Streaming Function Secret Sharing and Its Applications

Xiangfu Song[1], Jianli Bai[2], Ye Dong[3,✉], Yijian Liu[4,5], Yu Zhang[4,5], Xianhui Lu[4,5], Tianwei Zhang[1]
[1]*Nanyang Technological University* [2]*Singapore Management University* [3]*National University of Singapore*
[4]*Institute of Information Engineering, Chinese Academy of Sciences*
[5]*School of Cyber Security, University of Chinese Academy of Sciences*
Emails: {xiangfu.song,tianwei.zhang}@ntu.edu.sg, baijianli0812@gmail.com, dongye@nus.edu.sg,
{liuyijian,zhangyu1999,luxianhui}@iie.ac.cn, ✉ corresponding author

Abstract

Collecting statistics from users of software and online services is crucial to improve service quality, yet obtaining such insights while preserving individual privacy remains a challenge. Recent advances in function secret sharing (FSS) make it possible for scalable privacy-preserving measurement (PPM), which leads to ongoing standardization at the IETF. However, FSS-based solutions still face several challenges for streaming analytics, where messages are continuously sent, and secure computation tasks are repeatedly performed over incoming messages. We introduce a new cryptographic primitive called streaming function secret sharing (SFSS), a new variant of FSS that is particularly suitable for secure computation over streaming messages. We formalize SFSS and propose concrete constructions, including SFSS for point functions, predicate functions, and feasibility results for generic functions. SFSS powers several promising applications in a simple and modular fashion, including conditional transciphering, policy-hiding aggregation, and attribute-hiding aggregation. In particular, our SFSS formalization and constructions identify security flaws and efficiency bottlenecks in existing solutions, and SFSS-powered solutions achieve the expected security goal with asymptotically and concretely better efficiency and/or enhanced functionality.

1 Introduction

A (two-party) function secret sharing (FSS) [18, 19] secretly shares a function via a pair of keys, and any single key reveals no more information about the shared function. Concrete FSS constructions include distributed point function (DPF) [18, 19, 39], distributed comparison function (DCF) [18, 19, 20], and distributed multi-point function (DMPF) [17]. FSS has found applications in many secure computation protocols and applications [12, 13, 14, 20, 25, 29, 30, 50]. In particular, FSS is one of the main tools to build privacy-preserving measurement (PPM) systems [10, 29] in the distributed-trust model, which is deployed by Mozilla [43], Divvi Up [1], and Apple and Google [4], and is under standardization at IETF [38].

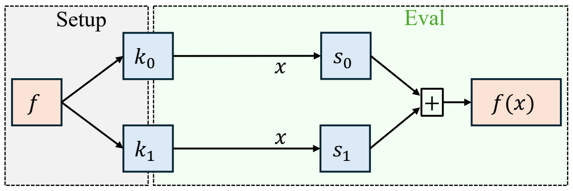
When applying FSS to applications, commonly, data is first encoded as a function of a certain family. Then FSS for that family is used to encode the function via a pair of FSS keys, which are distributed between two servers for the subsequent secure computation. Many applications, such as IoT services, healthcare monitoring, and federated learning, are naturally designed in the *streaming* setting: messages are continuously sent to the servers and secure computation is repeatedly performed over incoming messages. In this setting, simply encoding each new incoming message requires a new pair of FSS keys. Since the size of FSS keys is concretely high, when most portions of the streaming messages remain static, naïvely encoding data via new FSS keys incurs redundant overhead. We provide a convincing example in §2.1. This motivates a new primitive that is better suitable for secure computation over streaming data.

1.1 Formal Definition & Constructions

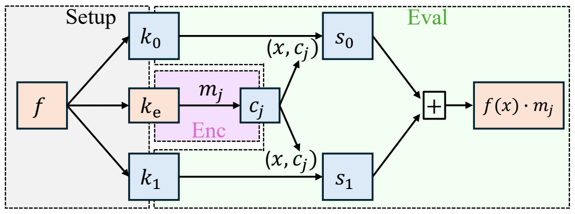
We initiate a new primitive called streaming function secret sharing (SFSS), capturing core efficiency and security properties for *reusable* secure computation over streaming messages.

SFSS definition. Fig. 1 sketches the difference between FSS and SFSS in a nutshell. A two-party FSS scheme consists of a Setup algorithm and an Eval algorithm. The Setup algorithm takes as input a function f, and outputs a pair of FSS keys (k_0, k_1). Each FSS key k_b ($b \in \{0,1\}$), when evaluated over an input x by Eval, outputs a share s_b such that $s_0 + s_1 = f(x)$. Namely, the local evaluation essentially shares $f(x)$.

SFSS captures how to decouple the (shared) function from a data stream to support key reuse, as well as efficiency properties such as *non-interactive* streaming encryption and *non-interactive* evaluation. It differs from FSS mainly in the following aspects: (1) The SFSS Setup algorithm additionally outputs a secret encryption key k_{e} besides the SFSS key pair. (2) SFSS introduces a streaming encryption algorithm Enc. Enc takes as input k_{e} and message m_j, outputs a streaming ciphertext c_j; here j is a counter to identify the j-th streaming message and ciphertext. (3) The SFSS key k_b is evaluated over



(a) The sketched FSS workflow



(b) The sketched SFSS workflow

Figure 1: Comparison of FSS and SFSS in a nutshell.

x and c_j, and outputs a share s_b such that $s_0 + s_1 = f(x) \cdot m_j$.

We stress that the SFSS Enc algorithm can be invoked possibly many times, and security requires that any single SFSS key k_b and all streaming ciphertexts are pseudorandom, meaning that $(\mathsf{k}_b, \{c_j\}_j)$ reveals no more information about the function f or messages $\{m_j\}_j$.

SFSS constructions. We develop SFSS constructions for a broad class of functions. Overall, we "compile" FSS to SFSS.

1) SFSS for single-point functions. We begin with single-point functions $f_{\alpha,\beta}(x)$ that output β for $x = \alpha$ and 0 otherwise. Our construction, called *streaming distributed point function (SDPF)*, efficiently compiles any DPF scheme to its streaming version. The transformation only additionally requires a pseudorandom function (PRF). We design an efficient compiler by 1) incorporating *pseudorandom shares* from DPF outputs as PRF keys, and 2) designing a new streaming masking and unmasking mechanism that introduces negligible overhead. We start with $\beta = 1$ and extend to the case $\beta \geq 1$.

2) SFSS for predicate functions. We extend SFSS beyond single-point functions to general predicate functions. A direct extension encounters a fundamental barrier: the previous technique fails to work if the function has multiple non-zero outputs. To overcome this, we design a new compiler that relies on a *key-homomorphic pseudorandom function* (KH-PRF) [11]. We require a KH-PRF with a *special zero-key* property, which is not generally satisfied by all KH-PRFs but can be instantiated from the KH-PRF based on the *Learning-With-Rounding* (LWR) assumption [5].

3) SFSS for generic functions. We further study how to support generic functions for SFSS. We introduce a *decomposition* technique that converts any function into its predicate form, and then apply SFSS for predicate functions again. This provides a guiding solution for generic functions.

1.2 Applications

SFSS provides a modular approach for privacy-preserving applications over streaming data. Our formalization also helps us identify security/efficiency flaws in existing solutions.

Policy-hiding aggregation. We can use SFSS for designing *policy-hiding aggregation* (`phAgg`) [25]. In `phAgg`, each client attaches a secret policy to its data stream. The policy defines under which conditions the stream can be used for aggregation. Beyond hiding the data stream, `phAgg` additionally protects: 1) the policy itself, and 2) whether a data stream is involved in an aggregation task (*i.e.*, data stream access patterns).

SFSS provides a modular design for `phAgg`, by simply configuring the policy function in its setup phase. Our study reveals that Vizard [25], based on equality testing using DPF, suffers from security and efficiency flaws. In contrast, our SDPF-based solution securely realizes the functionality that Vizard failed to achieve, meanwhile offering strictly improved efficiency (even compared with the flawed Vizard). Moreover, by configuring SFSS with predicate functions or generic functions, the SFSS-based solution extends beyond equality-based policies, supporting a richer class of policy functions, highlighting the versatility of our formalization.

Attribute-hiding aggregation. We introduce *attribute-hiding aggregation* (`ahAgg`), in which each client attaches attributes to its data stream, and an aggregation task specifies a filtering policy that selects which streams should contribute to the aggregation. Indeed, `ahAgg` is the *dual* definition of `phAgg`. The goal is to aggregate over the selected streams while hiding which streams are involved, thereby preserving anonymity and preventing leakage through access patterns. This approach is particularly suited for aggregation tasks grouped by client attributes (*e.g.*, age or region), where data stream, attributes, and attribute membership need protection, We show that SDPF suffices for practical `ahAgg` instantiations and propose a non-interactive query mechanism for arbitrary filtering functions.

Window-based aggregation. We further show how SFSS-based metadata-hiding aggregation supports efficient window-based aggregation with only two calls of SFSS evaluation per data stream, while existing native FSS-based approaches perform FSS evaluation calls that is linear in the window size. This optimization demonstrates how to further reduce computational overhead when SFSS is used in concrete applications.

Conditional transciphering. SFSS provides a new concept called *conditional transciphering* (CT). CT converts a ciphertext to an additive sharing of its encrypted message only if certain conditions are met; 0 is shared otherwise. We note that CT is promising for secure streaming computation beyond metadata-hiding aggregation. More details defer to §5.2.

1.3 Implementation

Besides these promising asymptotic efficiency results, we implement SFSS and report concrete performance. For basic primitives, we evaluate SFSS for single-point functions and comparison functions (as a representative predicate function). Our results confirm that SDPF introduces negligible overhead compared to DPF, while SFSS for predicate functions is more expensive due to the relatively expensive KH-PRF. Nevertheless, this cost is acceptable in practice, which can be further atomized for window-based aggregation. For SFSS applications, our policy-hiding aggregation achieves even better efficiency properties than Vizard, while significantly outperforming the DPF-based baseline. For example, the DPF-based baseline incurs a total of 569.34 MB of client-server communication for 100 data streams with an average of 10K messages, compared to only 13.28 MB for our SDPF-based solution.

1.4 Contributions

We summarize our main contributions as follows.

- We propose streaming function secret sharing, a new cryptographic primitive with a formal definition and concrete constructions. We study feasibility results on SFSS over specialized and generic functions. This is the first systematic study of the streaming notion of FSS.
- SFSS facilitates promising streaming applications, with connections to real-world applications and standardization. SFSS-powered applications are conceptually simple and modular, with improved security, efficiency, and/or enhanced functionality compared with existing approaches.
- We develop a range of techniques and optimizations by carefully exploiting properties of underlying primitives, some of which may be of independent interest.
- We have implemented our schemes and open-sourced the code. We expect our implementation would facilitate research and applications in the related fields.

Connection to IETF PPM standardization. IETF PPM WG is actively standardizing PPM [38]. SFSS provides a promising extension for FSS-based PPM to the streaming setting. We will report our results to IETF PPM WG.

Organization. §2 presents motivation, SFSS definition, and system/threat model for related applications. §3 defines useful notations and definitions. §4 gives concrete SFSS constructions. §5 presents SFSS applications. §6 reports performance. We provide related works in §7 and conclude in §8.

2 Problem Statement

2.1 Example: Weighted Histogram Collection

To concretely emphasize why SFSS is useful, we start by reviewing the DPF-based histogram collection in the two-server model [29]. Let h be the histogram domain size and N_c the number of clients. Each client i holds an input $x^{(i)} \in [h]$ and encodes it as a single-point function $f_{x^{(i)},1}$; $f_{x^{(i)},1}$ outputs 1 at $x^{(i)}$ and 0 elsewhere, and distributes the corresponding DPF keys between two non-colluded servers. To compute the histogram value at position $p \in [h]$, the servers evaluate the DPF keys and jointly compute $\sum_{i\in[N_c]} f_{x^{(i)},1}(p)$ in a secret-shared fashion. The final result is obtained by combining the server outputs. This technique ensures input privacy while enabling efficient, non-interactive server-side computation.

Challenges in the streaming setting. The prior approach does not scale well to streaming analytics, where clients contribute data across multiple rounds. Assume each input $x^{(i)}$ is a static attribute i (*e.g.*, region or device type). In round j, client i submits a new message $m_j^{(i)}$. The goal is to compute a weighted histogram:

$$\sum_{i\in[N_c]} f_{x^{(i)},1}(p)\cdot m_j^{(i)}, \forall p \in [h].$$

A straightforward solution is to generate a fresh DPF for $f_{x^{(i)},m_j^{(i)}}$ in each round, where $f_{x^{(i)},m_j^{(i)}}$ outputs $m_j^{(i)}$ at $x^{(i)}$ and zero elsewhere. This requires each client to regenerate and distribute a new pair of DPF keys per round, even though $x^{(i)}$ remains unchanged, and each client only wants to send one streaming message of constant size. Note that each DPF key is of size $O(\lambda \log h)$ [18, 19, 40], which is still concretely high for large h. Regenerating and resending new DPF keys introduce substantial computation and communication overhead, especially in high-frequency streaming environments.

Streaming function secret sharing. To reduce this overhead, we expect a streaming variant of DPF. Ideally, each client performs a one-time setup by distributing a pair of *streaming DPF keys*. In each round j, the client simply sends a *constant-size* streaming ciphertext $c_j^{(i)}$, encrypting the new message $m_j^{(i)}$. Given the reusable streaming DPF keys and the ciphertext $c_j^{(i)}$, the servers can *non-interactively* compute additive sharing of $f_{x^{(i)},1}(p)\cdot m_j^{(i)}$ for any $p \in [h]$. This leads to an efficient protocol for *weighted* histogram aggregation with *optimal* per-message cost. This is what we want for a streaming DPF scheme, and SFSS generalizes the concept.

2.2 Formal Definition

Def. 1 presents the formal SFSS definition. First, the key generation algorithm Gen outputs a pair of SFSS keys and a streaming encryption key k_e. Second, SFSS introduces an encoding algorithm Enc that encrypts streaming messages, decoupling function setup from data availability. Third, the evaluation algorithm Eval takes a streaming ciphertext as input, enabling key reuse across multiple streaming messages.

Definition 1 (Streaming Function Secret Sharing). *A two-party streaming FSS (SFSS) scheme* $\Pi_{\mathsf{SFSS}} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Eval})$ *for a function family* $\mathcal{F}$ *is defined with the following syntax.*

- $(\mathsf{st}_f, \mathsf{k}_\mathsf{e}, \mathsf{k}_0, \mathsf{k}_1) \leftarrow \mathsf{Gen}(1^\lambda, \hat{f})$. *On input* 1^λ *and a description* $\hat{f} \in \{0,1\}^*$ *for* $f : \mathbb{G}_\mathsf{in} \to \mathbb{G}_\mathsf{out} \in \mathcal{F}$, *output a key pair* $(\mathsf{k}_0, \mathsf{k}_1)$, *a streaming encryption key* k_e, *and a state* st_f.
- $(\mathsf{st}_f, c) \leftarrow \mathsf{Enc}(\mathsf{st}_f, \mathsf{k}_\mathsf{e}, m)$. *On input the state* st_f, *the streaming encryption key* k_e, *and a message* $m \in \mathbb{G}_\mathsf{out}$, *output a ciphertext* $c \in \mathbb{G}_\mathsf{out}$ *and an (updated) state* st_f.
- $s_b \leftarrow \mathsf{Eval}(b, \mathsf{k}_b, x, c)$. *On input the party identifier* $b \in \{0,1\}$, *the party's key* k_b, *an evaluation input* $x \in \mathbb{G}_\mathsf{in}$, *and an streaming ciphertext* $c \in \mathbb{G}_\mathsf{out}$, *output a share* $s_b \in \mathbb{G}_\mathsf{out}$.

We can define $\mathsf{st}_f = \perp$ *for a stateless definition. Thus, this definition captures both state-based and stateless constructions.*

***B*-bounded correctness**. For correctness, the ideal goal for evaluation is to share $m \cdot f(x)$. We define a more general correctness notion called B-bounded correctness, allowing the result to deviate from $m \cdot f(x)$ by at most B. Formally, an SFSS is correct if there exists $B \in \mathbb{G}_\mathsf{out}$ such that for any function $f \in \mathcal{F}$ with description $\hat{f}$, any $x \in \mathbb{G}_\mathsf{in}$, and any $m \in \mathbb{G}_\mathsf{out}$,

$$\Pr\left[\begin{array}{l}(\mathsf{st}_f, \mathsf{k}_\mathsf{e}, \mathsf{k}_0, \mathsf{k}_1) \leftarrow \mathsf{Gen}(1^\lambda, \hat{f}),\\ (\mathsf{st}_f, c) \leftarrow \mathsf{Enc}(\mathsf{st}_f, \mathsf{k}_\mathsf{e}, m):\\ \qquad \sum_{b\in\{0,1\}} \mathsf{Eval}(b, \mathsf{k}_b, x, c) = m \cdot f(x) + e\end{array}\right] = 1$$

for some error $e \in [0, B]$. SFSS is *perfectly correct* if $B = 0$.

Security. SFSS requires that any single SFSS key, together with a polynomial many streaming ciphertexts, reveals no more information than permitted by the leakage function Leak, formally captured by a simulation-based definition:

$\mathbf{Real}^{\mathcal{A}}_{f,b}(1^\lambda)$**:**

```
1: $(\mathsf{st}_f, k_e, k_0, k_1) \leftarrow \mathsf{Gen}(1^\lambda, \hat{f})$
2: $\mathsf{st}_\mathcal{A} \leftarrow \mathcal{A}_0(1^\lambda, \mathsf{st}_f, \mathsf{k}_b)$
3: for $j \in [1, q]$:
4:     $(\mathsf{st}_\mathcal{A}, m_j) \leftarrow \mathcal{A}_j(1^\lambda, \mathsf{st}_\mathcal{A}, \mathsf{st}_f, \mathsf{k}_b, (m_1, \cdots, m_{j-1}), (c_1, \cdots, c_{j-1}))$
5:     $(\mathsf{st}_f, c_j) \leftarrow \mathsf{Enc}(\mathsf{st}_f, \mathsf{k}_e, m_j)$
6: return output $= (\mathsf{st}_\mathcal{A}, \mathsf{k}_b, (c_1, \cdots, c_q))$
```

$\mathbf{Ideal}^{\mathcal{S}}_{f,b}(1^\lambda)$**:**

```
1: $(\mathsf{st}_\mathcal{S}, \mathsf{st}_f, \mathsf{k}_b) \leftarrow \mathcal{S}_0(1^\lambda, b, \mathsf{Leak}(\hat{f}))$
2: $\mathsf{st}_\mathcal{A} \leftarrow \mathcal{A}_0(1^\lambda, \mathsf{st}_f, \mathsf{k}_b)$
3: for $j \in [1, q]$:
4:     $(\mathsf{st}_\mathcal{A}, m_j) \leftarrow \mathcal{A}_j(1^\lambda, \mathsf{st}_\mathcal{A}, \mathsf{st}_f, \mathsf{k}_b, (m_1, \cdots, m_{j-1}), (c_1, \cdots, c_{j-1}))$
5:     $(\mathsf{st}_\mathcal{S}, (\mathsf{st}_f, c_j)) \leftarrow \mathcal{S}_j(1^\lambda, b, \mathsf{st}_\mathcal{S}, \mathsf{Leak}(\hat{f}, (m_1, \cdots, m_j))$
6: return output $= (\mathsf{st}_\mathcal{A}, \mathsf{k}_b, (c_1, \cdots, c_q))$
```

SFSS is *adaptively secure* if for any function $f \in \mathcal{F}$, any $b \in \{0,1\}$, and any probabilistic-polynominal-time (PPT) adversary $\mathcal{A} = (\mathcal{A}_0, \cdots, \mathcal{A}_q)$ with $q = \mathsf{poly}(\lambda)$, there exists a PPT simulator $\mathcal{S} = (\mathcal{S}_0, \cdots, \mathcal{S}_q)$, s.t. for any PPT distinguisher $\mathcal{D}$:

$$\Big|\Pr\Big[\mathsf{output} \leftarrow \mathbf{Real}^{\mathcal{A}}_{f,b}(1^\lambda) : \mathcal{D}(1^\lambda, \mathsf{output}) = 1\Big] - \Pr\Big[\mathsf{output} \leftarrow \mathbf{Ideal}^{\mathcal{S}}_{f,b}(1^\lambda) : \mathcal{D}(1^\lambda, \mathsf{output}) = 1\Big]\Big| \le \mathsf{negl}(\lambda),$$

where the probabilities are over the coins of Gen and Enc, and $\mathsf{Leak}(\hat{f}, m_1, \cdots, m_j)$ reveals the function family $\mathcal{F}$, $\mathbb{G}_\mathsf{in}$, $\mathbb{G}_\mathsf{out}$, size j, and message/ciphertext length $|\mathbb{G}_\mathsf{out}|$.

2.3 Applications: System and Threat Model

System model. Same as previous FSS-based privacy-preserving systems [29, 30, 38], applications considered in this paper consist of three types of participants: ① *data provider*, ② *computing server*, and ③ *data consumer*. In particular, data providers periodically send encrypted/shared messages to the servers for certain secure computation tasks. Then, two non-colluded computing servers conduct secure computation over encrypted/shared messages from data providers. The final computation is typically shared between servers at the end of secure computation. Finally, data consumers receive the computation result from the servers. In many cases, consumers are the computing servers themselves. We also use *client* to denote the data provider for short.

Threat model. We assume two servers are non-colluded. Servers and data contributors are semi-honest, meaning that they will faithfully follow protocol specification but may try to learn more information from protocol execution.

3 Preliminary

3.1 Notations

We use λ and κ to denote computational and statistical security parameters, respectively. $n = \mathsf{poly}(\lambda)$ denotes that $n \in \mathbb{N}$ is polynomial in λ. $\mathsf{negl}(\cdot)$ denotes some unspecified negligible function. Let $a \leftarrow b$ denote that a is assigned with value b. For a set A, $a \xleftarrow{\$} A$ denotes that a is uniformly randomly sampled from A. We use bold lowercase letters $\boldsymbol{a}$ to denote a vector. $\mathbb{G}$ denotes an Abelian group and $\mathbb{F}$ denotes a finite field.

Function family. A function family $\mathcal{F}$ contains a set of functions of the same type. For a function $f \in \mathcal{F}$, we use $\hat{f}$ to denote the *description* of f. $\hat{f}$ contains $\mathbb{G}_\mathsf{in}$, $\mathbb{G}_\mathsf{out}$, and a size parameter $\mathbb{S}$. Below, we show several commonly used function families in FSS.

Single-point function. A single-point function $f_{\alpha,\beta}(x) : \mathbb{G}_\mathsf{in} \to \mathbb{G}_\mathsf{out}$ outputs β for $x = \alpha$ and 0 otherwise.

Multi-point function. A t-point function $f_{A,B}(x) : \mathbb{G}_\mathsf{in} \to \mathbb{G}_\mathsf{out}$ for $A = \{\alpha_1, \cdots, \alpha_t\} \in \mathbb{G}^t_\mathsf{in}$ and $B = \{\beta_1, \cdots, \beta_t\} \in \mathbb{G}^t_\mathsf{out}$ evaluates to β_i on input α_i for $i \in [1, t]$ and 0 otherwise.

Comparison function. A comparison function $f^<_{\alpha,\beta}(x) : \mathbb{G}_\mathsf{in} \to \mathbb{G}_\mathsf{out}$ evaluates to β on input $x < \alpha$ and 0 otherwise.

Predicate function. A predicate function $f(x) : \mathbb{G}_\mathsf{in} \to \mathbb{G}_\mathsf{out}$ outputs $0/1 \in \mathbb{G}_\mathsf{out}$ for any $x \in \mathbb{G}_\mathsf{in}$. Note that single-point functions, multi-point functions, and comparison functions with only binary outputs are all predicate functions.

Definition 2 (Tuple product of functions)**.** *Let* $f : \mathbb{G}_\mathsf{in} \to \mathbb{G}_\mathsf{out}$ *and* $p : \mathbb{G}_\mathsf{in} \to \mathbb{R}$ *be two functions defined over the same domain* $\mathbb{G}_\mathsf{in}$. *The tuple product function* $g = f \times p : \mathbb{G}_\mathsf{in} \to \mathbb{G}_\mathsf{out} \times \mathbb{R}$ *is defined as*

$$g(x) = (f(x), p(x)), \forall x \in \mathbb{G}_\mathsf{in}.$$

3.2 Linear Secret Sharing

We use $[\![x]\!]$ to denote an additive linear secret sharing (LSS) of $x \in \mathbb{G}$. In the two-party case, $[\![x]\!]$ denotes the party P_b holds a share $[\![x]\!]_b$ such that $x = [\![x]\!]_0 + [\![x]\!]_1$. LSS supports affine transformation using public $\alpha, \beta, \gamma \in \mathbb{G}$ via *local* computation:

- $[\![z]\!] \leftarrow \alpha \cdot [\![x]\!] + \beta \cdot [\![y]\!] + \gamma$: P_b computes $[\![z]\!]_b \leftarrow \alpha \cdot [\![x]\!]_b + \beta \cdot [\![y]\!]_b + b \cdot \gamma$.

$\mathbb{G}$ is a ring or field to support multiplication. We can define *secret-shared multiplication* $[\![z]\!] \leftarrow [\![x]\!] \cdot [\![y]\!]$, which requires interaction between the parties. A typical approach [6] is using the Bevear's multiplication triple $([\![a]\!], [\![b]\!], [\![c]\!])$ with $c = a \cdot b$: the parties compute $[\![e]\!] \leftarrow [\![x]\!] - [\![a]\!]$, $[\![f]\!] = [\![y]\!] - [\![b]\!]$, and open e and f. The parties compute $[\![z]\!] \leftarrow f \cdot [\![a]\!] + e \cdot [\![b]\!] + [\![c]\!] + e \cdot f$. One can check $z = x \cdot y$.

3.3 Function Secret Sharing

Definition 3 (Function Secret Sharing [18])**.** *A two-party FSS scheme for function family $\mathcal{F}$ is defined as follows.*

- $(\mathsf{k}_0, \mathsf{k}_1) \leftarrow \mathsf{Gen}(1^\lambda, \hat{f})$. *On input 1^λ and description $\hat{f} \in \{0,1\}^*$ of a function $f : \mathbb{G}_{\mathsf{in}} \to \mathbb{G}_{\mathsf{out}} \in \mathcal{F}$, output a key pair* $(\mathsf{k}_0, \mathsf{k}_1)$.
- $[\![f(x)]\!]_b \leftarrow \mathsf{Eval}(b, \mathsf{k}_b, x)$. *On input a party identifier $b \in \{0,1\}$, key k_b, and $x \in \mathbb{G}_{\mathsf{in}}$, output a share $[\![f(x)]\!]_b \in \mathbb{G}_{\mathsf{out}}$.*

An FSS scheme $(\mathsf{Gen}, \mathsf{Eval})$ *is secure for the function family $\mathcal{F}$ with leakage* $\mathsf{Leak} : \{0,1\}^* \to \{0,1\}^*$ *if the following holds:*

- ***Correctness**. For any function $f \in \mathcal{F}$ and any $x \in \mathbb{G}_{\mathsf{in}}$,*

$$\Pr\left[\begin{array}{l} (\mathsf{k}_0, \mathsf{k}_1) \leftarrow \mathsf{Gen}(1^\lambda, \hat{f}) : \\ \quad \mathsf{Eval}_0(0, \mathsf{k}_0, x) + \mathsf{Eval}_1(1, \mathsf{k}_1, x) = f(x) \end{array}\right] = 1$$

- ***Security**. For any PPT adversary $\mathcal{A}$, there exists a PPT simulator $\mathcal{S}$ such that for any $f \in \mathcal{F}$ and $b \in \{0,1\}$:*

$$\Big| \Pr\left[(k_0, k_1) \leftarrow \mathsf{Gen}(1^\lambda, \hat{f}) : \mathcal{A}(1^\lambda, \mathsf{k}_b) = 1\right] - \Pr\left[\mathsf{k}_b \leftarrow \mathcal{S}(1^\lambda, b, \mathsf{Leak}(\hat{f})) : \mathcal{A}(1^\lambda, \mathsf{k}_b) = 1\right] \Big| \leq \mathsf{negl}(\lambda),$$

where $\mathsf{Leak}(\hat{f})$ reveals domain $\mathbb{G}_{\mathsf{in}}$ and range $\mathbb{G}_{\mathsf{out}}$ of f.

FSS with pseudorandom share. We say an FSS scheme for $\mathcal{F}$ has pseudorandom-share property if for any $f \in \mathcal{F}$, $b \in \{0,1\}$, and k_b with $(\mathsf{k}_0, \mathsf{k}_1) \leftarrow \mathsf{Gen}(1^\lambda, \hat{f})$ over the coin of Gen, and any $x \in \mathbb{G}_{\mathsf{in}}$, the distribution of $[\![f(x)]\!]_b = \mathsf{Eval}(b, \mathsf{k}_b, x)$ is computationally indistinguishable from uniform distribution over $\mathbb{G}_{\mathsf{out}}$, which requires $\mathsf{Eval}(b, \mathsf{k}_b, \cdot)$ beheaves like a pseudorandom function. This property is formalized by Boyle *et al.* [18] for poly-spanning function families (§4.1, [18]), which includes useful and practical functions considered in existing FSS work, such as point functions, comparison functions, etc.

3.4 Key-Homomorphic PRF

Definition 4 (B-almost Key-Homomorphic Pseudorandom Function (KH-PRF))**.** *Let $F : \mathbb{K} \times \mathbb{X} \to \mathbb{Y}$ be a secure pseudorandom function and $(\mathbb{K}, +_{\mathbb{K}})$ is a group. We say f is B-almost key-homomorphic if for any $k_1, k_2 \in \mathbb{K}$ and $x \in \mathbb{X}$, there exits a bounded error $e_{\mathsf{kh}} \in [0, B]$ for some $B \in \mathbb{G}_{\mathsf{out}}$ such that*

$$F(k_1 +_{\mathbb{K}} k_2, x) = F(k_1, x) +_{\mathbb{Y}} F(k_2, x) +_{\mathbb{Y}} e_{\mathsf{kh}}.$$

Special zero-key property. We say that a KH-PRF F has the *special zero-key* property if $F(0, x) = 0 \in \mathbb{G}_{\mathsf{out}}$ holds for any $x \in \mathbb{G}_{\mathsf{in}}$. We note that the *special zero-key* property does not generally hold for all KH-PRFs or any group $\mathbb{G}_{\mathsf{out}}$. For example, the DDH-based PRF [51] $F(k, x) = H(x)^k$, assuming a random oracle $H : \{0,1\}^* \to \mathbb{G}$, is a KH-PRF, but $F(0, x) = H(x)^0 = 1_{\mathbb{G}}$.

LWR-based KH-PRF. We can instantiate KH-PRF with a special zero-key from the seminal LWR-based KH-PRF [5]. Formally, let $q \geq p \geq 2$ be two integers. The rounding function $\lfloor x \rceil_p : \mathbb{Z}_q \to \mathbb{Z}_p$ outputs $\lfloor x \rceil_p = i$, where i is the largest multiple of q/p that does not exceed x. Let $H_1 : \{0,1\}^* \to \mathbb{Z}_q^d$ be a hash function modeled as a random oracle. For $\boldsymbol{k} \in \mathbb{Z}_q^d$, $F_{\mathsf{LWR}}(\boldsymbol{k}, x) : \mathbb{Z}_q^d \times \{0,1\}^* \to \mathbb{Z}_p$ is defined as:

$$F_{\mathsf{LWR}}(\boldsymbol{k}, x) = \lfloor \langle H_1(x), \boldsymbol{k} \rangle \rceil_p.$$

Note that $F_{\mathsf{LWR}}(\boldsymbol{k}, x)$ is B-almost key-homomorphic with $B = 1$, as the rounding introduces 1 bit of error at most. We note that LWR-based KH-PRF has a special zero-key.

4 Streaming Function Secret Sharing

4.1 Possible Approaches

Recall that FSS provides a mechanism to additively share $f(x)$, for a secret-shared function f and any public $x \in \mathbb{G}_{\mathsf{in}}$, via *non-interactive* evaluation. SFSS extends this capability by additively sharing $f(x) \cdot m$ *non-interactively*, where $x \in \mathbb{G}_{\mathsf{in}}$ is public and c is a streaming ciphertext encrypting m. Although no prior work formalizes this streaming variant of FSS, several approaches resemble some aspects of this notion. However, these methods do not align with the non-interactive semantics of (S)FSS and therefore fail to realize SFSS.

Homomorphic-encryption-based approach. Linear homomorphic encryption (LHE) seems to be a match to compute $f(x) \cdot m$. We compare the conceptual differences between SFSS and LHE-based approaches.

Suppose we have an LHE scheme with message space $\mathcal{M}$, ciphertext space $\mathcal{C}$, and $c = \mathsf{Enc}(\mathsf{pk}, m)$ be the ciphertext of message m. LHE supports homomorphic addition ($\boxplus$) and scalar multiplication ($\boxdot$) over ciphertexts:

- $\mathsf{Enc}(\mathsf{pk}, x) \boxplus \mathsf{Enc}(\mathsf{pk}, y) = \mathsf{Enc}(\mathsf{pk}, x +_{\mathcal{M}} y)$
- For any $y \in \mathcal{M}$, $\mathsf{Enc}(\mathsf{pk}, x) \boxdot y = \mathsf{Enc}(\mathsf{pk}, x \cdot_{\mathcal{M}} y)$

A simple “FSS+LHE” solution is sketched as follows:

- A streaming message m is encrypted as $c = \mathsf{Enc}(\mathsf{pk}, m)$.
- For $b \in \{0,1\}$, evaluate FSS to obtain $[\![f(x)]\!]_b$ over $\mathcal{M}$. Compute $c'_b \leftarrow c \boxdot [\![f(x)]\!]_b$. By definition, $c'_0 \boxplus c'_1 = \mathsf{Enc}(\mathsf{pk}, f(x) \cdot_{\mathcal{M}} m)$.

As we can see, ciphertexts c'_0 and c'_1 are $\boxplus$-shared over the ciphertext space $\mathcal{C}$. While it is possible to recover or additively share $f(x) \cdot m$ via secret-shared decryption protocols for certain LHE schemes, this approach is inherently interactive. We provide a concrete example in Appendix §A.3.

Vizard’s approach. Vizard [25] resorts to a similar “FSS+LHE” concept but combines DPF over an (symmetric) homomorphic stream encryption (HSE) scheme [24]. However, we find that Vizard cannot achieve its claimed security and efficiency goals, let alone SFSS.

In Vizard, a client i generates a pair of DPF keys for a function $f^{(i)} \in \mathcal{F}$ and distributes the keys between two servers, serving as a secret policy function. For the j-th message, the client i shares the streaming message $[\![m_j^{(i)}]\!]$ between the servers. To share $[\![f^{(i)}(x) \cdot m_j^{(i)}]\!]$ for any $x \in \mathbb{G}_{\mathsf{in}}$, the servers non-interactively generate a random sharing $[\![r_j^{(i)}]\!]$, compute $[\![c_j^{(i)}]\!] = [\![m_j^{(i)}]\!] + [\![r_j^{(i)}]\!]$, and open $c_j^{(i)} = m_j^{(i)} + r_j^{(i)}$, where $r_j^{(i)}$ perfectly hides $m_j^{(i)}$. Then, the servers jointly compute

$$[\![f^{(i)}(x) \cdot m_j^{(i)}]\!] \leftarrow c_j^{(i)} \cdot [\![f^{(i)}(x)]\!] - [\![f^{(i)}(x)]\!] \cdot [\![r_j^{(i)}]\!].$$

However, computing $[\![f(x)]\!] \cdot [\![r_j^{(i)}]\!]$ requires *interactive* secret-shared multiplication.

Directly applying the above method for policy-hiding aggregation over ciphertexts from N_c clients incurs $O(N_c)$ communication complexity, as the servers must perform secret-shared multiplication per data stream. To reduce the overhead, Vizard *reuses* a mask r_j for all N_c clients in round j, *i.e.*, $r_j^{(i)} = r_j$ for all $i \in [N_c]$, thus we have $c_j^{(i)} = m_j^{(i)} + r_j$ for all $i \in [N_c]$. In this way, the server computes the aggregation

$$[\![s]\!] \leftarrow \sum_{i \in [N_c]} c_j^{(i)} \cdot [\![f^{(i)}(x)]\!] - \Big(\sum_{i \in [N_c]} [\![f^{(i)}(x)]\!]\Big) \cdot [\![r_j]\!].$$

Now the servers only conduct one single secret-shared multiplication $(\sum_{i \in [N_c]} [\![f^{(i)}(x)]\!]) \cdot [\![r_j]\!]$ with $O(1)$ communication complexity.

Unfortunately, this optimization introduces a security flaw, as the secret key r_j is reused to encrypt messages for all clients. Thus, for any $i_1, i_2 \in [N_c]$, we have

$$m_j^{(i_1)} - m_j^{(i_2)} = c_j^{(i_1)} - c_j^{(i_2)},$$

which breaks security trivially. It is unclear how to fix the flaw without blowing up to $O(N_c)$ communication complexity.

Summary. Existing approaches are inherently interactive, not aligned with SFSS non-interaction semantics. In general, communication and interaction are more likely to be efficiency bottlenecks for real-world applications. Though the LHE-based approach may be promising for specific aggregation applications, SFSS evaluation generates additive sharing without interaction, which would benefit broader applications beyond aggregation. Vizard necessitates communication to generate a ciphertext, and its evaluation phase is interactive (and flawed). Looking ahead, we design non-interactive streaming encryption and non-interactive evaluation as required by SFSS.

4.2 SFSS for Single-point Functions

We first propose an SFSS compiler for single-point functions $f_{\alpha,\beta}$, called streaming DPF (SDPF). We start with the case $\beta = 1$ and extend it to the non-binary case in Appendix §A.1.

4.2.1 Streaming DPF with $\beta = 1$

Technique overview. Fig. 2 sketches the high-level idea, and Fig. 3 presents a formal scheme. At the core are *non-interactive* PRF-based streaming masking and unmasking mechanisms, which use the pseudorandom shares from DPF as the PRF keys. Now we elaborate our idea step by step.

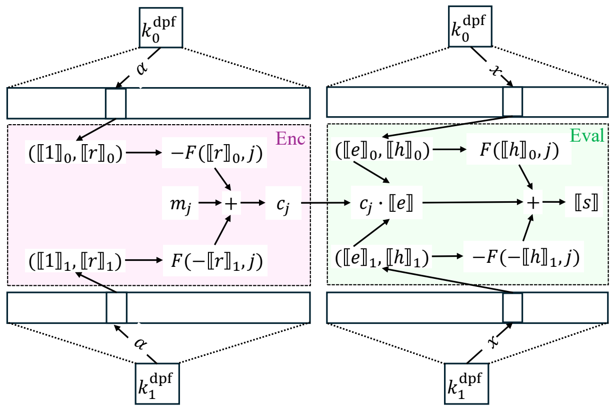


Figure 2: **Sketched streaming encryption and evaluation**. Two pseudorandom shares $([\![r]\!]_0, [\![r]\!]_1)$ evaluated at α serve as the PRF keys over counter j to derive two masks for encryption. For evaluation over any $x \in \mathbb{G}_{\mathsf{in}}$, a local unmasking strategy using the pseudorandom shares $([\![e]\!]_b, [\![h]\!]_b)$ essentially shares $[\![s]\!]$ such that $s = m_j$ if $x = \alpha$, and $s = 0$ otherwise.

First, we rely on a pseudorandom distributed point function (PDPF) to design SDPF. PDPF additionally embeds a pseudorandom payload r at the target point α:

$$f_{\alpha,(1,r)} = \begin{cases} (1,r), i = \alpha \\ (0,0), i \neq \alpha, \end{cases}$$

We note that it is essentially free to obtain PDPF from existing PRG-based DPF constructions [12, 18, 19, 40]. And we use the DPF from [19] in our implementation. The remaining question is how to compile PDPF to SDPF.

SDPF setup phase simply generates a pair of PDPF keys $(k_0^{\mathsf{dpf}}, k_1^{\mathsf{dpf}})$ via DPF key generation. From the DPF keys, as depicted in Fig. 2, we use the DPF pseudorandom shares evaluated at α as a pair of PRF keys to derive pseudorandom masks for streaming encryption. Specifically, let

$$(\llbracket 1 \rrbracket_b, \llbracket r \rrbracket_b) \leftarrow \Pi_{\mathsf{DPF}}.\mathsf{Eval}(b, \mathsf{k}_b, \alpha)$$

for $b \in \{0,1\}$, the pseudorandom shares $(\llbracket r \rrbracket_0, \llbracket r \rrbracket_1)$ are used as the PRF keys. To encrypt the j-th streaming message m_j, we take the following masking strategy:

$$c_j \leftarrow m_j - F(\llbracket r \rrbracket_0, j) + F(-\llbracket r \rrbracket_1, j).$$

Namely, m_j is double-masked by two pseudorandom masks derived from $\llbracket r \rrbracket_0$ and $\llbracket r \rrbracket_1$ over the counter j using PRF F.

It remains to design the SFSS evaluation to correctly share $f_{\alpha,1}(x) \cdot m_j$, by evaluation using k_b and c_j for any $x \in \mathbb{G}_{\mathsf{in}}$. As depicted in Fig. 2, we use the following non-interactive unmasking strategy:

$$(\llbracket e \rrbracket_b, \llbracket h \rrbracket_b) \leftarrow \Pi_{\mathsf{DPF}}.\mathsf{Eval}(b, \mathsf{k}_b, x),$$

$$\llbracket s \rrbracket_b \leftarrow \llbracket e \rrbracket_b \cdot c_j + (-1)^b \cdot F((-1)^b \cdot \llbracket h \rrbracket_b, j).$$

We prove $\llbracket s \rrbracket_0 + \llbracket s \rrbracket_1 = f_{\alpha,1}(x) \cdot m_j$ for any $x \in \mathbb{G}_{\mathsf{in}}$ as follows.

Correctness. We prove $\llbracket s \rrbracket_0 + \llbracket s \rrbracket_1 = m_j$ for $x = \alpha$ and $\llbracket s \rrbracket_0 + \llbracket s \rrbracket_1 = 0$ for $x \neq \alpha$. To see this, $\llbracket s \rrbracket_0 + \llbracket s \rrbracket_1 = \llbracket e \rrbracket_0 \cdot c_j + (-1)^0 \cdot F((-1)^0 \cdot \llbracket h \rrbracket_0, j) + \llbracket e \rrbracket_1 \cdot c_j + (-1)^1 \cdot F((-1)^1 \cdot \llbracket h \rrbracket_1, j) = e \cdot (m_j - F(\llbracket r \rrbracket_0, j) + F(-\llbracket r \rrbracket_1, j)) + F(\llbracket h \rrbracket_0, j) - F(-\llbracket h \rrbracket_1, j)$. There are two cases depending on whether $x = \alpha$ or $x \neq \alpha$:

- Case $x = \alpha$: In this case, we have $e = 1$, $\llbracket h \rrbracket_0 = \llbracket r \rrbracket_0$, and $\llbracket h \rrbracket_1 = \llbracket r \rrbracket_1$, thus $\llbracket s \rrbracket_0 + \llbracket s \rrbracket_1 = m_j - F(\llbracket r \rrbracket_0, j) + F(-\llbracket r \rrbracket_1, j) + F(\llbracket h \rrbracket_0, j) - F(-\llbracket h \rrbracket_1, j) = m_j$.
- Case $x \neq \alpha$: In this case, we have $e = 0$, $\llbracket h \rrbracket_0 + \llbracket h \rrbracket_1 = 0 \Leftrightarrow \llbracket h \rrbracket_0 = -\llbracket h \rrbracket_1$, thus $e \cdot c_j + F(\llbracket h \rrbracket_0, j) - F(-\llbracket h \rrbracket_1, j) = 0$.

Overall, $\llbracket s \rrbracket_0 + \llbracket s \rrbracket_1 = f_{\alpha,1}(x) \cdot m_j, \forall x \in \mathbb{G}_{\mathsf{in}}$.

Security. Intuitively, a secure DPF scheme ensures that k_b is pseudorandom and hides α and r. $\llbracket r \rrbracket_{1-b}$ is a pseudorandom share. PRF F guarantees that $F(\llbracket r \rrbracket_{1-b}, j)$ is pseudorandom. As a result, the distribution of the ciphertext $c_j = m_j - F(\llbracket r \rrbracket_0, j) + F(-\llbracket r \rrbracket_1, j)$ is computationally indistinguishable from uniform distribution over $\mathbb{G}_{\mathsf{out}}$, as $F(\llbracket r \rrbracket_{1-i}, j)$ serves as a pseudorandom mask that hides other values. Therefore, the tuple $(\mathsf{k}_b, c_1, c_2, \ldots)$ is computationally indistinguishable from random in the key and ciphertext spaces. Note that ctr should never repeat. Thus, we take sufficiently large $\mathbb{G}_{\mathsf{ctr}}$ and re-setup SFSS keys if necessary; the same goes for other SFSS constructions. Theorem 1 formalizes the security intuition, with a full proof in Appendix §D.1.

Theorem 1. *Let Π_{DPF} be a DPF scheme with pseudorandom-share property for a single-point function family with domain $\mathbb{G}_{\mathsf{in}}$ and range $\mathbb{G}_{\mathsf{out}} \times \{0,1\}^\lambda$, PRF $F : \{0,1\}^\lambda \times \mathbb{G}_{\mathsf{ctr}} \to \mathbb{G}_{\mathsf{out}}$ with sufficiently large $\mathbb{G}_{\mathsf{ctr}}$, then Π_{SDPF} in Fig. 3 is an adaptively secure SDPF scheme with domain $\mathbb{G}_{\mathsf{in}}$ and range $\mathbb{G}_{\mathsf{out}}$.*

```
Parameters: PRF $F : \{0,1\}^\lambda \times \mathbb{G}_{\mathsf{ctr}} \to \mathbb{G}_{\mathsf{out}}$; PRF $F : \{0,1\}^\lambda \times \mathbb{G}_{\mathsf{ctr}} \to \mathbb{G}_{\mathsf{out}}$; Function family $\mathcal{F}_{\mathbb{G}_{\mathsf{in}}, \mathbb{G}_{\mathsf{out}} \times \{0,1\}^\lambda}$ of single-point functions with domain $\mathbb{G}_{\mathsf{in}}$ and range $\mathbb{G}_{\mathsf{out}} \times \{0,1\}^\lambda$; DPF scheme $\Pi_{\mathsf{DPF}}$ for function family $\mathcal{F}_{\mathbb{G}_{\mathsf{in}}, \mathbb{G}_{\mathsf{out}} \times \{0,1\}^\lambda}$; $|\mathbb{G}_{\mathsf{ctr}}| \geq 2^\lambda$.
$\mathsf{Gen}(1^\lambda, \hat{f}_{\alpha,(1,r)} \in \mathcal{F}_{\mathbb{G}_{\mathsf{in}}, \mathbb{G}_{\mathsf{out}} \times \{0,1\}^\lambda})$:        $\triangleright$ Here $r \xleftarrow{\$} \{0,1\}^\lambda$
  1: Set $\mathsf{ctr} = 1$, $\mathsf{st}_f \leftarrow \{\mathsf{ctr}\}$
  2: $(\mathsf{k}_0^{\mathsf{dpf}}, \mathsf{k}_1^{\mathsf{dpf}}) \leftarrow \Pi_{\mathsf{DPF}}.\mathsf{Gen}(1^\lambda, \hat{f}_{\alpha,(1,r)})$
  3: For $b \in \{0,1\}$, $(\llbracket 1 \rrbracket_b, \llbracket r \rrbracket_b) \leftarrow \Pi_{\mathsf{DPF}}.\mathsf{Eval}(b, \mathsf{k}_b^{\mathsf{dpf}}, \alpha)$
  4: $\mathsf{k}_e = \{\llbracket r \rrbracket_0, \llbracket r \rrbracket_1\}$, $\mathsf{k}_0 \leftarrow \{\mathsf{k}_0^{\mathsf{dpf}}\}$, $\mathsf{k}_1 \leftarrow \{\mathsf{k}_1^{\mathsf{dpf}}\}$
  5: Return $(\mathsf{st}_f, \mathsf{k}_e, \mathsf{k}_0, \mathsf{k}_1)$.
$\mathsf{Enc}(\mathsf{st}_f, \mathsf{k}_e, m)$:
  1: Parse $\mathsf{st}_f = \{\mathsf{ctr}\}$, $\mathsf{k}_e = \{\llbracket r \rrbracket_0, \llbracket r \rrbracket_1\}$.
  2: $c_{\mathsf{ctr}} \leftarrow m - F(\llbracket r \rrbracket_0, \mathsf{ctr}) + F(-\llbracket r \rrbracket_1, \mathsf{ctr})$
  3: Set $c \leftarrow (\mathsf{ctr}, c_{\mathsf{ctr}})$, update $\mathsf{st}_f \leftarrow \{\mathsf{ctr}+1\}$
  4: Return $(\mathsf{st}_f, c)$
$\mathsf{Eval}(b, \mathsf{k}_b, x, c)$:
  1: Parse $c = (j, c_j)$ and $\mathsf{k}_b = \{\mathsf{k}_b^{\mathsf{dpf}}\}$
  2: $(\llbracket e \rrbracket_b, \llbracket h \rrbracket_b) \leftarrow \Pi_{\mathsf{DPF}}.\mathsf{Eval}(b, \mathsf{k}_b^{\mathsf{dpf}}, x)$
  3: $\llbracket s \rrbracket_b \leftarrow \llbracket e \rrbracket_b \cdot c_j + (-1)^b \cdot F((-1)^b \cdot \llbracket h \rrbracket_b, j)$
  4: Return $(\llbracket s \rrbracket_b)$
```

Figure 3: A SDPF construction with $\beta = 1$.

4.3 SFSS for Predicate Functions

SDPF introduces little overhead compared to DPF, as it only additionally relies on PRFs. However, this framework cannot easily be extended beyond single-point functions.

4.3.1 Challlenge

To elaborate the challenge concretely, we show an attempt of designing SFSS for a multi-point function $f_{A,B}$ with $A = \{\alpha_1, \alpha_2\}$ and $B = \{\beta_1, \beta_2\}$, where $\alpha_1 \neq \alpha_2$ and $\beta_1 = \beta_2 = 1$. As before, we define an argumented two-point function $f_{A,(B,r)} : \mathbb{G}_{\mathsf{in}} \to \mathbb{G}_{\mathsf{out}} \times \{0,1\}^\lambda$ as

$$f_{A,(B,r)}(x) = \begin{cases} (1,r), x \in A \\ (0,0), x \notin A, \end{cases}$$

and generate the MDPF keys $(\mathsf{k}_0, \mathsf{k}_1) \leftarrow \Pi_{\mathsf{MDPF}}.\mathsf{Gen}(1^\lambda, \hat{f})$ via any existing MDPF scheme such as [17].

The remaining question is how to perform streaming encryption. In particular, how to choose the binding randomness used in the encryption process and correctly unmask it during evaluation. Without loss of generality, suppose the correlated randomness associated with index α_1 is selected:

$$(\llbracket \beta_1 \rrbracket_b, \llbracket r \rrbracket_b) \leftarrow \Pi_{\mathsf{MDPF}}.\mathsf{Eval}(b, \mathsf{k}_b, \alpha_1),$$

and the encryption algorithm encrypts the j-th streaming message as $c_j \leftarrow m_j - F(\llbracket r \rrbracket_0, j) + F(-\llbracket r \rrbracket_1, j)$ as before. Similialry, the evaluation algorithm $\mathsf{Eval}(b, \mathsf{k}_b, (j, c_j), x)$ computes:

$$(\llbracket e \rrbracket_b, \llbracket h \rrbracket_b) \leftarrow \Pi_{\mathsf{MDPF}}.\mathsf{Eval}(b, \mathsf{k}_b, x),$$
$$\llbracket s \rrbracket_b \leftarrow \llbracket e \rrbracket_b \cdot c_j + (-1)^b \cdot F((-1)^b \cdot \llbracket h \rrbracket_b, j).$$

One can verify that correctness still holds for $x \neq \alpha_2$. However, correctness fails to hold at $x = \alpha_2$. To see this, $\llbracket s \rrbracket_0 + \llbracket s \rrbracket_1 = e \cdot c_j + F(\llbracket h \rrbracket_b, j) - F(-\llbracket h \rrbracket_b, j) = e \cdot (m_j - F(\llbracket r \rrbracket_0, j) + F(-\llbracket r \rrbracket_1, j)) + (F(\llbracket h \rrbracket_0, j) - F(-\llbracket h \rrbracket_1, j)) = m_j - F(\llbracket r \rrbracket_0, j) + F(-\llbracket r \rrbracket_1, j)) + F(\llbracket h \rrbracket_0, j) - F(-\llbracket h \rrbracket_1, j)$. Since the shares evaluated at α_1 and α_2 are pseudorandom and inherently randomized, the shares $\llbracket r \rrbracket_0 \neq \llbracket h \rrbracket_0$ and $\llbracket r \rrbracket_1 \neq \llbracket h \rrbracket_1$ hold with overwhelming probability – no way to take off the masks as we do for SDPF.

4.3.2 New KH-PRF-based Solution

The prior approach attempted to embed correlated randomness at non-zero outputs to derive streaming masks, but failed to ensure correctness. *The issue arises as the embedded randomness r is not directly used to derive masks; instead, its randomized shares are used to derive masking keys, leading to inconsistency during evaluation*. Hence, the evaluation cannot correctly take off the masks from the ciphertext. An intuition comes: can we derive random masks from r directly?

We propose a new masking strategy that directly derives masks from a shared secret. The core challenge is to compute a shared mask from an additively shared secret key, allowing decryption to be performed in a secret-shared manner. We resolve this via *key-homomorphic pseudorandom function* (KH-PRF), which supports local PRF evaluation over shared keys. By embedding a KH-PRF key in the function output, servers can recover consistent key shares at evaluation points and use key homomorphism to jointly derive a shared PRF output, enabling secure and correct masking and unmasking.

The proposed construction. Fig. 4 describes the formal construction. To encode the KH-PRF key into a predicate function, we first define a payload function $p : \mathbb{G}_{\mathsf{in}} \to \mathbb{K}_{\mathsf{kh}}$.

$$p(x) = \begin{cases} \mathsf{k}_{\mathsf{kh}}, & f(x) = 1, \\ 0, & f(x) = 0, \end{cases}$$

where k_{kh} is the KH-PRF key. Then, we define an argumented function $h = f \times p$ as the tuple product of f and p (c.f., Def. 2). By definition, we have

$$h(x) = \begin{cases} (1, \mathsf{k}_{\mathsf{kh}}), & f(x) = 1, \\ (0, 0), & f(x) = 0. \end{cases}$$

Let Π_{FSS} be an FSS scheme for the function family $\mathcal{F}$ that h belongs to. We can generate a pair of FSS keys $(\mathsf{k}_0^{\mathsf{FSS}}, \mathsf{k}_1^{\mathsf{FSS}}) \leftarrow \Pi_{\mathsf{FSS}}.\mathsf{Gen}(1^\lambda, \hat{h})$. To encrypt the j-th streaming message m_j, the streaming encryption algorithm simply computes:

$$c_j \leftarrow m_j + F(\mathsf{k}_{\mathsf{kh}}, j),$$

```
Parameters: A $B$-almost KH-PRF $F : \mathbb{K}_{\mathsf{kh}} \times \mathbb{G}_{\mathsf{ctr}} \to \mathbb{G}_{\mathsf{out}}$; A function family $\mathcal{F}_{\mathbb{G}_{\mathsf{in}},\mathbb{G}_{\mathsf{out}}}$ for predicate functions with domain $\mathbb{G}_{\mathsf{in}}$ and range $\mathbb{G}_{\mathsf{out}}$; A function family $\mathcal{P}_{\mathbb{G}_{\mathsf{in}},\mathbb{K}_{\mathsf{kh}}}$ for payload functions with domain $\mathbb{G}_{\mathsf{in}}$ and range $\mathbb{K}_{\mathsf{kh}}$; A function family $\mathcal{H} = \{f \times p : f \in \mathcal{F}_{\mathbb{G}_{\mathsf{in}},\mathbb{G}_{\mathsf{out}}}, p \in \mathcal{P}_{\mathbb{G}_{\mathsf{in}},\mathbb{K}_{\mathsf{kh}}}\}$ with domain $\mathbb{G}_{\mathsf{in}}$ and range $\mathbb{G}_{\mathsf{out}} \times \mathbb{K}_{\mathsf{kh}}$; An FSS scheme $\Pi_{\mathsf{FSS}}$ for the function family $\mathcal{H}$; $|\mathbb{G}_{\mathsf{ctr}}| \geq 2^\lambda$.
$\mathsf{Gen}(1^\lambda, \hat{f} \in \mathcal{F}_{\mathbb{G}_{\mathsf{in}},\mathbb{G}_{\mathsf{out}}})$:
  1: $\mathsf{k}_{\mathsf{kh}} \xleftarrow{\$} \mathbb{K}_{\mathsf{kh}}$, $\mathsf{ctr} = 1$.
  2: Define function $p \in \mathcal{P}$ s.t. $p(x) = k_{\mathsf{kh}} \cdot f(x), \forall x \in \mathbb{G}_{\mathsf{in}}$
  3: Let $h = f \times p$, compute $(k_0^{\mathsf{fss}}, k_1^{\mathsf{fss}}) \leftarrow \Pi_{\mathsf{FSS}}.\mathsf{Gen}(1^\lambda, \hat{h})$
  4: $\mathsf{st}_f \leftarrow \{\mathsf{ctr}\}$, $\mathsf{k}_{\mathsf{e}} \leftarrow \{\mathsf{k}_{\mathsf{kh}}\}$, $\mathsf{k}_0 \leftarrow \{\mathsf{k}_0^{\mathsf{fss}}\}$, $\mathsf{k}_1 \leftarrow \{\mathsf{k}_1^{\mathsf{fss}}\}$
  5: Return $(\mathsf{st}_f, \mathsf{k}_{\mathsf{e}}, \mathsf{k}_0, \mathsf{k}_1)$
$\mathsf{Enc}(\mathsf{st}_f, \mathsf{k}_{\mathsf{e}}, m)$:
  1: Parse $\mathsf{st}_f = \{\mathsf{ctr}\}$ and $\mathsf{k}_{\mathsf{e}} = \{\mathsf{k}_{\mathsf{kh}}\}$
  2: $c_{\mathsf{ctr}} \leftarrow m + F(\mathsf{k}_{\mathsf{kh}}, \mathsf{ctr})$
  3: Set $c \leftarrow (\mathsf{ctr}, c_{\mathsf{ctr}})$, update $\mathsf{st}'_f = \{\mathsf{ctr} + 1\}$
  4: Return $(\mathsf{st}'_f, c)$
$\mathsf{Eval}(b, \mathsf{k}_b, x, c)$:
  1: Parse $c = (j, c_j)$ and $\mathsf{k}_b = \{\mathsf{k}_b^{\mathsf{fss}}\}$
  2: $(\llbracket e \rrbracket_b, \llbracket \mathsf{k} \rrbracket_b) \leftarrow \Pi_{\mathsf{FSS}}.\mathsf{Eval}(b, \mathsf{k}_b^{\mathsf{fss}}, x)$
  3: $\llbracket s \rrbracket_b \leftarrow \llbracket e \rrbracket_b \cdot c_j - F(\llbracket \mathsf{k} \rrbracket_b, j)$
  4: Return $(\llbracket s \rrbracket_b)$
```

Figure 4: A SFSS framework for predicate functions.

where $F(\mathsf{k}_{\mathsf{kh}}, j)$ computationally hides m_j.

Two evaluation algorithms should share $f(x) \cdot m_j$ by correctly taking off the mask $F(\mathsf{k}_{\mathsf{kh}}, j)$ in a secret-shared fashion. To this end, the SFSS evaluation algorithm computes

$$(\llbracket e \rrbracket_b, \llbracket \mathsf{k} \rrbracket_b) \leftarrow \Pi_{\mathsf{FSS}}.\mathsf{Eval}(b, \mathsf{k}_b^{\mathsf{fss}}, x),$$
$$\llbracket s \rrbracket_b \leftarrow \llbracket e \rrbracket_b \cdot c_j - F(\llbracket \mathsf{k} \rrbracket_b, j).$$

We now argue its correctness and security as follows.

Correctness. For any $x \in \mathbb{G}_{\mathsf{in}}$, we show that $\llbracket s \rrbracket_0 + \llbracket s \rrbracket_1 = f(x) \cdot m + e_{\mathsf{kh}}$, where $e_{\mathsf{kh}} \in [0, B]$ is a small and bounded error. Thus, our KH-PRF-based SFSS satisfies the B-bounded correctness, and we use KH-PRF with a special zero-key property (c.f. §3.4). By the definition, we have:

$$\llbracket s \rrbracket_0 + \llbracket s \rrbracket_1 = e \cdot c_j - F(\llbracket \mathsf{k} \rrbracket_0, j) - F(\llbracket \mathsf{k} \rrbracket_1, j)$$

There are two cases depending on whether $f(x) = 0$ or 1:

- Case $f(x) = 1$: We have $e = 1$ and $\mathsf{k} = \mathsf{k}_{\mathsf{kh}}$, hence $\llbracket s \rrbracket_0 + \llbracket s \rrbracket_1 = m_j + F(\mathsf{k}_{\mathsf{kh}}, j) - (F(\llbracket \mathsf{k}_{\mathsf{kh}} \rrbracket_0, j) + F(\llbracket \mathsf{k}_{\mathsf{kh}} \rrbracket_1, j)) = m_j + F(\mathsf{k}_{\mathsf{kh}}, j) - (F(\mathsf{k}_{\mathsf{kh}}, j) - e_{\mathsf{kh}}) = m_j + e_{\mathsf{kh}}$,
- Case $f(x) = 0$: We have $e = 0$, $\mathsf{k} = 0$, and $\llbracket s \rrbracket_0 + \llbracket s \rrbracket_1 = 0 - (F(\llbracket 0 \rrbracket_0, j) + F(\llbracket 0 \rrbracket_1, j)) = -(F(0, j) - e_{\mathsf{kh}}) = e_{\mathsf{kh}}$,

where $e_{\mathsf{kh}} \in [0, B]$ is the bounded error introduced from B-almost KH-PRF. The last equation holds in Case 2 as $F(0, j) = 0$, following the required *special zero-key* property.

Security. We prove that any single SFSS key k_b and a polynomially many ciphertexts reveal no more information than allowed. Intuitively, the security of FSS for the predicate function family ensures that $\mathsf{k}_b^{\mathsf{fss}}$ is pseudorandom, thus hiding the function within its family, which implies that the embedded KH-PRF key k_{kh} is computationally hidden. Then the security of KH-PRF F ensures that $F(\mathsf{k}_{\mathsf{kh}}, j)$ is pseudorandom. Consequently, the distribution of the ciphertext $c_j = m_j + F(\mathsf{k}_{\mathsf{kh}}, j)$ is computationally indistinguishable from uniform distribution over $\mathbb{G}_{\mathsf{out}}$. Thus, the tuple $(\mathsf{k}_b, c_1, c_2, \ldots, c_q)$ is computationally indistinguishable from a uniform distribution over the key and ciphertext spaces. Theorem 2 formalizes security, with a proof in Appendix §D.2.

Theorem 2. *Given an FSS scheme Π_{FSS} for a predicate function family $\mathcal{F}$ with domain $\mathbb{G}_{\mathsf{in}}$ and range $\mathbb{G}_{\mathsf{out}}$, a B-almost KH-PRF $F : \mathbb{K}_{\mathsf{kh}} \times \mathbb{G}_{\mathsf{ctr}} \to \mathbb{G}_{\mathsf{out}}$ with the special zero-key property, then the construction in Fig. 4 is an adaptively secure SFSS scheme of B-bounded correctness with domain $\mathbb{G}_{\mathsf{in}}$ and range $\mathbb{G}_{\mathsf{out}}$.*

Error reduction & Removal. The shared output from the KH-PRF-based SFSS contains a bounded error. Looking ahead, errors will accumulate proportionally with the number of data streams when SFSS is used for aggregation. While the errors are acceptable for applications where approximate results are sufficient, we propose a "scale-then-truncate" method inspired by [47], reducing the error to at most one bit. We can further use a simple two-party protocol to completely remove the one-bit error. Refer to Appendix §A.2 for details.

4.3.3 A Concrete Construction: From MDPF to SMDPF

To understand the compiler from Fig. 4 concretely, we show how to compile MDPF to streaming MDPF (SMDPF).

Step 1: Decompose a t-point function to t single-point functions. For any t-point function $f_{A,B}(x) : \mathbb{G}_{\mathsf{in}} \to \mathbb{G}_{\mathsf{out}}$ with $A = \{\alpha_1, \cdots, \alpha_t\} \in \mathbb{G}_{\mathsf{in}}^t$ and $B = \{1, \cdots, 1\} \in \mathbb{G}_{\mathsf{out}}^t$, we can always decompose $f_{A,B}$ as t single-point functions: $f_{A,B} = \sum_{i \in [t]} f_{\alpha_i,1}^{(i)}$, where $f_{\alpha_i,1}^{(i)}(x)$ outputs 1 only at $x = \alpha_i$ and 0 otherwise.

Step 2: Couple each single-point function with a KH-PRF secret key. Following our KH-PRF-based compiler, we extend the range of the i-th single-point function and define a new argumented single-point function:

$$f_{\alpha_i,(1,\mathsf{k}_{\mathsf{kh}})}^{(i)}(x) = \begin{cases} (1, \mathsf{k}_{\mathsf{kh}}), x = \alpha_i, \\ (0, 0), \quad x \neq \alpha_i. \end{cases}$$

Let $h = \sum_{i \in [t]} f_{\alpha_i,(1,\mathsf{k}_{\mathsf{kh}})}^{(i)}$, it is not hard to see:

$$h(x) = \sum_{i \in [t]} f_{\alpha_i,(1,\mathsf{k}_{\mathsf{kh}})}^{(i)}(x) = \begin{cases} (1, \mathsf{k}_{\mathsf{kh}}), x \in A, \\ (0, 0), \quad x \notin A. \end{cases}$$

Step 3: Define a DPF for each argumented single-point function. The remaining is simply designing a DPF for each argumented single-point function $f_{\alpha_i,(1,\mathsf{k}_{\mathsf{kh}})}^{(i)}$. These t DPFs sum together to form an FSS for the argumented t-point function h. Then, following the compiler from Fig. 4, we directly obtain a KH-PRF-based SFSS for MDPF.

We note that the above strategy can be applied to compile recent MDPF constructions [17] with improved efficiency to their streaming versions.

Cost analysis. We analyze the overhead from SMDPF compared with MDPF. Using the DPF construction from [19], encoding the KH-PRF key k_{kh} requires one more correction word of size $|\mathbb{K}_{\mathsf{kh}}|$ for each DPF key. Let $\mathsf{Keysize}(\Pi)$ be the keysize of a scheme Π, then we have $\mathsf{Keysize}(\Pi_{\mathsf{SMDPF}}) = \mathsf{Keysize}(\Pi_{\mathsf{MDPF}}) + t \cdot |\mathbb{K}_{\mathsf{kh}}|$. As for evaluation, SMDPF additionally conducts the KH-PRF evaluation that requires inner-product computation between vectors in the LWR-based KH-PRF. The KH-PRF key contributes to the main overhead in key size, and the KH-PRF evaluation contributes additional computational overhead. Nevertheless, KH-PRF-based SFSS, including SMDPF, is still promising for certain applications as we will show in §5.1.2 and §6.

4.4 SFSS Extensions

SFSS for generic functions. Based on SFSS for predicate functions, we further propose SFSS for generic functions. The high-level idea is to decompose each function output as a binary vector and then apply our SFSS technique for predicate functions. We move the details to Appendix §A.4.

Stateless SFSS. All prior SFSS constructions are state-based as they require the client to maintain a non-repeatable counter. We can slightly modify the state-based construction to be stateless by randomly sampling a fresh random value from $\mathbb{G}_{\mathsf{ctr}}$ each time. This requries $|\mathbb{G}_{\mathsf{ctr}}| \geq 2^{2\lambda}$ to ensure collusion probability of $2^{-\lambda}$ under the birthday bound.

5 SFSS-based Applications

5.1 Policy-hiding Aggregation

Policy-hiding aggregation [25] allows clients to attach policy functions to their data streams, restricting the condition under which the streaming message can be used for aggregation. It aims to protect both the data stream and client-defined policy (*i.e.*, metadata privacy). Fig. 5 defines the ideal functionality with commands SETUP, SEND, and AGGREGATE. In SETUP, a client registers its policy function. In SEND, client i submits its j-th streaming message $m_j^{(i)}$. AGGREGATE computes the result as $\sum_{i \in [N_c]} f^{(i)}(att) \cdot m_j^{(i)}$, where $f^{(i)}$ is the client's policy function and *att* denotes the attribute for an aggregation task. The functionality does not reveal stream messages, policy, or whether a stream is involved in an aggregation task.

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

Parameters: N_c denotes the number of clients; A policy function family $\mathcal{F}$; An attribute list $\mathcal{A}$; A dictionary Val storing policies; A dictionary Msg storing messages.

Functionality:

- SETUP: on input $(\mathsf{setup}, i, f^{(i)} \in \mathcal{F})$ from client i, store $\mathsf{Val}[i] \leftarrow f^{(i)}$.
- SEND: on input streaming message $(\mathsf{send}, i, j, m_j^{(i)})$ from client i in round j, store $\mathsf{Msg}[i,j] \leftarrow m_j^{(i)}$.
- AGGREGATE: on receiving an aggregation task (agg, att, j) from an aggregation receiver, broadcast att to all parties, return $\sum_{i\in[N_c]} f^{(i)}(att) \cdot m_j^{(i)}$.

Figure 5: Ideal policy-hiding aggregation functionality

Parameters: N_c denotes the number of clients; A function family $\mathcal{F}$ with domain $\mathbb{G}_{\mathsf{in}}$ and range $\mathbb{G}_{\mathsf{out}}$; A SFSS scheme Π_{SFSS} for $\mathcal{F}$.

```
Setup(1^λ, f̂^(i); ⊥; ⊥):
  1: The client generates SFSS keys:
       $(\mathsf{st}_f^{(i)}, \mathsf{k}_\mathsf{e}^{(i)}, \mathsf{k}_0^{(i)}, \mathsf{k}_1^{(i)}) \leftarrow \Pi_{\mathsf{SFSS}}.\mathsf{Gen}(1^\lambda, \hat{f}^{(i)})$
  2: Client $i$ stores $\mathsf{st}^{(i)}$ and $\mathsf{k}_\mathsf{e}^{(i)}$ and sends $\mathsf{k}_0^{(i)}$ and $\mathsf{k}_1^{(i)}$ to
     the server 0 and 1, respectively. Server $b$ stores $\mathsf{k}_b^{(i)}$.
Send((st_f^(i), k_e^(i), m_j^(i)); ⊥; ⊥):                    ▷ j-th message
  1: Client $i$ runs
       $(\mathsf{st}_f^{(i)}, c_j^{(i)}) \leftarrow \Pi_{\mathsf{SFSS}}.\mathsf{Enc}(\mathsf{st}_f^{(i)}, \mathsf{k}_\mathsf{e}^{(i)}, m_j^{(i)})$
     and sends $c_j^{(i)}$ to servers.
  2: Both servers receive and store $c_j^{(i)}$.
Aggregate(b, j, x, {c_j^(i)}_{i∈[N_c]}):                      ▷ j-th message
  1: Server $b$ sets $[\![s]\!]_b \leftarrow \sum_i \Pi_{\mathsf{SFSS}}.\mathsf{Eval}(b, \mathsf{k}_b^{(i)}, x, c_j^{(i)})$.
  2: Return $[\![s]\!]_b$.
```

Figure 6: SFSS-based policy-hiding aggregation

5.1.1 Existing Approaches

Two possible approaches realize policy-hiding aggregation.

Non-streaming FSS-based solution. We can apply non-streaming FSS as outlined in §2.1, by defining a weighted function with messages encoded as the policy function output, and sharing the function via a pair of FSS keys. However, the clients must send new FSS keys for each message, even though the policy remains unchanged. Moreover, servers must store all FSS keys for future aggregation. This results in high costs in communication and server-side storage.

Vizard's solution. As discussed in §4.1, Vizard still requires inter-server communication during the message sending phase and the aggregation phase. To avoid linear communication in the aggregation phase, Vizard reuses the same key for masking messages from different users, which introduces a security flaw. Even the flawed Vizard requires $\omega(1)$ communication complexity in the aggregation phase; Vizard only achieves constant inter-server communication when *all* data streams contain the same number of messages.

5.1.2 SFSS-powered Policy-hiding Aggregation

SFSS provides a simple and modular design for policy-hiding aggregation, as shown in Fig. 6. When SFSS is configured by SDPF, our solution realizes the same functionality that Vizard failed to realize securely, meanwhile eliminating server-to-server communication in the sending and aggregation phases.

Table 1 compares the communication complexity of three different approaches. For single-point functions, we additionally compare with Vizard. For DPF-related `phAgg`, Vizard incurs linear inter-server communication during the sending phase and $\omega(1)$ communication during aggregation, while our construction eliminates inter-server interaction. Compared to the DPF-based baseline, our solution significantly reduces the client-to-server communication in the sending phase (and hence the server-side storage). Notably, the total communication in the SFSS setup phase is independent of the number of messages per client, and each client only sets up one pair of SDPF keys for its data stream. As we will show later in §6, this reduces client-to-server communication and server-side storage significantly. A similar trend goes for (S)MDPF-based and (S)DCF-based phAgg. Additionally, the key for KH-PRF-based SFSS is larger than its non-streaming FSS key, as the KH-PRF-based SFSS key must additionally encode the KH-

Table 1: **Communication Comparison of policy-hiding aggregation schemes**. N_c denotes the number of data contributors; w denotes the number of messages from each contributor; $\mathbb{G}_{\mathsf{in}}$ denotes the (S)FSS domain. For KH-PRF, we use LWR-based $F_{\mathsf{LWR}}(\boldsymbol{k}, x) = \lfloor \langle H_1(x), \boldsymbol{k} \rangle \rfloor_p$ with $H_1 : \{0,1\}^* \to \mathbb{Z}_q^d$. For SMDPF, we use the concrete construction from §4.3.3. C$\to$S denotes the total client-to-server communication, and S$\leftrightarrow$S denotes the total inter-server communication. Highlighted cells denote the comparable better efficiency.

	Setup		**Send**		**Aggregate**	
	C$\to$S	S$\leftrightarrow$S	C$\to$S	S$\leftrightarrow$S	C$\to$S	S$\leftrightarrow$S
DPF-based [18]	No	No	$O(N_c \cdot w \cdot \lambda \cdot \log\lvert\mathbb{G}_{\mathsf{in}}\rvert)$	No	No	No
Vizard [25]	$O(N_c \cdot \lambda \cdot \log\lvert\mathbb{G}_{\mathsf{in}}\rvert)$	No	$O(N_c \cdot w)$	$O(N_c \cdot w)$	No	$\omega(1)$
SDPF-based	$O(N_c \cdot \lambda \cdot \log\lvert\mathbb{G}_{\mathsf{in}}\rvert)$	No	$O(N_c \cdot w)$	No	No	No
MDPF-based [18]	No	No	$O(t \cdot N_c \cdot w \cdot \lambda \cdot \log\lvert\mathbb{G}_{\mathsf{in}}\rvert)$	No	No	No
SMDPF-based	$O(t \cdot N_c \cdot (\lambda \cdot \log\lvert\mathbb{G}_{\mathsf{in}}\rvert + d\log q))$	No	$O(N_c \cdot w)$	No	No	No
DCF-based [18]	No	No	$O(N_c \cdot w \cdot \lambda \cdot \log\lvert\mathbb{G}_{\mathsf{in}}\rvert)$	No	No	No
SDCF-based	$O(N_c \cdot (\lambda + d\log q) \cdot \log\lvert\mathbb{G}_{\mathsf{in}}\rvert)$	No	$O(N_c \cdot w)$	No	No	No

PRF key when necessary. For instance, an SMDPF key additionally encodes t KH-PRF keys following the instantiation from §4.3.3, and SDCF additionally encodes a KH-PRF key vector for each bit of input following the DCF construction from [18]. Nevertheless, each data stream only requires one single SFSS key, and each client sends a streaming ciphertext of constant size in each round, which saves significant communication in the message-sending phase than the FSS-based approach. We will report concrete comparison in §6.

Security. Fig. 6 securely realizes $\mathcal{F}_{\mathsf{phAgg}}$. Correctness follows from SFSS, which ensures that any single SFSS key and streaming ciphertexts reveal no more information.

5.2 More SFSS-powered Applications

For attribute-hiding aggregation and window-based aggregation, please refer to Appendix §C.1 and §C.2, respectively. Here, we discuss conditional transciphering (CT) from SFSS, which may facilitate broader applications beyond aggregation.

Conditional transciphering. We can view SFSS as a CT primitive: an SFSS streaming ciphertext can be non-interactively converted to an additive sharing of its encrypted message only if certain conditions are met; otherwise, the servers share 0. CT can be applied beyond aggregation. Consider the following scenario: a lightweight client device (*e.g.*, a smartwatch) encrypts its data and uploads the ciphertexts to a storage server. At a later time, two computing servers want to perform a secure computation task over the stored ciphertexts. Before proceeding, they first transcipher the ciphertexts into additive shares suitable for subsequent secure processing. CT allows the transformation to occur only when predefined conditions are met, while hiding the filtering policy or attributes specified by the device. Also note that the transmission of SFSS ciphertexts does not require an additional secret channel, as the ciphertexts are pseudorandom by themselves. This is particularly appealing for applications where encrypted channels are hard or expensive to set up. This demonstrates that CT is not simply a tool for aggregation but a primitive with potential applications in a broad sense.

6 Performance Evaluation

This section reports the performance of SFSS and related policy-hiding aggregation. For attribute-hiding aggregation, please refer to Appendix C.1.

6.1 Implementation

We implement SFSS using C++ and open-source the code at `https://zenodo.org/records/17910080`. We conduct experiments on a PC equipped with Intel(R) Core(TM) i7-9750H CPU @ 2.6 GHz and 64 GB RAM, running Ubuntu 20.04. We simulate a local-area network (LAN, RTT: 0.2 ms, 10 Gbps) and a wide-area network (WAN, RTT: 20 ms, 500 Mbps) via Linux tc command. We set $\lambda = 128$.

6.2 SFSS Basic Primitives

Table 2 and Table 3 report the performance of SFSS compared to non-streaming FSS. We first compare SDPF with DPF. For predicate functions, we compare SDCF and DCF, with the LWR parameter setting in Table 6, Appendix §B.1.

Table 2: **Performance comparison of generating/evaluating 1K instances of DPF, SDPF, DCF, and SDCF.** We set $\mathbb{G}_{\mathsf{out}} = \mathbb{Z}_{2^{32}}$. $|\mathtt{CTX}|$ denotes the ciphertext length. Key size, encryption time, and evaluation times are summed over 1K invocations. Note that key size and (Gen/Enc/Eval) running time are summed over 1K instances.

Scheme	Domain $\mathbb{G}_{\mathsf{in}}$	Key Size (KB)	Gen (ms)	Enc (ms)	$\lvert\mathtt{CTX}\rvert$ (B)	Eval (ms)
DPF	$\mathbb{Z}_{2^{16}}$	309	25.0	-	4	0.5
	$\mathbb{Z}_{2^{32}}$	597	25.8	-	4	0.8
	$\mathbb{Z}_{2^{64}}$	1,173	27.5	-	4	1.7
SDPF	$\mathbb{Z}_{2^{16}}$	305	25.5	$\sim$0.1	4	0.5
	$\mathbb{Z}_{2^{32}}$	593	27.2	$\sim$0.1	4	0.9
	$\mathbb{Z}_{2^{64}}$	1,169	27.9	$\sim$0.1	4	1.8
DCF	$\mathbb{Z}_{2^{16}}$	369	27.6	-	4	0.9
	$\mathbb{Z}_{2^{32}}$	721	26.6	-	4	2.1
	$\mathbb{Z}_{2^{64}}$	1,425	27.7	-	4	3.0
SDCF	$\mathbb{Z}_{2^{16}}$	82,617	423.2	$\sim$7	4	162.9
	$\mathbb{Z}_{2^{32}}$	165,209	821.9	$\sim$7	4	328.4
	$\mathbb{Z}_{2^{64}}$	330,393	1,626.6	$\sim$7	4	660.3

Table 3: **Performance comparison between DPF, SDPF, DCF, and SDCF with $\mathbb{G}_{\mathsf{in}} = \mathbb{Z}_{2^{64}}$ and different configurations of $\mathbb{G}_{\mathsf{out}} \in \{\mathbb{Z}_{2^{20}}, \mathbb{Z}_{2^{40}}, \mathbb{Z}_{2^{60}}\}$.** Key size and (Gen/Enc/Eval) running time are summed over 1K invocations.

$\mathbb{G}_{\mathsf{out}}$	Scheme	Key Size (KB)	Gen (ms)	Enc (ms)	$\lvert\mathtt{CTX}\rvert$ (B)	Eval (ms)
$\mathbb{Z}_{2^{20}}$	DPF	1173	19.04	-	4	2.10
	SDPF	1,169	17.46	0.12	4	2.13
	DCF	1,425	21.70	-	4	3.86
	SDCF	206,745	1,045	4.21	4	411.98
$\mathbb{Z}_{2^{40}}$	DPF	1,177	17.70	-	8	2.09
	SDPF	1,169	18.00	0.13	8	2.13
	DCF	1,681	20.74	-	8	3.15
	SDCF	565,401	2,644.68	12.43	8	1,072.85
$\mathbb{Z}_{2^{60}}$	DPF	1,177	17.63	-	8	2.07
	SDPF	1,169	17.63	0.13	8	2.14
	DCF	1,681	17.18	-	8	3.37
	SDCF	1,210,521	7,600.13	28.77	8	2,455.08

Key and ciphertext size. Key and ciphertext size directly affect the communication and storage consumption in SFSS-based applications. All schemes exhibit roughly linear scaling in key size with respect to the bit size of $\mathbb{G}_{\mathsf{in}}$. DPF and SDPF remain lightweight and scale efficiently, whereas DCF and SDCF show significantly steeper growth. This disparity arises from the increased structural complexity of SDCF constructions. We use the DCF from [18] in our SDCF implementation. The DCF from [18] follows a tree structure during its key generation and evaluation, where each layer of the tree produces a correction word. When implementing this DCF for SDCF, we must embed a KH-PRF key at every layer of the tree using a correction word of size $|\mathbb{K}_{\text{kh}}|$, requiring $\log|\mathbb{G}_{\text{in}}|$ correction words in total. Since the LWR-based KH-PRF key is concretely large, the SDCF key is dominated by the correction words for encoding these large key vectors. For instance, with $\mathbb{G}_{\text{in}} = \mathbb{Z}_{2^{64}}$ and $\mathbb{G}_{\text{out}} = \mathbb{Z}_{2^{32}}$, SDCF requires encoding 64 correction words, each for the KH-PRF key $\boldsymbol{k} \in \mathbb{Z}_{2^{127}}^{320}$, which requires 320 KB and counts 97% of the total SDCF key size.

We note that SFSS achieves optimal constant ciphertext size. This property is particularly beneficial in streaming settings, where SFSS keys are not uploaded frequently, but streaming ciphertexts are transmitted continuously over time.

Running time. SDPF introduces negligible overhead compared to DPF. Across all different settings of $\mathbb{G}_{\text{in}}$, SDPF introduces only $\sim$0.1 ms per encrypted message while maintaining similar key and generation/evaluation time. This confirms that SDPF is highly efficient.

SDCF is a representative SFSS construction for predicate functions. SDCF incurs higher overhead compared to DCF for key generation and evaluation, due to the use of LWR-based KH-PRF. The SDCF key generation and evaluation times increase by more than two orders of magnitude compared to DCF. For example, when $\mathbb{G}_{\text{in}} = \mathbb{Z}_{2^{64}}$, SDCF requires 1.6 s for generating 1K SDCF keys, and 660 ms for 1K invocations of SDCF evaluation. In contrast, DCF only requires 27.7 ms and 3 ms, respectively.

There are two main reasons for the efficiency gap. First, SDCF embeds the LWR-based KH-PRF key vector at every layer of the tree-based DCF. As a result, key generation and evaluation must repeatedly handle large vectors: expanding seeds, copying vectors in memory, and computing over them. These operations dominate the running time. In fact, they account for about 98% of the total SDCF cost in our measurements for $\mathbb{G}_{\text{in}} = \mathbb{Z}_{2^{64}}$ and $\mathbb{G}_{\text{out}} = \mathbb{Z}_{2^{30}}$. Second, SDPF benefits from hardware-accelerated PRF and PRG implementations in EMP-tool [59]. Our KH-PRF implementation currently does not support hardware acceleration. This lack of acceleration further widens the efficiency gap. We believe that adding hardware support for KH-PRF would narrow the efficiency gap, which is a promising direction for future engineering work.

Even though our implementation is not deeply optimized, SDCF is still practical for real-world applications: each SDCF requires $\sim$330 KB of key size, $\sim$1.7 ms for key generation, $\sim$0.007 ms for streaming encryption, and $\sim$0.6 ms for evaluation for $\mathbb{G}_{\text{in}} = \mathbb{Z}_{2^{64}}$ and $\mathbb{G}_{\text{out}} = \mathbb{Z}_{2^{32}}$. We stress that SDCF is not designed to outperform DCF in a single-instance setting. SDCF, like other SFSS schemes, targets the streaming setting. Its advantages appear when evaluating many messages over long-lived keys. SFSS properties such as optimal ciphertext size and efficient support for window-based aggregation make SDCF-based applications significantly more attractive than DCF-based counterparts, as we will show in §6.3.2.

6.3 SFSS-based Policy-Hiding Aggregation

6.3.1 Equality-based Policy-hiding Aggregation

We can directly rely on SDPF to implement policy-hiding aggregations to support equality-based policy functions, as considered in Vizard [25]. We compare our SDPF-based policy-hiding aggregation with Vizard, in the scope of window-based aggregation that aggregates over a window range $[l, r]$ for each data stream (c.f., Appenidx C.2). We conduct experiments with different window size and #contributors. Table 4 presents the communication and computation cost of SFSS-based `phAgg`, DPF-based non-streaming construction, and Vizard, with $\mathbb{G}_{\text{in}} = \mathbb{Z}_{2^{32}}$, under LAN and WAN settings. Highlighted cells mean our solution achieves better efficiency.

Table 4: **Comparison of communication (MB) and running time (ms) for policy-hiding aggregation**. Note that Vizard requires secret-shared multiplication during aggregation; thus, we report its running time in the LAN (*i.e.*, Vizard (L)) and WAN (*i.e.*, Vizard (W)) settings, respectively.

Win. Size	Scheme	Number of Data Contributors					
		5K		10K		50K	
		Com.	Time	Com.	Time	Com.	Time
10	DPF-based	0	78.1	0	155.9	0	786.3
	Vizard (L)	≥ 16	15.3	≥ 16	26.9	≥ 16	97.4
	Vizard (W)	≥ 16	343.0	≥ 16	429.5	≥ 16	199.2
	Our	0	13.0	0	26.0	0	80.9
50	DPF-based	0	389.5	0	781.9	0	3866.4
	Vizard (L)	≥ 16	27.6	≥ 16	49.8	≥ 16	168.0
	Vizard (W)	≥ 16	128.9	≥ 16	152.8	≥ 16	279.3
	Our	0	13.5	0	27.1	0	87.2
100	DPF-based	0	797.4	0	1539.8	0	7755.1
	Vizard (L)	≥ 16	39.9	≥ 16	67.5	≥ 16	255.5
	Vizard (W)	≥ 16	140.5	≥ 16	172.3	≥ 16	361.3
	Our	0	14.5	0	27.9	0	90.4

Running time. The DPF-based policy-hiding aggregation does not support efficient window-based aggregation, as the server must perform DPF evaluation over each DPF key, which incurs DPF evaluation complexity that is linear in the product #contributors $\times$ #windows. Table 4 shows that the evaluation time is inefficient for a large number of contributors and a large window size, around 90$\times$ that of SDPF-based construction. Both Vizard and our SDPF-based constructions support efficient window-based aggregation, where the servers only evaluate the DPF key once for each data stream. SDPF-based policy-hiding aggregation is more efficient than Vizard [25]. For instance, when the window size is 100 and the number of contributors is 10K, our SDPF-based construction only requires 14.5 ms both in LAN and WAN settings, while Vizard [25] requires 39.9 ms in LAN and 140.5 ms in WAN, *i.e.*, 2.7× and 9× slower, respectively.

Communication. Both DPF-based and SDPF-based policy-hiding aggregation do not require inter-server communication. Vizard [25] requires server-side secret-shared multiplication during aggregation. For $\mathbb{G}_{\mathsf{in}} = \mathbb{Z}_{2^{32}}$, Vizard requires at least 16 bytes of communication if all contributors have the same window size. However, when different contributors have different window sizes, Vizard will incur the worst communication overhead that is linear in the number of contributors; this is why we have ≥ 16 in Table 4, where 16 is the best efficiency we can hope for Vizard. SDPF-based policy-hiding aggregation does not require any inter-server communication.

Storage. Each server in our SDPF-based construction stores only one DPF key pair per stream, independent of the number of ciphertexts. In contrast, DPF-based baseline stores one key pair per message, leading to storage overhead linear in the total number of messages. Experimental results confirm this difference: DPF-based aggregation incurs $10-80\times$ higher storage than SDPF-based for $\mathbb{G}_{\mathsf{out}} = \mathbb{Z}_{2^{32}}$, and $10-40\times$ for $\mathbb{G}_{\mathsf{out}} = \mathbb{Z}_{2^{128}}$. Refer to Table 7 in Appendix §B.2 for details.

6.3.2 Solution from KH-PRF-based SFSS

Table 5 compares efficiency between DCF-based and SDCF-based policy-hiding aggregation. The main communication of the SDCF-based solution lies in transferring keys, with extremely small ciphertext-transfer communication, and the SDCF-based solution outperforms the DCF-based approach over communication and running time for large windows. This is because 1) SFSS enjoys an optimal streaming ciphertext size, which reduces communication significantly than the DCF-based approach. 2) SDCF-based only evaluates twice per data stream via window-based optimization, while the DCF-based approach must evaluate for each time slot.

Table 5: **Comparison of communication (MB) and running time (ms) for (S)DCF-based policy-hiding aggregation**. We set $\mathbb{G}_{\mathsf{in}} = \mathbb{Z}_{2^{40}}$ and $\mathbb{G}_{\mathsf{out}} = \mathbb{Z}_{2^{30}}$. Communication for SDCF-based contains offline key transfer and online ciphertext transfer. For example, (197, 7.6) denotes 197 MB offline communication and 7.6 MB online communication.

Win. Size	**Scheme**	**Number of Data Contributors**					
		10		100		1000	
		Com.	Time	Com.	Time	Com.	Time
10	DPF-based	<0.1	<1	0.1	1.3	1.3	12.8
	SDCF-based	(2,<0.001)	<1	(2,<0.01)	7.1	(197,<0.1)	70.3
100	DPF-based	<0.1	1.3	1.3	12.8	100.8	129.0
	SDCF-based	(2,<0.01)	<1	(20,<0.1)	7.3	(197,0.8)	73.9
1000	DPF-based	1.3	12.8	12.9	125.8	1008.0	1257.2
	SDCF-based	(2, <0.1)	<1	(20, 0.8)	8.0	(197, 7.6)	84.2

7 Related Work

Function secret sharing. Gilboa and Ishai introduces the notion of distributed point function (DPF) [39]. Boyel, Gilboa, and Ishai [18] extended DPF and formalized FSS, in which they also proposed a two-party DPF and distributed comparison function (DCF) with key size $O(\lambda \log n)$. Following [18], many optimizations [12, 19, 40] and extensions [17, 21, 22] are proposed on FSS itself. FSS has found broad applications in secure computation protocols and applications, including secure RAM and database applications [31, 32, 56, 58], pseudorandom correlation generator (PCG) [13, 14, 15, 16], mix-mode secure computation [12, 20], anonymous communication [30, 35], private set intersection [23, 37], and privacy-preserving machine learning [41, 42, 45, 53, 57].

Secure aggregation. Secure aggregation aggregates private data from different parties, which can be widely used in many secure computation protocols such as e-voting and federated learning. Secure aggregation can be constructed using generic secure multi-party computation [7, 49], dining cryptographers (DC) networks [27], pairwise additive masking [2, 8, 34], homomorphic or functional encryption [26, 28, 55]. Recently, FSS-based secure aggregation systems [10, 25, 29, 50] in the two-server distributed-trust model have gained attention due to their efficiency and simplicity, including Prio [29] for general statistics collection, Poplur [10] for heavy-hitter collection, Mastic [50] for attribute-based aggregation, and Vizard for policy-hiding aggregation [25]. In particular, FSS-based aggregation has been deployed by Mozilla [43] for browser data collection, Divvi Up [1] for private website analytics and surveys, and Apple and Google [4] for measuring the effectiveness of Covid-19 exposure notification. IETF is currently standardizing the FSS-based private aggregation [38]. Vizard is the only existing FSS-based work for streaming aggregation. As mentioned, Vizard cannot achieve its claimed security and efficiency properties simultaneously.

8 Conclusion

This paper introduces streaming function secret sharing, a new primitive tailored for secure computation over streaming data. We formalize the SFSS model, present concrete constructions, and demonstrate SFSS applications. Our SFSS-based solutions improve efficiency, strengthen security, and broaden functionality compared to existing approaches.

Limitations & future work. Our work leaves several interesting directions. First, a promising direction is *verifiable SFSS*, enabling verifiable streaming aggregation that resists malicious contributors. While zero-knowledge proofs for secret-shared data based on fully linear PCP [9, 29] offer a generic way to enforce well-formed SFSS keys, designing concretely efficient mechanisms that exploit SFSS structural properties remains an open challenge. Second, our current SFSS for predicate functions relies on LWR-based KH-PRF with a special zero-key, which contributes the main overhead for the resulting construction. It remains to design SFSS beyond single-point functions based on more concretely efficient primitives.

Acknowledgment

We thank the anonymous reviewers and shepherd for their insightful suggestions and comments. This work is supported by the National Research Foundation, Singapore, Cyber Security Agency of Singapore under its National Cybersecurity R&D Programme and CyberSG R&D Cyber Research Programme Office, and National Natural Science Foundation of China under Grant 62302118. Any opinions, findings and conclusions or recommendations expressed in these materials are those of the author(s) and do not reflect the views of National Research Foundation, Singapore, Cyber Security Agency of Singapore as well as CyberSG R&D Programme Office, Singapore.

Ethical Considerations

Stakeholder Analysis. Stakeholders include data contributors whose data is being aggregated, service providers who perform the aggregation, aggregation receivers who benefit from aggregated statistics, standardization groups such as IETF PPM WG, and privacy and security researchers. Data contributors benefit from improved privacy when their data is aggregated. Service providers can offer privacy-preserving services, enhancing user trust and compliance with privacy regulations. aggregation receivers benefit from aggregated statistics without compromising individual privacy. Standardization groups can incorporate SFSS into guidelines for new extensions. Researchers may get inspired to perform future research for new primitives and applications.

Ethical Principles. We follow the four guiding principles outlined in the Menlo Report. We uphold *respect for persons* by ensuring individual data remains private during the aggregation. We promote *beneficence* by enabling service providers to derive useful insights without compromising individual privacy. We support *justice* by designing provably secure and efficient protocols, minimizing the resources required for real-world deployment. We adhere to *respect for law and public interest* by conducting all experiments on synthetic data.

Security Assumption, Possible Mismatch, and Mitigation. SFSS-based applications assume non-colluded servers. While this is a common assumption for FSS-based aggregation systems, there exists a possible extreme case where both servers are corrupted and/or colluded, and security collapses in this case. Therefore, any future applications based on our work must validate whether this assumption holds. Some possible seconarios: ① Two servers are separately run by two organizations without intentions to actively collude with each other due to conflicts of interest and/or regulations. ② One of the servers is run by an independent auditing organization, such as privacy regulation agencies. Protecting individual data is the ultimate goal of such agencies. ③ Design a game-theoretic incentive mechanism to discourage collusion among servers.

SFSS-based applications currently support semi-honest security; achieving a certain level of malicious security is future work. While the difference between semi-honest and malicious security is relatively well-known in the security and cryptography community, here we explicitly introduce the differences for the benefit of broader audiences, to prevent possible implementers from using SFSS with unmatched security assumptions. In particular, in a semi-honest secure protocol, the computing parties are assumed to follow the protocol specification, but they may try to learn additional information from the protocol execution. While for malicious security, corrupted parties may perform arbitrary deviations to undermine privacy and/or integrity.

Any applications attempting to achieve a certain level of malicious security when using SFSS should deploy additional mechanisms to check the well-formedness of the inputs from the users and/or correct execution of the protocol. In particular: ① The data contributors may be malicious, and they may submit malformed SFSS keys and/or ciphertexts to the servers. To mitigate this issue, the servers can ask the users to provide zero-knowledge proofs to prove the well-formedness of their inputs. ② The aggregation servers may be malicious, and they may deviate from the protocol specification to learn additional information about the users' inputs. To mitigate this issue, the servers can provide zero-knowledge proofs to prove that they have followed the protocol specification. For example, in secure aggregation, the servers can provide proofs to prove that they have correctly evaluated the SFSS ciphertexts and computed the final aggregation result.

Wellbeing for Team Members. We maintain open communication among team members, encourage sharing ideas and concerns, and provide support for any challenges faced during the research. While SFSS provides the first study of SFSS on security, efficiency, and functionality, with a promising connection to existing standardization, the authors may suffer from possible rejection, especially facing biased and/or irresponsible reviews, and misunderstandings. We support team members to defend authors' rights when applicable, maintain a healthy work-life balance, and seek support when needed.

Decision to Publish. We proceed with publication for the following reasons: ① *Advancing knowledge*: Publishing our research contributes knowledge for privacy-preserving technologies, providing valuable insights and techniques that benefit the broader research community and practitioners working on privacy and security. ② *Positive impact*: Our work has the potential to impact the design and implementation of secure aggregation systems and standardization efforts. By sharing our findings, we enable others to build upon our work and develop more secure and efficient solutions. ③ *Ethical considerations*: We have adhered to ethical research practices. We have also provided security considerations to guide future implementers. Overall, we believe that the benefits of publishing our research outweigh potential risks, and we are committed to promoting ethical standards in the dissemination and application of our work.

Open Science

The SFSS implementation we used for benchmarking can be found at `https://zenodo.org/records/17910080`. In the following, we provide an overview of our implementation and how to obtain the performance data in this paper.

Hardware requirement. SFSS currently supports 64-bit macOS and Ubuntu systems. Some cryptographic primitives, including pseudorandom generator (PRG), pseudorandom function (PRF), hash functions, require hardware acceleration (*i.e.*, SSE2 for x86_64 and ARM NEON). The hardware requirement is due to the involvement of emp-tool (`https://github.com/emp-toolkit/emp-tool`).

Software dependency. Our SFSS implementation depends on the following libraries:

- **CMake** (`https://cmake.org`). We use CMake (version >= 3.12) to build the code.
- **emp-tool** (`https://github.com/emp-toolkit/emp-tool`). We use the hardware-accelerated implementations of PRG, PRF, and hash functions from emp-tool.
- **GMP** (`https://gmplib.org`). We use `__uint128_t` to emulate modular operation over $\mathbb{Z}_{2^k}$ for $k < 128$. For $k \geq 128$, we use GMP and wrap our own big integer class `MyBigInteger` (in header file `field.h`).

Implementation overview. The project file structure is as follows:

```
SFSS/
 ├─ CMakeLists.txt
 ├─ util.h
 ├─ twokeyprp.h
 ├─ field.h
 ├─ field.cpp
 ├─ sfss.h
 ├─ sharing.h
 ├─ runner.h
 ├─ sfss_main.cpp
 ├─ client_runner_main.cpp
 ├─ server_runner_main.cpp
 └─ README.md
```

Compile & Run Please refer to `README.MD` on how to compile the code and run benchmarks.

Evaluation. Evaluating the benchmarking requires successfully compiling and generating the target executable files `sfss_main`, `client_runner_main`, and `server_runner_main`.

1) Basic primitives. The evaluation of basic primitive report efficiency comparison between DPF and SDPF, as well as DCF and SDCF. To obtain the results in Table 2 and Table 3. Simply put `bench_basic_sfss();` to the `main` function in `sfss_main.cpp`. Then open a terminal and work in the `SFSS/build` directory. Compile via `make` and run `./bin/sfss_main`. The evaluation results on key size and running time will be shown in the terminal.

2) Policy-hiding aggregation. To obtain the performance result on Table 4, do as follows:

2.1) Benchmarking Vizard. To obtain the benchmarking results for Vizard, do the following steps:

- Put `bench_vizard_main(argc, argv);` to the `main` function in `client_runnner_main.cpp`; Put `bench_vizard_main(argc, argv);` to the `main` function in `server_runnner_main.cpp`.
- Compile the project by running `make` in `SFSS/build`.
- Open three terminals. Run `./bin/client_runner_main` in terminal 1, `./bin/server_runner_main 0 12345` in terminal 2, and `./bin/server_runner_main 0 12346` in terminal 3.
- The performance results will be shown in the terminals.

2.2) Benchmarking DPF-based solution.

- Put `bench_dpf_window();` in the `main` function in `sfss_main.cpp`;
- Compile the project by running `make` in `SFSS/build`.
- Open a terminal and run `./bin/sfss_main` in a terminal.
- The performance results will be shown in the terminal.

2.3) Benchmarking SDPF-based solution. To obtain the benchmarking results for our SDPF-based policy-hiding aggregation, do the following steps:

- Put `bench_sdpf_main(argc, argv);` in the `main` function in `client_runnner_main.cpp`; Put `bench_sdpf_main(argc, argv);` in the `main` function in `server_runnner_main.cpp`.
- Compile the project by running `make` in `SFSS/build`.
- Open three terminals. Run `./bin/client_runner_main` in terminal 1, `./bin/server_runner_main 0 12345` in terminal 2, and `./bin/server_runner_main 0 12346` in terminal 3.
- The performance results will be shown in the terminals.

3) Attribute-hiding aggregation. To obtain the performance data, Do as follows:

- Put `bench_attribute_hiding();` in the `main` function in `sfss_main.cpp`.
- Compile the project by running `make` in `SFSS/build`.
- Open a terminal and run `./bin/sfss_main` in the terminal.
- The performance results will be shown in the terminal.

References

[1] About divvi up. https://divviup.org/about/#open-source. Accessed: 2025-08-09.

[2] Gergely Ács and Claude Castelluccia. I have a dream!(differentially private smart metering). In *International Workshop on Information Hiding*, pages 118–132. Springer, 2011.

[3] Martin R. Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of Learning with Errors. *Journal of Mathematical Cryptology*, 9(3):169–203, 2015.

[4] Apple and Google. Exposure notification privacy-preserving analytics (enpa) white paper. 2021.

[5] Abhishek Banerjee, Chris Peikert, and Alon Rosen. Pseudorandom functions and lattices. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 719–737. Springer, 2012.

[6] Donald Beaver. Efficient multiparty protocols using circuit randomization. In *Annual International Cryptology Conference*, pages 420–432. Springer, 1991.

[7] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Providing sound foundations for cryptography: on the work of Shafi Goldwasser and Silvio Micali*, pages 351–371. 2019.

[8] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. Practical secure aggregation for privacy-preserving machine learning. In *proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1175–1191, 2017.

[9] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Zero-knowledge proofs on secret-shared data via fully linear pcps. In *Annual international cryptology conference*, pages 67–97. Springer, 2019.

[10] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Lightweight techniques for private heavy hitters. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 762–776. IEEE, 2021.

[11] Dan Boneh, Kevin Lewi, Hart Montgomery, and Ananth Raghunathan. Key homomorphic prfs and their applications. In *Annual Cryptology Conference*, pages 410–428. Springer, 2013.

[12] Elette Boyle, Nishanth Chandran, Niv Gilboa, Divya Gupta, Yuval Ishai, Nishant Kumar, and Mayank Rathee. Function secret sharing for mixed-mode and fixed-point secure computation. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 871–900. Springer, 2021.

[13] Elette Boyle, Geoffroy Couteau, Niv Gilboa, and Yuval Ishai. Compressing vector ole. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 896–912, 2018.

[14] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, Peter Rindal, and Peter Scholl. Efficient two-round ot extension and silent non-interactive secure computation. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 291–308, 2019.

[15] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Efficient pseudorandom correlation generators: Silent ot extension and more. In *Annual International Cryptology Conference*, pages 489–518. Springer, 2019.

[16] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Efficient pseudorandom correlation generators from ring-lpn. In *Annual International Cryptology Conference*, pages 387–416. Springer, 2020.

[17] Elette Boyle, Niv Gilboa, Matan Hamilis, Yuval Ishai, and Yaxin Tu. Improved constructions for distributed multi-point functions. In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 2414–2432. IEEE, 2025.

[18] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 337–367. Springer, 2015.

[19] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing: Improvements and extensions. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 1292–1303, 2016.

[20] Elette Boyle, Niv Gilboa, and Yuval Ishai. Secure computation with preprocessing via function secret sharing. In *Theory of Cryptography: 17th International Conference, TCC 2019, Nuremberg, Germany, December 1–5, 2019, Proceedings, Part I 17*, pages 341–371. Springer, 2019.

[21] Elette Boyle, Niv Gilboa, Yuval Ishai, and Victor I Kolobov. Programmable distributed point functions. In *Annual International Cryptology Conference*, pages 121–151. Springer, 2022.

[22] Elette Boyle, Lisa Kohl, Zhe Li, and Peter Scholl. Direct fss constructions for branching programs and more from prgs with encoded-output homomorphism. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 266–298. Springer, 2024.

[23] Dung Bui, Gayathri Garimella, Peihan Miao, and Phuoc Van Long Pham. New framework for structure-aware psi from distributed function secret sharing. *Cryptology ePrint Archive*, 2025.

[24] Lukas Burkhalter, Anwar Hithnawi, Alexander Viand, Hossein Shafagh, and Sylvia Ratnasamy. {TimeCrypt}: Encrypted data stream processing at scale with cryptographic access control. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 835–850, 2020.

[25] Chengjun Cai, Yichen Zang, Cong Wang, Xiaohua Jia, and Qian Wang. Vizard: A metadata-hiding data analytic system with end-to-end policy controls. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 441–454, 2022.

[26] T-H Hubert Chan, Elaine Shi, and Dawn Song. Privacy-preserving stream aggregation with fault tolerance. In *International Conference on Financial Cryptography and Data Security*, pages 200–214. Springer, 2012.

[27] David Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of cryptology*, 1(1):65–75, 1988.

[28] Jérémy Chotard, Edouard Dufour Sans, Romain Gay, Duong Hieu Phan, and David Pointcheval. Decentralized multi-client functional encryption for inner product. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 703–732. Springer, 2018.

[29] Henry Corrigan-Gibbs and Dan Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In *14th USENIX symposium on networked systems design and implementation (NSDI 17)*, pages 259–282, 2017.

[30] Henry Corrigan-Gibbs, Dan Boneh, and David Mazières. Riposte: An anonymous messaging system handling millions of users. In *2015 IEEE Symposium on Security and Privacy*, pages 321–338. IEEE, 2015.

[31] Emma Dauterman, Mayank Rathee, Raluca Ada Popa, and Ion Stoica. Waldo: A private time-series database from function secret sharing. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 2450–2468. IEEE, 2022.

[32] Jack Doerner and Abhi Shelat. Scaling oram for secure computation. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 523–535, 2017.

[33] Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, and Frederik Vercauteren. Saber: Module-lwr based key exchange, cpa-secure encryption and cca-secure kem. In *International Conference on Cryptology in Africa*, pages 282–305. Springer, 2018.

[34] Tariq Elahi, George Danezis, and Ian Goldberg. Privex: Private collection of traffic statistics for anonymous communication networks. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1068–1079, 2014.

[35] Saba Eskandarian, Henry Corrigan-Gibbs, Matei Zaharia, and Dan Boneh. Express: Lowering the cost of metadata-hiding communication with cryptographic privacy. In *30th USENIX Security Symposium*, pages 1775–1792, 2021.

[36] Pierre-Alain Fouque, Paul Kirchner, Thomas Pornin, and Yang Yu. BAT: Small and Fast KEM over NTRU Lattices. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022:240–265, 02 2022.

[37] Gayathri Garimella, Mike Rosulek, and Jaspal Singh. Structure-aware private set intersection, with applications to fuzzy matching. In *Annual International Cryptology Conference*, pages 323–352. Springer, 2022.

[38] Tim Geoghegan, Christopher Patton, Eric Rescorla, and Christopher A Wood. Distributed aggregation protocol for privacy preserving measurement. *IETF*, 2023.

[39] Niv Gilboa and Yuval Ishai. Distributed point functions and their applications. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 640–658. Springer, 2014.

[40] Xiaojie Guo, Kang Yang, Xiao Wang, Wenhao Zhang, Xiang Xie, Jiang Zhang, and Zheli Liu. Half-tree: Halving the cost of tree expansion in cot and dpf. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 330–362. Springer, 2023.

[41] Kanav Gupta, Neha Jawalkar, Ananta Mukherjee, Nishanth Chandran, Divya Gupta, Ashish Panwar, and Rahul Sharma. Sigma: Secure gpt inference with function secret sharing. *Cryptology ePrint Archive*, 2023.

[42] Kanav Gupta, Deepak Kumaraswamy, Nishanth Chandran, and Divya Gupta. Llama: A low latency math library for secure inference. *Cryptology ePrint Archive*, 2022.

[43] Bobby Holley. Built for privacy: Partnering to deploy oblivious http and prio in firefox. https://blog.mozilla.org/en/firefox/partnership-ohttp-prio. Accessed: 2025-08-09.

[44] Zhicong Huang, Wen-jie Lu, Cheng Hong, and Jiansheng Ding. Cheetah: Lean and fast secure {Two-Party} deep neural network inference. In *31st USENIX Security Symposium*, pages 809–826, 2022.

[45] Neha Jawalkar, Kanav Gupta, Arkaprava Basu, Nishanth Chandran, Divya Gupta, and Rahul Sharma. Orca: Fss-based secure training and inference with gpus. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 597–616. IEEE, 2024.

[46] Marcel Keller, Emmanuela Orsini, and Peter Scholl. Mascot: faster malicious arithmetic secure computation with oblivious transfer. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 830–842, 2016.

[47] Tung Le, Rouzbeh Behnia, Jorge Guajardo, and Thang Hoang. {MUSES}: Efficient {Multi-User} searchable encrypted database. In *33rd USENIX Security Symposium*, pages 2581–2598, 2024.

[48] MATZOV. Report on the Security of LWE: Improved Dual Lattice Attack. 2022.

[49] Silvio Micali, Oded Goldreich, and Avi Wigderson. How to play any mental game. In *Proceedings of the Nineteenth ACM Symp. on Theory of Computing, STOC*, pages 218–229. ACM New York, 1987.

[50] Dimitris Mouris, Christopher Patton, Hannah Davis, Pratik Sarkar, and Nektarios Georgios Tsoutsos. Mastic: Private weighted heavy-hitters and attribute-based metrics. *Proceedings on Privacy Enhancing Technologies*, 2025.

[51] Moni Naor, Benny Pinkas, and Omer Reingold. Distributed pseudo-random functions and kdcs. In *International conference on the theory and applications of cryptographic techniques*, pages 327–346. Springer, 1999.

[52] Deevashwer Rathee, Mayank Rathee, Nishant Kumar, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. CrypTFlow2: Practical 2-Party Secure Inference. In *CCS*, pages 325–342, 2020.

[53] Théo Ryffel, Pierre Tholoniat, David Pointcheval, and Francis Bach. Ariann: Low-interaction privacy-preserving deep learning via function secret sharing. *arXiv preprint arXiv:2006.04593*, 2020.

[54] Kazue Sako and Joe Kilian. Secure voting using partially compatible homomorphisms. In *Annual international cryptology conference*, pages 411–424. Springer, 1994.

[55] Elaine Shi, HTH Chan, Eleanor Rieffel, Richard Chow, and Dawn Song. Privacy-preserving aggregation of time-series data. In *Annual Network & Distributed System Security Symposium (NDSS)*. Internet Society, 2011.

[56] Adithya Vadapalli, Ryan Henry, and Ian Goldberg. Duoram: A {Bandwidth-Efficient} distributed {ORAM} for 2-and 3-party computation. In *32nd USENIX Security Symposium*, pages 3907–3924, 2023.

[57] Sameer Wagh. Pika: Secure computation using function secret sharing over rings. *Proceedings on Privacy Enhancing Technologies*, 2022.

[58] Frank Wang, Catherine Yun, Shafi Goldwasser, Vinod Vaikuntanathan, and Matei Zaharia. Splinter: Practical private queries on public data. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 299–313, 2017.

[59] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit, 2016.

A Additional Constructions

A.1 SDPF with Weighted Output

Fig. 7 shows the construction for single-point functions with weighted output. The only difference is that the streaming encryption algorithm computes the streaming ciphertext as

$$c_{\mathsf{ctr}} \leftarrow m - (F([\![r]\!]_0, \mathsf{ctr}) - F(-[\![r]\!]_1, \mathsf{ctr})) \cdot \beta^{-1},$$

which requries $\mathbb{G}_{\mathsf{out}} = \mathbb{F}$ for some finte field to support the inverse β^{-1}. The correctness and security analysis are essentially the same as in §4.2.

A.2 Error Reduction & Removal

Our starting point is a simple and efficient *scale-then-truncate* approach [47] to reduce the error to at most one bit. Specifically, we modify the SFSS encryption algorithm to scale the message m as

$$m' = m \cdot 2^{\lceil \log(B+1) \rceil}.$$

where B is the upper bound on the error. To reduce the error to only 1 bit, it is sufficient to truncate each evaluation share by $\lceil \log(B+1) \rceil$ bits and then add 1 to the truncated shared secret. This approach only leaves at most 1 bit of error from

Parameters: PRF $F : \{0,1\}^\lambda \times \mathbb{G}_{\mathsf{ctr}} \to \mathbb{G}_{\mathsf{out}}$; Function family $\mathcal{F}_{\mathbb{G}_{\mathsf{in}},\mathbb{G}_{\mathsf{out}}\times\{0,1\}^\lambda}$ of point functions with domain $\mathbb{G}_{\mathsf{in}}$ and range $\mathbb{G}_{\mathsf{out}} \times \{0,1\}^\lambda$; DPF scheme Π_{DPF} for function family $\mathcal{F}_{\mathbb{G}_{\mathsf{in}},\mathbb{G}_{\mathsf{out}}\times\{0,1\}^\lambda}$; We require $\mathbb{G}_{\mathsf{out}} = \mathbb{F}$ for a finete field $\mathbb{F}$, and $|\mathbb{G}_{\mathsf{ctr}}| \geq 2^\lambda$.

```
Gen(1^λ, f̂_{α,(β,r)} ∈ F_{G_in,G_out×{0,1}^λ}):    ▷ Here $r \xleftarrow{\$} \{0,1\}^\lambda$
  1: Set $\mathsf{ctr} = 1$
  2: $(k_0^{\mathsf{dpf}}, k_1^{\mathsf{dpf}}) \leftarrow \Pi_{\mathsf{DPF}}.\mathsf{Gen}(1^\lambda, \hat{f}_{\alpha,(\beta,r)})$
  3: For $b \in \{0,1\}$, $([\![\beta]\!]_b, [\![r]\!]_b) \leftarrow \Pi_{\mathsf{DPF}}.\mathsf{Eval}(b, \mathsf{k}_b^{\mathsf{dpf}}, \alpha)$
  4: $\mathsf{st}_f \leftarrow \{\mathsf{ctr}\}$, $\mathsf{k_e} \leftarrow \{[\![r]\!]_0, [\![r]\!]_1\}$, $\mathsf{k}_0 \leftarrow \{\mathsf{k}_0^{\mathsf{dpf}}\}$, $\mathsf{k}_1 \leftarrow \{\mathsf{k}_1^{\mathsf{dpf}}\}$
  5: Return $(\mathsf{st}_f, \mathsf{k}_0, \mathsf{k}_1)$.
Enc(st_f, k_e, m):
  1: Parse $\mathsf{st}_f = \{\mathsf{ctr}\}$ and $\mathsf{k_e} = \{[\![r]\!]_0, [\![r]\!]_1\}$.
  2: $c_{\mathsf{ctr}} \leftarrow m - (F([\![r]\!]_0, \mathsf{ctr}) - F(-[\![r]\!]_1, \mathsf{ctr})) \cdot \beta^{-1}$
  3: Set $c \leftarrow (\mathsf{ctr}, c_{\mathsf{ctr}})$ and update $\mathsf{st}_f = \{\mathsf{ctr}+1\}$
  4: Return $(\mathsf{st}_f, c)$
Eval(b, k_b, x, c):
  1: Parese $c = (j, c_j)$ and $\mathsf{k}_b = \{\mathsf{k}_b^{\mathsf{dpf}}\}$
  2: $([\![e]\!]_b, [\![h]\!]_b) \leftarrow \Pi_{\mathsf{DPF}}.\mathsf{Eval}(b, \mathsf{k}_b^{\mathsf{dpf}}, x)$
  3: $[\![s]\!]_b \leftarrow [\![e]\!]_b \cdot c_j + (-1)^b \cdot F((-1)^b \cdot [\![h]\!]_b, j)$
  4: Return $[\![s]\!]_b$
```

Figure 7: The Proposed SDPF construction

the truncation error. We can extend the idea to the multiple-client case. Suppose n clients contribute SFSS keys, and the i-th client encodes its message m_i as

$$m'^{(i)} = m_i \cdot 2^{\lceil \log(nB+1) \rceil}.$$

By the definition of SFSS evaluation, the parties can compute the aggregate

$$\sum_i m'^{(i)} + \sum_i e^{(i)} = \sum_i m_i \cdot 2^{\lceil \log(nB+1) \rceil} + \sum_i e^{(i)},$$

where each $e^{(i)} \in [B]$ is the bounded error introduced by the i-th evaluation. Since $\sum_i e^{(i)} \leq n \cdot B$, it affects only the least $\lceil \log(nB+1) \rceil$ bits of $\sum_i m'^{(i)}$, thus the error does not interfere with the message bits. Each evaluator can locally truncate the least significant of $\lceil \log(nB+1) \rceil$ bits, which reduces the error to at most 1 bit (due to the truncation error).

Taking a step further, we can totally remove the truncation error via 2-party computation (2PC). Since the error affects only the least significant $\lceil \log(B+1) \rceil$ bits, the parties can engage in a 2PC faithful truncation protocol to jointly extract and correct these bits, thereby recovering the exact result. Note that this error-elimination step incurs only constant communication, regardless of the number of participating clients.

Similarly, if the results are of ℓ bits, the communication cost is of $12(\ell + \lceil \log n + \log B \rceil)$ bits in $\log_2 \ell + 1$ rounds; If the results are of $\ell - \lceil \log n + \log B \rceil$ bits, the communication is of $12\lceil \log n + \log B \rceil + \ell$ bits in $\log \lceil \log n + \log B \rceil + 1$ rounds.

Parameters: $[\![x]\!]$ with $x \in \mathbb{Z}_{2^\ell}$; truncation bits f; 2PC compare protocol $[\![\mathbf{1}\{x > y\}]\!]^B \leftarrow \mathsf{CMP}(x,y)$; Bit to Arithmetic conversion protocol $[\![x]\!] \leftarrow \mathsf{B2A}([\![x]\!]^B)$.

```
  Wrap-around Error:
  1: $S_0$ and $S_1$ call CMP protocol: $([\![w]\!]_0^B, [\![w]\!]_1^B) \leftarrow \mathsf{CMP}([\![x]\!]_0, 2^\ell - 1 - [\![x]\!]_1)$.
  2: Two servers invoke B2A protocol: $([\![w]\!]_0^f, [\![w]\!]_1^f) \leftarrow \mathsf{B2A}([\![w]\!]_0^B, [\![w]\!]_1^B)$.
  Least bit Error:
  3: $S_b$ extracts the least $f$ bits of $[\![x]\!]_b$ as $u_b$ locally.
  4: $S_0$ and $S_1$ call CMP protocol: $([\![c]\!]_0^B, [\![c]\!]_1^B) \leftarrow \mathsf{CMP}(u_0, 2^f - 1 - u_1)$.
  5: Two servers invoke B2A protocol: $([\![c]\!]_0, [\![c]\!]_1) \leftarrow \mathsf{B2A}([\![c]\!]_0^B, [\![c]\!]_1^B)$.
  6: $S_b$ sets $[\![z]\!]_b = ([\![x]\!]_b \gg f) - [\![w]\!]_b^f \cdot 2^{\ell-f} + [\![c]\!]_b$.
  7: Return $[\![z]\!]_b$.
```

Figure 8: Faithful Two-Party Truncation.

Given an ℓ-bit integer x and its arithmetic shares $([\![x]\!]_0, [\![x]\!]_1)$, Figure 8 presents the 2PC protocol for its faithful truncation following [44, 52]. First, w denotes whether the sum of two shares wraps around modulo 2^ℓ. If so, there will be a large error due to local truncation, and it should be corrected by subtracting 2^ℓ. c comes from the last-bit error that is decided by whether the sum of the least significant f bits of the two shares exceeds 2^f. Let $w = \mathbf{1}\{[\![x]\!]_0 + [\![x]\!]_1 > 2^\ell - 1\}$, u_b denotes the last f bits of $[\![x]\!]_b$ and $c = \mathbf{1}\{u_0 + u_1 > 2^f - 1\}$, we have $([\![x]\!]_0 \gg f) + ([\![x]\!]_1 \gg f) - w \cdot 2^\ell + c = (x \gg f)$. By using the VOLE style-OT, the communication complexity is $12\ell + 12f$ bits in $\log_2 \ell + 1$ rounds.

A.3 LHE-based Approach

We give a concrete "FSS+LHE" construction using the exponential ElGamal encryption [54] that supports additive homomorphism in the exponent.

In particular, the servers joinly generate a public key $h = g^{\mathsf{sk}}$, where sk is the secret key additively shared between the servers and g is the generator for a DDH-hard cyclic group $\mathbb{G}$ with group order p. Suppose we have an FSS scheme that can be used to additively share $f(x)$ over $\mathbb{Z}_p$. With a ciphertext

$$c = (c_1, c_2) = (g^r, h^r \cdot g^m),$$

servers compute as follows:

1. For $b \in \{0,1\}$, server b samples a random share $[\![\gamma]\!]_b \leftarrow \mathbb{Z}_q$,

computes

$$c'_{1,b} \leftarrow g^{[\![\gamma]\!]_b} \cdot c_1^{[\![f(x)]\!]_b}, c'_{2,b} \leftarrow h^{[\![\gamma]\!]_b} \cdot c_2^{[\![f(x)]\!]_b},$$

and sends $(c'_{1,b}, c'_{2,b})$ to server $1-b$.

2. For $b \in \{0,1\}$, server b computes

$$c' = (c'_1, c'_2) \leftarrow (c'_{1,0} \cdot c'_{1,1}, c'_{2,0} \cdot c'_{2,1}).$$

3. Server 0 defines $f_0 \leftarrow (c'_1)^{-[\![\mathsf{sk}]\!]_0}$, and server 1 defines $f_1 \leftarrow c'_2 \cdot (c'_1)^{-[\![\mathsf{sk}]\!]_1}$.

We can check that $f_0 \cdot f_1 = g^{f(x)\cdot m}$, meaning that f_0 and f_1 multiplicatively share $g^{f(x)\cdot m}$ (equivalently, additive sharing of $f(x)\cdot m$ in the exponent). Note that the parties can reconstruct $g^{f(x)\cdot m}$ by disclosing shares f_0 and f_1 to each other. When $f(x)\cdot m$ is small enough, the result $f(x)\cdot m$ can be recovered by exhaustive search.

The above protocol is essentially a classic secret-shared decryption protocol for the ElGamal encryption scheme. Correctness is easy to check:

$$c'_{1,b} \leftarrow g^{[\![\gamma]\!]_b + r\cdot[\![f(x)]\!]_b}, c'_{2,b} \leftarrow h^{[\![\gamma]\!]_b + r\cdot[\![f(x)]\!]_b} \cdot g^{m\cdot[\![f(x)]\!]_b},$$

$$c'_1 = c'_{1,0} \cdot c'_{1,1} = g^{\gamma + r\cdot f(x)}, c'_2 = c'_{2,0} \cdot c'_{2,1} = h^{\gamma + r\cdot f(x)} \cdot g^{m\cdot f(x)}.$$

Thus, $c' = (c'_1, c'_2)$ is indeed a ciphertext of $f(x)\cdot m$ in the exponent. As follows,

$$f_0 \cdot f_1 = (c'_1)^{-[\![\mathsf{sk}]\!]_0} \cdot c'_2 \cdot (c'_1)^{-[\![\mathsf{sk}]\!]_1} = g^{m\cdot f(x)}.$$

As for privacy, $g^{[\![\gamma]\!]_b}$ and $h^{[\![\gamma]\!]_b}$ are used to randomize $(c_1^{[\![f(x)]\!]_b}), c_2^{[\![f(x)]\!]_b})$, so that nothing about $[\![f(x)]\!]_b$ is revealed; this is a standard randomization trick.

A.4 SFSS for Generic Functions

For the generic case with function outputs not limited to prediate values (0/1), we design a generic SFSS framework via a "predicate decomposition" trick.

Tool: bit decomposition. Given any $x \in \mathbb{G}$, one can decompose x into a vector of values $(x_{\ell-1}, \cdots, x_1, x_0) \in \mathbb{G}^\ell$ such that $x = \sum_{i\in[\ell]} g^i \cdot x_i$ for some base $g \in \mathbb{G}$. Following existing notations [46], we use a gadget vector

$$g = (1, g^1, g^2, \cdots, g^{\ell-1}) \in \mathbb{G}^\ell.$$

$\boldsymbol{g}^{-1}: \mathbb{G} \to \mathbb{G}^\ell$ denotes the bit decomposition function that maps $x \in \mathbb{G}$ into a *binary* vector $\boldsymbol{x} = (x_{\ell-1}, \cdots, x_1, x_0) = \boldsymbol{g}^{-1}(x) \in \mathbb{G}^\ell$. We can map $\boldsymbol{x}$ back to x by inner-produt $x = \langle \boldsymbol{g}, \boldsymbol{x} \rangle$. The base g depends on $\mathbb{G}$. For example, we can have $g = 2$ for $\mathbb{G} = \mathbb{F}_p$ and $g = X$ for $\mathbb{G} = \mathbb{F}_{2^\ell}$.

Predicate decomposition. Our framework follows a simple idea: any function output $f(x)$ can be decomposed into a binary form $\boldsymbol{y} = (y_0, y_1, \cdots, y_{\ell-1}) \leftarrow \boldsymbol{g}^{-1}(y)$. To compute $m \cdot y$, it's sufficient to compute

$$m \cdot \langle \boldsymbol{g}, \boldsymbol{y} \rangle = \sum_i m \cdot g_i \cdot y_i = \langle \boldsymbol{g}, m \cdot \boldsymbol{y} \rangle.$$

Therefore, we can rely on SFSS for predicate functions to compute each $y_i \cdot m$ separately, and then linearly combine the bit-wise outputs using $\boldsymbol{g}$.

Correctness follows from the construction and the correctness of SFSS for prediate functions. For any $x \in \mathbb{G}_{\mathsf{in}}$ such that $y = f(x) = \langle \boldsymbol{g}, \boldsymbol{g}^{-1}(y) \rangle$, we have $[\![s_i]\!]_0 + [\![s_i]\!]_1 = y_i \cdot m + e_i$, where $\boldsymbol{e} \in [0,B]^\ell$ is an error vector with bounded ℓ_1-norm introduced by the KH-PRF. Overall, we have $\boldsymbol{s} = m \cdot \boldsymbol{y} + \boldsymbol{e} = m \cdot \mathsf{BitDecom}(f(x)) + \boldsymbol{e}$ as requried. The servers can use the error reduction/elimination method in §4.3 and Appendix §A.2 to remove these errors before conducting the remaining linear combination to share $m \cdot f(x)$.

Security of the generic SFSS construction follows directly from the security of SFSS for predicate functions, as the only additional step is decomposing function outputs into binary form. And no further leakage is introduced. Thus, we do not present the same proof again.

Remark 1. *One may be concerned about the feasibility of SFSS construction for general functions, because this SFSS framework requires an existing FSS scheme for $\mathcal{F}$, which can be difficult to realize for generic functions [18, 19] based on well-studied assumptions. Moreover, bit-decomposition for every output $f(x)$ could be computationally infeasible for*

Parameters: A function family $\mathcal{F}_{\mathbb{G}_{\mathsf{in}},\mathbb{G}_{\mathsf{out}}}$ with domain $\mathbb{G}_{\mathsf{in}}$ and range $\mathbb{G}_{\mathsf{out}}$; The bit-decomposed function family of $\mathcal{F}_{\mathbb{G}_{\mathsf{in}},\mathbb{G}_{\mathsf{out}}}$ denoted as $\mathcal{H}_{\mathbb{G}_{\mathsf{in}},\mathbb{G}^\ell_{\mathsf{out}}}$ with domain $\mathbb{G}_{\mathsf{in}}$ and range $\mathbb{G}^\ell_{\mathsf{out}}$; An SFSS scheme Π_{SFSS} for the predicate function family $\mathcal{H}$ with domain $\mathbb{G}_{\mathsf{in}}$ and range $\mathbb{G}^\ell_{\mathsf{out}}$;

```
Gen($1^\lambda, \hat{f} \in \mathcal{F}_{\mathbb{G}_{\mathsf{in}},\mathbb{G}_{\mathsf{out}}}$):
  1: Let $h \in \mathcal{H}$ be the output-decomposed function of $f$
  2: $(\mathsf{st}_h^{\mathsf{sfss}}, \mathsf{k}_\mathsf{e}^{\mathsf{sfss}}, k_0^{\mathsf{sfss}}, k_1^{\mathsf{sfss}}) \leftarrow \Pi_{\mathsf{SFSS}}.\mathsf{Gen}(1^\lambda, \hat{h})$
  3: $\mathsf{st}_f \leftarrow \{\mathsf{st}_h^{\mathsf{sfss}}\}, \mathsf{k}_\mathsf{e} \leftarrow \{\mathsf{k}_\mathsf{e}^{\mathsf{sfss}}\}, \mathsf{k}_0 \leftarrow \{\mathsf{k}_0^{\mathsf{sfss}}\}, \mathsf{k}_1 \leftarrow \{\mathsf{k}_1^{\mathsf{sfss}}\}$
  4: Return $(\mathsf{st}_f, \mathsf{k}_0, \mathsf{k}_1)$
Enc($\mathsf{st}_f, \mathsf{k}_\mathsf{e}, m$):
  1: Parse $\mathsf{st}_f = \{\mathsf{st}_h^{\mathsf{sfss}}\}$ and $\mathsf{k}_\mathsf{e} = \{\mathsf{k}_\mathsf{e}^{\mathsf{sfss}}\}$
  2: $(\mathsf{st}_f^{\mathsf{sfss}}, c) \leftarrow \Pi_{\mathsf{SFSS}}.\mathsf{Enc}(\mathsf{st}_f^{\mathsf{sfss}}, \mathsf{k}_\mathsf{e}^{\mathsf{sfss}}, m)$
  3: Update $\mathsf{st}_f = \{\mathsf{st}_h^{\mathsf{sfss}}\}$
  4: Return $(\mathsf{st}_f, c)$
Eval($b, \mathsf{k}_b, x, c$):
  1: Parse $\mathsf{k}_b = \{\mathsf{k}_b^{\mathsf{sfss}}\}$
  2: $[\![\boldsymbol{s}]\!]_b \leftarrow \Pi_{\mathsf{SFSS}}.\mathsf{Eval}(b, \mathsf{k}_b^{\mathsf{sfss}}, x, c)$
  3: Return $[\![\boldsymbol{s}]\!]_b = ([\![s_{\ell-1}]\!], \cdots, [\![s_1]\!], [\![s_0]\!])$.
```

Figure 9: The SFSS framework for generic functions.

large domains. Nevertheless, this SFSS framework is cheap to realize for many useful functions, including multi-point functions and comparison functions with weighted outputs, where efficient FSS for these functions are already available, and the output bit-decomposition can be done efficiently. Moreover, these functions are the focus of existing FSS works, facilitating a range of applications. Our SFSS framework provides a universal compiler to convert these FSS schemes to their streaming varints efficiently.

B Additional Implementation & Performance

B.1 LWR-based KH-PRF Parameter Setting

The goal of LWR parameter setting is to determine d and q in the LWR-based KH-PRF $F_{\mathrm{LWR}}(\boldsymbol{k},x) : \mathbb{Z}_q^d \times \{0,1\}^* \to \mathbb{Z}_p$ according to p, where p depends on application settings. We adopt a parameter selection strategy that maximizes efficiency while maintaining security guarantees. For concrete security estimation, we follow the standard methodology widely adopted in lattice-based cryptography literature. Our security analysis relies on current analyses of the concrete hardness of LWR, which indicate that no known attack method provides an advantage against LWR compared to *Learning-With-Error* (LWE).

Table 6: **Parameter setting for LWR-based KH-PRF**.

p	2^{10}	2^{20}	2^{30}	2^{40}	2^{50}	2^{60}
d	200	200	320	550	840	1180
q	2^{30}	2^{100}	2^{127}	2^{127}	2^{127}	2^{127}

Following the approach established in prior works such as Saber [33] and BAT [36], we estimate the security level of our LWR-based construction to be equivalent to that of LWE, using the lattice estimator[1] from [3] with the cost model from [48]. In practice, we observe that the bit length of the modulus q has a minor impact on computational efficiency when it remains below 128 bits because we use `__uint128_t` in our implementation to emulate the modular arithmetic over $\mathbb{Z}_q$ in this case, while efficiency is highly sensitive to the lattice dimension d. Therefore, targeting 128-bit classical security, we select parameters that minimize d to achieve optimal efficiency in running time, as detailed in Table 6.

B.2 Storage for Policy-hiding Aggregation

Table 7 shows detailed storage comparison between DPF-based, Vizard, and SDPF-based policy-hiding aggregation. The SDPF-based approach achieves the optimal server-side storage as Vizard, yet significantly outperforms the non-streaming DPF-based approach. As mentioned, the reason is that SFSS-based solutions only requires one single ciphertext for each streaming message, while the FSS-based approach encodes a streaming message together with a policy function using a pair of FSS keys, which is concretely high for a large $\mathbb{G}_{\mathsf{in}}$.

[1]https://github.com/malb/lattice-estimator

C Additional Applications

C.1 Attribute-hiding Aggregation

Generic method from 2PC. We can resort to two-party computation (2PC) for secure attribute-hiding aggregation. In this approach, each client shares its attributes and data items with the servers. To perform attribute-based aggregation, the servers run a 2PC protocol to securely evaluate the policy function over the shared attributes and include the data item in the output only if the attributes satisfy the predicate. However, this method incurs communication that is (super-)linear in the number of clients and requires multiple rounds of interaction between the servers, especially for complex predicates.

In contrast, we propose a simple attribute-encoding method via single-point functions. This enables a solution using SDPF that supports *non-interactive aggregation*, requiring no communication between the servers during evaluation.

High-level idea. Suppose a client has a message m and N_a attributes

$$\alpha = (\alpha_1, \alpha_2, \cdots \alpha_{N_a}) \in \{0,1\}^{\ell_1} \times \{0,1\}^{\ell_2} \cdots \{0,1\}^{\ell_{N_a}},$$

where $|\alpha_i| = \ell_i$ and $\ell = \sum_{i\in[N_a]} \ell_i$. We set $\alpha = (\alpha_1, \alpha_2, \cdots, \alpha_{N_a})$ and define a single-point function

$$f_{\alpha,m}(x) = \begin{cases} m, x = \alpha, \\ 0, x \neq \alpha. \end{cases}$$

Now, suppose an aggregation task defines a (filtering) predicate policy function

$$p(x_1, x_2, \cdots x_{N_a}).$$

Our goal is to compute $p(\alpha_1, \alpha_2, \cdots \alpha_{N_a}) \cdot m$. To this end, we first define a set $\mathcal{X}$ that contains all *accepting* attributes

$$\mathcal{X} = \{x \,|\, \forall x \in \{0,1\}^{\ell} \text{ s.t. } p(x_1, x_2, \cdots, x_{N_a}) = 1\}.$$

Now we claim $\sum_{x\in\mathcal{X}} f_{\alpha,m}(x) = p(\alpha_1, \alpha_2, \cdots, \alpha_{N_a}) \cdot m$. Correctness is easy to see: if $p(\alpha) = 1$, it must have $\alpha \in \mathcal{X}$, then

$$\sum_{x\in\mathcal{X}} f_{\alpha,m}(x) = f_{\alpha,m}(\alpha) + \sum_{x\in\mathcal{X}\setminus\{\alpha\}} f_{\alpha,m}(x) = m.$$

Otherwise, we have

$$p(\alpha) = 0 \Leftrightarrow \alpha \notin \mathcal{X} \Leftrightarrow \sum_{x\in\mathcal{X}} f_{\alpha,m}(x) = 0.$$

Overall, we have

$$\sum_{x\in\mathcal{X}} f_{\alpha,m}(x) = p(\alpha_1, \alpha_2, \cdots, \alpha_{N_a}) \cdot m.$$

Table 7: **Comparison of storage consumption (MB) of different schemes**.

Message Space	#Ciphertexts Per User	Scheme	$\mathbb{G}_{\text{in}} = \mathbb{Z}_{2^{32}}$ Number of contributors 100	1000	10000	100000	$\mathbb{G}_{\text{in}} = \mathbb{Z}_{2^{64}}$ Number of contributors 100	1000	10000	100000
$\mathbb{G}_{\text{out}} = \mathbb{Z}_{2^{32}}$	10	DPF-based	0.57	5.69	56.93	569.34	1.12	11.19	111.87	1118.66
		Vizard [25]	0.06	0.65	6.46	64.56	0.12	1.19	11.95	119.50
		Our	0.06	0.64	6.42	64.18	0.12	1.19	11.91	119.11
	100	DPF-based	5.69	56.93	569.34	5693.44	11.19	111.87	1118.66	11186.60
		Vizard [25]	0.13	1.33	13.32	133.23	0.19	1.88	18.82	188.16
		Our	0.13	1.33	13.28	132.85	0.19	1.88	18.78	187.78
	1000	DPF-based	56.93	569.34	5693.44	56934.36	111.87	1118.66	11186.60	111866.00
		Vizard [25]	0.82	8.20	81.99	819.87	0.87	8.75	87.48	874.81
		Our	0.82	8.19	81.95	819.49	0.87	8.74	87.44	874.42
$\mathbb{G}_{\text{out}} = \mathbb{Z}_{2^{64}}$	10	DPF	0.59	5.88	58.84	588.42	1.12	11.22	112.25	1122.47
		Vizard [25]	0.07	0.73	7.26	72.57	0.13	1.28	12.75	127.51
		Our	0.07	0.72	7.18	71.81	0.13	1.27	12.67	126.74
	100	DPF-based	5.88	58.84	588.42	5884.17	11.22	112.25	1122.47	11224.75
		Vizard [25]	0.21	2.10	20.99	209.90	0.26	2.65	26.48	264.84
		Our	0.21	2.09	20.91	209.14	0.26	2.64	26.41	264.07
	1000	DPF-based	58.84	588.42	5884.17	58841.71	112.25	1122.47	11224.75	112247.47
		Vizard [25]	1.58	15.83	158.32	1583.19	1.64	16.38	163.81	1638.13
		Our	1.58	15.82	158.24	1582.43	1.64	16.37	163.74	1637.36
$\mathbb{G}_{\text{out}} = \mathbb{Z}_{2^{128}}$	10	DPF-based	0.58	5.81	58.08	580.79	1.13	11.30	113.01	1130.10
		Vizard [25]	0.09	0.89	8.86	88.60	0.14	1.44	14.35	143.53
		Our	0.09	0.87	8.71	87.07	0.14	1.42	14.20	142.00
	100	DPF-based	5.81	58.08	580.79	5807.88	11.30	113.01	1130.10	11301.04
		Vizard [25]	0.36	3.63	36.33	363.25	0.42	4.18	41.82	418.19
		Our	0.36	3.62	36.17	361.73	0.42	4.17	41.67	416.66
	1000	DPF-based	58.08	580.79	5807.88	58078.77	113.01	1130.10	11301.04	113010.41
		Vizard [25]	3.11	31.10	310.98	3109.84	3.16	31.65	316.48	3164.77
		Our	3.11	31.08	310.83	3108.31	3.16	31.63	316.32	3163.24

Formal description. Recall that SDPF supports decoupling the static attributes $\alpha_1, \alpha_2, \cdots, \alpha_{N_a}$ from the dynamic streaming message m. Thus, the client can encode its attributes $\alpha_1, \alpha_2, \cdots, \alpha_{N_a}$ using a single-point function $f_{(\alpha_1,\alpha_2,\cdots,\alpha_{N_a}),(1,r)}$ in the SDPF setup phase, and invoke the SDPF encryption algorithm to encrypt m. The servers obviously share $p(\alpha_1, \alpha_2, \cdots, \alpha_{N_a}) \cdot m$. Fig. 10 shows the formal non-interactive attribute-hiding aggregation based on SDPF.

Performance. We report the performance of the attribute-hiding aggregation. Note that the accepting query set $\mathcal{X}$ is at most the size of $\mathbb{G}_{\text{in}}$. Therefore, we report the *worst* efficiency when $\mathcal{X}$ contains all the elements in $\mathbb{G}_{\text{in}}$, which corresponds to the full-domain evaluation of SDPF. In this case, Table 8 reports the query efficiency with different sizes of $\mathbb{G}_{\text{in}}$ and number of contributors N_c. In particular, the evaluation time grows linearly with the number N_c of clients and is exponentially with the bit size $\log|\mathbb{G}_{\text{in}}|$. Overall, the running time grows linearly with N_c. Our attribute-hiding naturally supports evaluation speedup via multi-thread optimization. We depict the trend in Fig. 11.

The query efficiency would be intolerable for large $\mathbb{G}_{\text{in}}$ in the extreme case, which would be computationally expensive to evaluate. Therefore, our attribute-hiding aggregation is practical over $\mathbb{G}_{\text{in}}$ with small and moderate sizes. Nevertheless, this already suffices for real-world applications because we can leverage quantization, approximate data structure/encoding methods [25, 29] to reduce the bit size of $\mathbb{G}_{\text{in}}$, which already suffices in a practical sense. Note the SPDF-based `ahAgg` supports any filtering function – it does not require the contributors to know the function during setup.

Table 8: **Running time (s) with varying $\mathbb{G}_{\text{in}}$ and N_c**.

N_c	#Threads	$\lvert\mathbb{G}_{\text{in}}\rvert$ 2^{10}	2^{12}	2^{14}	2^{16}	2^{18}	2^{20}
100	1 Thread	0.02	0.05	0.19	0.74	2.94	11.98
	2 Thread	0.01	0.03	0.09	0.37	1.57	6.18
	4 Thread	0.01	0.02	0.05	0.23	0.91	3.94
	8 Thread	<0.01	0.01	0.04	0.20	0.88	3.42
1000	1 Thread	0.12	0.45	1.88	7.33	32.07	125.13
	2 Thread	0.06	0.24	0.96	3.78	15.43	62.67
	4 Thread	0.04	0.12	0.55	2.73	10.75	41.05
	8 Thread	0.02	0.08	0.38	2.30	8.53	33.52

```
Parameters: A SFSS scheme $\Pi_{\mathsf{SFSS}}$ for a function family $\mathcal{F}_{\mathbb{G}_{\mathsf{in}},\mathbb{G}_{\mathsf{out}}}$ with domain $\mathbb{G}_{\mathsf{in}}$ and range $\mathbb{G}_{\mathsf{out}}$; $N_a$ denotes the number of attributes; $\ell_i$ denotes the bit size of $i$-th attribute, $\ell = \sum_{i\in[N_a]} \ell_i$, and $\log|\mathbb{G}_{\mathsf{in}}| = \ell$;
$\mathsf{Setup}(1^\lambda, (\alpha_1, \alpha_2, \cdots, \alpha_{N_a}); \perp; \perp)$:
  1: The client $i$ defines $\alpha = \alpha_1||\alpha_2||\cdots||\alpha_{N_a}$ and a point function $f_{\alpha,1}$. The client $i$ generates SDPF keys
       $(\mathsf{st}_f, \mathsf{k}_0^{(i)}, \mathsf{k}_1^{(i)}) \leftarrow \Pi_{\mathsf{SDPF}}.\mathsf{Gen}(1^\lambda, \hat{f}_{\alpha,1})$,
     and sends $\mathsf{k}_b^{(i)}$ to the server $b$ for $b \in \{0,1\}$.
  2: Client $i$ stores $\mathsf{st}^{(i)}$ locally. Server $b$ stores $\mathsf{k}_b^{(i)}$.
$\mathsf{Send}((\mathsf{st}_f, \mathsf{k}_\mathsf{e}^{(i)}, m_j^{(i)}); \perp; \perp)$:
  1: Client $i$ encrypt its $j$-th streaming message $m_j^{(i)}$
       $(\mathsf{st}_f^{(i)}, c_j^{(i)}) \leftarrow \Pi_{\mathsf{SDPF}}.\mathsf{Enc}(\mathsf{st}_f^{(i)}, \mathsf{k}_\mathsf{e}^{(i)}, m_j^{(i)})$.
  2: Sends $c_j^{(i)}$ to both servers.
$\mathsf{Aggregate}(b, j, p, \{c_j^{(i)}\}_{i\in[N_c]})$:
  1: : Define the accepting set
       $\mathcal{X} = \{x \,|\, \forall x \in \{0,1\}^\ell \text{ s.t. } p(x_1, x_2, \cdots, x_{N_a}) = 1\}$.
  2: Server $b$ computes
       $[\![s]\!]_b \leftarrow \sum_{i\in[N_c]} \sum_{x\in\mathcal{X}} \Pi_{\mathsf{SDPF}}.\mathsf{Eval}(b, \mathsf{k}_b^{(i)}, x, c_j^{(i)})$.
```

Figure 10: SDPF-based attribute-hiding aggregation.

C.2 Window-based Aggregation

Many privacy-preserving aggregation systems require window-based aggregation. In window-based aggregation, the server aggregates streaming messages over a window range $[l,r]$. Note that the SFSS-based approach already supports window-based aggregation by invoking the SFSS Enc algorithm repeatedly over the range $[l,r]$. However, this requires the servers to conduct DPF and PRF evaluation that is linear to the window width, which could be expensive for a large window range. In the following, we construct a new window-based aggregation that only requires $O(1)$ DPF and PRF calls per data stream following the method from [24].

Window-based aggregation: SFSS for point function. Fig. 13 presents the window-based policy-hiding aggregation from SDPF. The idea is inspired by the two-server homomorphic stream encryption from [24], which is also used in Vizard. To encrypt the j-th message, the client will use the masks for encryptng the j-th and $(j+1)$-th streaming messages, such that once the streaming ciphertexts between $[l,r]$ are aggregated, all intermidiate masks canclled out so that only the l-th mask and $(r+1)$-th mask remain in the aggregated ciphertext. Note that ciphertext aggregation only requires simple addition over $\mathbb{G}_{\mathsf{out}}$, and the servers need to compute and take off two masks for window-based aggregation.

Window-based aggregation: SFSS for predicate functions. We can use the same double-key mechanism for the SFSS over predicate functions. In particular, the client uses the masks derived from the KH-PRF for the ctr and ctr + 1 to encrypt the ctr-th streaming message. For an aggregation task over a window $[l,r]$, the servers first aggregate over the range by summing up the streaming ciphertexts. By the definition, the only remaining masks are $F(\mathsf{k}_{\mathsf{kh}}, l)$ and $F(\mathsf{k}_{\mathsf{kh}}, r+1)$. The

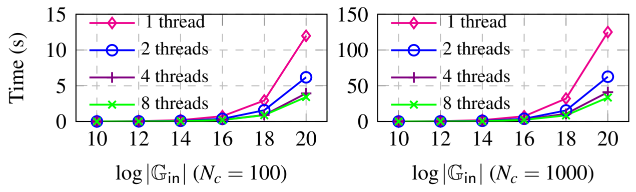


Figure 11: Evaluation speedup via multi-thread optimization.

```
Parameters: A SFSS scheme $\Pi_{\mathsf{SFSS}}$ for a function family $\mathcal{F}_{\mathbb{G}_{\mathsf{in}},\mathbb{G}_{\mathsf{out}}}$ with domain $\mathbb{G}_{\mathsf{in}}$ and range $\mathbb{G}_{\mathsf{out}}$;
$\mathsf{Setup}(i, 1^\lambda, \hat{f} \in \mathcal{F}; \perp; \perp)$:
     Client $i$:
  1: $(\mathsf{st}^{(i)}, \mathsf{k}_\mathsf{e}^{(i)}, \mathsf{k}_0^{(i)}, \mathsf{k}_1^{(i)}) \leftarrow \Pi_{\mathsf{SFSS}}.\mathsf{Gen}(1^\lambda, \hat{f})$
  2: Send $\mathsf{k}_0^{(i)}$ to servers 0 and and $\mathsf{k}_1^{(i)}$ to server 1
     Servers:
  3: Server $b$ receives and stores $\mathsf{k}_b^{(i)}$ locally
$\mathsf{Send}((i, \mathsf{st}^{(i)}, \mathsf{k}_\mathsf{e}^{(i)}); \mathsf{k}_0^i; \mathsf{k}_1^i)$:
     Client $i$:
  1: Parse $\mathsf{st}^{(i)} = \{\mathsf{ctr}\}$ and $\mathsf{k}_\mathsf{e}^{(i)} = \{\mathsf{k}_{\mathsf{kh}}\}$
  2: $c^{(i,\mathsf{ctr})} \leftarrow m - F(\mathsf{k}_{\mathsf{kh}}, \mathsf{ctr}) + F(\mathsf{k}_{\mathsf{kh}}, \mathsf{ctr}+1)$
  3: Update $\mathsf{st}^{(i)} \leftarrow \{\mathsf{ctr}+1\}$
     Servers:
  4: Both servers receive and store $c^{(i,\mathsf{ctr})}$.
$\mathsf{Aggregate}(b, \mathsf{win} = [l,r], \mathsf{att}, \mathsf{k}_b^{(i)}, \{c^{(i,j)}\}_{i\in[n], j\in\mathsf{win}})$:
     Server $S_b$:
  1: for $i \in [n]$ do:
  2:     Let $c^{(i)} \leftarrow \sum_{j\in\mathsf{win}} c^{(i,j)}$
  3:     $([\![e^{(i)}]\!]_b, [\![\mathsf{k}^{(i)}]\!]_b) \leftarrow \Pi_{\mathsf{FSS}}.\mathsf{Eval}(b, \mathsf{k}_b^{(i)}, \mathsf{att})$
  4:     $[\![s^{(i)}]\!]_b \leftarrow [\![e^{(i)}]\!]_b \cdot c^{(i)} + F([\![\mathsf{k}]\!]_b, l) - F([\![\mathsf{k}]\!]_b, r+1)$
  5: Set $[\![s]\!]_b \leftarrow \sum_{i\in[n]} [\![s^{(i)}]\!]_b$
  6: Return $[\![s]\!]_b$
```

Figure 12: Window-based aggregation with bounded errors for SFSS over predicate function

Parameters: A SDPF scheme Π_{SFSS} for a function family $\mathcal{F}_{\mathbb{G}_{\mathsf{in}},\mathbb{G}_{\mathsf{out}}}$ with domain $\mathbb{G}_{\mathsf{in}}$ and range $\mathbb{G}_{\mathsf{out}}$;

```
Setup$(1^\lambda, \hat{f}_{\alpha,(\beta,r)} \in \mathcal{F}; \perp; \perp)$:
    Client $i$:
1:  $r \xleftarrow{\$} \mathbb{F}$, $\mathsf{ctr} = 1$.
2:  $(\mathsf{k}_0^{\mathsf{dpf}}, \mathsf{k}_1^{\mathsf{dpf}}) \leftarrow \Pi_{\mathsf{DPF}}.\mathsf{Gen}(1^\lambda, \hat{f}_{\alpha,(\beta,r)})$
3:  $([\![\beta]\!]_b, [\![r]\!]_b) \leftarrow \Pi_{\mathsf{DPF}}.\mathsf{Eval}(b, \mathsf{k}_b^{\mathsf{dpf}}, \alpha)$ for $b \in \{0,1\}$
4:  $\mathsf{st}^{(i)} \leftarrow \{\mathsf{ctr}\}$, $\mathsf{k}_\mathsf{e} \leftarrow \{[\![r]\!]_0, [\![r]\!]_1\}$, $\mathsf{k}_0^{(i)} \leftarrow \{\mathsf{k}_0^{\mathsf{dpf}}\}$, $\mathsf{k}_1^{(i)} \leftarrow$
    $\{\mathsf{k}_1^{\mathsf{dpf}}\}$
5:  Store $(\mathsf{st}^{(i)}, \mathsf{k}_\mathsf{e}^{(i)}, \mathsf{k}_0^{(i)}, \mathsf{k}_1^{(i)})$ and send $\mathsf{k}_b^{(i)}$ to server $b$.
    Servers:
6:  Server $b$ receives and stores $\mathsf{k}_b^{(i)}$ locally.
Send$((\mathsf{st}^{(i)}, \mathsf{k}_\mathsf{e}, m_j^{(i)}); \mathsf{k}_0^i; \mathsf{k}_1^i)$:
    Client $i$:
1:  Parse $\mathsf{st}^{(i)} = \{\mathsf{ctr}\}$ and $\mathsf{k}_\mathsf{e} = \{[\![r]\!]_0, [\![r]\!]_1\}$
2:  $c^{(i,\mathsf{ctr})} \leftarrow m - (F([\![r]\!]_0, \mathsf{ctr}) - F(-[\![r]\!]_1, \mathsf{ctr})) \cdot \beta^{-1} +$
    $(F([\![r]\!]_0, \mathsf{ctr}+1) - F(-[\![r]\!]_1, \mathsf{ctr}+1)) \cdot \beta^{-1}$
3:  Update $\mathsf{st}^{(i)} \leftarrow \{\mathsf{ctr}+1\}$
    Servers:
4:  Both servers receive and store $c^{(i,\mathsf{ctr})}$.
Aggregate$(b, \mathsf{win} = [l,r], \mathsf{att}, \mathsf{k}_b^{(i)}, \{c^{(i,j)}\}_{i\in[n], j\in\mathsf{win}})$:
    Server $S_b$:
1:  for $i \in [n]$ do:
2:      Let $c^{(i)} \leftarrow \sum_{j\in\mathsf{win}} c^{(i,j)}$
3:      $([\![e^{(i)}]\!]_b, [\![h^{(i)}]\!]_b) \leftarrow \Pi_{\mathsf{DPF}}.\mathsf{Eval}(b, \mathsf{k}_b^{(i)}, \mathsf{att})$
4:      Compute $[\![s^{(i)}]\!]_b \leftarrow [\![e^{(i)}]\!]_b \cdot c^{(i)} + (-1)^b \cdot F((-1)^b \cdot$
        $[\![h]\!]_b, l) - (-1)^b \cdot F((-1)^b \cdot [\![h]\!]_b, r+1)$
5:  Set $[\![s]\!]_b \leftarrow \sum_{i\in[n]} [\![s^{(i)}]\!]_b$
6:  Return $[\![s]\!]_b$
```

Figure 13: Window-based aggregation for SDPF

server only needs to take off two masks during window-based aggregation to share the underlying aggregated value.

D Security Proof

D.1 Proof of Theorem 1

Proof. Correctness refers to §4.2. Below we prove Theorem 1. Overall, we prove that the SDPF key k_b and ciphertexts $\{c_1, c_2, \cdots, c_q\}$ are pseudorandom; thus revealing no more non-trivial information. The proof is similar to the previous proof strategy for FSS [19, 40]. Since we use a DPF construction in a black-box fashion, we can directly use the existing simulation strategy [18, 19, 40] for DPF.

For any PPT adversary $\mathcal{A} = (\mathcal{A}_0, \cdots, \mathcal{A}_q)$, we construct a PPT simulator $\mathcal{S} = (\mathcal{S}_0, \cdots, \mathcal{S}_q)$ such that for any point function $f_{\alpha,(\beta,r)}$, any $b \in \{0,1\}$, $\mathcal{S}$ simulates a DPF key k_b and streaming ciphertexts $\{c_j\}_{j\in[1,q]}$ from the allowed leakage function $\mathsf{Leak}(\hat{f}, (m_1, \cdots, m_j)) = (\mathbb{G}_{\mathsf{in}}, \mathbb{G}_{\mathsf{out}} \times \{0,1\}^\lambda, q)$.

$\mathcal{S}_0(1^\lambda, b, \mathsf{Leak}(\hat{f}))$:

```
1:  Invoke $\mathcal{S}^{\mathsf{DPF}}$ to obtain $\mathsf{k}_b \leftarrow \mathcal{S}^{\mathsf{DPF}}(1^\lambda, b, \mathsf{Leak}(\hat{f}))$
2:  Return $\mathsf{k}_b$
```

While for $j \in [1,q]$, the simulator $\mathcal{S}_j$ works as follows: $\mathcal{S}_j(1^\lambda, b, \perp, \mathsf{Leak}(\hat{f}, (m_1, \cdots, m_j)))$:

```
1:  Sample $c_j \xleftarrow{\$} \mathbb{G}_{\mathsf{out}}$ and return $(j, c_j)$
```

We prove that the distribution of the simulated key and ciphertexts is indistinguishable from that of the original Π_{SDPF} key generation and encoding, via a sequence of hybrid games.

- Hyb_0: This is exactly the Π_{SDPF} construction in Fig. 7.
- Hyb_1: Hyb_1 only differs with Hyb_0 by calling $\mathcal{S}^{\mathsf{DPF}}(1^\lambda, b, \mathsf{Leak}(\hat{f}))$ to generate $\mathsf{k}_b^{\mathsf{dpf}}$. The adversary can only distinguishes the view from Hyb_0 and Hyb_1 with negligible probability from the pseudorandomenss of $\mathsf{k}_b^{\mathsf{dpf}}$; a concrete $\mathcal{S}^{\mathsf{DPF}}$ can be found from [19, 40].
- Hyb_2: Hyb_2 differs with Hyb_1 on the generation of c. In Hyb_1, we have
$$c_j \leftarrow m_j - ((-1)^b \cdot F((-1)^b \cdot [\![r]\!]_b, j) \\ + (-1)^{1-b} \cdot F((-1)^b \cdot [\![r]\!]_{1-b}, j)),$$
while in Hyb_2, we have $k \xleftarrow{\$} \mathbb{G}_{\mathsf{out}}$ and Hyb_2 computes
$$c_j \leftarrow m_j - ((-1)^b \cdot F((-1)^b \cdot [\![r]\!]_b, j) + (-1)^{1-b} \cdot F(k, j)).$$
The replacement is computationally indistinguishable from the pseudorandom-share property of FSS.
- Hyb_3: Hyb_3 differs with Hyb_2 on the generation of c. In Hyb_2, we have
$$c_j \leftarrow m_j - ((-1)^b \cdot F((-1)^b \cdot [\![r]\!]_b, j) + (-1)^{1-b} \cdot F(k, j)),$$
while in Hyb_3, we have $\mu \xleftarrow{\$} \mathbb{G}_{\mathsf{out}}$ and Hyb_3 computes
$$c_j \leftarrow m_j - ((-1)^b \cdot F((-1)^b \cdot [\![r]\!]_b, j) + (-1)^{1-b} \cdot \mu).$$
We prove Hyb_3 and Hyb_2 are computationally indistinguishable via Claim 1.

Claim 1. *There exists a polynominal p' such that for any $(T, \varepsilon_{\mathsf{PRF}})$-secure pseudorandom function F, then for any $b \in \{0,1\}$ and any non-uniform PPT adversary $\mathcal{D}$ running in time $T' \leq T - p'(\lambda)$, it holds that*

$$\left| \Pr\left[\begin{array}{c} (\mathsf{k}_b, \{c_j\}_{j\in[q]}) \leftarrow \mathsf{Hyb}_1(1^\lambda, b) : \\ \mathcal{D}(1^\lambda, \mathsf{st}_\mathcal{A}, (\mathsf{k}_b, \{c_j\}_{j\in[q]})) = 1 \end{array} \right] - \Pr\left[\begin{array}{c} (\mathsf{k}_b, \{c_j\}_{j\in[q]}) \leftarrow \mathsf{Hyb}_2(1^\lambda, b) : \\ \mathcal{D}(1^\lambda, \mathsf{st}_\mathcal{A}, (\mathsf{k}_b, \{c_j\}_{j\in[q]})) = 1 \end{array} \right] \right| \leq \mathsf{negl}(\lambda),$$

Proof. Given an distinguisher $\mathcal{D}$ that can distinguish Hyb_2 and Hyb_3 with advantage ε, we construct an adversary $\mathcal{B}$ for PRF F. Note that $\mathcal{B}$ can query a PRF oracle O that is either instantiated as PRF F or a random function f.

- For each streaming encryption, $\mathcal{B}$ query $O(j)$ and receives μ.
- $\mathcal{B}$ computes the ciphertext with $c_1 \leftarrow m - ((-1)^b \cdot F((-1)^b \cdot [\![r]\!]_b, j) + (-1)^{1-b} \cdot \mu)$.
- $\mathcal{B}$ perform other computation as Hyb_2 does for key generation and streaming encryption.
- Return $(\mathsf{k}_b, \{c_j\}_{j\in[q]})$.

Clearly, if O is instanized by F, then $(\mathsf{k}_b, \{c_j\}_{j\in[q]})$ generated by $\mathcal{B}$ is identically distributed as the Hyb_2. If O is instanized with a random function, then $(\mathsf{k}_b, \{c_j\}_{j\in[q]})$ is identically distributed as Hyb_3. Therefore, $\mathcal{B}$'s advantage in distinguishing the PRF from random functions equals to the advantage that $\mathcal{D}$ distinguishes Hyb_2 from Hyb_3. Since $\mathcal{B}$ runs additional time of $p'(\lambda)$ than $\mathcal{A}$, we have that for any $T' \leq T - p'(\lambda)$, $\mathcal{A}$ distinguishes Hyb_1 and Hyb_2 is bounded by $\varepsilon = \varepsilon_{\mathsf{PRF}}$. □

- Hyb_4: The difference between Hyb_4 and Hyb_3 is that Hyb_4 replace $c_j \leftarrow m - ((-1)^b \cdot F((-1)^b \cdot [\![r]\!]_b, j) + (-1)^{1-b} \cdot \mu)$ with $c_j \xleftarrow{\$} \mathbb{G}_{\mathsf{out}}$, which is uniformly distributed over $\mathbb{G}_{\mathsf{out}}$. Note that μ is uniformly distrbitued in Hyb_3, thus $m - ((-1)^b \cdot F((-1)^b \cdot [\![r]\!]_b, j) + (-1)^{1-b} \cdot \mu)$ is uniformally distribtued. This means that Hyb_3 and Hyb_4 are identically distributed. We note that Hyb_4 is exactly the ideal world.

Combining the above hybrid games completes the proof. □

D.2 Proof of Theorem 2

Proof. Since we use an FSS scheme for a function family $\mathcal{F}$ in a black-box fashion in Fig. 4, we will use the existing simulation strategy for FSS directly. The simulator follows the same proof strategy for Π_{SDPF}. For any PPT adversary $\mathcal{A} = (\mathcal{A}_0, \cdots, \mathcal{A}_q)$, we construct a PPT simulator $\mathcal{S} = (\mathcal{S}_0, \cdots, \mathcal{S}_q)$ such that for any function $f \in \mathcal{F}$, any $b \in \{0,1\}$, $\mathcal{S}$ simulates a DPF key k_b and streaming ciphertexts $\{c_j\}_{j\in[1,q]}$ from the allowed leakage function $\mathsf{Leak}(\hat{f}, (m_1, \cdots, m_j)) = (\mathbb{G}_{\mathsf{in}}, \mathbb{G}_{\mathsf{out}} \times \{0,1\}^\lambda, j)$ for $j \in [0,q]$.

$\underline{\mathcal{S}_0(1^\lambda, b, \mathsf{Leak}(\hat{f}))}$:

```
1: Invoke $\mathcal{S}^{\mathsf{FSS}}$ to obtain $\mathsf{k}_b \leftarrow \mathcal{S}^{\mathsf{FSS}}(1^\lambda, b, \mathsf{Leak}(\hat{f}))$
2: Return $\mathsf{k}_b$
```

While for $j \in [1,q]$, the simulator $\mathcal{S}_i$ works as follows: $\underline{\mathcal{S}_j(1^\lambda, b, \mathsf{Leak}(\hat{f}, (m_1, \cdots, m_j)))}$:

```
1: Sample $c_j \xleftarrow{\$} \mathbb{G}_{\mathsf{out}}$ and return $(j, c_j)$
```

We show that the distribution of simulated keys and ciphertexts is indistinguishable from the distribution of Π_{SFSS} key generation and encoding, via a sequence of hybrid games.

- Hyb_0: This is exactly the Π_{SFSS} construction in Fig. 4.
- Hyb_1: Hyb_1 only differs with Hyb_0 by calling $\mathcal{S}_0(1^\lambda, b, \mathsf{Leak}(\hat{f}))$ to generate k_b. The adversary can only distinguish the view from Hyb_0 and Hyb_1 with negligible probability from the pseudorandomness of k_b.
- Hyb_2: The difference between Hyb_1 and Hyb_2 is in the generation of c. In Hyb_1, we have $c_j \leftarrow m + F(\mathsf{k}_{\mathsf{kh}}, j)$, while in Hyb_2, $\mu \xleftarrow{\$} \mathbb{F}$ is uniformally distributed and Hyb_2 computes $c_j \leftarrow m + u$. If $\mathcal{A}$ can distinguish Hyb_1 and Hyb_2 with non-negligible probability, then we can build an adversary $\mathcal{B}$ to distinguish F from random functions with the same advantage. Thus, by the security of F, Hyb_1 and Hyb_2 are indistinguishable.
- Hyb_3: Hyb_3 generates $c_j \leftarrow u$, where $u \xleftarrow{\$} \mathbb{G}_{\mathsf{out}}$ is uniformly distributed. Note that Hyb_3 works exactly as $\mathcal{S}$. We note that $m+u$ and u follow the identical uniform distribution. Hence, Hyb_3 and Hyb_2 are identically distributed.

Combining the above hybrid games completes the proof. □

Remark 2. *It is worth mentioning that our KH-PRF-based SFSS compiler does not require the pseudorandom-share property for the underlying FSS scheme. Intuitively, the pseudorandomness of the FSS key k_b computationally hides the argumented predicate function within its function family, which implies that the embedded KH-PRF key k_{kh} is computationally hidden. Thus, $c_j = m_j + F(\mathsf{k}_{\mathsf{kh}}, j)$ is computationally indistinguishable from $c_j = m_j + F(\mathsf{k}_{\mathsf{kh}}, j)$ over uniformly choosen key $\mathsf{k}_{\mathsf{kh}} \xleftarrow{\$} \mathbb{K}_{\mathsf{kh}}$. This is different from the proof for Theorem 1, where the pseudorandom-share property is necessary because $c_j = m_j - F([\![r]\!]_0, j) + F(-[\![r]\!]_1, j)$, in which $F(-[\![r]\!]_{1-b}, j)$ is the only secret part to hide m_j as the distinguisher $\mathcal{D}$ knowing k_b can compute $F((-1)^b \cdot [\![r]\!]_b, j)$. In this case, we need the pseudorandom-share property for $[\![r]\!]_{1-b}$ as PRF F requires random keys. But again, we note that the pseudorandom-share property is essentially free to obtain following the PRG-based constructions from [18, 19].*