can be summarized into 3 specific conditions, as illustrated above. When the high-order bits of two numbers are identical, the comparison depends solely on the lower bits, with a 1 in the lower bit indicating the larger value. If the high-order bits differ but the lower bits are the same, the comparison hinges on the high-order bit. In cases where both high and lower-order bits differ, including the special case (00_2 vs. 11_2), the high-order bit determines the outcome. Specifically, for 01_2 vs. 10_2, the number with 1 in the high-order bit is larger. ■

Therefore, to compare two integers that differ by 1, it suffices to extract their last bit l_b and second to last bit h_b and calculate Equation 4 with one AND operation.

$$\begin{aligned} &(1 \oplus h_0 \oplus h_1) \wedge l_1 \oplus (h_0 \oplus h_1) \wedge ((h_0 \oplus l_0) \oplus h_1 \oplus 1) \\ =&l_1 \oplus (h_0 \oplus h_1) \wedge (l_1 \oplus h_1 \oplus l_0 \oplus h_0 \oplus 1) \end{aligned} \tag{4}$$

Complexity. Given the input and payload bit length to be ℓ, our dealer-less DPF key generation requires $9\ell + 5$ offline rounds with the communication size of $18\lambda\ell + 5\lambda + 13\ell + 2$ bits. The evaluation process is identical to prior work [13], which requires ℓ bits of communication size within 1 round.

Optimization from Half-tree [10]. Our protocol follows the DPF construction [13], in which all labels are pseudo-randomly generated. Recently, Half-tree [10] proposed an improved DPF construction with correlated labels for intermediate layers and pseudo-random labels only for the last

Algorithm 2 Dealer-less DPF Key Generation

$\mathsf{Gen}_{\mathsf{DPF}}^{\ell_{\mathsf{in}},\ell_{\mathsf{out}}}(b, \langle\alpha\rangle_b^{\ell_{\mathsf{in}}}, \langle\beta\rangle_b^{\ell_{\mathsf{out}}})$:
Input: P_b holds shares of position $\langle\alpha\rangle_b^{\ell_{\mathsf{in}}}$ and payload $\langle\beta\rangle_b^{\ell_{\mathsf{out}}}$.
Output: P_b obtains a DPF Key k_b.
1: P_b invokes $\mathcal{F}_{\mathsf{BitDec}}^{\ell_{\mathsf{in}}}$ with input $\langle\alpha\rangle_b^{\ell_{\mathsf{in}}}$ to get Boolean shares $\langle\alpha\rangle_b^B \in \{0,1\}^{\ell_{\mathsf{in}}}$
2: P_b samples a random seed $s_b^{0,0}$ from $\{0,1\}^\lambda$ and sets $t_b^{0,0} := b$.
3: **for** $i \in [\ell_{\mathsf{in}}]$ **do**
4: P_b sets $s_b^{i+1,2j} \| t_b^{i+1,2j} \| s_b^{i+1,2j+1} \| t_b^{i+1,2j+1} := G(s_b^{i,j})$ for $j \in [2^i]$
5: P_b sets $S_b^{i,p} := \bigoplus_{j\in[2^i]} s_b^{i+1,2j+p}$ for $p \in \{0,1\}$
6: P_b sets $T_b^{i,p} := \bigoplus_{j\in[2^i]} t_b^{i+1,2j+p}$ for $p \in \{0,1\}$
7: P_b invokes $\mathcal{F}_{\mathsf{MUX}}^{B,\lambda}$ with input $S_b^{i,0}, S_b^{i,1}, \langle\alpha[\ell_{\mathsf{in}}-1-i]\rangle_b^B \oplus b$ to get $\sigma_b^i := S_b^{i,\alpha[\ell_{\mathsf{in}}-1-i]\oplus 1}$.
8: P_b sets $\tau_b^{i,0} := T_b^{i,0} \oplus \langle\alpha[\ell_{\mathsf{in}}-1-i]\rangle_b^B \oplus b$ and $\tau_b^{i,1} := T_b^{i,1} \oplus \langle\alpha[\ell_{\mathsf{in}}-1-i]\rangle_b^B$.
9: P_b sends $\sigma_b^i, \tau_b^{i,0}, \tau_b^{i,1}$ to P_{1-b} to reconstruct $\sigma^i, \tau^{i,0}, \tau^{i,1}$.
10: **for** $j \in [2^i]$ **do**
11: P_b sets $s_b^{i+1,2j} \| s_b^{i+1,2j+1} := s_b^{i+1,2j} \| s_b^{i+1,2j+1} \oplus t_b^{i,j} \cdot \sigma^i$.
12: P_b sets $t_b^{i+1,2j} := t_b^{i+1,2j} \oplus t^{i,j} \cdot \tau^{i,0}$ and $t_b^{i+1,2j+1} := t_b^{i+1,2j+1} \oplus t^{i,j} \cdot \tau^{i,1}$
13: **end for**
14: P_b sets $CW_i := \sigma^i \| \tau^{i,0} \| \tau^{i,1}$ for $i \in [\ell_{\mathsf{in}}]$.
15: **end for**
16: P_b evaluates $w_b^i := \mathsf{Convert}_{\mathbb{G}}(s_b^{\ell_{\mathsf{in}}-1,i})$ for $i \in [2^{\ell_{\mathsf{in}}}]$.
17: P_b sets $w_b := \sum_{i\in[2^{\ell_{\mathsf{in}}}]} w_b^i$, $t_b := \sum_{i\in[2^{\ell_{\mathsf{in}}}]} t_b^{\ell_{\mathsf{in}},i}$.
18: P_b invokes $\mathcal{F}_{\mathsf{CCMP}}$ with input t_b to get $\langle g\rangle_b^B$.
19: P_b sets $\langle W_{CW}^0\rangle_b^{\ell_{\mathsf{out}}} := \langle\beta\rangle_b^{\ell_{\mathsf{out}}} + (-1)^{1-b} \cdot w_b$
20: P_b sets $\langle W_{CW}^1\rangle_b^{\ell_{\mathsf{out}}} := -\langle\beta\rangle_b^{\ell_{\mathsf{out}}} + (-1)^{b} \cdot w_b$
21: P_b invokes $\mathcal{F}_{\mathsf{MUX}}^{A,\ell^{\mathsf{out}}}$ with input $\langle W_{CW}^0\rangle_b^{\ell_{\mathsf{out}}}, \langle W_{CW}^1\rangle_b^{\ell_{\mathsf{out}}}, \langle g\rangle_b^B$ to get $\langle W_{CW}\rangle_b^{\ell_{\mathsf{out}}}$ where $W_{CW} = W_{CW}^g$.
22: P_b sends $\langle W_{CW}\rangle_b^{\ell_{\mathsf{out}}}$ to P_{1-b} to reconstruct W_{CW}.
23: P_b sets $k_b := s_b^{0,0} \| CW_0 \| \ldots \| CW_{\ell_{\mathsf{in}}-1} \| W_{CW}$.
24: **return** k_b

layer. This optimization can be seamlessly integrated into our protocol, reducing the number of invocations of $\mathcal{F}_{\mathsf{MUX}}^{B,\lambda}$ by ℓ_{in} in the tree traversing phase.

B. Dealer-less FSS key generation for DCF

Distributed comparison function is an important building block in a wide range of FSS-based protocols [5], [4]. We describe the detailed construction of our dealer-less DCF protocol that supports arithmetic-shared inputs and outputs.

We present our dealer-less DCF key generation protocol in Algorithm 3. Our construction can be viewed as a dealer-less extension of the dealer-based DCF [13]. The major difference between DCF and DPF key generation is calculating an extra correction word, namely V_{CW}. It corrects the random GGM tree and ensures the output satisfies the *DCF invariant*. Therefore, in our protocol, we compute V_{CW} for each layer (step 11), along with all the correction words in the DPF key (step 4-9).

To efficiently compute V_{CW}, we propose a *correlated DCF correction words generation* protocol, as outlined in Algorithm 4. We explain the insight of our design by first introducing how V_{CW} is computed in the prior dealer-based work [13]:

$$\begin{aligned} V_{CW} :=& (-1)^{t_1^{(i-1)}} \cdot (C_{\mathbb{G}}(v_1^{\mathsf{Lose}}) - C_{\mathbb{G}}(v_0^{\mathsf{Lose}}) - V_\alpha + \alpha[i]\cdot\beta) \\ V_\alpha :=& V_\alpha - C_{\mathbb{G}}(v_1^{\mathsf{Keep}}) + C_{\mathbb{G}}(v_0^{\mathsf{Keep}}) + (-1)^{t_1^{(i-1)}} \cdot V_{CW} \\ =& C_{\mathbb{G}}(v_0^{\mathsf{Keep}}) - C_{\mathbb{G}}(v_1^{\mathsf{Keep}}) - C_{\mathbb{G}}(v_0^{\mathsf{Lose}}) + C_{\mathbb{G}}(v_1^{\mathsf{Lose}}) + \alpha[i]\cdot\beta \end{aligned} \tag{5}$$

In the above equation, $t_1^{(i-1)}$ is the control bit of the node at the $(i-1)$-th layer held by P_1. The comparison label v_0^{Lose} (and v_1^{Lose}, respectively) is expanded from the node label at the previous layer held by P_0 (and P_1, respectively). The superscript Lose indicates the opposite side of the special path, while Keep denotes the same side as the special path.

Therefore, computing V_{CW} can be divided into two parts: evaluating the term $t_1^{(i-1)}$ and $C_{\mathbb{G}}(v_1^{\mathsf{Lose}}) - C_{\mathbb{G}}(v_0^{\mathsf{Lose}}) + \alpha[i]\cdot\beta$. Similar to our dealer-less DPF, it suffices to invoke our efficient Π_{CCMP} protocol to compute the former term. Without knowing the special path, the latter term is partitioned into $C_{\mathbb{G}}(v_1^{\mathsf{Lose}}) - C_{\mathbb{G}}(v_0^{\mathsf{Lose}})$ and $\alpha[i]\cdot\beta$, computed by two multiplexer invocations (step 3 and 5, Algorithm 4)

To update V_α, recalling the second relation in Equation 5, one may follow the same way that using multiplexer to obtain $-C_{\mathbb{G}}(v_1^{\mathsf{Keep}}) + C_{\mathbb{G}}(v_0^{\mathsf{Keep}})$. However, we observe that P_b can locally compute the following randomness by summing all left side (denoted with superscript L) and right side (denoted with superscript R) comparison labels v at the same layer without requiring knowledge of which side represents the special path (step 1, Algorithm 4):

$$\begin{aligned} &\Sigma_{b\in\{0,1\}}(-1)^b (C_{\mathbb{G}}(v_b^{\mathsf{L}}) + C_{\mathbb{G}}(v_b^{\mathsf{R}})) \\ =& \underbrace{C_{\mathbb{G}}(v_0^{\mathsf{L}}) - \Sigma C_{\mathbb{G}}(v_1^{\mathsf{L}})}_{\text{Part 1}} + \underbrace{C_{\mathbb{G}}(v_0^{\mathsf{R}}) - \Sigma C_{\mathbb{G}}(v_1^{\mathsf{R}})}_{\text{Part 2}} \end{aligned} \tag{6}$$

As we have obtained $C_{\mathbb{G}}(v_1^{\mathsf{Lose}}) - C_{\mathbb{G}}(v_0^{\mathsf{Lose}})$ (step 3, Algorithm 4), it corresponds to either part 1 or part 2 in the preceding equation. This suffices to compute $-C_{\mathbb{G}}(v_1^{\mathsf{Keep}}) + C_{\mathbb{G}}(v_0^{\mathsf{Keep}})$ by locally subtracting the output of the multiplexer (step 4, Algorithm 4) from the sum-up randomness in Equation 6. This approach effectively eliminates one multiplexer invocation of λ bits.

Complexity. Given the input and payload bit length to be ℓ, it requires $16\ell + 7$ offline rounds with the communication size of $\ell^2 + 17\lambda\ell + 2\lambda + 47\ell + 12$ bits to generate a DCF key. The evaluation process is identical to the prior work [13], which requires ℓ bits of communication size within 1 round.

Optimization for public payload. Although we provide general DCF construction for protecting both index α and payload β, in most cases, the payload is a public constant like 1. When there is no need to hide the payload, we can replace Step 5 in Algorithm 4 with a public multiplication, which reduces one $\mathcal{F}_{\mathsf{MUX}}^{A,\ell_{\mathsf{out}}}$ invocation.

Optimization from Half-tree [10]. With the correlated GGM construction of Half-tree [10], our DCF scheme further reduces two multiplexer invocations $\mathcal{F}_{\mathsf{MUX}}^{A,\ell_{\mathsf{out}}}$ for each layer during DCF correction word generation. As the result, the total

Algorithm 3 Dealer-less DCF Key Generation

$\mathsf{Gen}_{\mathsf{DCF}}^{\ell_{\mathsf{in}},\ell_{\mathsf{out}}}(b, \langle\alpha\rangle_b^{\ell_{\mathsf{in}}}, \langle\beta\rangle_b^{\ell_{\mathsf{out}}})$:

Input: P_b holds shares of position $\langle\alpha\rangle_b^{\ell_{\mathsf{in}}}$ and payload $\langle\beta\rangle_b^{\ell_{\mathsf{out}}}$.

Output: P_b obtains a DCF Key k_b.

1: P_b invokes $\mathcal{F}_{\mathsf{BitDec}}^{\ell_{\mathsf{in}}}$ with input $\langle\alpha\rangle_b^{\ell_{\mathsf{in}}}$ to get Boolean shares $\langle\alpha\rangle_b^B \in \{0,1\}^{\ell_{\mathsf{in}}}$.
2: P_b samples a random seed $s_b^{0,0}$ from $\{0,1\}^\lambda$ and sets $t_b^{0,0} := b, V_\alpha := 0$.
3: **for** $i \in [\ell_{\mathsf{in}}]$ **do**
4: P_b sets $Q^{i+1,2j} \| Q^{i+1,2j+1} := G(s_b^{i,j})$ for $j \in [2^i]$, where $Q^{i+1,2j+p} := s_b^{i+1,2j+p} \| v_b^{i+1,2j+p} \| t_b^{i+1,2j+p} \in \{0,1\}^\lambda \times \{0,1\}^\lambda \times \{0,1\}$ for $p \in \{0,1\}$
5: P_b sets $S_b^{i,p} := \bigoplus_{j\in[2^i]} s_b^{i+1,2j+p}$ for $p \in \{0,1\}$.
6: P_b sets $T_b^{i,p} := \bigoplus_{j\in[2^i]} t_b^{i+1,2j+p}$ for $p \in \{0,1\}$.
7: P_b invokes $\mathcal{F}_{\mathsf{MUX}}^{B,\lambda}$ with input $S_b^{i,0}, S_b^{i,1}, \langle\alpha[\ell_{\mathsf{in}}-1-i]\rangle_b^B \oplus b$ to get $\sigma_b^i := S_b^{i,\alpha[\ell_{\mathsf{in}}-1-i]\oplus 1}$.
8: P_b sets $\tau_b^{i,0} := T^{i,0} \oplus \langle\alpha[\ell_{\mathsf{in}}-1-i]\rangle_b^B \oplus b$ and $\tau_b^{i,1} := T^{i,1} \oplus \langle\alpha[\ell_{\mathsf{in}}-1-i]\rangle_b^B$.
9: P_b sends $\sigma_b^i, \tau_b^{i,0}, \tau_b^{i,1}$ to P_{1-b} to reconstruct $\sigma^i, \tau^{i,0}, \tau^{i,1}$.
10: P_b sets $\mathbf{v}[i] := v_b^{i+1,j}$ for $j \in [2^{i+1}]$
11: P_b sets $\mathbf{t}[i] := t_b^{i,j}$ for $j \in [2^i]$
12: P_b calls $\mathsf{ComputeVCW}(b, i, \mathbf{v}, \mathbf{t}, \langle\alpha[\ell_{\mathsf{in}}-1-i]\rangle_b^B, V_\alpha)$ to get (V_{CW}, V_α)
13: P_b sets $CW^i := \sigma^i \| V_{CW} \| \tau^{i,0} \| \tau^{i,1}$
14: **for** $j \in [2^i]$ **do**
15: P_b sets $s_b^{i+1,2j} \| s_b^{i+1,2j+1} := s_b^{i+1,2j} \| s_b^{i+1,2j+1} \oplus t_b^{i,j} \cdot \sigma^i$
16: P_b sets $t_b^{i+1,2j} := t_b^{i+1,2j} \oplus t_b^{i,j} \cdot \tau^{i,0}$ and $t_b^{i+1,2j+1} := t_b^{i+1,2j+1} \oplus t_b^{i,j} \cdot \tau^{i,1}$
17: **end for**
18: **end for**
19: P_b evaluates $w_b^i := \mathsf{Convert}_{\mathbb{G}}(s_b^{\ell_{\mathsf{in}},i})$ for $i \in [2^{\ell_{\mathsf{in}}}]$.
20: P_b sets $w_b := \sum_{i\in[2^{\ell_{\mathsf{in}}}]} w_b^i$, $t_b := \sum_{i\in[2^{\ell_{\mathsf{in}}}]} t_b^{\ell_{\mathsf{in}},i}$.
21: P_b invokes $\mathcal{F}_{\mathsf{CCMP}}$ with input t_b to get $\langle g\rangle_b^B$.
22: P_b sets $\langle W_{CW}^0\rangle_b^{\ell_{\mathsf{out}}} := (-1)^{1-b} \cdot w_b - V_\alpha$ and $\langle W_{CW}^1\rangle_b^{\ell_{\mathsf{out}}} := (-1)^b \cdot w_b - V_\alpha$.
23: P_b invokes $\mathcal{F}_{\mathsf{MUX}}^{A,\ell_{\mathsf{out}}}$ with input $\langle W_{CW}^0\rangle_b^{\ell_{\mathsf{out}}}, \langle W_{CW}^1\rangle_b^{\ell_{\mathsf{out}}}, \langle g\rangle_b^B$ to get $\langle W_{CW}\rangle_b^{\ell_{\mathsf{out}}}$ where $W_{CW} = W_{CW}^g$.
24: P_b sends $\langle W_{CW}\rangle_b^{\ell_{\mathsf{out}}}$ to P_{1-b} to reconstruct W_{CW}.
25: P_b sets $k_b := s_b^{0,0} \| CW^0 \| \ldots \| CW^{\ell_{\mathsf{in}}} \| W_{CW}$
26: **return** k_b

amount of $\mathcal{F}_{\mathsf{MUX}}^{A,\ell_{\mathsf{out}}}$ and $\mathcal{F}_{\mathsf{MUX}}^{B,\lambda}$ invocations can be reduced by $2\ell_{\mathsf{in}}$ and ℓ_{in} respectively.

V. Building Block Protocols

Built upon the 2PC-based FSS schemes presented in Section IV, this section provides efficient secure protocols for important building blocks used in the consequent framework for complex function evaluation, including equality test, comparison, truncation, interval containment, and digit decomposition.

A. Equality test and comparison

Equality test and comparison are two key components in our building block protocols and complex function evaluation framework.

Algorithm 4 Correlated DCF Correction Word Generation

$\mathsf{ComputeVCW}(b, i, \mathbf{v}, \mathbf{t}, \langle\alpha[i]\rangle_b^B, \langle\beta\rangle_b^{\ell_{\mathsf{out}}}, V_\alpha)$:

Input: P_b holds the depth of correction words i, seeds vector $\mathbf{v}$, control bits vector $\mathbf{t}$, shares of the input bit $\langle\alpha[i]\rangle_b^B$ and shares of payload $\langle\beta\rangle_b^{\ell_{\mathsf{out}}}$.

Output: P_b obtains public correction words of i-th layer V_{CW}, and the updated $\hat{V_\alpha}$.

1: P_b evaluates $V_b^{i,p} := \Sigma_{j\in[2^{i+1}]} \mathsf{Convert}_{\mathbb{G}}(\mathbf{v}[2j+p])$ for $p \in \{0,1\}$
2: P_b sets $t_b := \Sigma_{j\in[2^i]} \mathbf{t}[j]$ and invokes $\mathcal{F}_{\mathsf{CCMP}}$ with input t_b to get $\langle g\rangle_B^B$
3: P_b invokes the multiplexer functionality $\mathcal{F}_{\mathsf{MUX}}^{A,\ell_{\mathsf{out}}}$ with input $(-1)^{1-b} \cdot V_b^{i,0}, (-1)^{1-b} \cdot V_b^{i,1}, \langle\alpha[i]\rangle_b^B \oplus b$ to get $\langle\phi\rangle_b^{\ell_{\mathsf{out}}} := \langle V_1^{i,\alpha[i]\oplus 1} - V_0^{i,\alpha[i]\oplus 1}\rangle_b^{\ell_{\mathsf{out}}}$.
4: P_b sets $\langle\theta\rangle_b^{\ell_{\mathsf{out}}} := (-1)^b \cdot (V_b^{i,0} + V_b^{i,1}) + \langle\phi\rangle_b^{\ell_{\mathsf{out}}}$
5: P_b invokes $\mathcal{F}_{\mathsf{MUX}}^{A,\ell_{\mathsf{out}}}$ with input $\langle 0\rangle_b^{\ell_{\mathsf{out}}}, \langle\beta\rangle_b^{\ell_{\mathsf{out}}}, \langle\alpha[i]\rangle_b^B$ to get $\langle\eta\rangle_b^{\ell_{\mathsf{out}}} := \langle\alpha[i] \cdot \beta\rangle_b$.
6: P_b sets $\langle V_{CW}^0\rangle_b^{\ell_{\mathsf{out}}} := \langle\phi\rangle_b^{\ell_{\mathsf{out}}} - (1-b) \cdot V_\alpha + \langle\eta\rangle_b^{\ell_{\mathsf{out}}}$, $\langle V_{CW}^1\rangle_b^{\ell_{\mathsf{out}}} := -\langle\phi\rangle_b^{\ell_{\mathsf{out}}} + (1-b) \cdot V_\alpha - \langle\eta\rangle_b^{\ell_{\mathsf{out}}}$
7: P_b invokes $\mathcal{F}_{\mathsf{MUX}}^{A,\ell_{\mathsf{out}}}$ with input $\langle V_{CW}^0\rangle_b^{\ell_{\mathsf{out}}}, \langle V_{CW}^1\rangle_b^{\ell_{\mathsf{out}}}, \langle g\rangle_b^B$ to get $\langle V_{CW}\rangle_b^{\ell_{\mathsf{out}}}$ where $V_{CW} = V_{CW}^g$.
8: P_b sends $\langle V_{CW}\rangle_b^{\ell_{\mathsf{out}}}$ to P_{1-b} to reconstruct V_{CW}
9: P_b sets $\langle\hat{V_\alpha}\rangle_b^{\ell_{\mathsf{out}}} := \langle\theta\rangle_b^{\ell_{\mathsf{out}}} - \langle\phi\rangle_b^{\ell_{\mathsf{out}}} + \langle\eta\rangle_b^{\ell_{\mathsf{out}}}$
10: P_b sends $\langle\hat{V_\alpha}\rangle_b^{\ell_{\mathsf{out}}}$ to P_{1-b} to reconstruct $\hat{V_\alpha}$
11: **return** $(V_{CW}, \hat{V_\alpha})$

At the offline stage, the equality test takes as input the arithmetic shares of target value $\langle\alpha\rangle^{\ell_{\mathsf{in}}} \in \mathbb{Z}_{2^{\ell_{\mathsf{in}}}}$ and payload $\langle\beta\rangle^{\ell_{\mathsf{out}}} \in \mathbb{Z}_{2^{\ell_{\mathsf{out}}}}$ and outputs equality test keys for two parties. At the online stage, it takes as input the arithmetic shares of $\langle x\rangle^{\ell_{\mathsf{in}}} \in \mathbb{Z}_{2^{\ell_{\mathsf{in}}}}$, and outputs $\langle\beta\rangle^{\ell_{\mathsf{out}}} \in \mathbb{Z}_{2^{\ell_{\mathsf{out}}}}$ if $x = \alpha$, otherwise $\langle 0\rangle^{\ell_{\mathsf{out}}} \in \mathbb{Z}_{2^{\ell_{\mathsf{out}}}}$. We follow the construction based on the FSS offset gate [15]. It transforms checking $x = \alpha$ into $x + r = \alpha + r$ by generating a random mask r at the offline stage, thus preserving the correctness of the equality test. We present the detailed protocol in Algorithm 5.

The comparison operation takes as input the arithmetic shares of target value $\langle\alpha\rangle^{\ell_{\mathsf{in}}} \in \mathbb{Z}_{2^{\ell_{\mathsf{in}}}}$ and payload $\langle\beta\rangle^{\ell_{\mathsf{out}}} \in \mathbb{Z}_{2^{\ell_{\mathsf{out}}}}$, and outputs comparison keys for two parties at the offline stage. Correspondingly, at the online stage, it takes as input the arithmetic shares of $\langle x\rangle^{\ell_{\mathsf{in}}} \in \mathbb{Z}_{2^{\ell_{\mathsf{in}}}}$, and output $\langle\beta\rangle^{\ell_{\mathsf{out}}} \in \mathbb{Z}_{2^{\ell_{\mathsf{out}}}}$ if $x < \alpha$, otherwise $\langle 0\rangle^{\ell_{\mathsf{out}}} \in \mathbb{Z}_{2^{\ell_{\mathsf{out}}}}$. To be compatible with the arithmetic-shared input, it is necessary to determine if an overflow occurs after adding the random mask to the input share. If an overflow occurs when adding α and r_{in} in ring $\mathbb{Z}_{2^{\ell_{\mathsf{in}}}}$, the correct comparison result should be $\mathbb{1}\{x + r_{\mathsf{in}} \in [(\alpha + r_{\mathsf{in}}) \mod 2^{\ell_{\mathsf{in}}}, r_{\mathsf{in}}]\}$. This problem can be easily resolved when the dealer has the knowledge of both α and r_{in} [13]. Without the dealer, to construct the correct comparison scheme supporting arithmetic shared input while maintaining extremely low online latency, we integrate secure comparison via invoking the Millionaire protocol [11] to check the overflow. Although this check can be performed directly by the comparison between $\alpha + r_{\mathsf{in}}$ and $2^{\ell_{\mathsf{in}}}$, we opt for the comparison between two ring elements r_{in} and $\alpha + r_{\mathsf{in}}$

Algorithm 5 DPF-based Equality Test

$\mathsf{Gen}_{\mathsf{EQ}}^{\ell_{\mathsf{in}},\ell_{\mathsf{out}}}(b, \langle\alpha\rangle_b^{\ell_{\mathsf{in}}}, \langle\beta\rangle_b^{\ell_{\mathsf{out}}})$:

Input: P_b holds shares of target value $\langle\alpha\rangle_b^{\ell_{\mathsf{in}}}$ and payload $\langle\beta\rangle_b^{\ell_{\mathsf{out}}}$.

Output: P_b obtains an Equality Test Key k_b.

1: P_b samples $\langle r\rangle_b^{\ell_{\mathsf{in}}} \in \mathbb{Z}_{2^{\ell_{\mathsf{in}}}}$.
2: P_b evaluates $\hat{k}_b := \mathsf{Gen}_{\mathsf{DPF}}^{\ell_{\mathsf{in}},\ell_{\mathsf{out}}}(b, \langle r\rangle_b^{\ell_{\mathsf{in}}} + \langle\alpha\rangle_b^{\ell_{\mathsf{in}}}, \langle\beta\rangle_b^{\ell_{\mathsf{out}}})$
3: P_b sets $k_b := \langle r\rangle_b^{\ell_{\mathsf{in}}} \| \hat{k}_b$
4: **return** k_b

$\mathsf{Eval}_{\mathsf{EQ}}^{\ell_{\mathsf{in}},\ell_{\mathsf{out}}}(b, k_b, \langle x\rangle_b^{\ell_{\mathsf{in}}})$:

Input: P_b holds shares of input value $\langle x\rangle_b^{\ell_{\mathsf{in}}}$ and key k_b.

Output: P_b obtains $\langle y\rangle_b^{\ell_{\mathsf{out}}}$ where $y = \beta \cdot \mathbb{1}\{x = \alpha\}$.

1: P_b parses k_b as $\langle r\rangle_b^{\ell_{\mathsf{in}}} \| \hat{k}_b$
2: P_b sends $\langle x\rangle_b^{\ell_{\mathsf{in}}} + \langle r\rangle_b^{\ell_{\mathsf{out}}}$ to P_{1-b} and reconstructs $x + r \in \mathbb{Z}_{2^{\ell_{\mathsf{in}}}}$
3: P_b evaluates $\langle y\rangle_b^{\ell_{\mathsf{out}}} := \mathsf{Eval}_{\mathsf{DPF}}^{\ell_{\mathsf{in}},\ell_{\mathsf{out}}}(b, x + r, \hat{k}_b)$
4: **return** $\langle y\rangle_b^{\ell_{\mathsf{out}}}$

mod $2^{\ell_{\mathsf{in}}}$ under fixed-point arithmetic within ℓ_{in} bits rather than $\ell_{\mathsf{in}} + 1$ bits. We present the detailed protocol Π_{CMP} in Algorithm 6.

Complexity. The equality test requires one DPF invocation. Therefore, it introduces $9\ell+5$ offline rounds with $18\lambda\ell+5\lambda+13\ell+2$ bits to communicate at the offline stage, and 1 round of ℓ bits data exchange at the online stage. The comparison requires 2 DCF invocations with one $\mathcal{F}_{\mathsf{MUL}}$, one $\mathcal{F}_{\mathsf{MILL}}$ and one $\mathcal{F}_{\mathsf{B2A}}$. Totally, it costs $32\ell + \log\ell + 16$ rounds with $2\lambda^2\ell + 2\lambda\ell^2 + 2\ell^2 + 35\lambda\ell + 5\lambda + 111\ell + 24$ bits at the offline stage and requires 1 round with ℓ bit communication for evaluation.

B. Truncation

The truncation operation on an ℓ-bit input $\langle x\rangle^{\ell} \in \mathbb{Z}_{2^\ell}$ can be represented as $y = x \gg s \in \mathbb{Z}_{2^{\ell-s}}$, where s is the truncation bit length. The functionality of our truncation protocol is identical to the truncate-and-reduce protocol proposed in SIRNN [8]. Given that $\langle x\rangle_b = \langle u\rangle_b \| \langle v\rangle_b$ with $u \in \{0,1\}^{\ell-s}$ and $v \in \{0,1\}^s$, the truncation operation can be further formulated as

$$\mathsf{Trunc}^s(\langle x\rangle_b^{\ell_{\mathsf{in}}}) := \begin{cases} \langle u\rangle_b, & \langle v\rangle_0 + \langle v\rangle_1 < 2^s \\ \langle u+1\rangle_b, & \langle v\rangle_0 + \langle v\rangle_1 \geq 2^s \end{cases} \quad (7)$$

Specifically, a secure comparison protocol is necessary on the last s bits to determine whether their sum results in an overflow into the $s+1$ bits. This can be achieved by employing an $(s+1)$-bit comparison. The detailed truncation protocol is given in Algorithm 7.

Complexity. Our truncation protocol invokes an $(s+1)$-bit comparison, which requires $s+1$ bits of communication within 1 round for evaluation.

C. Interval containment

The interval containment operation takes as input the public interval list at the offline stage and generates interval containment keys for the two parties. At the online stage, it inputs the arithmetic shares of $\langle x\rangle^{\ell_{\mathsf{in}}} \in \mathbb{Z}_{2^{\ell_{\mathsf{in}}}}$, and outputs

Algorithm 6 DCF-based Comparison

$\mathsf{Gen}_{\mathsf{CMP}}^{\ell_{\mathsf{in}},\ell_{\mathsf{out}}}(b, \langle\alpha\rangle_b^{\ell_{\mathsf{in}}}, \langle\beta\rangle_b^{\ell_{\mathsf{out}}})$:

Input: P_b holds shares of target value $\langle\alpha\rangle_b^{\ell_{\mathsf{in}}}$ and payload $\langle\beta\rangle_b^{\ell_{\mathsf{out}}}$.

Output: P_b obtains a Comparison Key k_b

1: P_b samples $\langle r\rangle_b^{\ell_{\mathsf{in}}} \in \mathbb{Z}_{2^{\ell_{\mathsf{in}}}}$
2: P_b invokes $\mathcal{F}_{\mathsf{MILL}}^{\ell_{\mathsf{in}}}$ with inputs $\langle r\rangle_b^{\ell_{\mathsf{in}}}$ and $\langle r\rangle_b^{\ell_{\mathsf{in}}} + \langle\alpha\rangle_b^{\ell_{\mathsf{in}}}$ to get $\langle g\rangle_b^B$
3: P_b invokes $\mathcal{F}_{\mathsf{B2A}}^{\ell_{\mathsf{out}}}$ with input $\langle g\rangle_b^B$ to get $\langle g\rangle_b^{\ell_{\mathsf{out}}}$
4: P_b invoked $\mathcal{F}_{\mathsf{MUL}}$ with inputs $\langle g\rangle_b^{\ell_{\mathsf{out}}}$ and $\langle\beta\rangle_b^{\ell_{\mathsf{out}}}$ to get $\langle\hat{g}\rangle_b^{\ell_{\mathsf{out}}}$
5: P_b evaluates $\hat{k}_b^0 := \mathsf{Gen}_{\mathsf{DCF}}^{\ell_{\mathsf{in}},\ell_{\mathsf{out}}}(b, \langle r\rangle_b^{\ell_{\mathsf{in}}}, -\langle\beta\rangle_b^{\ell_{\mathsf{out}}})$
6: P_b evaluates $\hat{k}_b^1 := \mathsf{Gen}_{\mathsf{DCF}}^{\ell_{\mathsf{in}},\ell_{\mathsf{out}}}(b, \langle r\rangle_b^{\ell_{\mathsf{in}}} + \langle\alpha\rangle_b^{\ell_{\mathsf{in}}}, \langle\beta\rangle_b^{\ell_{\mathsf{out}}})$
7: P_b sets $k_b := \langle r\rangle_b^{\ell_{\mathsf{in}}} \| \hat{k}_b^0 \| \hat{k}_b^1 \| \langle\hat{g}\rangle_b^{\ell_{\mathsf{out}}}$
8: **return** k_b

$\mathsf{Eval}_{\mathsf{CMP}}^{\ell_{\mathsf{in}},\ell_{\mathsf{out}}}(b, k_b, \langle x\rangle_b^{\ell_{\mathsf{in}}})$:

Input: P_b holds shares of input value $\langle x\rangle_b^{\ell_{\mathsf{in}}}$ and key k_b.

Output: P_b obtains $\langle y\rangle_b^{\ell_{\mathsf{out}}}$ where $y = \beta\{x < \alpha\}$.

1: P_b parses k_b as $\langle r\rangle_b^{\ell_{\mathsf{in}}} \| \hat{k}_b^0 \| \hat{k}_b^1 \| \langle\hat{g}\rangle_b^{\ell_{\mathsf{out}}}$
2: P_b sends $\langle x\rangle_b^{\ell_{\mathsf{in}}} + \langle r\rangle_b^{\ell_{\mathsf{out}}}$ to P_{1-b} and reconstructs $x + r \in \mathbb{Z}_{2^{\ell_{\mathsf{in}}}}$
3: P_b evaluates $\langle y^0\rangle_b^{\ell_{\mathsf{out}}} := \mathsf{Eval}_{\mathsf{DCF}}^{\ell_{\mathsf{in}},\ell_{\mathsf{out}}}(b, x + r, \hat{k}_b^0)$
4: P_b evaluates $\langle y^1\rangle_b^{\ell_{\mathsf{out}}} := \mathsf{Eval}_{\mathsf{DCF}}^{\ell_{\mathsf{in}},\ell_{\mathsf{out}}}(b, x + r, \hat{k}_b^1)$
5: P_b sets $\langle y\rangle_b^{\ell_{\mathsf{out}}} := \langle y^0\rangle_b^{\ell_{\mathsf{out}}} + \langle y^1\rangle_b^{\ell_{\mathsf{out}}} + \langle\hat{g}\rangle_b^{\ell_{\mathsf{out}}}$
6: **return** $\langle y\rangle_b^{\ell_{\mathsf{out}}}$

Algorithm 7 Truncate and Reduce, $\Pi_{\mathsf{TR}}^{\ell,s}$

Input: P_b holds $\langle x\rangle_b^{\ell}$ and public truncation bit length s.

Output: P_b obtains $\langle y\rangle_b^{\ell-s}$ where $y = x \gg s \bmod 2^{\ell-s}$.

Offline Phase:

1: P_b invokes $\mathsf{Gen}_{\mathsf{CMP}}^{\mathsf{s+1},\ell-\mathsf{s}}(b, \langle 2^s\rangle, \langle 1\rangle)$ to obtain k_b.

Online Phase:

1: P_b parses input $\langle x\rangle_b^{\ell}$ as an ℓ-bit string $u_b || v_b$, where $u_b \in \{0,1\}^{\ell-s}$, $v_b \in \{0,1\}^s$.
2: P_b invokes $\mathsf{Eval}_{\mathsf{CMP}}^{\mathsf{s+1},\ell-\mathsf{s}}(b, k_b, \langle t\rangle_b^{s+1})$ and learns $\langle c\rangle_b^{\ell-s}$, where $\langle t\rangle_b^{s+1} = v_b \in \{0,1\}^{s+1}$.
3: P_b sets $\langle y\rangle_b^{\ell-s} := u_b + \langle c\rangle_b^{\ell-s} \mod 2^{\ell-s}$.

the one-hot vector indicating the interval that x belongs to. Specifically, each interval $I_i = [d_i, d_{i+1})$, for $i \in [k]$, is defined by two public knots d_i and d_{i+1}, and this operation outputs the vector $\langle \mathbf{v}\rangle^{\ell_{\mathsf{out}}}$ satisfying $\mathbf{v}^{\ell_{\mathsf{out}}}[i] = \mathbb{1}\{x \in I_i\}$. Further, this operation can be represented as

$$\begin{aligned}\mathsf{Ctn}^k(\langle x\rangle_b^{\ell_{\mathsf{in}}})[i] :=& (1 - \mathbb{1}\{\langle x\rangle_0 + \langle x\rangle_1 \bmod 2^{\ell_{\mathsf{in}}} < d_i\}) \\ & \cdot \mathbb{1}\{\langle x\rangle_0 + \langle x\rangle_1 \bmod 2^{\ell_{\mathsf{in}}} < d_{i+1}\}\end{aligned} \quad (8)$$

Therefore, this operation can be securely evaluated with secure multiplications and comparison invocations. The detailed protocol is given in Algorithm 8.

Optimization when the public knot is 0 **or** $2^\ell - 1$**.** In most cases like an ℓ-bit spline approximation, the first interval is always $[0, d_0)$, and the last interval is $[d_k, 2^\ell)$. In this setting, we do not need to evaluate comparison for 0 and $2^\ell - 1$ as the result is fixed under integer representation, namely 0 for

Algorithm 8 Secure Containment, $\Pi^{\ell_{\mathsf{in}},\ell_{\mathsf{out}},k}_{\mathsf{Ctn}}$

Input: P_b holds $\langle x\rangle_b^{\ell_{\mathsf{in}}}$, knots number $k,(k>2)$, with $k-1$ intervals $[d_i,d_{i+1}), i\in[0,k-1], d_0=0, d_{k-1}=2^{\ell_{\mathsf{in}}-1}$, with ℓ_{in}-bit input

Output: P_b obtains a one-hot vector $\langle \mathbf{v}\rangle_b^{\ell_{\mathsf{out}}}$, where $\mathbf{v}[i]=\mathbb{1}\{x\in[d_i,d_{i+1})\}$

Offline Stage:

1: P_b invokes $\mathsf{Gen}^{\ell_{\mathsf{in}},\ell_{\mathsf{out}}}_{\mathsf{CMP}}(b,d_i,1)$ to obtain k_b^i for $i\in[0,k-2]$.

2: P_b performs $k-3$ offline operations of $\mathcal{F}_{\mathsf{MUL}}$

Online Stage:

1: P_b computes $\langle \mathbf{c}[i]\rangle_b^{\ell_{\mathsf{out}}} := \mathsf{Eval}^{\ell_{\mathsf{in}},\ell_{\mathsf{out}}}_{\mathsf{CMP}}(b,k_b^i,\langle x\rangle_b^{\ell_{\mathsf{in}}})$ for $i\in[0,k-1]$.

2: P_b computes $\langle \mathbf{v}\rangle_b^{\ell_{\mathsf{out}}}$ by evaluating $\langle \mathbf{v}[i]\rangle_b^{\ell_{\mathsf{out}}} := \mathcal{F}^{\ell_{\mathsf{out}}}_{\mathsf{MUL}}(\langle \mathbf{c}[i]\rangle_b^{\ell_{\mathsf{out}}},(b-\langle \mathbf{c}[i+1]\rangle_b^{\ell_{\mathsf{out}}}))$ for $i\in[1,k-3]$, $\langle \mathbf{v}[0]\rangle_b^{\ell_{\mathsf{out}}} := \langle \mathbf{c}[0]\rangle_b^{\ell_{\mathsf{out}}}$, and $\langle \mathbf{v}[k-2]\rangle_b^{\ell_{\mathsf{out}}} := b-\langle \mathbf{c}[2]\rangle_b^{\ell_{\mathsf{out}}}$.

comparing with 0, β for comparing with $2^\ell-1$. As such, the secure multiplication to evaluate $\mathbf{v}[0]$ and $\mathbf{v}[k+1]$ can also be removed and transformed to the local public constant multiplication.

Complexity. Given the knot number k, we invoke k comparisons on ℓ-bit input with addition $k-3$ parallel secure multiplications. Consequently, the online phase of $\Pi^{\ell_{\mathsf{in}},\ell_{\mathsf{out}},k}_{\mathsf{Ctn}}$ requires $\ell_{\mathsf{in}}(3k-2)$ bits of communication within 2 rounds.

D. Digit decomposition

A digit decomposition protocol $\Pi^{\ell,c,d}_{\mathsf{DigDec}}$ takes as input the original bit length ℓ, segment number c and new bit length $d=\ell/c$ at the offline stage. It inputs an ℓ-bit arithmetic value $\langle x\rangle^\ell$ at online stage, and decomposes it into c segments $x[c-1]\|\ldots\|x[0], x[i]\in\{0,1\}^d$ with $d=l/c$ bits for each segment. Our detailed digit decomposition protocol is given in Algorithm 9. For the i-th segment, given $\mathsf{carry}_i=1$ denoting the overflow from the $i-1$-th segment, we formulate the digit decomposition operation as

$$\mathsf{DigDec}^c(\langle x\rangle_b^\ell)[i] := \langle x[i]\rangle_0+\langle x[i]\rangle_1+\mathsf{carry}_i \tag{9}$$

To achieve this operation, the primary objective is to determine whether an overflow has occurred from the preceding segment and also within the current segment. We observe that there are two possible conditions for an overflow. First, if the sum of $\langle x[i]\rangle_0$ and $\langle x[i]\rangle_1$ exceeds 2^d-1, it indicates that the cumulative value of the current segment surpasses the maximum capacity of a d-bit value. Alternatively, an overflow can occur when $\langle x[i]\rangle_0+\langle x[i]\rangle_1=2^d-1$ and there is a carryover from the preceding segment. As such, we check 2 possible conditions for overflow in Step 2. Intuitively, one might need to evaluate the logical OR between these two conditions. However, we would like to emphasize that it is sufficient to add the evaluation results of the equality test and comparison for 2^d-1 together to accomplish the digit decomposition, without resorting to additional OR operations, as both conditions cannot happen simultaneously [8].

Complexity. We employ a $(d+1)$-bit comparison to check potential overflow in the current segment and a d-bit equality test plus a secure multiplication to verify if the overflow is

Algorithm 9 Secure Digit Decomposition Protocol, $\Pi^{\ell,c,d}_{\mathsf{DigDec}}$

Input: P_b holds $\langle x\rangle_b^\ell$, segment number c, digit bit length $d=\ell/c$.

Output: P_b obtains $\langle z\rangle_b^d$, where $x=z[c-1]||...||z[1]||z[0]$ with new bit length $d=\ell/c$.

Offline Stage:

1: P_b invokes equality test key generation as $\mathsf{Gen}^{d,d}_{\mathsf{EQ}}(b,\langle 2^d-1\rangle_b^d,\langle 1\rangle_b^d)$ to obtain $c-1$ DPF keys $k_b^{\mathsf{EQ},i}$ for $i\in[c-1]$.

2: P_b invokes comparison key generation as $\mathsf{Gen}^{d+1,d}_{\mathsf{CMP}}(b,\langle 2^d\rangle_b^{d+1},\langle 1\rangle_b^d)$ to obtain $c-1$ comparison keys $k_b^{\mathsf{LT},i}$ for $i\in[c-1]$.

3: P_b performs $c-1$ offline operations of $\mathcal{F}_{\mathsf{MUL}}$

Online Stage:

1: P_b parses $\langle x\rangle_b^\ell$ as $\langle p\rangle_b^d[c-1]||...||\langle p\rangle_b^d[1]||\langle p\rangle_b^d[0]$ and sets $\langle \mathbf{z}\rangle_b^d[0] := \langle p\rangle_b^d[0]$, $\langle u\rangle_b^d[0] := 0$

2: P_b invokes $\mathsf{Eval}^{d+1,d}_{\mathsf{CMP}}(b,k_b^{\mathsf{LT},i},\langle p[i]\rangle_b^{d+1})$ to obtain $\langle w\rangle_b^d[0],...,\langle w\rangle_b^d[i]$, and $\mathsf{Eval}^{d,d}_{\mathsf{EQ}}(b,k_b^{\mathsf{EQ},i},\langle p[i]\rangle_b^d)$ to obtain $\langle e\rangle_b^d[1],...,\langle e\rangle_b^d[i]$ for $i\in[0,c-2]$.

3: **for** $i\in[1,c-1]$ **do**

4: $\quad P_0$ and P_1 invokes $\mathcal{F}^d_{\mathsf{MUL}}(\langle u[i-1]\rangle_b^d,\langle e[i-1]\rangle_b^d)$ to obtain $\langle v[i-1]\rangle_b^d$

5: $\quad P_b$ sets $\langle u[i]\rangle_b^d := \langle v[i-1]\rangle_b^d+\langle w[i-1]\rangle_b^d$

6: $\quad P_b$ sets $\langle z[i]\rangle_b^d := \langle p[i]\rangle_b^d+\langle u[i]\rangle_b^d$

7: **end for**

attributed to the preceding carry. Consequently, the online phase of the protocol $\Pi^{\ell,c,d}_{\mathsf{DigDec}}$ requires $(c-1)(4d+1)$ bits of communication within $c+1$ rounds.

VI. Math Functions in Scientific Computation

Scientific computation is fundamental to systems supporting various time-critical applications, such as autonomous vehicle positioning [25], proximity detection in online navigation [26], [27], and biometric authentication [28]. These services demand low-latency processing of sensitive input data while maintaining high accuracy. We introduce our optimized implementation of widely used scientific functions, specifically trigonometric functions, built on the building blocks discussed in the previous section. Our design achieves minimal communication overhead and latency while preserving accuracy.

A. Optimized spline polynomial approximation and lookup table protocol

Many mathematical functions, such as sine, tangent, and exponential, pose challenges in efficiently evaluating them within the MPC environment [1]. There are two common methods for secure evaluation. The first is the lookup table (LUT) technique [3], [8], which requires storing all possible input-output pairs. For an ℓ_{in}-bit secret-shared input, this method involves $2^{\ell_{\mathsf{in}}}$ secure equality tests, resulting in a communication complexity of $O(2^{\ell_{\mathsf{in}}})$, which is impractically expensive. The second method approximates functions using piecewise polynomials [4]. With k intervals, $O(k)$ secure comparisons are needed to identify the input's interval, which becomes costly as k increases. To address the inefficiencies of these methods, we propose optimized constructions leveraging our previously described building blocks.

Algorithm 10 Public Lookup Table Protocol, $\Pi_{\mathsf{pubLUT}}^{\ell_{\mathsf{in}},\ell_{\mathsf{out}},T_{\mathsf{pub}}}$

Input: P_b holds $\langle x\rangle_b^{\ell_{\mathsf{in}}}$ and a public lookup table T with $2^{\ell_{\mathsf{in}}}$ ℓ_{out}-bit entries.
Output: P_b obtains $\langle y\rangle_b^{\ell_{\mathsf{out}}}$ where $y = T[x]$.
Offline Phase:
1: P_b randomly samples $\langle r\rangle_b^{\ell_{\mathsf{in}}} \in \{0,1\}^{\ell_{\mathsf{in}}}$.
2: P_b invokes $\mathsf{Gen}_{\mathsf{DPF}}^{\ell_{\mathsf{in}},\ell_{\mathsf{out}}}(b, \langle r\rangle_b^{\ell_{\mathsf{in}}}, \langle 1\rangle_b^{\ell_{\mathsf{out}}})$ to obtain k_b.
Online Phase:
1: P_b sends $\langle x\rangle_b^{\ell_{\mathsf{in}}} - \langle r\rangle_b^{\ell_{\mathsf{in}}}$ to P_{1-b}, to reconstruct $x - r$.
2: P_b computes $\langle \mathbf{z}[i]\rangle_b^{\ell_{\mathsf{out}}} := \mathsf{Eval}_{\mathsf{DPF}}^{\ell_{\mathsf{in}},\ell_{\mathsf{out}}}(b, k_b, i)$ for $i \in [2^{\ell_{\mathsf{in}}}]$.
3: P_b circularly shifts the vector $\langle \mathbf{z}\rangle_b^{\ell_{\mathsf{out}}}$ by $x-r$ to get $\langle \mathbf{z}'\rangle_b^{\ell_{\mathsf{out}}}$.
4: P_b outputs $\langle y\rangle_b^{\ell_{\mathsf{out}}} := \sum_{i\in[2^{\ell_{\mathsf{in}}}]} \langle \mathbf{z}'[i]\rangle_b^{\ell_{\mathsf{out}}} \cdot T[i]$.

Efficient public lookup table protocol. By carefully utilizing the DPF technique, we propose an efficient lookup table protocol that achieves a communication complexity of $O(\ell_{\text{in}})$, instead of $O(2^{\ell_{\mathsf{in}}})$ complexity in prior works. During the offline stage, the parties generate the DPF key on a randomly sampled location r in the table T. This ensures the offline stage is input-independent. During the online stage, parties jointly evaluate the DPF on the public masked input $x+r$ and get the one-hot vector $\mathbf{z}$, where $\mathbf{z}[r] = \beta$. To get the correct result, namely $T(x)$, we circularly shift $\mathbf{z}$ by $x-r$. Based on this vector, the parties can easily obtain the secret-shared output $y = T(x)$, by executing the inner product operation once. It is worth noting that the offline overhead of our protocol is sub-linear with the table size, and in the online phase, the parties only communicate 2 ring elements within 1 round. We present our public lookup table protocol Algorithm 10.

Extending to the private table. Although the private LUT protocol can be derived from $\Pi_{\mathsf{pubLUT}}^{\ell_{\mathsf{in}},\ell_{\mathsf{out}},T_{\mathsf{pub}}}$, we provide an alternative for private LUT illustrated in Algorithm 11 to avoid costly 2^{ℓ}_{in} secure multiplications. In our protocol, we generate DPF keys for each table entry shifted by r at the offline stage, whose payload is the secret value in the table. Then the party evaluates DPF for every position to get respective DPF output on public FSS input $x + r$. For the x-th result, it will be shared in the private table and shares of 0 in other indexes. Finally, we sum them together to get the protocol output. Our protocol may suffer from large overhead at the offline stage, but achieve extreme efficiency at the online stage with 1 round of interaction to communicate 1 ring element. We carefully invoke this protocol to reduce offline overhead in our math function implementation. We present the complete protocol for our private lookup table in Algorithm 11.

Efficient spline polynomial approximation protocol. Combining the FSS offset gate with polynomial approximation requires two key steps: generating correct coefficients for the shifted function $f(x-r)$ and performing public multiplication on $(x + r)^i$ with the corresponding coefficients. First, at the offline stage, the polynomial's coefficients are shifted by a random value, transforming $\sum a_i \cdot x^i$ into $\sum a_i \cdot (x - r)^i$ using secure multiplication. Second, at the online stage, we determine which spline the masked input falls into. Notably, we avoid direct DCF-based interval containment, as large numbers of splines lead to excessive overhead for range

Algorithm 11 Private Lookup Table Protocol, $\Pi_{\mathsf{priLUT}}^{\ell_{\mathsf{in}},\ell_{\mathsf{out}},T_{\mathsf{pri}}}$

Input: P_b holds $\langle x\rangle_b^{\ell_{\mathsf{in}}}$ and a secret-shared lookup table $\langle T\rangle_b^{\ell_{\mathsf{out}}}$ with $2^{\ell_{\mathsf{in}}}$ entries.
Output: P_b obtains $\langle y\rangle_b^{\ell_{\mathsf{out}}}$ where $y = T[x]$.
Offline Phase:
1: P_b samples random number $\langle r\rangle_b^{\ell_{\mathsf{in}}}$
2: P_b invokes Π_{BitDec} to get $\langle i + r\rangle_b^B$ for $i \in [2^{\ell_{\mathsf{in}}}]$
3: P_b invokes $\mathsf{Gen}_{\mathsf{DPF}}^{\ell_{\mathsf{in}},\ell_{\mathsf{out}}}(b, \langle i + r\rangle_b^B, \langle T_{\mathsf{pri}}[i]\rangle_b^{\ell_{\mathsf{out}}})$ to obtain k_b^i for $i \in [2^{\ell_{\mathsf{in}}}]$ with random mask r
Online Phase:
1: P_b sends $\langle x\rangle_b^{\ell_{\mathsf{in}}} + \langle r\rangle_b^{\ell_{\mathsf{in}}}$ to P_{1-b} to reconstruct $x + r$.
2: P_b computes $\langle \mathbf{z}[i]\rangle_b^{\ell_{\mathsf{out}}} := \mathsf{Eval}_{\mathsf{DPF}}^{\ell_{\mathsf{in}},\ell_{\mathsf{out}}}(b, k_b^i, x + r)$ for $i \in [2^{\ell_{\mathsf{in}}}]$.
3: P_b outputs $\langle y\rangle_b^{\ell_{\mathsf{out}}} := \sum_{i\in[2^{\ell_{\mathsf{in}}}]} \langle \mathbf{z}[i]\rangle_b^{\ell_{\mathsf{out}}}$.

Algorithm 12 Secure Spline Polynomial Approximation Protocol, $\Pi_{\mathsf{Approx}}^{\ell,d,u,s,f_{i,i\in[u]}}$

Input: P_b holds $\langle x\rangle_b^{\ell}$, public d-degree spline polynomials $\{f_i^{\ell}\}_{i\in[u]}$ with ℓ-bit coefficients, truncation bit length s.
Output: P_b obtains $\langle y\rangle_b^{\ell}$, where $y = f_i^{\ell}(x)$ and x belongs to the i-th piece.
Offline Stage:
1: P_b invokes the offline sub-protocol of $\Pi_{\mathsf{TR}}^{\ell,s}$ to get k_b^{TR}.
2: P_b invokes $\mathcal{F}_{\mathsf{MUL}}$ on $f_{i,j,i\in[u],j\in[d]}^{\ell}$ to get shares of the updated polynomial approximation coefficients with offset r, i.e. $\langle \hat{f}_{i,j,i\in[u],j\in[d]}\rangle_b^{\ell}$
3: P_b invokes $\mathsf{Gen}_{\mathsf{priLUT}}^{\ell-s,\ell,\langle \hat{f}_{i,,j,i\in[u],j\in[d]}\rangle_b^{\ell}}$ to get private LUT protocol keys k_b^{priLUT} for all table entries and $\langle r\rangle_b^{\ell}$, where $\langle r\rangle_b^{\ell}$ is the share of mask of LUT key
4: P_b parses $\langle r\rangle_b^{\ell} \parallel k_b^{\mathsf{TR}} \parallel k_b^{\mathsf{priLUT}} \parallel \langle \hat{f}_{i,j,i\in[u],j\in[d]}\rangle_b^{\ell}$ as k_b
Online Stage:
1: P_b parses k_b as $\langle r\rangle_b^{\ell} \parallel k_b^{\mathsf{TR}} \parallel k_b^{\mathsf{priLUT}} \parallel \langle \hat{f}_{i,j,i\in[u],j\in[d]}\rangle_b^{\ell}$
2: P_b sends $\langle x\rangle_b^{\ell} + \langle r\rangle_b^{\ell}$ to P_{1-b} to reconstruct $x + r$.
3: P_b evaluates $\langle x'\rangle_b^{\ell-s} := \mathsf{Eval}_{\mathsf{TR}}^{\ell,s}(b, k_b^{\mathsf{TR}}, x + r)$ to get truncated input
4: P_b evaluates $\langle \hat{f}_{j,j\in[d]}\rangle_b^{\ell} := \mathsf{Eval}_{\mathsf{priLUT}}^{\ell-s,\ell,}(b, k_b^{\mathsf{priLUT}}, \langle x'\rangle_b^{\ell-s})$ to get corrected polynomial coefficients
5: P_b locally computes $\langle y\rangle_b^{\ell} := \sum_{j=0}^{d} \langle \hat{f}_j\rangle_b^{\ell} \cdot (x + r)^d$

checks. Instead, for 2^u segments, calculating the higher u bits suffices for range determination. Given $x^{\ell} = x^u \| x^{\ell-u}$, we have $x^{\ell} \in I_{x^u} = [x^u \cdot 2^{\ell-u}, (x^u + 1) \cdot 2^{\ell-u})$. This allows for interval containment via an $(\ell - u)$-bit DPF on a truncated input, reducing the need for multiple ℓ-bit comparisons. With secret-shared coefficients, our optimized spline approximation protocol employs the private LUT protocol to retrieve the correct coefficients using lower bit lengths. The protocol $\Pi_{\mathsf{Approx}}^{\ell,d,u,s,f_{i,i\in[u]}}$ is detailed in Algorithm 12.

B. Trigonometric functions

Trigonometric functions, such as sine, cosine, and tangent, are essential in scientific computation, particularly for spatial problems [7]. We propose an efficient protocol that leverages the periodic properties of these functions along with our FSS-

Algorithm 13 Secure Sine Protocol, $\Pi_{\mathsf{Sin}}^{\ell,s,c,d,u,k}$

Input: P_b holds $\langle x\rangle_b^A$, with bit length ℓ, scale s, LUT segmentation number c, polynomial approximation degree d with spline u, truncation bits k,

Output: P_b obtains $\langle y\rangle_b^A$ where $y = \sin(\pi x)$

Offline Stage:

1: P_b invokes $\mathsf{Gen}_{\mathsf{ModPow2}}^{2+s,2}$ to obtain non-negative comparison key k_b^{ModPow2}
2: P_b invokes $\mathsf{Gen}_{\mathsf{Ctn}}^{s+1,5}$ with knots on $\{0, 0.5, 1, 1.5, 2\}$ to get the interval containment key k_b^{Ctn}
3: P_b invokes offline stage of $\mathcal{F}_{\mathsf{MUL}}$ to get multiplication triplets k_b^{MUL} of both ℓ and $s+2$ bit length
4: **if** *Lookup table implementation* **then**
5: P_b invokes $\mathsf{Gen}_{\mathsf{DigDec}}^{s,c,s/c}$ to get digit decomposition key k_b^{DigDec}
6: P_b invokes $\mathsf{Gen}_{\mathsf{pubLUT}}^{s/c,\ell,T_f}$ for c times to get public LUT keys $k_b^{\mathsf{pubLUT}}[i]$ for $i \in [c]$
7: **else if** *Spline polynomial approximation implementation* **then**
8: P_b invokes $\mathsf{Gen}_{\mathsf{Approx}}^{s,d,u,k,\sin(\pi x)_{i,i\in[u]}}$ to get spline approximation key k_b^{Approx}
9: **end if**
10: P_b outputs the final key $k_b^{\mathsf{Sin}} := k_b^{\mathsf{ModPow2}} \parallel k_b^{\mathsf{Ctn}} \parallel k_b^{\mathsf{MUL}} \parallel k_b^{\mathsf{DigDec}} \parallel k_b^{\mathsf{pubLUT}}[i] \parallel k_b^{\mathsf{Approx}}$

Online Stage:

1: P_b locally drops $\ell-(s+2)$ high order bits of $\langle x\rangle_b^{\ell_{\mathsf{in}}}$ to get $\langle x'\rangle_b^{s+2}$
2: P_b evaluates $\mathsf{Eval}_{\mathsf{ModPow2}}^{s+2,2}(b, k_b^{\mathsf{Mod}}, \langle x'\rangle_b^{s+2})$ get $\langle x_{\mathsf{Mod}}\rangle_b^{s+2}$
3: P_b evaluates $\langle \mathbf{v}\rangle_b^{s+2} := \mathsf{Eval}_{\mathsf{Ctn}}^{s+2,5}(b, k_b^{\mathsf{Ctn}}, \langle x_{\mathsf{Mod}}\rangle_b^{s+2})$
4: P_b locally compute $\langle m_i\rangle_b^{s+2} := \langle \mathbf{v}\rangle_b \cdot D_i$ for $i \in [3]$
5: P_b evaluates $\mathcal{F}_{\mathsf{MUL}}(\langle m_0\rangle_b^{s+2}, \langle x_{\mathsf{Mod}}\rangle_b^{s+2}) + \langle m_1\rangle_b^{s+2}$ to get $\langle x_{\mathsf{Transform}}\rangle_b^{s+2}$
6: P_b locally locally drops 2 high order bits of $\langle x_{\mathsf{Transform}}\rangle_b^{s+2}$ to get $\langle x_{\mathsf{Frac}}\rangle_b^{s}$
7: **if** *Lookup table implementation* **then**
8: P_b evaluates $\mathsf{Eval}_{\mathsf{DigDec}}^{s,c,s/c}(b, k_b^{\mathsf{DigDec}}, \langle x_{\mathsf{Frac}}\rangle_b^{s})$ to get the digit decomposition results $\langle x_{\mathsf{Seg}}[i]\rangle_b^{s/c}$
9: P_b evaluates $\mathsf{Eval}_{\mathsf{pubLUT}}^{s/c,\ell,T_f^i}(b, k_b^{\mathsf{pubLUT}}[i], \langle x_{\mathsf{Seg}}[i]\rangle_b^{s/c})$ to get $\langle y_i\rangle_b^{\ell}$ for $i \in [c]$, where $T_f^i[j] = f(j \times 2^{(s-1)/c-s})$ for $f \in \{\sin(\pi x), \cos(\pi x)\}, i \in [c], j \in [2^{s/c}]$
10: P_b evaluates $\langle y'\rangle_b^{\ell} := \mathcal{F}_{\mathsf{MUL}}^{\ell}(\langle y_i\rangle_b^{\ell}, \langle y_j\rangle_b^{\ell})$ for $i, j \in [c]$ according to sum-to-product identity of trigonometric functions.
11: **else if** *Polynomial approximation implementation* **then**
12: P_b evaluates $\mathsf{Eval}_{\mathsf{Approx}}^{s,d,u,k,\sin(\pi x)_{i,i\in[u]}}(b, k_b^{\mathsf{Approx}}, \langle x_{\mathsf{Frac}}\rangle_b)$ to get $\langle y'\rangle_b^{\ell}$
13: **end if**
14: P_b evaluates $\langle y\rangle_b^{\ell} := \mathcal{F}_{\mathsf{MUL}}^{\ell}(\langle m_2\rangle_b^{\ell}, \langle y'\rangle_b^{\ell})$

based building blocks. This approach enables effective and accurate implementations for both LUT and approximation methods. Our protocol consists of four major steps, using *sine* as a representative example.

Utilizing the periodic properties. Due to the periodic property, we can calculate the function result in one period, which requires transforming $x \in \mathbb{R}$ into a smaller domain that $x' \in [0, 2)$ ($x' \in [0, 1)$ for *tangent*). This can be done by local bits dropping and a non-negative comparison, which is realized by a $(2+s)$-bit DCF, where s is the scale in fixed-point arithmetic. We clarify the correctness of this local operation in Theorem 1 and provide the proof in Appendix C. The protocol for secure modular is detailed in Appendix B2.

Theorem 1. *Given* $\mathsf{Fix}(k)$ *to be the fixed-point representation of constant k, s to be the scale,* $N = 2^{\ell} > \mathsf{Fix}(4)$. *The transformation from* $x \in \mathbb{R}$ *to* $x' \in [0, 2)$ *under* ℓ*-bit fixed-point arithmetic can be correctly performed by locally dropping high* $\ell - 2 - s$ *bit of x following an extra secure modular with* $\mathsf{Fix}(2)$.

Calculating on one-forth-period. An intriguing property of trigonometric functions is that they retain the same shape within one-fourth of their period, enabling simple transformations like flipping or inverting. Notably, it suffices to compute the function within one-fourth of the period (i.e., $[0, 0.5]$) using additional transformations. Formally, there exists coefficients a, b and c that satisfies $g_x^{\mathsf{f}}(y') = af[\pi(bx+c)]$ for $y' \in [0, 0.5], x \in [0, 2)$. This further reduces the bit length for LUT or spline polynomial approximation implementation.

Performing FSS-based trigonometric evaluation. Upon performing the above optimizations, it comes to evaluating the actual trigonometric function. We implement both LUT and approximation methods. For LUT implementation, we observe that we can decompose the s bit value into several segments (2 in our work) so that we have lower entry numbers per LUT because of the *sum-to-product* identity for *sine* and *cosine*. Correspondingly, we decompose an ℓ-bit input x^{ℓ} into $x_h^{\ell/2} \times 2^{\ell/2} + x_l^{\ell/2}$ and construct 2 LUTs with $2^{\ell/2}$ entries plus two extra multiplications. Furthermore, we can reuse the result of the one-hot vector generated during the LUT protocol so that we only have to evaluate $\ell/2$-bit DPF twice instead of four times directly. For spline approximation implementation, it suffices to invoke the optimized spline polynomial protocol in Algorithm 12 with the correct approximation coefficient for completion of the trigonometric evaluation.

We provide the complete implementation of the *sine* function in Algorithm 13 and *tangent* function in Appendix F. *Cosine* function follows the identical implementation with

TABLE II: Runtime and communication overhead of our FSS protocols with different bit lengths ℓ in (8, 16, 18).

Protocol	Params.	Time LAN		Time WAN		Comm.	
	ℓ	Gen(s)	Eval(μs)	Gen(s)	Eval(μs)	Gen(MB)	Eval(B)
Π_{DPF}	8	0.050	0.577	1.288	0.615	0.046	0
	16	0.092	1.150	2.450	1.151	0.080	0
	18	0.124	1.278	2.780	1.265	0.088	0
Π_{DCF}	8	0.085	1.818	2.417	1.866	0.145	0
	16	0.180	3.357	4.625	3.505	0.277	0
	18	0.280	3.739	5.247	3.718	0.310	0

different approximation coefficients and lookup table compared to *sine*. Theoretical analysis for online computation and communication complexity of our building blocks protocol and trigonometric evaluation framework is presented in Appendix D, Table VI.

VII. Evaluation

A. Experimental setup

We implemented our framework[1] built on top of EMP-toolkit [29] and EzPC [12] in C++. Similar to prior works [30], [1], we simulate the network connection under two different settings, where LAN with 10Gbps bandwidth and 0.05ms echo latency, and WAN with 40 Mbps bandwidth and 20ms echo latency. All experiments are performed using a single thread on AWS c5.9xlarge instances with Intel Xeon 8000 series CPUs at 3.6GHz.

Implementation details. Similar to prior works [4], [3], we set the computational security parameter to $\lambda = 128$ in our implementation. The default integer representation for fixed-point arithmetic is in $\mathbb{Z}_{2^{16}}$ with the scale of 9. When constructing a lookup table, we decompose it into 2 sub-tables. We use 2-degree polynomials with the Remez algorithm [31] to compute the correct coefficients for spline approximation.

Baselines. Our first baseline is MP-SPDZ [1], one of the state-of-the-art MPC frameworks aligning with our offline-online paradigm, providing implementations for various building blocks and trigonometric functions. MP-SPDZ is also compared in prior works [4], [32], [8]. The second baseline is EzPC-Secfloat [32], integrated into EzPC [11]. It is one of the state-of-the-art floating-point-based MPC libraries with efficient and practical approximation-based trigonometric function implementations.

B. Performance evaluation

We evaluate the performance of our framework from the three hierarchical levels as shown in Figure 1.

Performance of 2PC FSS key generation. In Table II, we show the performance of our 2PC DPF and DCF key generation schemes. We observe that our schemes are concretely efficient. Specifically, the DPF/DCF key can be generated in seconds, while the online phase for evaluating DPF and DCF requires only a few microseconds with no interaction and communication between parties. The high efficiency of these two protocols becomes the basis for our crucial building block protocols in Section V and efficient trigonometric evaluation framework in Section VI.

TABLE III: Runtime and communication overhead of our FSS-based building block protocols. *Aux* means auxiliary parameters for the tested protocol.

Protocol	Params.		Time LAN		Time WAN		Comm.	
	ℓ	Aux.	Gen(s)	Eval(ms)	Gen(s)	Eval(ms)	Gen(MB)	Eval(B)
Π_{EQ}	8	–	0.050	0.006	1.277	10.335	0.046	1
	16	–	0.092	0.006	2.464	10.337	0.080	2
	18	–	0.125	0.006	2.783	10.348	0.088	4
Π_{CMP}	8	–	0.162	0.006	4.819	10.345	0.281	1
	16	–	0.354	0.007	9.329	10.370	0.545	2
	18	–	0.539	0.007	10.605	10.355	0.611	4
Π_{TR}	8	$s = 5$	0.108	0.010	3.127	10.199	0.182	1
	16	$s = 9$	0.181	0.012	5.405	10.207	0.314	1
	18	$s = 9$	0.199	0.013	5.947	10.203	0.347	2
Π_{Ctn}	8	$k = 4$	0.608	0.190	18.891	31.014	1.099	12
	16	$k = 4$	1.365	0.207	37.234	31.023	2.156	24
	18	$k = 4$	2.157	0.210	42.296	30.982	2.421	48
Π_{DigDec}	8	$k = 2$	0.130	0.154	3.821	40.898	0.203	4
	16	$k = 2$	0.220	0.161	6.683	41.040	0.352	5
	18	$k = 2$	0.241	0.164	7.311	40.968	0.389	8
Π_{PubLUT}	8	$T = 2^{\ell}$	0.050	0.041	1.282	10.270	0.046	1
	16	$T = 2^{\ell}$	0.094	15.028	2.463	25.313	0.080	2
	18	$T = 2^{\ell}$	0.126	67.023	2.778	77.447	0.088	4
Π_{PriLUT}	8	$T = 2^{\ell}$	9.752	0.165	326.247	10.469	9.596	1
Π_{Approx}	8	$d = 2, u = 16$	1.071	0.058	36.244	50.889	1.180	5
	16	$d = 2, u = 16$	1.204	0.061	40.766	51.049	1.445	7
	18	$d = 2, u = 16$	1.391	0.060	41.854	51.031	1.151	9

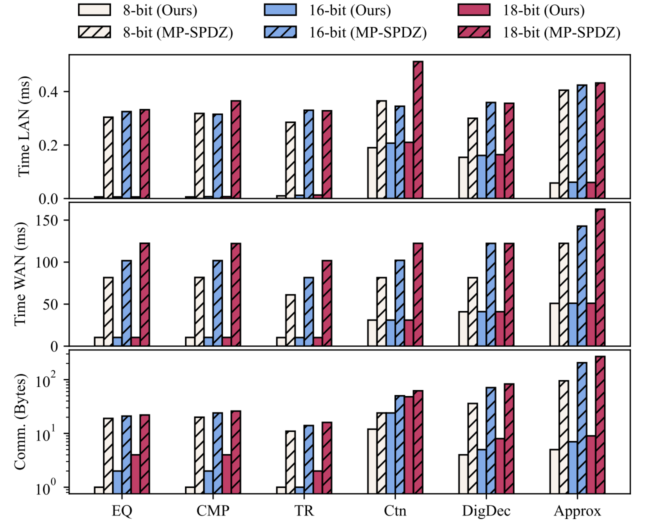

Fig. 6: Online performance comparison for building blocks with MP-SPDZ

Performance of building blocks. In Table III, we report the performance of our crucial building blocks based on our 2PC FSS scheme. We observe that the equality test protocol Π_{EQ} and comparison protocol Π_{CMP} are lightweight and highly efficient during evaluation because they directly invoke Π_{DPF} and Π_{DCF}. The remaining protocols built upon the equality test and comparison incur bytes of communication. In Figure 6, we present an online performance comparison of our building blocks with MP-SPDZ. Our results demonstrate significant performance improvements, with runtime reductions ranging from $1.67\times$ to $52.80\times$ in LAN environments and from $1.99\times$ to $11.83\times$ in WAN environments. Additionally, we observe up to a $29.89\times$ reduction in communication overhead. The

[1] Available at https://github.com/xingpz2008/dealerless-FSS_public

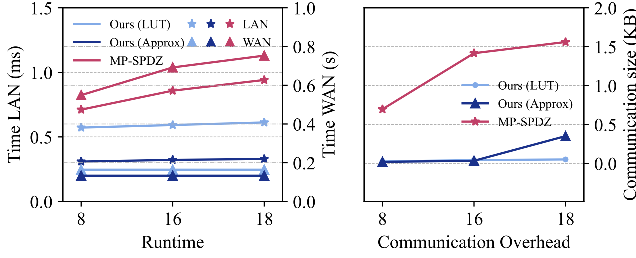

Fig. 7: Online performance of Sine function

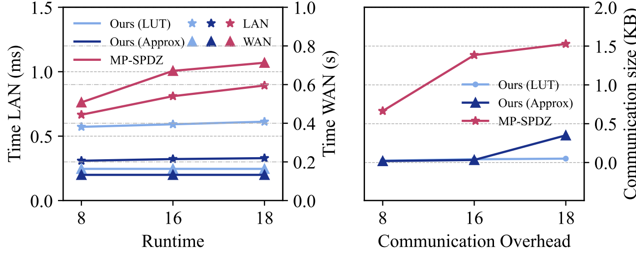

Fig. 8: Online performance of Cosine function

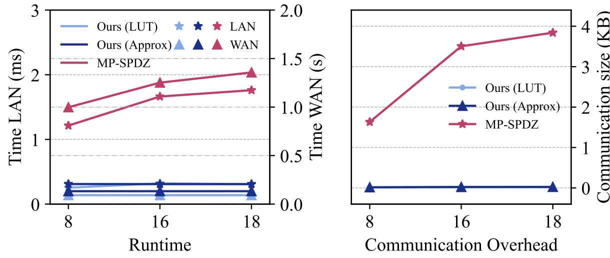

Fig. 9: Online performance of Tangent function

cost of our public lookup table protocol Π_{PubLUT} and private lookup table Π_{PriLUT} grows exponentially with input bit length or the table entries. However, during math function evaluation, we invoke these protocols in sufficiently low bit length (2 to 4 in our implementation) to alleviate the impact of massive table entries, thus lowering the overall overhead. Complete results for lookup table protocols with MP-SPDZ are reported in Appendix E.

Performance of trigonometric functions. We provide the performance of our efficient trigonometric function framework in Table IV. We compare the online performance of our framework with MP-SPDZ [1] and present the result in Figures 7, 8 and 9 for three respective functions[2]. It is observed that our framework is $1.16 \sim 5.74\times$ faster under LAN settings and $3.11 \sim 14.73\times$ faster under WAN settings. Moreover, our communication size is $27.67 \sim 184.42\times$ less than MP-SPDZ. When it comes to function accuracy[3], we use the criterion of Unit in Last Place (ULP) error, similar to [32], [8]. It represents the amount of representable numbers between protocol output and exact output. As seen from the last column, errors in all protocols are minor, resulting in negligible accuracy loss. Concretely, $\sin$ and $\cos$ all achieved around one ULP error, which equates to a one-bit LSB error. Unlike these two functions, the LUT-based $\tan$ does not employ accuracy-loss fixed-point multiplication, thus achieving accurate results without ULP error. We provide additional experimental results compared with EzPC-Secfloat [32] and report them in the Appendix E.

[2]Our LUT and approximation results in Figures 7 and 8 are identical due to the same underlying protocol implementation.

[3]We restrict the input to the range that the approximation coefficients can be correctly represented under the given fixed-point arithmetic settings.

C. Case studies

We apply our FSS-based protocols to two real-world scenarios: privacy-preserving biometric authentication and privacy-preserving proximity test. Table V shows unprecedented online efficiency for real-world applications. We present the complete experiment result comparing with MP-SPDZ [1] and EzPC-Secfloat [32] in Appendix E.

Privacy-preserving biometric authentication. A biometric system verifies user permission for operations by analyzing sensitive data such as voice, fingerprints, and iris images, generating a confidence score. It incorporates a *score fusion* model that evaluates multiple scores using a tangent approach [28]. We abstract the system and simulate it as four tangent functions, which output a final confidence score from four independent random data pairs. Compared with MP-SPDZ [1], we achieved $3.39 \sim 4.77\times$ and $7.46 \sim 14.49\times$ less latency on LAN and WAN respectively, as illustrated in Figure 10. It is worth noting that the advancement of our framework is greater with a larger bit length in a variety of real-world applications.

Privacy-preserving proximity test. The proximity test solutions [26] are usually connected with distance measurement in various online map services. It collects the user's position and checks if it is close enough to the target point like a gas station or restaurant. A privacy-preserving proximity test framework [32] securely computes the distance using Haversine's formula [33] without revealing the cleartext data: $\Delta = 2R \cdot \tan^{-1}\sqrt{\delta/(1-\delta)}$, where $\delta = \sin\pi^2[(x_A - x_B)/2] + \cos\pi x_A \cdot \cos\pi x_B \cdot \sin\pi^2[(y_A - y_B)/2]$. We test the performance of computing the vital component δ in privacy-preserving systems. We present the performance for our framework and MP-SPDZ-based implementation in Figure 11. The evaluation result reveals that the runtime of our polynomial approximation implementation is $1.15 \sim 1.42\times$ faster than MP-SPDZ in LAN settings, and $2.18 \sim 2.96\times$ better in WAN settings, with a reduction of $38.55 \sim 58.90\times$ in communication.

VIII. CONCLUSION

This paper addresses the gap in function secret sharing by developing dealer-less key generation protocols for DPF and DCF, supporting arithmetic-shared inputs and outputs. We then integrate these into the trigonometric evaluation framework that leverages periodic properties to compute complex non-linear functions efficiently with low bit-length, ensuring exceptional online performance. Our experiments demonstrate the practical applicability of this approach.

A key limitation of our FSS protocols is the linear scaling of offline communication rounds with input bit length. Although Guo et al. [10] reduce PRG invocations with a

TABLE IV: Performance of trigonometric function evaluation. *Aux* means auxiliary parameters for the tested protocol. *Impl* means the underlying implementation method. Average ULP error is measured from 2^ℓ random data pairs.

Protocol	Impl.	Params.		Time LAN		Time WAN		Comm.		Error
		(ℓ, s)	Aux.	Gen(s)	Eval(ms)	Gen(s)	Eval(s)	Gen(MB)	Eval(KB)	(ULP)
Π_{Sin}	LUT	$(8,5)$	$d=2$	0.643	0.572	20.115	0.164	1.144	0.024	0.687
		$(16,9)$	$d=2$	1.118	0.592	33.816	0.164	1.896	0.042	1.477
		$(18,9)$	$d=2$	1.313	0.612	35.007	0.164	1.962	0.050	1.477
	Approx	$(8,5)$	$d=2, u=16$	1.540	0.309	51.086	0.133	2.039	0.019	0.750
		$(16,9)$	$d=2, u=16$	2.018	0.322	64.712	0.133	3.479	0.033	1.387
		$(18,9)$	$d=2, u=16$	2.178	0.329	65.985	0.133	3.562	0.035	1.387
Π_{Cos}	LUT	$(8,5)$	$d=2$	0.643	0.572	20.115	0.164	1.144	0.024	0.188
		$(16,9)$	$d=2$	1.118	0.592	33.816	0.164	1.896	0.042	0.360
		$(18,9)$	$d=2$	1.313	0.612	35.007	0.164	1.962	0.050	0.360
	Approx	$(8,5)$	$d=2, u=16$	1.540	0.309	51.086	0.133	2.039	0.019	0.750
		$(16,9)$	$d=2, u=16$	2.018	0.322	64.712	0.133	3.479	0.033	1.117
		$(18,9)$	$d=2, u=16$	2.178	0.329	65.985	0.133	3.562	0.035	1.117
Π_{Tan}	LUT	$(8,5)$	–	0.295	0.253	9.135	0.092	0.712	0.012	0.000
		$(16,9)$	–	0.576	0.321	16.605	0.092	1.223	0.019	0.000
		$(18,9)$	–	0.841	0.307	17.840	0.092	1.305	0.021	0.000
	Approx	$(8,5)$	$d=2, u=16$	1.266	0.310	42.420	0.133	1.755	0.016	0.750
		$(16,9)$	$d=2, u=16$	1.578	0.309	51.380	0.133	2.442	0.023	0.875
		$(18,9)$	$d=2, u=16$	1.785	0.309	56.629	0.133	2.525	0.025	1.053

TABLE V: Performance of our case studies. *Aux* means auxiliary parameters for the tested protocol. *Impl* means the underlying implementation method. Average ULP error is measured from 2^ℓ random data pairs.

Task	Impl.	Params.		Time LAN		Time WAN		Comm.		Error
		(ℓ, s)	Aux.	Gen(s)	Eval(ms)	Gen(s)	Eval(s)	Gen(MB)	Eval(KB)	(ULP)
Biometric Authentication	LUT	$(8,5)$	–	1.130	1.071	36.232	0.368	2.007	0.048	0.000
		$(16,9)$	–	2.272	1.240	66.343	0.369	3.659	0.076	0.000
		$(18,9)$	–	3.120	1.429	71.343	0.369	3.924	0.084	0.000
	Approx	$(8,5)$	$d=2, u=16$	4.997	1.275	168.904	0.531	6.079	0.064	1.000
		$(16,9)$	$d=2, u=16$	6.281	1.255	204.858	0.530	8.194	0.092	3.438
		$(18,9)$	$d=2, u=16$	7.396	1.252	210.050	0.530	8.458	0.100	2.594
Proximity Test	LUT	$(8,5)$	$d=2$	2.549	2.373	80.167	0.696	4.552	0.110	0.563
		$(16,9)$	$d=2$	4.439	2.417	134.994	0.696	7.558	0.188	2.203
		$(18,9)$	$d=2$	5.277	2.409	140.135	0.697	10.454	0.220	2.094
	Approx	$(8,5)$	$d=2, u=16$	3.809	1.317	204.370	0.572	8.129	0.090	0.875
		$(16,9)$	$d=2, u=16$	8.003	1.394	258.941	0.572	11.300	0.152	1.078
		$(18,9)$	$d=2, u=16$	8.708	1.394	263.986	0.572	14.211	0.160	0.926

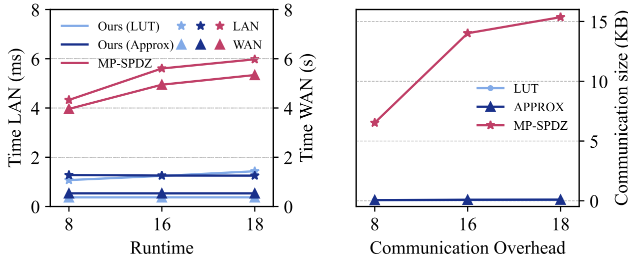

Fig. 10: Online performance of bio-authentication

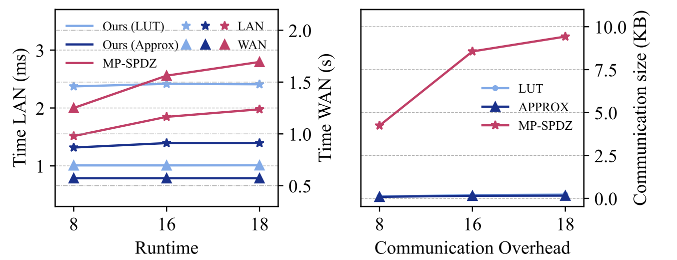

Fig. 11: Online performance of proximity test

GGM-tree, achieving sublinear communication remains an open challenge. Additionally, optimizations [13] like reduced DCF keys apply only in dealer mode. While 2PC protocols could enable these optimizations, designing efficient protocols with unknown randomness is a promising area for future research. This work focuses on semi-honest parties, assuming no protocol deviations. For malicious security, Castro et al. [34] proposed verifiable FSS in the dealer model. Extending our approach to the malicious setting requires verifying input and output consistency and key correctness [35], which could involve authenticated outputs via information-theoretic MACs and efficient ZKPs [36], [37] for our future work.

ACKNOWLEDGMENT

We would like to express our deepest gratitude for the invaluable help provided by our shepherd and for all the reviewers' constructive comments. This work is supported by the National Key R&D Program of China under Grant 2022YFB3103500, the National Natural Science Foundation of China under Grant 62020106013, the Chengdu Science and Technology Program under Grant 2023-XT00-00002-GX, the Fundamental Research Funds for Chinese Central Universities under Grant Y030232063003002.

References

[1] M. Keller, "Mp-spdz: A versatile framework for multi-party computation," in *Proceedings of ACM CCS*, 2020, pp. 1575–1590.

[2] E. Boyle, N. Gilboa, and Y. Ishai, "Function secret sharing," in *Proceedings of EUROCRYPT*, 2015, pp. 337–367.

[3] S. Wagh, "Pika: Secure computation using function secret sharing over rings," in *Proceedings of PETS*, 2022, pp. 351–377.

[4] K. Gupta, D. Kumaraswamy, N. Chandran, and D. Gupta, "Llama: A low latency math library for secure inference," in *Proceedings of PETS*, 2022, pp. 274–294.

[5] T. Ryffel, P. Tholoniat, D. Pointcheval, and F. R. Bach, "Ariann: Low-interaction privacy-preserving deep learning via function secret sharing," in *Proceedings of PETS*, 2020, pp. 291–316.

[6] D. Boneh, E. Boyle, H. Corrigan-Gibbs, N. Gilboa, and Y. Ishai, "Lightweight techniques for private heavy hitters," in *Proceedings of IEEE S&P*, 2021, pp. 762–776.

[7] B. Hemenway, S. Lu, R. Ostrovsky, and W. W. IV, "High-precision secure computation of satellite collision probabilities," ePrint 2016/319, 2016.

[8] D. Rathee, M. Rathee, R. K. K. Goli, D. Gupta, R. Sharma, N. Chandran, and A. Rastogi, "Sirnn: A math library for secure rnn inference," in *Proceedings of IEEE S&P*, 2021, pp. 1003–1020.

[9] J. Doerner and A. Shelat, "Scaling oram for secure computation," in *Proceedings of ACM CCS*, 2017, pp. 523–535.

[10] X. Guo, K. Yang, X. Wang, W. Zhang, X. Xie, J. Zhang, and Z. Liu, "Half-tree: Halving the cost of tree expansion in cot and dpf," in *Proceedings of EUROCRYPT*, 2023, pp. 330–362.

[11] D. Rathee, M. Rathee, N. Kumar, N. Chandran, D. Gupta, A. Rastogi, and R. Sharma, "Cryptflow2: Practical 2-party secure inference," in *Proceedings of ACM CCS*, 2020, pp. 325–342.

[12] N. Chandran, D. Gupta, A. Rastogi, R. Sharma, and S. Tripathi, "Ezpc: Programmable and efficient secure two-party computation for machine learning," in *Proceedings of EuroS&P*, 2019, pp. 496–511.

[13] E. Boyle, N. Chandran, N. Gilboa, D. Gupta, Y. Ishai, N. Kumar, and M. Rathee, "Function secret sharing for mixed-mode and fixed-point secure computation," in *Proceedings of EUROCRYPT*, 2021, pp. 871–900.

[14] E. Boyle, N. Gilboa, and Y. Ishai, "Function secret sharing: Improvements and extensions," in *Proceedings of ACM CCS*, 2016, pp. 1292–1303.

[15] ——, "Secure computation with preprocessing via function secret sharing," in *Proceedings of TCC*, 2019, pp. 341–371.

[16] A. Agarwal, S. Peceny, M. Raykova, P. Schoppmann, and K. Seth, "Communication-efficient secure logistic regression," in *Proceedings of EuroS&P*, 2024, pp. 440–467.

[17] D. Kim, Y. Son, D. Kim, A. Kim, S. Hong, and J. Cheon, "Privacy-preserving approximate gwas computation based on homomorphic encryption," *BMC Medical Genomics*, vol. 13, 2020.

[18] G. Dessouky, F. Koushanfar, A.-R. Sadeghi, T. Schneider, S. Zeitouni, and M. Zohner, "Pushing the communication barrier in secure computation using lookup tables," in *Proceedings of NDSS*, 2017.

[19] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious rams," *Journal of the ACM*, vol. 43, no. 3, pp. 431–473, 1996.

[20] R. Canetti, "Security and composition of multiparty cryptographic protocols," *Journal of Cryptology*, vol. 13, no. 1, pp. 143–202, 2000.

[21] A. Shamir, "How to share a secret," *Communications of the ACM*, vol. 22, no. 11, pp. 612–613, 1979.

[22] G. R. Blakley, "Safeguarding cryptographic keys," in *Proceedings of Workshop on MARK*, 1979, pp. 313–318.

[23] O. Goldreich, S. Goldwasser, and S. Micali, "How to construct random functions," *Journal of the ACM*, vol. 33, no. 4, pp. 792–807, 1986.

[24] I. Abraham, C. W. Fletcher, K. Nayak, B. Pinkas, and L. Ren, "Asymptotically tight bounds for composing oram with pir," in *Proceedings of PKC*, 2017, pp. 91–120.

[25] Y. Li, X. Tao, X. Zhang, J. Liu, and J. Xu, "Privacy-preserved federated learning for autonomous driving," *IEEE Transactions on Intelligent Transportation Systems*, vol. 23, no. 7, pp. 8423–8434, 2022.

[26] J. Šeděnka and P. Gasti, "Privacy-preserving distance computation and proximity testing on earth, done right," in *Proceedings of ASIACCS*, 2014, pp. 99–110.

[27] M. Li, Y. Chen, S. Zheng, D. Hu, C. Lal, and M. Conti, "Privacy-preserving navigation supporting similar queries in vehicular networks," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 2, pp. 1133–1148, 2022.

[28] A. Herbadji, N. Guermat, L. Ziet, Z. Akhtar, and D. Dasgupta, "Weighted quasi-arithmetic mean based score level fusion for multi-biometric systems," *IET Biometrics*, vol. 9, pp. 91–99, 2020.

[29] X. Wang, A. J. Malozemoff, and J. Katz, "EMP-toolkit: Efficient MultiParty computation toolkit," https://github.com/emp-toolkit, 2016.

[30] Z. Huang, W. jie Lu, C. Hong, and J. Ding, "Cheetah: Lean and fast secure Two-Party deep neural network inference," in *Proceedings of USENIX Security*, 2022, pp. 809–826.

[31] S. Hocevar, "Polynomial Approximations using the Remez Algorithm," https://github.com/samhocevar/lolremez, 2020.

[32] D. Rathee, A. Bhattacharya, R. Sharma, D. Gupta, N. Chandran, and A. Rastogi, "Secfloat: Accurate floating-point meets secure 2-party computation," in *Proceedings of IEEE S&P*, 2022, pp. 576–579.

[33] R. W. Sinnott, "Virtues of the haversine," 1984.

[34] L. de Castro and A. Polychroniadou, "Lightweight, maliciously secure verifiable function secret sharing," in *Proceedings of EUROCRYPT*, 2022, pp. 150–179.

[35] W. Zhang, X. Guo, K. Yang, R. Zhu, Y. Yu, and X. Wang, "Efficient actively secure dpf and ram-based 2pc with one-bit leakage," in *Proceedings of IEEE S&P*, 2024, pp. 561–577.

[36] C. Weng, K. Yang, Z. Yang, X. Xie, and X. Wang, "Antman: Interactive zero-knowledge proofs with sublinear communication," in *Proceedings of ACM CCS*, 2022, pp. 2901–2914.

[37] K. Yang, P. Sarkar, C. Weng, and X. Wang, "Quicksilver: Efficient and affordable zero-knowledge proofs for circuits and polynomials over any field," in *Proceedings of ACM CCS*, 2021, pp. 2986–3001.

[38] D. Beaver, "Efficient multiparty protocols using circuit randomization," in *Proceedings of CRYPTO*, 1992, pp. 420–432.

Appendix

A. 2PC functionalities

Our protocols adopt the following 2-party functionalities.

Multiplexer (MUX). The ℓ-bit Boolean MUX functionality, $\mathcal{F}^{\ell,B}_{\mathsf{MUX}}$, takes as input $\{\langle x_i \rangle^B, \langle y_i \rangle^B\}_{i\in[1,\ell]} \in \{0,1\}^{2\ell}$ and the choice bit $\langle c \rangle^B \in \{0,1\}$, and outputs $\{\langle z_i \rangle^B\}_{i\in[1,\ell]} \in \{0,1\}^{\ell}$ such that $z_i = x_i$ if $c = 0$ and $z_i = y_i$ otherwise. It can be implemented via 2 parallel COTs. Besides, we also consider the ℓ-bit arithmetic MUX functionality, $\mathcal{F}^{\ell,A}_{\mathsf{MUX}}$, that takes as input $\langle x \rangle^\ell, \langle y \rangle^\ell \in \mathbb{Z}_{2^\ell}$ and the choice bit $\langle c \rangle^B \in \{0,1\}$, and outputs $\langle z \rangle^\ell$ such that $z = x$ if $c = 0$ and $z = y$ otherwise. We utilize the method in SIRNN [8] to instantiate this functionality, which requires $2(\lambda + \ell)$ bits of communication.

Multiplication (MUL). The ℓ-bit multiplication functionality, $\mathcal{F}^{\ell}_{\mathsf{MUL}}$, takes as input $\langle x \rangle^\ell$ and $\langle y \rangle^\ell$, and outputs $\langle z \rangle^\ell$ such that $z = x \cdot y \mod 2^\ell$. We use the Beaver multiplication protocol [38] to instantiate this functionality. Based on pre-generated Beaver triplets, this protocol requires ℓ bits of communication within 1 round.

AND. The AND functionality $\mathcal{F}_{\mathsf{AND}}$ takes as input $\langle x \rangle^B$ and $\langle y \rangle^B$, and outputs $\langle z \rangle^B$ such that $z = a \wedge b$. Given the bit triplets generated using the protocol in Cryptflow2 [11], this functionality requires 2 bits of communication within 1 round. Note that a variant of this functionality takes as input $a \in \{0,1\}$ from P_0 and $b \in \{0,1\}$ from P_1, and outputs

$\langle z \rangle^B \in \{0,1\}$ such that $z = a \wedge b$. This can be easily achieved via invoking $\mathcal{F}_{\mathsf{AND}}$ with another share of a (and b) is 0.

OR. The OR functionality $\mathcal{F}_{\mathsf{OR}}$ takes as input $\langle x \rangle^B$ and $\langle y \rangle^B$, and outputs $\langle z \rangle^B$ such that $z = a \vee b$. Given that $a \vee b = \neg(\neg a \wedge \neg b)$, this functionality can be implemented via invoking the above $\mathcal{F}_{\mathsf{AND}}$.

Millionaire (MILL). The Millionaire functionality $\mathcal{F}^{\ell}_{\mathsf{MILL}}$ takes as input $x^B \in \{0,1\}^{\ell}$ from P_0 and $y^B \in \{0,1\}^{\ell}$ from P_1, and outputs $\langle z \rangle^B$ such that $z = \mathbb{1}\{\mathsf{x} < \mathsf{y}\}$. We utilize the method in Cryptflow2 [11] which requires the communication size of $\lambda\ell + 14\ell$ bits with $\log \ell$ rounds.

Boolean-to-Arithmetic (B2A). The Boolean-to-Arithmetic functionality $\mathcal{F}^{\ell}_{\mathsf{B2A}}$ takes as input boolean shares $\langle x \rangle^B$ and outputs the respective arithmetic shares in the same value $\langle y \rangle^{\ell} \in \mathbb{Z}_{2^{\ell}}$ such that $x = y$. We utilize the implementation from Cryptflow2 [11] with communication of $\lambda + \ell$ bits.

B. Supporting Protocols

1) Bit decomposition in DPF and DCF key generation: A bit decomposition protocol is used in 2PC DPF and DCF key generation to obtain the binary representation of the shared target index. The core of this protocol is to check if an overflow comes from the lower position. More concretely, the carry bit equals 1 only if: 1) The shares of $x[i]$ incur an overflow, or 2) The shares of $x[i] = 1$ and the previous carry bit equals 1 thus incur an overflow. Therefore, we invoke 2 $\mathcal{F}_{\mathsf{AND}}$ and 1 $\mathcal{F}_{\mathsf{OR}}$ for the protocol. We present our implementation of the bit decomposition protocol in Algorithm 14.

Algorithm 14 Bit Decomposition, $\Pi^{\ell}_{\mathsf{BitDec}}$

Input: P_b holds $\langle x \rangle^{\ell}_b$.
Output: P_b obtains $\langle \mathbf{y} \rangle^B_b \in \{0,1\}^{\ell}$ where $\mathbf{y}[i] = x[i]$.

1: P_b sets $q := 0$
2: **for** $i \in [\ell]$ **do**
3: $\quad$ P_b sets $\langle \mathbf{y} \rangle^B_b[i] := \langle x \rangle^{\ell}_b[i] \wedge q$
4: $\quad$ P_b invokes $\mathcal{F}_{\mathsf{AND}}$ with input $\langle x \rangle^{\ell}_b[i]$ to get $\langle t \rangle^B_b$
5: $\quad$ P_b invokes $\mathcal{F}_{\mathsf{AND}}$ with input $\langle x \rangle^{\ell}_b[i]$ and q to get $\langle u \rangle^B_b$
6: $\quad$ P_b invokes $\mathcal{F}_{\mathsf{OR}}$ with input $\langle u \rangle^B_b$ and q to get $\langle v \rangle^B_b$
7: $\quad$ P_b sets $q := \langle v \rangle^B_b$
8: **end for**

2) Secure Modular with Power-of-2: A Power-of-2 modular protocol reflects the input into a period 2^s which is the power-of-2. The protocol first drops the redundant $\ell - s - 1$ bit because this will be canceled in modular operations. Now the input lies in the $[0, 2^{s+1})$ and we consequently perform the comparison with 2^s to finalize the modular. This protocol is highly efficient as it only requires an $s+1$-bit DCF-based comparison. We present our protocol in Algorithm 15.

C. Proof of Theorem 1

Proof: For positive inputs x, the correctness holds obviously.

For negative input $x^- < 0$, let its fixed-point representation be $N - x^+$.

Algorithm 15 Power-of-2 Modular Protocol, $\Pi^{\ell,s}_{\mathsf{ModPow2}}$

Input: P_b holds $\langle x \rangle^{\ell}_b$ and a public power-of-2 modular 2^s.
Output: P_b obtains $\langle y \rangle^{s+1}_b$ where $y = x \bmod 2^s$.
Offline Phase:
1: P_b invokes $\mathsf{Gen}^{s+1,s+1}_{\mathsf{CMP}}(2^s, 1)$ to obtain k_b.
Online Phase:
1: P_b parses input $\langle x \rangle^{\ell}_b$ as an ℓ-bit string $u_b || v_b$, where $u_b \in \{0,1\}^{\ell-s-1}$, $v_b \in \{0,1\}^{s+1}$.
2: P_b evaluates $\langle z \rangle^{s+1}_b := b - \mathsf{Eval}^{s+1,s+1}_{\mathsf{CMP}}(b, k_b, v_b)$.
3: P_b sets $\langle y \rangle^{s+1}_b := v_b - 2^s \cdot \langle z \rangle^{s+1}_b$.

We firstly assume that $x^+ < \mathsf{Fix}(2)$. The correct output of this operation should be $\mathsf{Fix}(2) - x^+$. When performing bit-dropping, the input becomes $N - x^+ \mod \mathsf{Fix}(4) = \mathsf{Fix}(4) - x^+$ as we have $N = k \cdot \mathsf{Fix}(4)$. Followed by the comparison, we have $\mathsf{Fix}(4) - x^+ - \mathsf{Fix}(2) \times \mathbf{1}\{\mathsf{Fix}(4) - x^+ > \mathsf{Fix}(2)\} = \mathsf{Fix}(2) - x^+$

When $\mathsf{Fix}(2) \le x^+ < \mathsf{Fix}(4)$, the correct output becomes $2 \times \mathsf{Fix}(2) - x^+$. Results after comparison becomes $\mathsf{Fix}(4) - x^+ = 2 \times \mathsf{Fix}(2) - x^+$ as $\mathbf{1}\{\mathsf{Fix}(4) - x^+ > \mathsf{Fix}(2)\} = 0$

For $\mathsf{Fix}(4) \le x^+$, we can divide x^+ into two parts: $0 < x^+_{\mathsf{Frac}} < \mathsf{Fix}(4)$ and x^+_{Four} which is times of $\mathsf{Fix}(4)$. Correct output becomes $k \times \mathsf{Fix}(2) - x^+_{\mathsf{Frac}} - x^+_{\mathsf{Four}} = k' \times \mathsf{Fix}(2) - x^+_{\mathsf{Frac}}$ accordingly. It is clear that x^+_{Four} will be canceled during bit-dropping, while the remaining part can be converted to conditions on $\mathsf{Fix}(2) \le x^+ < \mathsf{Fix}(4)$ or $x^+ < \mathsf{Fix}(2)$.

■

D. Theoretical overhead analysis for building block protocols

In Table VI, we report the theoretical overhead of our building blocks in terms of online communication rounds and size.

TABLE VI: Online communication complexity for building blocks with input and output bit length ℓ, security parameter λ, multiplication times q for sum-to-product identity.

Protocols	Impl.	Rounds	Comm.(Bits)
$\mathsf{Eval}^{\ell}_{\mathsf{EQ}}$	–	1	ℓ
$\mathsf{Eval}^{\ell}_{\mathsf{CMP}}$	–	1	ℓ
$\mathsf{Eval}^{\ell,s}_{\mathsf{TR}}$	–	1	$s+1$
$\mathsf{Eval}^{\ell,k}_{\mathsf{Ctn}}$	–	2	$\ell(3k-2)$
$\mathsf{Eval}^{\ell,c,d}_{\mathsf{DigDec}}$	–	$c+1$	$(c-1)(4d+1)$
$\mathsf{Eval}^{\ell}_{\mathsf{PubLUT}}$	–	1	2ℓ
$\mathsf{Eval}^{\ell}_{\mathsf{PriLUT}}$	–	1	ℓ
$\mathsf{Eval}^{\ell,d,u,s,f}_{\mathsf{Approx}}$	–	$3+d$	$s+\ell+(d+1)\log u$
$\mathsf{Eval}^{\ell,s,c,d,u,k}_{\mathsf{Sin}}$	LUT	$2c+6+q$	$2\ell q + 17s + 2\ell$ $+c+(4-4s)/c+34$
$\mathsf{Eval}^{\ell,s,c,d,u,k}_{\mathsf{Sin}}$	Approx	$d+8$	$3\ell+17s+(d+1)\log u+32$
$\mathsf{Eval}^{\ell,s,d,u,k}_{\mathsf{Tan}}$	LUT	6	$12s+2\ell+8$
$\mathsf{Eval}^{\ell,s,d,u,k}_{\mathsf{Tan}}$	Approx	$d+8$	$11s+3\ell+(d+1)\log u+10$

E. Additional Experiment Results

Lookup table protocols. We provide results for lookup table protocols and compare with MP-SPDZ in Table VII.

TABLE VII: Online performance of 8-bit lookup table protocols with 2^8 table entries. *Impl* means the underlying implementation method.

Protocol	Impl.	Runtime LAN (ms)	Runtime WAN (ms)	Comm. (B)	Rounds
Π_{PubLUT}	Ours	0.041	10.270	1	2
	MP-SPDZ	0.384	81.762	672	10
Π_{PriLUT}	Ours	0.165	10.469	1	2
	MP-SPDZ	0.440	81.898	1184	12

TABLE VIII: Online performance of trigonometric evaluation and case studies compared with MP-SPDZ and EzPC-Secfloat. Bolded term stands for the best result.

Protocol	Impl.	Runtime LAN (ms)	Runtime WAN (s)	Comm. (KB)	Error (ULP)
Π_{Sin}	Ours (LUT)	0.612	0.164	0.050	1.477
	Ours (Approx)	**0.329**	**0.133**	**0.035**	1.387
	MP-SPDZ	0.941	0.753	1.560	0.629
	EzPC-Secfloat	1.614	0.380	26.014	**0.318**
Π_{Cos}	Ours (LUT)	0.612	0.164	0.050	0.360
	Ours (Approx)	**0.329**	**0.133**	**0.035**	1.117
	MP-SPDZ	0.892	0.713	1.528	1.070
	EzPC-Secfloat	1.625	0.382	26.080	**0.318**
Π_{Tan}	Ours (LUT)	**0.307**	**0.092**	**0.021**	**0**
	Ours (Approx)	0.309	0.133	0.025	1.053
	MP-SPDZ	1.761	1.357	3.840	5.088
	EzPC-Secfloat	2.185	0.533	36.043	0.244
Biometric Authentication	Ours (LUT)	1.429	**0.369**	**0.084**	**0**
	Ours (Approx)	**1.252**	0.530	0.100	2.549
	MP-SPDZ	5.977	5.339	15.360	3.707
	EzPC-Secfloat	9.369	2.372	165.293	2.370
Proximity Test	Ours (LUT)	2.409	0.697	0.220	2.094
	Ours (Approx)	**1.394**	**0.572**	**0.160**	**0.926**
	MP-SPDZ	1.977	1.694	9.424	2.266
	EzPC-Secfloat	8.210	2.030	173.322	6.299

Benefiting from extremely low overhead, we achieve up to $9.32\times$ improvements in lookup table protocols.

Extended trigonometric and case studies results. We perform extended comparison with EzPC-Secfloat [32], along with additional error statistics from MP-SPDZ [1]. We provide these results in Table VIII. Compared with EzPC-Secfloat, we achieve significant performance improvements, e.g., $5.89\times$ faster runtime and $858\times$ reduced communication for proximity tests. Additionally, experiment results show superior accuracy, e.g., we achieve 3.44 ULP-error for proximity tests, compared to MP-SPDZ's 4.06 and EzPC-Secfloat's 6.30.

F. Secure Tangent Protocol

We provide the complete implementation of our secure tangent protocol Π_{Tan} in Algorithm 16. Note that the period of tangent is 1 instead of 2, we adjust the bit length to be dropped in Step 1 and the knots for interval containment in Step 3. The tangent evaluation is simpler than sine evaluation in Section VI-B during LUT implementation as the *sum-to-product* identity is inapplicable for tangent in our design due to costly division.

Algorithm 16 Secure Tangent Protocol, $\Pi^{\ell,s,d,u,k}_{\mathsf{Tan}}$

Input: P_b holds $\langle x\rangle^A_b$, with bit length ℓ, scale s, polynomial approximation degree d with spline u, truncation bits k,
Output: P_b obtains $\langle y\rangle^A_b$ where $y = \tan(\pi x)$

Offline Stage:

1: P_b invokes $\mathsf{Gen}^{1+s,1}_{\mathsf{ModPow2}}$ to obtain non-negative comparison key k^{ModPow2}_b
2: P_b invokes $\mathsf{Gen}^{s+1,3}_{\mathsf{Ctn}}$ with knots on $\{0, 0.5, 1\}$ to get the interval containment key k^{Ctn}_b
3: P_b invokes offline stage of $\mathcal{F}_{\mathsf{MUL}}$ to get multiplication triplets k^{MUL}_b of both ℓ and $s+1$ bit length
4: **if** *Lookup table implementation* **then**
5: P_b invokes $\mathsf{Gen}^{s,\ell,T_{\mathsf{f}}}_{\mathsf{pubLUT}}$ public LUT key k^{pubLUT}_b
6: **else if** *Spline polynomial approximation implementation* **then**
7: P_b invokes $\mathsf{Gen}^{s,d,u,k,\tan(\pi x)_{i,i\in[u]}}_{\mathsf{Approx}}$ to get spline approximation key k^{Approx}_b
8: **end if**
9: P_b outputs the final key k^{Tan}_b as $k^{\mathsf{ModPow2}}_b \parallel k^{\mathsf{Ctn}}_b \parallel k^{\mathsf{MUL}}_b \parallel k^{\mathsf{pubLUT}}_b \parallel k^{\mathsf{Approx}}_b$

Online Stage:

1: P_b locally drops $\ell-(s+1)$ high order bits of $\langle x\rangle^{\ell_{\mathsf{in}}}_b$ to get $\langle x'\rangle^{s+1}_b$
2: P_b evaluates $\mathsf{Eval}^{s+1,1}_{\mathsf{ModPow2}}(b, k^{\mathsf{Mod}}_b, \langle x'\rangle^{s+1}_b)$ get $\langle x_{\mathsf{Mod}}\rangle^{s+1}_b$
3: P_b evaluates $\langle \mathbf{v}\rangle^{s+1}_b := \mathsf{Eval}^{s+1,3}_{\mathsf{Ctn}}(b, k^{\mathsf{Ctn}}_b, \langle x_{\mathsf{Mod}}\rangle^{s+1}_b)$
4: P_b locally compute $\langle m_i\rangle^{s+1}_b := \langle \mathbf{v}\rangle_b \cdot D_i$ for $i \in [3]$
5: P_b evaluates $\mathcal{F}_{\mathsf{MUL}}(\langle m_0\rangle^{s+1}_b, \langle x_{\mathsf{Mod}}\rangle^{s+1}_b) + \langle m_1\rangle^{s+1}_b$ to get $\langle x_{\mathsf{Transform}}\rangle^{s+1}_b$
6: P_b locally locally drops 1 high order bits of $\langle x_{\mathsf{Transform}}\rangle^{s+1}_b$ to get $\langle x_{\mathsf{Frac}}\rangle^{s}_b$
7: **if** *Lookup table implementation* **then**
8: P_b evaluates $\mathsf{Eval}^{s,\ell,T^i_{\mathsf{f}}}_{\mathsf{pubLUT}}(b, k^{\mathsf{pubLUT}}_b[i], \langle x_{\mathsf{frac}}\rangle^s_b)$ to get $\langle y'\rangle^{\ell}_b$
9: **else if** *Polynomial approximation implementation* **then**
10: P_b evaluates $\mathsf{Eval}^{s,d,u,k,\tan(\pi x)_{i,i\in[u]}}_{\mathsf{Approx}}(b, k^{\mathsf{Approx}}_b, \langle x_{\mathsf{Frac}}\rangle_b)$ to get $\langle y'\rangle^{\ell}_b$
11: **end if**
12: P_b evaluates $\langle y\rangle^{\ell}_b := \mathcal{F}^{\ell}_{\mathsf{MUL}}(\langle m_2\rangle^{\ell}_b, \langle y'\rangle^{\ell}_b)$