# Batched Distributed Point Function from Sparse LPN and Homomorphic Secret Sharing

Lucas Piske
Arizona State University
lpiske@asu.edu

Jaspal Singh
Purdue University
sing1361@purdue.edu

Ni Trieu
Arizona State University
nitrieu@asu.edu

May 26, 2024

**Abstract**

A function secret sharing (FSS) scheme ($\mathsf{Gen}$,$\mathsf{Eval}$) for a class of programs $\mathcal{F}$ allows a dealer to secret share any function $f \in \mathcal{F}$, such that each function share hides the function, and the shares can be used to non-interactively compute additive shares of $f(x)$ for any input $x$. All FSS related applications often requires the dealer to generate and share secret sharings for a batch of functions. We initiate the study of *batched function secret sharing* - where the objective is to secret share a set of functions from the class $\mathcal{F}$ while minimizing the size of the collection of FSS keys.

We use standard homomorphic secret sharing (HSS) schemes, variant of the Learning with Parity Noise assumption and the Quadratic Residuosity assumption to construct batched FSS schemes for point functions with single-bit and multi-bit output. Our scheme is asymptotically superior than naively batching state of the art FSS schemes for point functions. Concretely our batch key sizes are smaller by a factor of $3 - 80\times$ as batch size is increased from $2^{13}$ to $2^{19}$. Although our protocol relies on public key operations, it exhibits inefficiency in a LAN setting. However, it demonstrates up to a 120-fold improvement in a WAN setting with slow network bandwidth.

As a building block in our protocols, we introduce a new HSS ciphertext compression algorithm, that can decompress a short compressed ciphertext to give low noise ciphertexts of array of input message bits. This primitive might be of independent interest for other HSS related applications.

## 1 Introduction

A two-party function secret sharing (FSS) scheme [11] ($\mathsf{Gen}, \mathsf{Eval}$) for a class of functions $\mathcal{F}$ can be used to secret share any function $f \in \mathcal{F}$ using the $\mathsf{Gen}$ function $((f_0, f_1) \leftarrow \mathsf{Gen}(f))$, such that each share $f_i$ hides $f$ and $\mathsf{Eval}(f_0, x) + \mathsf{Eval}(f_1, x) = f(x)$ for each $x$ in the domain. The efficiency of an FSS scheme is measured by the size of function secret shares and by the computation complexity of $\mathsf{Eval}$. FSS for general polytime computation functions is only known to exist from ideal obfuscation, Spooky Encryption [24], and one-way functions [11]. Hence, a number of works have focused on designing efficient FSS schemes tailored to specific application-oriented function classes. These include point functions, which consist of functions like $f_{a,b}$ that yield 0 everywhere except at $a$, where they output $b$, as well as, multi-point functions, comparison functions and d-dimensional intervals [11, 14, 7, 21].

FSS serves as a fundamental component in various cryptographic protocols and applications, including private information retrieval [11], privacy preserving analytics [19], oblivious transfer

extension [10], MPC with prepossessing [15, 7], anonymous messaging [20], oblivious RAM [25], private set intersection [23, 32], and privacy preserving machine learning [35]. In many of these scenarios, there is often a need to generate a batch of FSS secret shares. This process typically dominates the overall communication complexity of the protocols. For instance, the MPC prepossessing paradigm in [15, 35] requires generating and sharing FSS keys corresponding to each complex-function gate in the circuit. This results in the offline MPC communication complexity being directly proportional to the total key size for a batch of FSS secret sharings.

One straightforward method to generate a batch of $m$ FSS for a function class is to perform $m$ calls to the underlying FSS scheme. This approach, known as the default batching technique, has been utilized in all previous works that incorporate FSS as a building block. Consequently, the communication complexity for all FSS-based applications scales proportionally to $O(m * [\mathsf{key\text{-}size}])$, where $[\mathsf{key\text{-}size}]$ is the key size required for secret sharing a single function from the underlying function class. Several FSS constructions for function classes, such as multi-point [21] and disjoint d-dimensional intervals [32], are constructed from a combination of more primitive FSS constructions for point functions and d-dimensional intervals, respectively. Therefore, developing an efficient FSS batching technique for point functions and d-dimensional intervals would directly contribute to improved FSS constructions for multi-point and union of interval function classes.

Hence, a natural question to consider is whether it is possible to secret share a batch of $m$ functions from a function class more efficiently than naive batching, thereby avoiding the multiplicative $m$ factor in the joint key size. This work provides an affirmative answer to this question for the class of point function, which can be employed in a number of cryptographic applications including oblivious transfer extension [10], private set intersection [23], and privacy-preserving machine learning [35]. An FSS scheme for the class of point functions is also called a *distributed point function* (DPF) scheme, and these will be the focus of current work with the goal of designing more efficient FSS batching schemes.

### 1.1 Our Contributions

We initiate the study of batch function secret sharing for point functions based on public-key assumptions. Our constructions are based on a variant of the learning with parity (LPN) assumption and any two-party homomorphic secret sharing (HSS) scheme, which can be thought of as a two-party analog of fully homomorphic encryption. An HSS scheme for a program $P$ allows a dealer to secret share an input $x$ with two parties, who can evaluate additive shares of $P(x)$ without any interaction. We assume an HSS scheme for restricted multiplication straight-line (RMS) programs, which informally allow four types of operations: load inputs into memory, add two memory values, multiply an input with a memory value, and output a memory value. HSS for RMS programs with negligible error probability and exponential message space size are known from public key assumptions including Learning with Errors [16] and Decisional Composite Residuosity [44, 41].

This work introduces two $m$-batch FSS constructions for point functions based on a sparse LPN assumption with structured noise, any HSS scheme for RMS program with negligible error probability and exponential message space size and the Quadratic Residuosity assumption. When instantiating our construction using the HSS scheme proposed in [41] with domain $\{0,1\}^n$ and range $\{0,1\}$ and $\{0,1\}^t$ (for any $t$), the generated keys have size $O(mn + \ell(\varepsilon + \sqrt{mn} \cdot \lambda))$ and $O(m(n+t) + \ell(\varepsilon + \sqrt{mn} \cdot \lambda))$ respectively, where $\varepsilon, \lambda$ are statistical and public security parameters, $k$ is the dimension parameter of the LPN assumption, and $\ell$ is a parameter of the HSS scheme of [41] that is related to the size of HSS input encodings and in our construction is independent of $m$. The computation complexity of $\mathsf{Eval}$ in both the batched DPF constructions is dominated by the HSS evaluation cost for executing $O(n)$ RMS operations. Section 5 delves into the asymptotic

complexity and the concrete performance of our protocol.

Due to our approach relying on asymmetric operations (e.g., exponentiations), while the naive approach solely utilizes symmetric operations (e.g., calls to a PRG), our solution is significantly lower performance in terms of runtime. However, it offers markedly improved key-size efficiency for relevant parameters. This trade-off between key-size and runtime becomes particularly advantageous in specific scenarios, such as when sending keys over slow networks. In Table 2 and Table 3, we show a more concrete comparison of key sizes and the runtime of the proposed batch FSS constructions alongside the naive batching technique. Specifically, we show a reduction of $80\times$ in size compared to the naive approach and a $5\times$ increase in speed on slow networks (e.g., WAN with bandwidth 100Mbps) when executing and sending the compression of $m = 2^{21}$ DPF instances with $n = 128$ bits. However, our DPF evaluation process requires 0.68 seconds while the naive approach completes in just 0.0014 milliseconds. When we consider the typical context in which DPF is utilized, particularly in outsourcing scenarios where robust servers perform the evaluation (as observed in applications like privacy-preserving machine learning [35]), we find the trade-off between key-size efficiency and runtime to be reasonable.

The core building block of our construction is an HSS Private Input Compression scheme (HSSIC), which we believe may be of independent interest.

### 1.2 Technical Overview

An $m$-batched Function Secret Sharing (BFSS) is a straightforward extension of FSS. It also consists of (Gen,Eval) but with some slight modifications: Gen now receives a vector of point functions in $\mathcal{F}$, and Eval includes a new index parameter $i$ specifying which function among those provided to Gen is to be evaluated. Our BFSS construction relies on Homomorphic Secret Sharing (HSS), enabling two parties to perform computations on secret-shared private inputs without interaction. To understand the overarching concept of our BFSS construction, we provide a brief overview of HSS (a more detailed explanation of HSS is available in Section 2.4) and a simple HSS-based non-batch DPF construction (presented in Figure 1). HSS [16] typically involves three main procedures: HSS.Setup, HSS.Input, and HSS.Eval. The HSS.Setup generates a public key pk along with a pair of evaluation keys $(\mathsf{ek}_0, \mathsf{ek}_1)$. The HSS.Input takes pk and an input $x$ to produce a type of secret-shared inputs, enabling each party to compute the secret-share of the function $f(x)$ on them using HSS.Eval.

$\underline{\mathsf{Gen}(\boldsymbol{\alpha} \in \{0,1\}^n)}$:
Execute $(\mathsf{pk}, (\mathsf{ek}_0, \mathsf{ek}_1)) \leftarrow \mathsf{HSS.Setup}(1^\lambda)$.
Execute $(\llbracket \boldsymbol{\alpha} \rrbracket_0, \llbracket \boldsymbol{\alpha} \rrbracket_1) \leftarrow \mathsf{HSS.Input}(\mathsf{pk}, \boldsymbol{\alpha})$.
Output $\mathsf{k}_0 = (\mathsf{pk}, \mathsf{ek}_0, \llbracket \boldsymbol{\alpha} \rrbracket_0)$ and $\mathsf{k}_1 = (\mathsf{pk}, \mathsf{ek}_1, \llbracket \boldsymbol{\alpha} \rrbracket_1)$.
$\underline{\mathsf{Eval}(\sigma, \mathsf{k}_\sigma, \boldsymbol{x})}$:
Parse $\mathsf{k}_\sigma$ as $(\mathsf{pk}, \mathsf{ek}_\sigma, \llbracket \boldsymbol{\alpha} \rrbracket_\sigma)$.
Non-interactively execute $(\llbracket \neg \boldsymbol{x} \rrbracket_0, \llbracket \neg \boldsymbol{x} \rrbracket_1) \leftarrow \mathsf{HSS.Input}(\mathsf{pk}, \neg \boldsymbol{x})$.
Execute $f_\sigma \leftarrow \mathsf{HSS.Eval}(\sigma, \mathsf{ek}_\sigma, (\llbracket \neg \boldsymbol{x} \rrbracket_\sigma, \llbracket \boldsymbol{\alpha} \rrbracket_\sigma), P^{\mathsf{DPF}}_{n,\boldsymbol{x}})$.
Output $f_\sigma$.
RMS program $\underline{P^{\mathsf{DPF}}_{n,\boldsymbol{x}}(\neg \boldsymbol{x}, \boldsymbol{\alpha})}$ :
Set $h_i \leftarrow \neg x_i - \alpha_i$ if $x_i = 0$, and $h_i \leftarrow \alpha_i$ otherwise, for $i \in [n]$.
Output $\Pi_{i=0}^{n-1} h_i$.

Figure 1: Naive HSS-based DPF Construction.

In the following, we denote a vector $\boldsymbol{\alpha}$ using bold letters, while each bit of the vector is repre-

sented by a normal letter (e.g., $\alpha_i$). The single instance DPF described above involves $\mathsf{Gen}$ utilizing $\mathsf{HSS.Setup}$ and $\mathsf{HSS.Input}$ to generate HSS keys and a type of secret-shared inputs for the non-zero position $\boldsymbol{\alpha}$ of the point function. The $\mathsf{Eval}$ algorithm then employs the same public key $\mathsf{pk}$ to execute $\mathsf{HSS.Input}(\mathsf{pk}, \boldsymbol{x})$, which in turn feeds into $\mathsf{HSS.Eval}$. The $\mathsf{HSS.Eval}$ evalues the program $P^{\mathsf{DPF}}$ (in Figure 1) on the secret share of $\neg\boldsymbol{x}$ and $\boldsymbol{\alpha}$. The $P^{\mathsf{DPF}}$ computes $\Pi_{i=0}^{n-1}(1 \oplus x_i \oplus \alpha_i)$. Thus, $\mathsf{Eval}$ returns 1 if $\boldsymbol{x} = \boldsymbol{\alpha}$, and 0 otherwise, as desired.

In the BDPF approach, we repeat the $\mathsf{Gen}$ algorithm with a set of $m$ input vectors $\{\boldsymbol{\alpha}_i\}_{i\in[m]}$, where each $\boldsymbol{\alpha}_i$ has $n$ bits (i.e., $\boldsymbol{\alpha}_i \in \{0,1\}^n$. Thus, it is feasible to reuse the HSS keys $(\mathsf{pk}, \mathsf{ek}_0, \mathsf{ek}_1)$ generated from $\mathsf{HSS.Setup}$ since they are independent of the input. However, the batch FSS key size is still $O(mn\lambda)$ (for security parameter $\lambda$) - which is no better than naively batching any DPF scheme. To mitigate the influence of the $\lambda$ factor, we introduce an innovative technique for compressing the lengthy input vector to the HSS scheme based on Learning Parity with Noise (LPN) assumption. This technique, called HSS Private Input Compression ($\mathsf{HSSIC}$), comprises two algorithms: $\mathsf{HSSIC.BIn}$ and $\mathsf{HSSIC.Expand}$.

**HSS Input Compression.** We represent the input set as $\boldsymbol{u}$ as $\boldsymbol{\alpha}_1||\ldots||\boldsymbol{\alpha}_m$. The $\mathsf{HSSIC.BIn}$ aims to compress $\boldsymbol{u} \in (\{0,1\})^{mn}$ by computing $\boldsymbol{w} = A\boldsymbol{s} + \boldsymbol{e} + \boldsymbol{u} \pmod 2$, where $A$ is randomly sampled from the set of generating matrices of a linear code ensemble, and $\boldsymbol{e}$ is the random noise vector (Section 2.1 provides further details on selecting $A$ and $\boldsymbol{e}$). Leveraging the LPN assumption, disclosing $(\boldsymbol{w}, A)$ reveals nothing about the input $\boldsymbol{\alpha}$. Therefore, we can include them in the BFSS keys $(\rho_1, \rho_2)$ along with the secret share of $\boldsymbol{s}$. Importantly, rather than requiring $O(mn\lambda)$ in the naive batched FSS, the size of FSS's keys is now $O(\lambda + nm)$ as $w$ has a size of $nm$, and $A$ can be derived from a PRF seed.

Using the HSS compressed input $\boldsymbol{w}$, we develop $\mathsf{HSSIC.Expand}$, an expansion function that outputs the share of a HSS private input encoding $[\![z_i]\!]_\sigma$ of $z_i$, where $z_i$ encodes each bit $u_i$ of $\boldsymbol{u}$ along with a special noise $e_i$ for $i \in [mn]$. This process relies on the evaluation of an RMS program by each party on the $i^{th}$-bit compressed input $w_i$ and the secret-shared inputs $[\![\boldsymbol{s}]\!]$ and, resulting in the output $z_i = A_{i,*}\boldsymbol{s} + w_i$, where $A_{i,*}$ is the $i^{th}$ row of the matrix $A$. This allows $\mathsf{LSB}(z_i) = e_i + u_i \pmod 2$. Figure 4 presents the exact RMS program.

Because of the additional noise, we need to modify the evaluation algorithm in Figure 1. First, we want to operate on the least significant bits of two HSS private input value encodings $[\![x]\!]$ and $[\![y]\!]$. This can be done by choosing a sufficiently large modulo $M$ during HSS scheme instantiation to prevent any "wrap-around" occurrences while computing intermediary values during $\mathsf{HSS.Eval}$ execution. Second, we want to mitigate the noise introduced by $\mathsf{HSSIC}$ to our compressed input during DPF evaluation. To do this, we first focus on 1-bit output DPF, followed by demonstrating the extension to an arbitrary $t$-bit output DPF.

**1-bit Output DPF.** In the single instance of the DPF (Figure 1) for 1-bit output, we provide the encodings of $\boldsymbol{\alpha}_i, \boldsymbol{x}$ and an RMS program to $\mathsf{HSS.Eval}$ so we can compute the following, where $\mathsf{HD}$ is the hamming distance function.

$$f_{\boldsymbol{\alpha_i}}(\boldsymbol{x}) = \begin{cases} 1, & \text{if } \mathsf{HD}(\boldsymbol{x}, \boldsymbol{\alpha}_i) = 0 \\ 0, & \text{otherwise} \end{cases} \tag{1}$$

However, following the HSS input compression, each encoding of $\boldsymbol{\alpha}_{i\in[m]}$ is "compressed" to $[\![\boldsymbol{\alpha}_i^\star]\!] := \big([\![z_{n\cdot i}]\!], [\![z_{n\cdot i+1)}]\!]_\sigma, \ldots, [\![z_{n\cdot i+n-1}]\!]\big)$. Thus, $\mathsf{HD}(\boldsymbol{x}, \boldsymbol{\alpha}_i)$ might not equal $\mathsf{HD}(\boldsymbol{x}, \boldsymbol{\alpha}_i^\star)$. Therefore, Equation (1) no longer holds for $\boldsymbol{\alpha}_i'$. Fortunately, we design $\mathsf{HSSIC}$ with a structured noise $e$ drawn from regular noise distribution (see 'last bit correctness' in Definition 8) to guarantee $\mathsf{HD}(\boldsymbol{\alpha}_i', \boldsymbol{\alpha}_i) \leq$

1. Hence, we rely on HSS.Eval to address the above issue. Specifically, we design an RMS program (see Figure 6) which takes input $\boldsymbol{z}^\star$ and $\boldsymbol{x}$, and computes the following function.

$$f^\star_{\boldsymbol{\alpha}_i^\star}(\boldsymbol{x}) = \begin{cases} 1, & \text{if } \mathsf{HD}(\boldsymbol{x}, \boldsymbol{\alpha}_i^\star) \leq 1 \\ 0, & \text{otherwise} \end{cases}$$

However, the function $f^\star$ does not operate as a point function since it may yield multiple $\boldsymbol{x}$ for which $f^\star$ outputs 1. To handle this, we modify the input of HSS.Input in the DPF generation, taking $H(\boldsymbol{\alpha}_i)$ instead of of $\boldsymbol{\alpha}_i$, where $H$ is a random oracle. This adjustment is made because when $\boldsymbol{x} = \boldsymbol{\alpha_i}$, $H(\boldsymbol{x}) = H(\boldsymbol{\alpha}_i) = H(\boldsymbol{\alpha}'_i)$, resulting in $f'$ returning 1. For cases where $\boldsymbol{x} \neq \boldsymbol{\alpha}'_i$, the functionality of our HSSIC ensure that $\mathsf{HD}(H(\boldsymbol{x}), H(\boldsymbol{\alpha}_i)) > 1$ except with negligible probability, indicating that that $f'$ returns 0.

**Multiple-bit Output BDPF.** To extend our 1-bit output BDPF to arbitrary-bit output $\boldsymbol{\beta}_i \in \{0,1\}^t$, we introduce a new building block called GM Ciphertext Compression (GMCC). This enables the compression of the lengthy GM ciphertext of $\boldsymbol{\beta}_i$ into a compressed structure cc, which mostly has the same size of the input $\boldsymbol{\beta}_i$. At a high level, this compression can be achieved using a pseudorandom function. Additionally, as a result of this compression, we introduce a new function within GMCC. This function is used to expand the compressed structure cc and retrieve the ciphertext. We present detailed information regarding the construction of GMCC in Section 3.2.

**Analysis on the key length of BDPF.** By harnessing our two building blocks, HSSIC and GMCC, we achieve a notable asymptotic advantage in key-size efficiency over the naive batch DPF, which previously stood as the most efficient scheme in terms of key size (for example, the naive $t$-bit output $m$-instance DPF has the key size $O(m(n+t)\kappa)$). Employing the most efficient HSS construction [41] with the HSS ciphertext size $\ell \in O\left(\frac{\lambda}{\lambda - \varepsilon - n \cdot \log_2(d)}\right)$ where $d$ is the locality parameter for the Local Code used (as explained in Section 2.1), the key size complexities for both our 1-bit and $t$-bit variants (as elaborated in Section 5.1.1) are as follows.

- For 1-bit output BDPF: $O(\ell \cdot (\varepsilon + k \cdot \lambda) + \kappa + m \cdot n)$
- For $t$-bit output BDPF: $O(\ell \cdot (\varepsilon + k \cdot \lambda) + \kappa + m \cdot (n + t))$

Therefore, as the number of DPF executions increases (i.e., the value of $m$ increases), the performance gap between our approach and the naive solution widens. We demonstrate an improvement ranging from $3 - 80\times$ when $m$ increases from $2^{13}$ to $2^{19}$.

### 1.3 Related Work

In cryptographic literature, introducing and constructing a batched version of a fundamental cryptographic primitive is often aimed at amortizing computational costs, communication costs, or other efficiency measures. In this study, we define and construct a non-trivial BFSS scheme with the aim of amortizing the size of keys generated by the Gen algorithm, potentially resulting in improvements in communication costs for protocols requiring batched FSS. Specifically, we develop BFSS schemes for point functions (DPF).

To the best of our knowledge, no existing BFSS scheme for any function family achieves an asymptotic improvement in key sizes over the naive approach where the key generation algorithm of a non-batched FSS scheme is simply invoked $m$ times, and the multiple keys generated are used to form the pair of keys for the batched scheme. This indicates that the efficiency of currently

known BFSS constructions (prior to our work) relies solely on the efficiency of their non-batched counterparts.

There have been notable advancements in FSS schemes tailored for point function families (distributed point function – DPF), including [33], [11], and [14]. Among these schemes, the most efficient in terms of key size exhibits a complexity of $O(n \cdot \kappa)$, where $\kappa$ represents the security parameter and $n$ denotes the bit-length of domain elements in the encoded point function. Consequently, the most efficient batched FSS scheme for the point function family entails a key size complexity of $O(m \cdot n \cdot \kappa)$, where $m$ indicates the batch size of encoded point functions.

**Hybrid HSS Techniques.** The idea of compressing HSS ciphertexts was first suggested in [13] using a "hybrid HSS" approach. Here the high level idea is to use the output of a log-depth PRG (or PRF) to mask the plaintext input of the HSS scheme, and generate HSS ciphertext of just the randomness input to the PRG to get HSS input for the compressed scheme. To evaluate any RMS program $R$, the evaluating parties can first use the HSS scheme to evaluate log depth PRG on the input randomness, xor it with the masked input and then evaluate RMS program $R$ on this decompressed input. It should be noted that the computation complexity of this compressed input HSS scheme does increase compared to the original program $R$ by a factor proportional to the depth complexity of the assumed PRG. While this overall approach closely resembles our HSS compression scheme (except for the occasional errors due to noise), our use of the sparse LPN assumption leads to a protocol with improved computation complexity. Specifically, decompressing each HSS input for program $R$ in our scheme can be done by just taking a linear combinations of a constant subset of inputs given the constant locality property of the sparse LPN matrix. Hence, the computation complexity of our resulting batch DPF scheme is just the cost to decompress HSS inputs, plus the computation complexity of the non-batch DPF scheme - leading to essentially no overhead in the computation overhead for our compressed scheme variant. This improves over a batch DPF construction that can be obtained using hybrid HSS framework of [13] even when assuming a 'local PRG' [2, 4] (which has constant depth), which gives a constant computational overhead over the RMS program of a point function.

**Ciphertext Compression.** In this work, we define and design a homomorphic ciphertext compression (HSSIC) scheme to compress HSS input encodings/ciphertexts more efficiently. Analogous but different notion of compressing ciphertexts has been described in previous works [29, 28, 17, 26]. However, none of these are suitable for our batch DPF construction based on HSS for RMS programs. For example, the ciphertext compression primitive proposed in [29, 28] allows one to compress an encrypted dataset. Further to decompress the ciphertext the algorithm outputs the message that was encrypted initially in plaintext. Whereas, in our HSSIC scheme, we require an HSSIC.Expand algorithm - that inputs a compressed ciphertext for a sequence of messages, which can be split into ciphertexts of individual messages. Brakerski et al. [17] introduce a rate-1 linear homomorphic ciphertext compression scheme based on a range of public key assumptions including Quadratic Residuosity and Learning with Errors. However, their scheme only supports performing linear operations over the compressed ciphertext inputs, whereas our HSSIC scheme requires evaluation of arbitrary RMS programs over the compressed ciphertext inputs.

## 2 Preliminaries

We use $[n]$ to denote $\{0, 1, \ldots, n-1\}$, $\mathbb{J}_N$ denote the elements of $\mathbb{Z}_N$ with Jacobi symbol 1. We use $\varepsilon, \kappa, \lambda$ to denote statistical, computational, and public security parameters, respectively.

The vector is denoted by $\boldsymbol{\alpha}$, and $\alpha_i$ represents the $i$-th bit of $\boldsymbol{\alpha}$. We use $\neg\boldsymbol{v}$ to denote $\boldsymbol{v}' \in \mathbb{Z}_2^n$ where $\boldsymbol{v}'_i = \neg\boldsymbol{v}_i$ for every $i \in [n]$. Let $\mathsf{LSB}(x)$ denote the least significant bit of $x$'s binary expansion. For $n \in \mathbb{N}$ and $\boldsymbol{x} \in (\mathbb{Z}^+)^n$, we define $\boldsymbol{y} = \mathsf{LSB}(\boldsymbol{x}) \in \mathbb{Z}_2^n$ is such that $y_i = \mathsf{LSB}(x_i)$ for every $i \in [n]$. The relationship between mod 2 addition and integer addition is summarized in Appendix B.

Matrices are denoted by capital letters. For a matrix $A \in \mathbb{Z}_M^{n\times m}$, we use $A_{j,*}$ to denote the vector $\boldsymbol{v} \in \mathbb{Z}_M^m$, where $v_i = A_{j,i}$ for every $i \in [m]$. For $n \in \mathbb{N}, M \in \mathbb{Z}^{\geq 2}$ and $A, B \subseteq \mathbb{Z}_M^n$, we use $A \otimes B$ to denote the set $\{\ a \mid\mid b \mid a \in A \ \wedge\ b \in B\ \}$. For $n \in \mathbb{N}, M \in \mathbb{Z}^{\geq 2}$ and $A$, we use $A^{\otimes 0}$ to denote $\emptyset$, $A^{\otimes 1}$ to denote $A$, $A^{\otimes 2}$ to denote $A \otimes A$, $A^{\otimes 3}$ to denote $A \otimes A \otimes A$, and so on.

The function $\mathsf{HD}\colon \{0,1\}^n \times \{0,1\}^n \to \mathbb{Z}^+$ outputs the hamming distance between its two arguments. For a vector $\boldsymbol{v}$ of length $n$ that contains GM ciphertexts, we use $\mathsf{GM.Dec}(\mathsf{sk}, \boldsymbol{v})$ to denote $\mathsf{GM.Dec}(\mathsf{sk}, v_0) \mid\mid \ \ldots\ \mid\mid \mathsf{GM.Dec}(\mathsf{sk}, v_{n-1})$.

### 2.1 Learning Parity with Noise Assumption

In the construction of our $\mathsf{HSSIC}$ Scheme, we utilize the Learning Parity with Noise (LPN) assumption [6] over $\mathbb{Z}_2$. In simple terms, the decision version of this assumption states that an adversary cannot distinguish between $(A, A\boldsymbol{s} + \boldsymbol{e})$ and $(A, \boldsymbol{b})$, where $A$ is randomly sampled from the set of generating matrices of a linear code ensemble, $\boldsymbol{s}$ is a uniformly sampled vector over $\mathbb{Z}_2$, $\boldsymbol{e}$ is a noise vector (or error vector) sampled from a distribution over $\mathbb{Z}_2$-vectors, and $\boldsymbol{b}$ is a uniformly random vector over $\mathbb{Z}_2$. We present the formal definition of the decision version of the LPN assumption below.

**Definition 1** (LPN Assumption [6])**.** *Let $\mathcal{D} = \{\mathcal{D}_{k,q}\}_{k,q\in\mathbb{N}}$ denote a family of distributions over $\mathbb{Z}_2$, such that for any $k, q \in \mathbb{N}$, $\mathsf{Im}(\mathcal{D}_{k,q}) \subseteq \mathbb{Z}_2^q$. Let $\mathbf{C}$ be a probabilistic code generation algorithm such that $\mathbf{C}(k, q)$ outputs a (description of a) matrix $A \in \mathbb{Z}_2^{q\times k}$. For dimension $k = k(\kappa)$, number of samples (or block length) $q = q(\kappa)$, the $(\mathcal{D}, \mathbf{C}) - \mathsf{LPN}(k, q)$ assumption states that*

$$\{(A, \boldsymbol{b}) \mid A \xleftarrow{\$} \mathbf{C}(k, q), \boldsymbol{e} \xleftarrow{\$} \mathcal{D}_{k,q}, \boldsymbol{s} \xleftarrow{\$} \mathbb{Z}_2^k, \boldsymbol{b} \leftarrow A \cdot \boldsymbol{s} + \boldsymbol{e}\}$$
$$\overset{c}{\approx} \{(A, \boldsymbol{b}) \mid A \xleftarrow{\$} \mathbf{C}(k, q), \boldsymbol{b} \xleftarrow{\$} \mathbb{Z}_2^q\}$$

Note that the definition does not specify the code generation algorithm or the noise distribution $\mathcal{D}$. The standard LPN assumption typically involves $\mathbf{C}(k, q)$ outputting a matrix $A$ sampled uniformly from $\mathbb{Z}_2^{q\times k}$ and the noise distribution being the Bernoulli distribution $\mathsf{Ber}_r^q(\mathbb{Z}_2)$, where every component of noise vector $\boldsymbol{e}$ is equal to 1 with probability $r$ and 0 with probability $1 - r$. However, employing the standard LPN assumption may sometimes result in inefficient constructions, prompting researchers to investigate variants of the LPN assumption that could yield more efficient constructions. These efficiency enhancements typically stem from the utilization of different code generation algorithms and noise distributions.

In this work, we use a Local Code $\mathbf{C}_\mathsf{L}$ as our probabilistic code generation algorithm as previously proposed in [1].A local linear code with constant locality $\mathbf{C}_\mathsf{L}$ and parameter $d$ is characterized by each row of the generator matrix being $d$-sparse, meaning it contains $d$ non-zero entries. In this work, we choose $d = 10$ as recommended in [3], as it provides a sufficient level of security against well-known attacks with a suitably large dimension $k$.

For the noise vector distribution $\mathcal{D}$, we select the regular noise distribution $\mathsf{RHW} = \{\mathsf{RHW}_{\tau,q}\}_{\tau,q\in\mathbb{N}}$, which is a uniform distribution with support $\mathrm{supp}(\mathsf{RHW}_{\tau,q}) \subseteq \mathbb{Z}_2^q$. In this distribution, every vector in $\mathrm{supp}(\mathsf{RHW}_{\tau,q}) \subseteq \mathbb{Z}_2^q$ is subdivided into $\tau$ consecutive sub-vectors of length $\lfloor q/\tau \rfloor$, where each sub-vector contains exactly one non-zero component. The regular noise distribution has been commonly utilized in previous works [8, 9, 45] when implementing the LPN assumption, and its concrete security has recently been extensively examined in[40].

## 2.2 GM Encryption Scheme

The Goldwasser-Micali (GM) encryption scheme [34] is a public-key cryptosystem that involves generating public and private keys from large prime numbers and employs quadratic non-residues to obscure plaintext information during encryption. In this work, we utilize a variant of the GM cryptosystem introduced in [39]. This particular variant is necessary as it enables threshold decryption of a ciphertext when the primes $p$ and $q$ generated during the key generation phase satisfy the conditions $p \equiv 3 \pmod 4$ and $q \equiv 3 \pmod 4$. A description of this variant is provided in Figure 8. We use the randomized algorithm $\mathsf{GenModulus}(1^\lambda)$ to generate $(N, p, q)$, where $\lambda$ is the security parameter. This algorithm samples two random primes $p$ and $q$, each of length $\ell = \ell(\lambda)$, and outputs $(N, p, q)$.

## 2.3 Distributed Discrete Log

In our batched Distributed Point Function (DPF), we adopt the Distributed Discrete Log ($\mathsf{DDL}^{\mathsf{GM}}$) technique [41, 26] to transform the multiplicative shares of the encrypted bit from the GM scheme to additive shares. The $\mathsf{DDL}^{\mathsf{GM}}$ method is presented in Figure 9. Notably, when $a_0, a_1 \in \mathbb{Z}_N^*$ satisfy $a_1/a_0 = (-1)^b$, we have $\mathsf{DDL}^{\mathsf{GM}}(a_0) \oplus \mathsf{DDL}^{\mathsf{GM}}(a_1) = b$.

## 2.4 Homomorphic Secret Sharing

Homomorphic Secret Sharing (HSS) is a cryptographic technique that allows parties to share and jointly perform computations on secret data without revealing additional information. We adopt the following standard HSS definitions from [41, 16].

**Definition 2** (HSS Syntax)**.** *An HSS scheme consists of the following PPT algorithms:*

- $(\mathsf{pk}, (\mathsf{ek}_0, \mathsf{ek}_1)) \leftarrow \mathsf{HSS.Setup}(1^\lambda)$*: Given a security parameter* $1^\lambda$*, the setup algorithm outputs a public key* $\mathsf{pk}$ *and a pair of evaluation keys* $(\mathsf{ek}_0, \mathsf{ek}_1)$*.*
- $([\![x]\!]_0, [\![x]\!]_1) \leftarrow \mathsf{HSS.Input}(\mathsf{pk}, x)$*: Given public key* $\mathsf{pk}$ *and private input value* $x \in \mathbb{Z}_M$*, the input algorithm outputs private input encoding* $([\![x]\!]_0, [\![x]\!]_1)$*.*
- $(y_{0,\sigma}, \ldots, y_{t-1,\sigma}) \leftarrow \mathsf{HSS.Eval}(\sigma, \mathsf{ek}_\sigma, ([\![x_0]\!]_\sigma, \ldots, [\![x_{n-1}]\!]_\sigma), P)$*: On input a party index* $\sigma \in \{0, 1\}$*, evaluation key* $\mathsf{ek}_\sigma$*, vector of* $n$ *private input encodings and a program* $P \in \mathcal{P}$ *with* $n$ *input values, the evaluation algorithm outputs a vector of additive shares of output values* $y_0, \ldots, y_{t-1}$*.*

**Definition 3** (HSS Security)**.** *An HSS scheme* ($\mathsf{HSS.Setup}$, $\mathsf{HSS.Input}$, $\mathsf{HSS.Eval}$) *for a class of programs* $\mathcal{P}$ *is termed secure if it satisfies all the following properties:*

- ***Correctness****: For any security parameters* $\lambda \in \mathbb{N}$*, program* $P \in \mathcal{P}$ *with* $t$ *outputs, for private inputs* $x_0, x_1, \ldots, x_{n-1}$*, the following holds for all* $i \in [t]$*:*

$$Pr\Big(y_{i,0} + y_{i,1} = P(x_0, x_1, \ldots, x_{n-1})[i]\Big) \geq 1 - \mathsf{negl}(\lambda)$$

*where*

$$\begin{aligned} (\mathsf{pk}, (\mathsf{ek}_0, \mathsf{ek}_1)) &\leftarrow \mathsf{HSS.Setup}(1^\lambda) \\ ([\![x_j]\!]_0, [\![x_j]\!]_1) &\leftarrow \mathsf{HSS.Input}(\mathsf{pk}, x_j) && (\text{ for } j \in [n]) \\ (y_{0,\sigma}, \ldots, y_{t-1,\sigma}) &\leftarrow \mathsf{HSS.Eval}(\sigma, \mathsf{ek}_\sigma, ([\![x_0]\!]_\sigma, \ldots, [\![x_{n-1}]\!]_\sigma), P) && (\text{for } \sigma \in \{0, 1\}) \end{aligned}$$

- ***Privacy****: For any security parameter* $\lambda \in \mathbb{N}$*, party index* $\sigma \in \{0, 1\}$ *and non-uniform adversary* $\mathcal{A}$ *(of size polynomial in the security parameter* $\lambda$*), the following holds for all sufficiently large* $\lambda$*, where* $\mathsf{Exp}^{HSS,\mathsf{sec}}_{\mathcal{A},\sigma,0}(\lambda)$ *for* $b \in \{0, 1\}$ *is as defined in Figure 2.*

$\mathsf{Exp}^{\mathsf{HSSIC},\mathsf{sec}}_{\mathcal{A},\sigma,b}(\lambda)$:

$(x_0, x_1, \mathsf{state}) \leftarrow \mathcal{A}(1^\lambda)$

$(\mathsf{pk}, (\mathsf{ek}_0, \mathsf{ek}_1)) \leftarrow \mathsf{HSS.Setup}(1^\lambda)$

$(\llbracket x_b \rrbracket_0, \llbracket x_b \rrbracket_1) \leftarrow \mathsf{HSS.Input}(\mathsf{pk}, x_b)$

$b' \leftarrow \mathcal{A}(\mathsf{state}, \mathsf{pk}, \mathsf{ek}_\sigma, \llbracket x_b \rrbracket_\sigma)$

**return** $b'$

Figure 2: Security of HSS

$$\left| Pr\left[\mathit{Exp}^{HSS,\mathsf{sec}}_{\mathcal{A},\sigma,0}(\lambda) = 1\right] - Pr\left[\mathit{Exp}^{HSS,\mathsf{sec}}_{\mathcal{A},\sigma,1}(\lambda) = 1\right] \right| \leq \mathit{negl}(\lambda)$$

***Restricted Multiplication Straight-line Programs*** *In this work, we assume HSS schemes for the class of restricted multiplication straight-line (RMS) programs, which are known from a range of public-key assumptions including Learning with Errors [16] and Decisional Composite Residuosity [44, 41]. An RMS program is an arithmetic circuit with a set of inputs values ($\llbracket x \rrbracket$) where each memory value ($\langle x \rangle$) is bounded by some integer parameter $M$ and the circuit supports the following operations:*

- $\langle x \rangle \leftarrow \mathit{Load}(\llbracket x \rrbracket)$*: Load an input into memory*
- $\llbracket z \rrbracket \leftarrow \mathit{Add}(\llbracket x \rrbracket, \llbracket y \rrbracket)$*: Add two input values to output another input value where* $z = x + y$
- $\langle z \rangle \leftarrow \mathit{Add}(\langle x \rangle, \langle y \rangle)$*: Add two memory values to output another memory value where* $z = x + y$
- $\langle z \rangle \leftarrow \mathit{ResMult}(\llbracket x \rrbracket, \langle y \rangle)$*: Multiply an input and a memory value to output another memory value where* $z = x * y$*; its restricted since we cannot arbitrarily multiply memory values*
- $x \leftarrow \mathit{Output}(\langle x \rangle)$*: Output a memory value*

*If any memory value of the program ever exceeds the bound $M$, its output is undefined.*

**Remark:** Traditional $\mathsf{HSS.Eval}$ syntax only supports outputting memory values encodings. However, when the RMS program outputs just a linear combination of the inputs with scalars $\in \{0, 1\}$, all known HSS schemes [16, 44, 41] also supports outputting input encoding. To simplify our presentation, we will overload the $\mathsf{HSS.Eval}$ function to output input encodings when the input RMS program involves only $\mathsf{Add}$ operations.

## 2.5 Batched Function Secret Sharing

We introduce the definition of Batched Function Secret Sharing (BFSS) and Batched Distributed Point Function (BDPF) which is almost entirely based on the definition of Function Secret Sharing and Distributed Point Function given in [14], except for some small adaptations made to address that we are dealing with a batch of functions instead of a single one.

A *function family* is defined by a pair $\mathcal{F} = (P_\mathcal{F}, E_\mathcal{F})$, where $P_\mathcal{F} \subseteq \{0,1\}^*$ is an infinite collection of function descriptions $\hat{f}$, and $E_\mathcal{F} \colon P_\mathcal{F} \times \{0,1\}^* \to \{0,1\}^*$ is a polynomial-time algorithm defining the function described by $\hat{f}$. Concretely, each $\hat{f} \in P_\mathcal{F}$ describes a corresponding function $f \colon D_f \to R_f$ defined by $f(x) = E_\mathcal{F}(\hat{f}, x)$.

We assume by default that $D_f = \{0,1\}^n$ for a positive integer $n$ and always require $R_f$ to be a finite Abelian group, denoted by $\mathbb{G}$. When there is no risk of confusion, we will sometimes write

$f$ instead of $\hat{f}$ and $f \in \mathcal{F}$ instead of $\hat{f} \in P_{\mathcal{F}}$. We assume that $\hat{f}$ includes an explicit description of both $D_f$ and $R_f$ as well as a size parameter $S_f$.

We assume that $\hat{f}$ includes an explicit description of both $D_f$ and $R_f$. We let $\mathsf{Leak}(\hat{f})$ capture partial information about $\hat{f}$ that can be leaked by the scheme. When $\mathsf{Leak}$ is omitted it is understood to output $D_f$ and $R_f$.

**Definition 4** (Batched FSS: Syntax)**.** *A 2-party batched function secret sharing (BFSS) scheme is a pair of algorithms* $(\mathsf{Gen}, \mathsf{Eval})$ *with the following syntax:*

- $\mathsf{Gen}(1^\lambda, \hat{\boldsymbol{f}})$ *is a PPT algorithm, which on input* $1^\lambda$ *(security parameter) and* $\hat{\boldsymbol{f}} \in (\{0,1\}^*)^m$ *(a vector of length* $m$ *containing function descriptions) outputs a pair of keys* $(k_0, k_1)$*. We assume that every* $\hat{f}_i \in \hat{\boldsymbol{f}}$ *explicitly contains an input length* $1^n$*, group description* $\mathbb{G}$*, and size parameter* $S$ *(see above).*
- $\mathsf{Eval}(\sigma, k_\sigma, i, x)$*: is a polynomial-time evaluation algorithm, which on input* $\sigma \in \{0,1\}$ *(party index),* $(k_\sigma, i)$ *(key and index defining* $f_i \colon \{0,1\}^n \to \mathbb{G}$*) and* $x \in \{0,1\}^n$ *(input for* $f_i$*) outputs a group element* $y_i \in \mathbb{G}$ *(the* $i$*-th share of* $f_i(x)$*).*

**Definition 5** (Batched FSS: Security)**.** *Let* $\mathcal{F} = (P_{\mathcal{F}}, E_{\mathcal{F}})$ *be a function family and* $\mathsf{Leak} \colon \{0,1\}^* \to \{0,1\}^*$ *be a function specifying the allowable leakage. A 2-party 1-secure FSS for* $\mathcal{F}$ *with leakage* $\mathsf{Leak}$ *is a pair* $(\mathsf{Gen}, \mathsf{Eval})$ *as in 4, satisfying the following requirements.*

- ***Correctness:*** *For all vectors* $\hat{\boldsymbol{f}}$ *of length* $m$ *where* $\hat{f}_i \in P_{\mathcal{F}}$ *describes* $f_i \colon \{0,1\}^n \to \mathbb{G}$ *for every* $i \in [m]$*, and every* $x \in \{0,1\}^n$*, if* $(k_0, k_1) \leftarrow \mathsf{Gen}(1^\lambda, \hat{\boldsymbol{f}})$ *then* $Pr[\mathsf{Eval}(0, k_0, i, x) + \mathsf{Eval}(1, k_1, i, x) = f_i(x)] \geq 1 - \mathsf{negl}(\lambda)$ *for every* $i \in [m]$*.*
- ***Secrecy:*** *For any corrupted party* $c \in \{0,1\}$*, there exists a PPT algorithm* $\mathsf{Sim}$ *(simulator), such that for every sequence* $\hat{\boldsymbol{f}}^{(0)}, \hat{\boldsymbol{f}}^{(1)}, \ldots$ *of polynomial-size vectors where every function description* $\hat{\boldsymbol{f}}^{(i)}_j$ *is a polynomial-size function description from* $P_{\mathcal{F}}$*, the outputs of the following experiments* $\mathsf{Real}$ *and* $\mathsf{Ideal}$ *are computationally indistinguishable:*
  - $\mathsf{Real}(1^\lambda)$ : $(k_0, k_1) \leftarrow \mathsf{Gen}(1^\lambda, \hat{\boldsymbol{f}}^{(\lambda)})$*;* *Output* $k_c$*.*
  - $\mathsf{Ideal}(1^\lambda)$ : *Output* $\mathsf{Sim}(1^\lambda, \mathsf{Leak}(\hat{\boldsymbol{f}}^{(i)}))$*.*

**Definition 6** (Batched Distributed Point Function)**.** *A point function* $f_{\boldsymbol{\alpha},\beta}$*, for* $\boldsymbol{\alpha} \in \{0,1\}^n$ *and* $\beta \in \mathbb{G}$*, is defined to be the function* $f \colon \{0,1\}^n \to \mathbb{G}$ *such that* $f(\boldsymbol{\alpha}) = \beta$ *and* $f(\boldsymbol{x}) = 0$ *for all* $\boldsymbol{x} \neq \boldsymbol{\alpha}$*. A Batched Distributed Point Function (BDPF) is a Batched FSS for the family of all point functions, with leakage function* $\mathsf{Leak}(\hat{\boldsymbol{f}}) = (1^n, \mathbb{G})$*.*

# 3 Our Building Blocks

## 3.1 HSS Private Input Compression

**Definition 7** (HSS Private Input Compression syntax)**.** *An HSS private input compression scheme consists of the following PPT algorithms:*

- $(\rho_0, \rho_1, \boldsymbol{z}) \leftarrow \mathit{HSSIC.BIn}(\mathit{pk}, \boldsymbol{m} = (m_0, m_1, \ldots, m_{n-1}))$*: takes as input the public encryption key and a vector* $\boldsymbol{m} \in (\{0,1\})^n$ *s.t* $\eta | n$*, and it outputs two compressed private input structures* $\rho_0, \rho_1$ *and the vector* $\boldsymbol{z} \in \mathbb{Z}^n$*.*

- $[\![z_i]\!]_\sigma \leftarrow \mathsf{HSSIC.Expand}(\sigma, \mathsf{ek}_\sigma, \rho_\sigma, i)$: *Expands the compressed private input structure $\rho_\sigma$ into an individual HSS private input share encoding of $m_i$.*

*The size of the compressed ciphertext $\rho_i$ divided by $n$ gives the rate of compression - which we would ideally want to be as close to 1 as possible. Next we present the formal security definition for our* HSSIC *scheme followed by a novel construction based on the LPN assumption with structured noise.*

**Definition 8** (HSS Private Input Compression security; $(\delta, \eta) - \mathsf{HSSIC}$)**.** *For any integers $\delta, \eta > 0$ and given any HSS scheme in Definition 2, an HSS private input compression scheme is said to be $(\delta, \eta)$ secure if:*

- ***Last bit correctness****: Let $\boldsymbol{e}_i \in \mathbb{Z}_2^\eta$ denote a standard unit vector with 1 in the $i$-th position, $\mathbf{0} \in \mathbb{Z}_2^\eta$ be the zero vector, and $E = \{\mathbf{0}, \boldsymbol{e}_0, \ldots, \boldsymbol{e}_{\eta-1}\}$. The following must hold*

$$\mathsf{LSB}(\boldsymbol{z}) \oplus \boldsymbol{m} \in E^{\otimes n/\eta}$$

- ***Bounded decompressed messages****: for any $i \in [n]$:*

$$0 \leq z_i \leq \delta$$

- ***Private input equivalency****: For any security parameters $\lambda \in \mathbb{N}$, program $P \in \mathcal{P}$ with $t$ outputs, for private inputs $x_0, \ldots, x_{n'-1}$, input $m \in \{0,1\}^n$, and any subset $J = \{j_1, \ldots, j_l\} \subseteq [n]$ the following holds for all $i \in [t]$:*

$$Pr\Big(y_{i,0} + y_{i,1} = P(z_{j_0}, \ldots, z_{j_l}, x_0, \ldots, x_{n'-1})[i]\Big) \geq 1 - \mathsf{negl}(\lambda)$$

  *where:*

  - $(\mathsf{pk}, (\mathsf{ek}_0, \mathsf{ek}_1)) \leftarrow \mathsf{HSS.Setup}(1^\lambda)$
  - $(\rho_0, \rho_1, \boldsymbol{z}) \leftarrow \mathsf{BIn}(\mathsf{pk}, \boldsymbol{m})$
  - $[\![z'_i]\!]_\sigma \leftarrow \mathsf{Expand}(\sigma, \mathsf{pk}, \rho_\sigma, i) \forall \sigma \in \{0,1\}, i \in [n]$.
  - $([\![x_j]\!]_0, [\![x_j]\!]_1) \leftarrow \mathsf{HSS.Input}(\mathsf{pk}, x_j) \forall j \in [n']$
  - $(y_{0,\sigma}, \ldots, y_{t-1,\sigma}) \leftarrow \mathsf{HSS.Eval}(\sigma, \mathsf{ek}_\sigma, ([\![z'_{j_0}]\!]_\sigma, \ldots, [\![z'_{j_j}]\!]_\sigma, [\![x_0]\!]_\sigma, \ldots, [\![x_{n'-1}]\!]_\sigma), P) \forall \sigma \in \{0,1\}$

  *It implies that the* HSS.Eval *gives the same output, whether its input is* HSS.Input *input encodings $[\![z_i]\!]$ or if its given the corresponding outputs of* HSSIC.BIn *$[\![z'_i]\!]$. Intuitively, it captures that the HSS scheme gives the correct output when its inputs are generated from the HSS input compression scheme* HSSIC *instead of using* HSS.Input *to generate corresponding input encodings.*

- ***Compressed private input indistinguishability****: For each $\sigma \in \{0,1\}$ and non-uniform adversary $\mathcal{A}$ (of size polynomial in the security parameter $\lambda$), it holds that*

$$\left| Pr\left[\mathsf{Exp}^{\mathsf{HSSIC},\mathsf{sec}}_{\mathcal{A},\sigma,0}(\lambda) = 1\right] - Pr\left[\mathsf{Exp}^{\mathsf{HSSIC},\mathsf{sec}}_{\mathcal{A},\sigma,1}(\lambda) = 1\right] \right| \leq \mathsf{negl}(\lambda)$$

  *for all sufficiently large $\lambda$, where $\mathsf{Exp}^{\mathsf{HSSIC},\mathsf{sec}}_{\mathcal{A},\sigma,0}(\lambda)$ for $b \in \{0,1\}$ is as defined in Figure 3.*

$\underline{\mathsf{Exp}^{\mathsf{HSSIC,sec}}_{\mathcal{A},\sigma,b}(\lambda)}$:
$(\boldsymbol{m}^{(0)}, \boldsymbol{m}^{(1)}, \mathsf{state}) \leftarrow \mathcal{A}(1^\lambda)$
$(\mathsf{pk}, (\mathsf{ek}_0, \mathsf{ek}_1)) \leftarrow \mathsf{HSS.Setup}(1^\lambda)$
$(\rho_0, \rho_1, \boldsymbol{z}) \leftarrow \mathsf{HSSIC.BIn}(\mathsf{pk}, \boldsymbol{m}^{(b)})$
$b' \leftarrow \mathcal{A}(\mathsf{state}, \mathsf{pk}, \mathsf{ek}_\sigma, \rho_\sigma)$
**return** $b'$

Figure 3: Security of HSSIC

Let $\mathbf{C}_\mathsf{L}$ be probabilistic code generation
Let $\mathsf{RHW}_{\tau,n}$ be regular noise distribution
Let $k$ define the LPN assumption dimension parameter.
$\underline{\mathsf{HSSIC.BIn}(\mathsf{pk}, \boldsymbol{m} = (m_1, m_1, \ldots, m_n))}$:
Sample $A \xleftarrow{\$} \mathbf{C}_\mathsf{L}(k, n)$, $\boldsymbol{e} \xleftarrow{\$} \mathsf{RHW}_{\tau,n}$, and $\boldsymbol{s} \in_R \mathbb{Z}_2^k$.
Compute $\boldsymbol{w} = A\boldsymbol{s} + \boldsymbol{e} + \boldsymbol{m} \pmod 2$.
Compute $z = A\boldsymbol{s} + \boldsymbol{w}$
Execute $(\llbracket \boldsymbol{s} \rrbracket_0, \llbracket \boldsymbol{s} \rrbracket_1) \leftarrow \mathsf{HSS.Input}(\mathsf{pk}, \boldsymbol{s})$.
Output $\rho_0 = (A, \boldsymbol{w}, \llbracket \boldsymbol{s} \rrbracket_0)$, $\rho_1 = (A, \boldsymbol{w}, \llbracket \boldsymbol{s} \rrbracket_1)$, and $\boldsymbol{z}$.
$\underline{\mathsf{HSSIC.Expand}(\sigma, \mathsf{ek}_\sigma, \rho_\sigma, i)}$:
Parse $\rho_\sigma$ as $(A, \boldsymbol{w}, \llbracket \boldsymbol{s} \rrbracket_\sigma)$.
Non-interactively execute $(\llbracket w_i \rrbracket_0, \llbracket w_i \rrbracket_1) \leftarrow \mathsf{HSS.Input}(\mathsf{pk}, w_i)$.
Execute $\llbracket z_i' \rrbracket_\sigma \leftarrow \mathsf{HSS.Eval}(\sigma, \mathsf{ek}_\sigma, (\llbracket \boldsymbol{s} \rrbracket_\sigma, \llbracket w_i \rrbracket_\sigma), P^{\mathsf{Ex}}_{k,i,A})$.
Output $\llbracket z_i' \rrbracket_\sigma$.
RMS program $\underline{P^{\mathsf{Ex}}_{k,i,A}(\boldsymbol{s}, w_i)}$:
Set $y \leftarrow w_i$.
Add $s_j$ to $y$ if $A_{i,j} = 1$, otherwise do nothing, for $j \in [k]$.
Output $y$.

Figure 4: Our HSSIC construction based on LPN Variant

**Our Protocol.** We adhere to the high-level technique overview outlined in Section 1.2. The compression algorithm $\mathsf{HSSIC.BIn}$ begins by calculating $w = A\boldsymbol{s} + \boldsymbol{e} + \boldsymbol{m} \pmod 2$, which can be seen as a ciphertext encrypting the binary vector $\boldsymbol{m}$ using the LPN variant. Subsequently, the algorithm encodes each component of the secret vector $\boldsymbol{s}$ as an HSS input, concluding by producing the compressed structures $\rho_0 = (A, \boldsymbol{w}, \llbracket \boldsymbol{s} \rrbracket_0)$ and $\rho_1 = (A, \boldsymbol{w}, \llbracket \boldsymbol{s} \rrbracket_1)$. Note that each $\rho_{\sigma \in \{0,1\}}$ does not reveal any information about $\boldsymbol{m}$ or $\boldsymbol{s}$, thanks to the security of HSS and the LPN assumption.

When provided with a compressed structure $\rho_\sigma$ and an HSS evaluation key $\mathsf{ek}_\sigma$, a party $P_\sigma$ can employ $\mathsf{HSSIC.Expand}$ to calculate an HSS input encoding share $\llbracket z_i \rrbracket_\sigma$ that encodes $m_i$ as its least significant bit. This is achieved by utilizing $\mathsf{HSS.Eval}$ to execute the RMS program $P^{\mathsf{Ex}}_{k,i,A}$ on imput $\llbracket \boldsymbol{s} \rrbracket_\sigma$ and $\llbracket w_i \rrbracket_\sigma$. The output $z_i$ of $P^{\mathsf{Ex}}_{k,i,A}$ is governed by the following expression when computed over the integers $z_i = A_{i,*}\boldsymbol{s} + w_i$, which means that $\mathsf{LSB}(z_i) = e_i + m_i$, since $\mathsf{LSB}(z_i) = A_{i,*}\boldsymbol{s} + w_i \pmod 2$. Since the noise vector $\boldsymbol{e}$ introduces structure noise deemed acceptable by our definition, as long as we operate over a large enough HSS modulo we have that $\llbracket \boldsymbol{z_i} \rrbracket_\sigma$ correctly encodes $m_i$.

**Theorem 9.** *Given the $(\mathcal{D}, \mathbf{C}) - \mathsf{LPN}(k, n)$ assumption where $\mathbf{C}$ has constant locality $d$ and structured noise distribution $\mathsf{RHW} = \{\mathsf{RHW}_{\tau,q}\}_{\tau,q \in \mathbb{N}}$, and any HSS construction, Figure 4 is a secure $(d+1, \lfloor n/\tau \rfloor)$-HSSIC scheme.*

Let $H\colon \{0,1\}^\kappa \times \{0,1\}^* \to \mathbb{J}_N^t$ be a keyed random oracle.

$\mathsf{GMCC.Comp}_{n,t}(\mathsf{pk}, \mathsf{sk}, \boldsymbol{w} = (w_0, \ldots, w_{n-1}))$:

Parse $\mathsf{pk}$ as $N$.
Sample a key $\mathsf{k_H} \in_R \{0,1\}^\kappa$ to $H$.
Compute $C \in \mathbb{J}_N^{n \times t}$, where $C_{i,*} \leftarrow H_{\mathsf{k_H}}(i)$ for every $i \in [n]$.
Compute $r_i = \mathsf{GM.Dec}(\mathsf{sk}, C_{i,*})$, for every $i \in [n]$.
Compute $e_i = w_i \oplus r_i$, for every $i \in [n]$.
Output $\mathsf{cc} = (\mathsf{k_H}, \boldsymbol{e})$, where $\boldsymbol{e} = (e_0, \ldots, e_{n-1})$.

$\mathsf{GMCC.Expand}_t(\mathsf{pk}, \mathsf{cc}, i)$:

Parse $\mathsf{pk}$ as $N$ and $\mathsf{cc}$ as $(\mathsf{k_H}, \boldsymbol{e})$.
Compute $\boldsymbol{c} \leftarrow H_{\mathsf{k_H}}(i)$.
Compute $\mathsf{ct}_j \leftarrow c_j \cdot (-1)^{e_{i,j}} \pmod N$, for every $j \in [t]$.
Output $(\mathsf{ct}_0, \ldots, \mathsf{ct}_{t-1})$.

Figure 5: Our GMCC Construction.

We present the formal proof of the above theorem in Appendix C.

### 3.2 GM Ciphertext Compression

To extend our concept of constructing a DPF supporting the range $\{0,1\}$ to one supporting the range $\{0,1\}^*$ while preserving our efficiency advantage over naive batched DPF constructions, we need a method for a dealer $D$ to send a sequence of GM ciphertexts encrypting a sequence of binary words to multiple receivers. To accomplish this, we introduce the GM Ciphertext Compression scheme, consisting of two algorithms ($\mathsf{GMCC.Comp}, \mathsf{GMCC.Expand}$).

The algorithm $\mathsf{GMCC.Comp}$ takes a GM cryptosystem secret key $\mathsf{sk}$ and a vector of binary words $\boldsymbol{w} \in (\{0,1\}^t)^n$ as input and outputs a compressed structure $\mathsf{cc}$. This compressed structure $\mathsf{cc}$, along with the public key $\mathsf{pk}$ associated with the secret key $\mathsf{sk}$ provided to $\mathsf{GMCC.Comp}$ to generate $\mathsf{cc}$, and an index $i \in [n]$, can then be provided to $\mathsf{GMCC.Expand}$. The $\mathsf{GMCC.Expand}$ outputs a vector of GM ciphertexts $(\mathsf{ct}_0, \ldots, \mathsf{ct}_{t-1})$ such that $\mathsf{GM.Dec}(\mathsf{sk}, \mathsf{ct}_0) \,||\, \ldots \,||\mathsf{GM.Dec}(\mathsf{sk}, \mathsf{ct}_{t-1}) = w_i$. This allows a receiver to compute a vector of GM ciphertexts that encrypt the binary word $w_i$ in a bitwise fashion. Appendix D.1 formally describes the definition of $\mathsf{GMCC}$.

**Protocol.** To implement our $\mathsf{GMCC}$, we utilize ciphertext compression techniques, which have been extensively employed in previous studies such as [41] and are widely recognized within the multi-party computation community. The pioneering use of these methods that we are familiar with dates back to the work of [37].

Let $N$ be a public key sampled by $\mathsf{GM.Gen}$. It is a well-known fact that $\mathbb{J}_N$ defines the set of valid ciphertexts for the GM cryptosystem when encryption is performed using public key $N$. Therefore, we can transmit GM ciphertexts of pseudorandom messages (bits) in a compressed way by sampling and transmitting a key $\mathsf{k_H} \in \{0,1\}^\kappa$ to a keyed random oracle $H\colon \{0,1\}^\kappa \times \{0,1\}^* \to \mathbb{J}_N$. This allows us to evaluate $H_{\mathsf{k_H}}(x)$ and receive as output a vector of GM ciphertexts of pseudorandom bits. We leverage this idea to construct the compression scheme, enabling us to send a vector of GM-encrypted binary words as defined by Definitions 12 and 13

The compression algorithm $\mathsf{GMCC.Comp}$ begins by parsing the public key as $N$ and sampling a key $\mathsf{k_H}$ for the function $H\colon \{0,1\}^\kappa \times \{0,1\}^* \to \mathbb{J}_N$, which will later be included as part of the compressed structure $\mathsf{cc}$. For each word $w_i$ provided as input for compression, we evaluate $H_{\mathsf{k_H}}$ at position $i$ and decrypt the output $H_{\mathsf{k_{prf}}}(i)$ using $\mathsf{GM.Dec}$ (where $\mathbb{J}_N$ defines the set of valid

ciphertexts), resulting in a binary word $r_i$ as output. Note that $r_i$ is a random word.

Having obtained $r_i$, we calculate $e_i = w_i \oplus r_i$, which is also random since $r_i$ is random. Then, we define the vector $\boldsymbol{e} = (e_0, \ldots, e_{n-1})$ and include this vector as part of the compressed structure $\mathsf{cc}$. Next, we explore how it is possible to utilize $\mathsf{cc}$ and the homomorphic property of GM ciphertexts to compute the ciphertexts of all the compressed words.

The $\mathsf{GMCC.Expand}$ begins by parsing the public key as $N$ and the compressed structure $\mathsf{cc}$ as $(\mathsf{k_H}, \boldsymbol{e})$. It then computes a vector $\boldsymbol{c} = (\mathsf{ct}'0, \ldots, \mathsf{ct}'t-1)$ of length $t$ of GM ciphertexts, which is equal to $C_{i,*}$, where $C$ is the same matrix computed during the execution of $\mathsf{GMCC.Comp}$ that generated $\mathsf{cc}$. Next, it calculates $\mathsf{ct}_i \leftarrow \mathsf{ct}'i \cdot (-1)^{e_i} \pmod N$, resulting in a GM ciphertext of $ei, j \oplus r_{i,j} = w_{i,j}$. Consequently, the output vector $(\mathsf{ct}_0, \ldots, \mathsf{ct}_{t-1})$ is a vector of GM ciphertexts encrypting the word $w_i$ bit-by-bit. The correctness and security of our $\mathsf{GMCC}$ are provided in Appendix D.2.

# 4 Batched Distributed Point Function Construction

## 4.1 1-bit Output Batched DPF

In the naive HSS-based DPF construction described in Section 1.2, the DPF key generation algorithm executes the HSS setup algorithm, encodes every bit of the vector $\boldsymbol{\alpha}_i$ as an HSS input, and then includes the HSS input encodings shares and respective evaluation keys in the batch DPF keys.

However, this approach does not directly work when the vector $\boldsymbol{\alpha}_i$ is compressed using $\mathsf{HSSIC}$ primitives - due to occasional errors when decompressing. To solve this in a computation efficient manner, the key idea is compress the vector $\{H(\boldsymbol{\alpha}_i)\}_{i\in[m]}$ using the $\mathsf{HSSIC}$ scheme (where $H$ is a random oracle), and the compressed structures $\rho_0, \rho_1$ are included in the DPF keys in place of the uncompressed bitwise HSS input encodings of every element in $\{\boldsymbol{\alpha}_i\}_{i\in[m]}$. The importance of the random oracle becomes evident when considering our intuitive explanation for the 1-bit output BDPF construction and its proof of correctness.

To evaluate the encoded point function indexed by $i$, the evaluation algorithm first uses $\mathsf{HSSIC.Expand}$ to obtain the following vector of HSS input encoding shares $[\![\boldsymbol{\alpha}^\star]\!]_\sigma = [\![z_{n\cdot i}]\!]_\sigma, [\![z_{n\cdot i+1}]\!]_\sigma, \ldots, [\![z_{n\cdot i+n-1}]\!]$. Note that by the definition of $(\delta, k) - \mathsf{HSSIC}$, we can pick values for $k$ such that $n|k$, and because of the way we build vector $\boldsymbol{u}$ and the Last Bit Correctness property of $\mathsf{HSSIC}$, we can be sure that $\mathsf{HD}(\mathsf{LSB}(\boldsymbol{\alpha}^\star), H(\boldsymbol{\alpha}_i)) \leq 1$.

Next our goal is to design an efficient RMS program to check if two input strings have hamming distance less than or equal to 1 with all but negligible probability. For any two strings $a, b \in \{a, b\}^n$, for odd $n$, let $c_i = [a_1 \stackrel{?}{=} b_1] \ldots [a_{i-1} \stackrel{?}{=} b_{i-1}][a_{i+1} \stackrel{?}{=} b_{i+1}] \ldots [a_n \stackrel{?}{=} b_n]$ i.e. bit $c_i$ equals 1 iff all bits of $a$ and $b$ match ignoring the $i^{th}$ bit. Then we have $\mathsf{HD}(a, b) \leq 1 \iff [\oplus_i c_i \stackrel{?}{=} 1]$. Each $c_i$ can be computed using $(n-2)$ RMS multiplications, leading to an RMS program of $O(n^2)$ complexity to check if two input strings have hamming distance less than or equal to 1. Hence, this natural approach does not lead to a computation efficient batch DPF. Our final construction is presented below:

**Our Final Construction**: After the HSS input expansion, we encode every bit of $\neg \boldsymbol{x}^H$ as HSS inputs and proceed to execute the RMS program $P_n^{\mathsf{DPF}}$ (presented in Figure 6) providing $\neg \boldsymbol{x}^H$ and $\boldsymbol{\alpha}^\star$ as arguments. This RMS program intuitively checks if the two inputs strings have hamming distance less than or equal to 1, and furthermore it does so only using a linear number of RMS operations! We analyze the exact computational complexity of this Protocol in Section 5. To be

more precise, this program $P_n^{\mathsf{DPF}}$ outputs three values $(f_0, f_1, f_2)$, such that following holds

$$\mathsf{LSB}(f_0) \oplus \mathsf{LSB}(f_1) \oplus \mathsf{LSB}(f_2) = 1 \iff \boldsymbol{x} = \boldsymbol{\alpha_i}$$

, except with negligible probability for large enough $n$.

From the Bounded Decompressed Messages property of $\mathsf{HSSIC}$ and the definition of $\mathsf{HSS}$, we know we can pick a large enough modulo to operate over when instantiating the $\mathsf{HSS}$ scheme, such that $f_{j,0}$ and $f_{j,1}$ are 2-out-of-2 additive secret shares over the integers, except with negligible probability. This implies the following identity is true except with negligible probability

$$\begin{aligned} y_0 \oplus y_1 &= \bigoplus_{j\in[3]} \mathsf{LSB}(f_{j,0}) \oplus \bigoplus_{j\in[3]} \mathsf{LSB}(f_{j,1}) \\ &= \mathsf{LSB}(f_0) \oplus \mathsf{LSB}(f_1) \oplus \mathsf{LSB}(f_2) \end{aligned}$$

Based on this, it is straightforward to see that $y_0 \oplus y_1 = 1 \iff \boldsymbol{x} = \boldsymbol{\alpha}_i$, except with negligible probability.

We now focus on analyzing the RMS program $P_n^{\mathsf{DPF}}$ and discussing how and why its outputs $(f_0, f_1, f_2)$ are governed by the previously described identity.

First, notice that by the expression that governs the value of $e_i$, we know that $\mathsf{LSB}(e_i) = 1 \iff x_i^H = \alpha_i^\star$, for all $i \in [n]$. Next, notice that we can think of the expression as dividing the vector $\boldsymbol{e}$ in 3 consecutive sub-vectors of equal size (here we assume the vector is divisible by 3 for simplicity's sake) and multiplying all the components of the subvector $i_0$ as $\left(\Pi_{j=i_0\cdot n/3}^{(i_0+1)\cdot n/3-1} e_j\right)$. So, $\mathsf{LSB}(f_0)$ is going to be equal to 1 when all the components of the first and second sub-vectors are equal to 1, $\mathsf{LSB}(f_1)$ is going to be equal to be 1 when all the components of the second and third sub-vectors are equal to 1, and $\mathsf{LSB}(f_2)$ is going to be equal to be 1 when all the components of the third and first sub-vectors are equal to 1.

Now, let the index of the point function being evaluated be $i$ and $\boldsymbol{\alpha}_i$ be the non-zero point encoding this function. Next, let the point being evaluated be $\boldsymbol{x}$ and suppose $\boldsymbol{x} = \boldsymbol{\alpha}_i$. Then, from the definition of $\mathsf{HSSIC}$ we have $\mathsf{HD}(\boldsymbol{x}^H, \boldsymbol{\alpha}^\star) \leq 1$. This implies that exactly 3 or 2 of the 3 sub-vectors will have all its components equal to 1, which in both cases leads to $\mathsf{LSB}(f_0) \oplus \mathsf{LSB}(f_1) \oplus \mathsf{LSB}(f_2) = 1$. Next, suppose $\boldsymbol{x} \neq \boldsymbol{\alpha}_i$. Since $x^H$ and $H(\boldsymbol{\alpha}_i)$ are independent random values and $\mathsf{HD}(H(\boldsymbol{\alpha}_i), \boldsymbol{\alpha}^\star) \leq 1$, we will only have exactly 3 or 1 of the sub-vectors with all-1 components only with negligible probability, which means we will have $\mathsf{LSB}(f_0) \oplus \mathsf{LSB}(f_1) \oplus \mathsf{LSB}(f_2) = 0$ except with negligible probability. Therefore, we can conclude that $\mathsf{LSB}(f_0) \oplus \mathsf{LSB}(f_1) \oplus \mathsf{LSB}(f_2) = 1 \iff \boldsymbol{x} = \boldsymbol{\alpha}_i$, except with negligible probability.

**Theorem 10.** *For statistical security parameter $\varepsilon$, $n \in \mathbb{Z}^+$, given any $(\delta, \eta)$-HSSIC scheme with modulus $M > 2^\varepsilon(\delta+1)^{2n/3}$, Figure 6 is a secure batched distributed point function with single bit output and $n$ bit input for $3(\varepsilon+1)/2 \leq n \leq \eta$ in the random oracle model.*

We present the proof of the above theorem in Appendix E.

### 4.2 Multiple-bit Output Batch DPF

To extend our previous 1-bit output BDPF scheme to allow for arbitrary binary string output $\boldsymbol{\beta}_i \in \{0,1\}^t$, we employ the GM cryptosystem to incorporate the information of $\boldsymbol{\beta}_i$ into the DPF keys in a privacy-preserving manner. This enables each evaluating party to compute GM ciphertexts for each DPF output $\boldsymbol{\beta}_{i\in[m]}$ without revealing the values to them. For communication efficiency, we leverage our $\mathsf{GMCC}$ (so is the GM variant described in Section 2.2) as the building block, which

$H : \{0,1\}^* \to \{0,1\}^n$ is a random oracle

$\underline{\mathsf{Gen}((\boldsymbol{\alpha_i})_{i=0\ldots m-1})}$:

Execute $(\mathsf{pk}, (\mathsf{ek}_0, \mathsf{ek}_1)) \leftarrow \mathsf{HSS.Setup}(1^\lambda)$.

Set $\boldsymbol{u} = H(\boldsymbol{\alpha}_0) \mid\mid H(\boldsymbol{\alpha}_1) \mid\mid \ldots \mid\mid H(\boldsymbol{\alpha}_{m-1}) \in \{0,1\}^{n \cdot m}$.

Execute $(\rho_0, \rho_1, \boldsymbol{z}) \leftarrow \mathsf{HSSIC.BIn}(\mathsf{pk}, \boldsymbol{u})$.

Output $\mathsf{k}_0 = (\mathsf{pk}, \mathsf{ek}_0, \rho_0)$ and $\mathsf{k}_1 = (\mathsf{pk}, \mathsf{ek}_1, \rho_1)$.

$\underline{\mathsf{Eval}(\sigma, \mathsf{k}_\sigma, i, \boldsymbol{x})}$:

Parse $\mathsf{k}_\sigma$ as $(\mathsf{pk}, \mathsf{ek}_\sigma, \rho_\sigma)$.

Execute $[\![\alpha_j^\star]\!]_\sigma \leftarrow \mathsf{HSSIC.Expand}(\sigma, \mathsf{ek}_\sigma, \rho_\sigma, n \cdot i + j) \forall j \in [n]$.

Compute $\boldsymbol{x}^H = H(\boldsymbol{x})$.

Non-interactively execute $([\![\neg \boldsymbol{x}^H]\!]_0, [\![\neg \boldsymbol{x}^H]\!]_1) \leftarrow \mathsf{HSS.Input}(\mathsf{pk}, \neg \boldsymbol{x}^H)$

Execute $(f_{0,\sigma}, f_{1,\sigma}, f_{2,\sigma}) \leftarrow \mathsf{HSS.Eval}(\sigma, \mathsf{ek}_\sigma, ([\![\neg \boldsymbol{x}^H]\!]_\sigma, [\![\boldsymbol{\alpha}^\star]\!]_\sigma), P_n^{\mathsf{DPF}})$

Output $y_\sigma = \bigoplus_{j \in [3]} \mathsf{LSB}(f_{j,\sigma})$.

RMS program $\underline{P_n^{\mathsf{DPF}}(\neg \boldsymbol{x}^H, \boldsymbol{\alpha}^\star)}$:

Set $e_i = \neg x_i^H + \alpha_i^\star, \forall i \in [n]$.

Set $f_{i_0} = \left(\Pi_{j=i_0 \cdot n/3}^{(i_0+1)\cdot n/3-1} e_j\right) \cdot \left(\Pi_{j=i_1 \cdot n/3}^{(i_1+1)\cdot n/3-1} e_j\right)$ for $(i_0, i_1) \in \{(0,1),(1,2),(2,0)\}$

Output $f_0, f_1, f_2$.

Figure 6: Our 1-bit Output Batched DPF Construction.

allows to compress the GM ciphertexts of $\boldsymbol{\beta}_{i\in[m]}$ and perform a "decrypt-and-multiply" operation utilized in constructing HSS schemes.

The proposed construction is presented in Figure 7, which is similar to our 1-bit output DPF construction, but in addition it has two function calls to $\mathsf{GM.Gen}$ and $\mathsf{GMCC.Comp}$ in its $\mathsf{Gen}$ and $\mathsf{Eval}$ functions respectively. The $\mathsf{GM.Gen}$ generates the public/secret key pair $(\mathsf{pk}_{\mathsf{GM}}, \mathsf{sk})$, while the $\mathsf{GMCC.Comp}$ constructs a compressed structure $\mathsf{cc}$ for $\{\boldsymbol{\beta}_i\}_{i\in[m]}$, enabling evaluators to compute GM ciphertexts $(\mathsf{ct}_0, \ldots, \mathsf{ct}_{t-1})$ of $\{\beta_i\}_{i\in[m]}$ in bitwise fashion during the DPF evaluation algorithm using $\mathsf{GMCC.Expand}$ with $\mathsf{pk}_{\mathsf{GM}}$. As a result, we include both $\mathsf{pk}_{\mathsf{GM}}$ and $\mathsf{cc}$ in the DPF keys. Furthermore, the DPF keys must include an HSS input encoding share of the GM private key $d = \mathsf{sk}$ corresponding to $\mathsf{pk}_{\mathsf{GM}}$. We also make a minor modification to the RMS program $P_n^{\mathsf{DPF}}$ such that it now returns $(f_0 \cdot d, f_1 \cdot d, f_2 \cdot d)$ as its output instead of $(f_0, f_1, f_2)$. Using the shares returned by $\mathsf{HSS.Eval}$ allows us to compute 2-out-of-2 additive share $h_{\sigma\in\{0,1\}}$ of $f_+ \cdot d = (f_0 + f_1 + f_2) \cdot d$ over the integers.

Using the GM variant described in Section 2.2, we have:

$$\mathsf{ct}_j^{h_0} \cdot \mathsf{ct}_j^{h_1} = \mathsf{ct}_j^{f_+ \cdot d} = (\mathsf{ct}_j^d)^{f_+} = ((-1)^{\beta_{i,j}})^{f_+} = (-1)^{\beta_{i,j} \cdot \mathsf{LSB}(f_+)} \pmod N$$

We can interpret $\mathsf{ct}_j^{h_0}, \mathsf{ct}_j^{h_1}$ as 2-out-of-2 multiplicative shares modulo $N$ such that $(-1)^{\beta_{i,j} \cdot \mathsf{LSB}(f_+)} \pmod N$. Thus, leveraging the property of $\mathsf{DDL}^{\mathsf{GM}}$ discussed in Section 2.3, we have $\mathsf{DDL}^{\mathsf{GM}}(\mathsf{ct}_j^{h_0}) \oplus \mathsf{DDL}^{\mathsf{GM}}(\mathsf{ct}_j^{h_1}) = \beta_{i,j} \cdot \mathsf{LSB}(f_+)$, indicating that $\mathsf{DDL}^{\mathsf{GM}}$ serves as a procedure enabling the conversion of multiplicative shares of $(-1)^{\beta_{i,j} \cdot \mathsf{LSB}(f_+)} \pmod N$ to additive shares of $\beta_{i,j} \cdot \mathsf{LSB}(f_+)$.

Given that $\mathsf{LSB}(f_+) = \mathsf{LSB}(f_0 + f_1 + f_2) = \mathsf{LSB}(f_0) \oplus \mathsf{LSB}(f_1) \oplus \mathsf{LSB}(f_2)$, the following holds for all $j \in [t]$.

$$z_{j,0} \oplus z_{j,1} = \begin{cases} \beta_{i,j} & , \text{ if } \boldsymbol{x} = \boldsymbol{\alpha}_i \\ 0 & , \text{ otherwise} \end{cases}$$

$H : \{0,1\}^* \to \{0,1\}^n$ is a random oracle

$\underline{\mathsf{Gen}(\{\boldsymbol{\alpha}_0, \ldots, \boldsymbol{\alpha}_{m-1}, \boldsymbol{\beta}_0, \ldots, \boldsymbol{\beta}_{m-1}\})}$:

Execute $(\mathsf{pk}_{\mathsf{GM}}, \mathsf{sk}) \leftarrow \mathsf{GM.Gen}(1^\lambda)$,

parse $\mathsf{pk}_{\mathsf{GM}}$ as $N$ and $\mathsf{sk}$ as $d \in \mathbb{Z}_N$.

Execute $(\mathsf{pk}, (\mathsf{ek}_0, \mathsf{ek}_1)) \leftarrow \mathsf{HSS.Setup}(1^\lambda)$.

Set $\boldsymbol{u} = H(\boldsymbol{\alpha}_0) \mid\mid H(\boldsymbol{\alpha}_1) \mid\mid \ldots \mid\mid H(\boldsymbol{\alpha}_{m-1}) \in \{0,1\}^{n \cdot m}$.

Execute $(\rho_0, \rho_1, \boldsymbol{z}) \leftarrow \mathsf{HSSIC.BIn}(\mathsf{pk}_{\mathsf{HSS}}, \boldsymbol{u})$.

Execute $\mathsf{cc} \leftarrow \mathsf{GMCC.Comp}(\mathsf{sk}, (\boldsymbol{\beta}_0, \ldots, \boldsymbol{\beta}_{m-1}))$.

Execute $(\llbracket d \rrbracket_0, \llbracket d \rrbracket_1) \leftarrow \mathsf{HSS.Input}(\mathsf{pk}_{\mathsf{HSS}}, d)$.

Output $\mathsf{k}_0 = (\mathsf{pk}_{\mathsf{HSS}}, \mathsf{ek}_0, \rho_0, \mathsf{pk}_{\mathsf{GM}}, \mathsf{cc}, \llbracket d \rrbracket_0)$ and
$\mathsf{k}_1 = (\mathsf{pk}_{\mathsf{HSS}}, \mathsf{ek}_1, \rho_1, \mathsf{pk}_{\mathsf{GM}}, \mathsf{cc}, \llbracket d \rrbracket_1)$.

$\underline{\mathsf{Eval}(\sigma, \mathsf{k}_\sigma, i, \boldsymbol{x})}$:

Parse $\mathsf{k}_\sigma$ as $(\mathsf{pk}_{\mathsf{HSS}}, \mathsf{ek}_\sigma, \rho_\sigma, \mathsf{pk}_{\mathsf{GM}}, \mathsf{cc}, \llbracket d \rrbracket_\sigma)$.

Execute $\llbracket \alpha_j^\star \rrbracket_\sigma \leftarrow \mathsf{HSSIC.Expand}(\sigma, \mathsf{ek}_\sigma, \rho_\sigma, n \cdot i + j) \forall \in [n]$.

Compute $\boldsymbol{x}^H = H(\boldsymbol{x})$.

Non-interactively execute $(\llbracket \neg \boldsymbol{x}^H \rrbracket_0, \llbracket \neg \boldsymbol{x}^H \rrbracket_1) \leftarrow \mathsf{HSS.Input}(\mathsf{pk}_{\mathsf{HSS}}, \neg \boldsymbol{x}^H)$

Execute $(w_{0,\sigma}, w_{1,\sigma}, w_{2,\sigma}) \leftarrow \mathsf{HSS.Eval}(\sigma, \mathsf{ek}_\sigma, (\llbracket \neg \boldsymbol{x}^H \rrbracket_\sigma, \llbracket \boldsymbol{\alpha}^\star \rrbracket_\sigma, \llbracket d \rrbracket_\sigma), P_n^{\mathsf{DPF}})$.

Set $h_\sigma \leftarrow \sum_{j \in [3]} w_{j,\sigma}$.

Execute $(\mathsf{ct}_0, \ldots, \mathsf{ct}_{t-1}) \leftarrow \mathsf{GMCC.Expand}(\mathsf{pk}_{\mathsf{GM}}, \mathsf{cc}, i)$.

Compute $z_{j,\sigma} = \mathsf{DDL}^{\mathsf{GM}}(\mathsf{ct}_j^{h_\sigma})$, for every $j \in [t]$.

Output $\boldsymbol{y}_\sigma = z_{0,\sigma} \mid\mid \ldots \mid\mid z_{t-1,\sigma}$.

RMS program $\underline{P_n^{\mathsf{DPF}}(\neg \boldsymbol{x}^H, \boldsymbol{\alpha}^\star, d)}$:

Set $e_i = \neg x_i^H + \alpha_i^\star$, for every $i \in [n]$.

Set $f_{i_0} = \left(\Pi_{j=i_0 \cdot n/3}^{(i_0+1) \cdot n/3 - 1} e_j\right) \cdot \left(\Pi_{j=i_1 \cdot n/3}^{(i_1+1) \cdot n/3 - 1} e_j\right)$, for $(i_0, i_1) \in \{(0,1), (1,2), (2,0)\}$.

Output $f_0 \cdot d, f_1 \cdot d, f_2 \cdot d$ as $w_0, w_1, w_2$, respectively.

Figure 7: Our Multiple-bit Output Batch DPF Construction.

Which means that the final output of $\mathsf{Eval}$:

$$\boldsymbol{y}_0 \oplus \boldsymbol{y}_1 = \begin{cases} \boldsymbol{\beta}_i & , \text{ if } \boldsymbol{x} = \boldsymbol{\alpha}_i \\ \{0\}^t & , \text{ otherwise} \end{cases}$$

Which proves the correctness of the proposed scheme in Figure 7. The secrecy of this batch DPF scheme follows a similar proof sketch as the secrecy proof in Theorem 10. The DPF key of each party contains an $\mathsf{HSSIC}$ ciphertext share, GM compressed ciphertext and an HSS input encoding - each of which is simulatable, making the adversary's entire view simulatable in the secrecy game. This gives us the following result:

**Theorem 11.** *For statistical security parameter $\varepsilon$, $n \in \mathbb{Z}^+$, given any $(\delta, \eta)$-HSSIC scheme with modulus $M > 2^\varepsilon (\delta + 1)^{2n/3}$ and a GM ciphertext compression scheme, Figure 7 is a secure batched distributed point function with $t$ bit output and $n$ bit input for $3(\varepsilon + 1)/2 \leq n \leq \eta$ in the random oracle model.*

# 5 Performance Analysis and Comparison

This section delves into an performance analysis of the two BFSS schemes for the family of the point functions. For the asymptotic complexity, we treat the HSS scheme as a black box, followed by instantiating it with the state-of-the-art HSS scheme [41]. In the concrete analysis, we utilize the key sizes from [41], standard parameter values to compute the size of the keys and the number of exponentiations generated/required by our FSS schemes.

## 5.1 Communication Efficiency

### 5.1.1 Asymptotic Complexity

Analyzing the asymptotic key size of the proposed BDPF constructions involves evaluating the asymptotic size of each component.The first component is the HSS public key $\mathsf{pk}$ and the HSS evaluation keys $\mathsf{ek}_0, \mathsf{ek}_1$. As we treat the HSS scheme as a black box, we denote the bit-length of these components using variables $|\mathsf{pk}_{\mathsf{HSS}}|$, $|\mathsf{ek}_0|$, and $|\mathsf{ek}_1|$, representing the bit-lengths of $\mathsf{pk}$, $\mathsf{ek}_0$, and $\mathsf{ek}_1$ respectively.

The second component is $\rho_\sigma = (A, \boldsymbol{w}, [\![s]\!]_\sigma)$, where $A \in \mathbb{Z}_2^{m\cdot n\times k}$, $\boldsymbol{w} \in \mathbb{Z}_2^{m\cdot n}$, and $[\![s]\!]_\sigma$ is a vector of length $k$. Indeed, the matrix $A$ is typically generated from PRG seeds of size $\kappa$. Therefore, we can conclude that the communication cost of $A$ is $O(\kappa)$. Since $[\![s]\!]_\sigma$ is a length-$k$ vector of HSS input value encoding share, we cannot ascertain its size without assuming a specific HSS scheme. Hence, we introduce the variable $|\mathsf{in}_\sigma|$ to represent a bit-length of an HSS input value encoding share, and thus, we claim that $[\![s]\!]_\sigma$ has a bit-length of $k \cdot |\mathsf{in}_\sigma|$. In summary, the size of $\rho_\sigma$ is $|\rho_\sigma| \approx m \cdot n + \kappa + k \cdot |\mathsf{in}_\sigma| \in O(m \cdot n + \kappa + k \cdot |\mathsf{in}_\sigma|)$.

The DPF key for our $t$-bit output BDPF contains two additional components compared to the 1-bit output version. Initially, we incorporate $\mathsf{GMCC}$, where the key size complexities of the GM public key $\mathsf{pk}_{\mathsf{GM}}$ and a single HSS input value encoding share $[\![d]\!]_\sigma$ are $O(\lambda)$ and $|\mathsf{in}_\sigma|$ respectively. Secondly, the compressed structure $\mathsf{cc} = (\mathsf{k}_{\mathsf{H}}, \boldsymbol{e})$ comprises a key $\mathsf{k}_{\mathsf{H}}$ of size $O(\kappa)$ for a keyed RO $H$ and $\boldsymbol{e} \in (\{0,1\}^t)^m$.

Having analyzed the size of each key component, we can determine that the key complexity of the DPF construction supporting the boolean range and the DPF construction supporting the range $\{0,1\}^t$ (for arbitrary $t$) are as follows:

- For 1-bit output BDPF: $O(|\mathsf{pk}| + |\mathsf{ek}_0| + |\mathsf{ek}_1| + k \cdot (|\mathsf{in}_0| + |\mathsf{in}_1|) + \kappa + m \cdot n)$
- For $t$-bit output BDPF: $O(|\mathsf{pk}| + |\mathsf{ek}_0| + |\mathsf{ek}_1| + k \cdot (|\mathsf{in}_0| + |\mathsf{in}_1|) + \kappa + \lambda + m \cdot (n + t))$

We have opted for the Paillier-based construction proposed in [41] to instantiate the HSS scheme. We initiate our analysis by examining the size of HSS input encodings produced by the construction detailed in [41]. In this construction, an input encoding share consists of a sequence of $\ell$ Paillier ciphertexts. Here, $\ell = \left\lceil \log_{B_{\mathsf{sk}}}(N^2) \right\rceil$ and $B_{\mathsf{sk}} = \frac{N}{B_{\mathsf{msg}} 2^\varepsilon}$, where $N$ is the public key generated by the Paillier cryptosystem, $\varepsilon$ is the statistical security parameter, and $B_{\mathsf{msg}}$ specifies an upper boundary that all input values and intermediary values computed during the execution of a program via $\mathsf{HSS.Eval}$ must adhere to guarantee the correctness of the computation.

Upon examining our algorithms and RMS programs, it becomes evident that in both BDPF schemes, no input or intermediate value surpasses the upper boundary $(d+2)^{\frac{2}{3}n}$, where $d$ is the locality parameter for the Local Code used (as explained in Section 2.1) and $n$ is the bit-length of

the input in the domain of the encoded point functions. We have,

$$\begin{aligned}\ell = \left\lceil \log_{B_{\mathsf{sk}}}(N^2)\right\rceil = \left\lceil 2\frac{\log_2(N)}{\log_2(B_{\mathsf{sk}})}\right\rceil &= \left\lceil 2\frac{\log_2(N)}{\log_2(N)-\log_2(B_{\mathsf{msg}}2^{\varepsilon})}\right\rceil \\ &= \left\lceil 2\frac{\log_2(N)}{\log_2(N)-\varepsilon-\frac{2}{3}n\cdot\log_2(d+2)}\right\rceil \\ &\approx \left\lceil 2\frac{\lambda}{\lambda-\varepsilon-\frac{2}{3}n\cdot\log_2(d+2)}\right\rceil \\ &\in O\left(\frac{\lambda}{\lambda-\varepsilon-n\cdot\log_2(d)}\right)\end{aligned}$$

Consider that when $n$ grows sufficiently large and $\kappa$ becomes sufficiently small, the final expression may yield a negative outcome. This indicates the absence of a valid input encoding for the specified parameter combination. Nevertheless, adjusting the $N$ parameter (by opting for a greater asymmetric security parameter) can accommodate these larger values, albeit at the expense of larger encoding sizes. During protocol instantiation, we experiment with varying $N$ values to determine which combination yields smaller-sized HSS input encodings.

Additionally, in [41], the two shares forming an input encoding are identical, meaning they consist of the same sequence of ciphertexts. Therefore, $|\mathsf{in}_0|, |\mathsf{in}_1| \approx \ell \cdot \lambda \in O(\ell \cdot \lambda)$.

The public key $\mathsf{pk}$ for this scheme consists of a Paillier public key and $\ell$ Paillier ciphertexts that encrypt the Paillier private key 'in pieces'. Similarly, both evaluation keys $\mathsf{ek}_0$ and $\mathsf{ek}_1$ consist of a key to a pseudorandom function and $\ell$ secret shares over the integers used to share the Paillier private key also 'in pieces'. Thus, we encounter the following size complexities

$$|\mathsf{pk}| \approx \ell \cdot (\lambda + 1) \in O(\ell \cdot \lambda)$$

$$|\mathsf{ek}_0|, |\mathsf{ek}_1| \approx \kappa + \ell \cdot \varepsilon + \lambda \in O(\kappa + \ell \cdot \varepsilon + \lambda)$$

By substituting these complexities back into the key size complexities shown earlier, we obtain the following two formulas.

- For 1-bit output BDPF: $O(\ell \cdot (\varepsilon + k \cdot \lambda) + \kappa + m \cdot n)$
- For $t$-bit output BDPF: $O(\ell \cdot (\varepsilon + k \cdot \lambda) + \kappa + m \cdot (n + t))$

### 5.1.2 Concrete Cost

In this section, we focus on the 1-bit output scenario, as the performance disparity (compression ratio) between our solution and the naive approach is mostly the same for any $t$-bit outputs. The total key size of our 1-bit output BDPF is

$$\begin{aligned}|\mathsf{k}_\sigma| &= |\mathsf{pk}| + |\mathsf{ek}_\sigma| + |\rho_\sigma| \\ &\approx \ell \cdot (\lambda + 1 + \varepsilon) + 2 \cdot \kappa + \lambda + k \cdot |\mathsf{in}_\sigma| + m \cdot n\end{aligned}$$

And, a naive batched DPF has keys with a bit-length of $m \cdot \mathsf{k}_{\mathsf{naive}}$, where $\mathsf{k}_{\mathsf{naive}}$ is the bit-length of a non-batched DPF construction. To ensure a fair comparison, we utilize the most efficient (in terms of key size) non-batched DPF construction, as proposed in [14], to instantiate the naive batched DPF, which has $\mathsf{k}_{\mathsf{naive}} = (n - \log_2(\kappa)) \cdot \kappa$. Thus, the naive batched DPF construction has the key length $|\mathsf{k}'_\sigma| \approx m \cdot (n - \log_2(\kappa)) \cdot \kappa$.

| Parameters | | | Attacks | | | | |
|---|---|---|---|---|---|---|---|
| $q$ | $k$ | $\tau$ | Gauss | SD | SD 2.0 | SD-ISD | BJMM-ISD |
| $2^{26}$ | $2^{13}$ | $2^{19}$ | 129.1 | 199.15 | 196.19 | 135.75 | 129.32 |
| $2^{24}$ | $2^{13}$ | $2^{17}$ | 129.11 | 199.18 | 196.23 | 131.32 | 126.15 |
| $2^{22}$ | $2^{13}$ | $2^{15}$ | 129.18 | 199.32 | 196.36 | 128.85 | 124.07 |
| $2^{20}$ | $2^{13}$ | $2^{13}$ | 129.46 | 199.87 | 196.9 | 124.68 | 122.26 |
| $2^{18}$ | $2^{13}$ | $2^{11}$ | 130.58 | 202.13 | 199.06 | 122.10 | 118.84 |
| $2^{16}$ | $2^{13}$ | $2^{9}$ | 135.44 | 211.95 | 208.42 | 123.96 | 120.12 |

Table 1: **Parameter Validation ($k = 2^{13}$ for $q = mn$ and $\tau = q/n = m$)**, ensuring bit-security against various attacks on LPN with regular noise: Pooled Gauss (Gauss) [27] , Statistical Decoding (SD) [38, 43, 30, 22], Statistical Decoding 2.0 (SD 2.0) [18], Stern-Dumer SD (SD-ISD) [36] and Becker-Joux-May-Meurer SD (BJMM-ISD) [5]

| Parameters | | | | Key-length Size | | | Runtime | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | Ours | Naive | Comp. | Key Generation | | Evaluation | |
| $m$ | $q$ | $\tau$ | $\lambda$ | | | Ratio | Ours | Naive | Ours | Naive |
| $2^{29}$ | $2^{36}$ | $2^{29}$ | | $70866972064 \approx 2^{36.05}$ | $2^{29} \cdot (2^7-7) \cdot 2^7$ | 117.33 | 555.75 | 263066.75 | | |
| $2^{25}$ | $2^{32}$ | $2^{25}$ | | $4831849888 \approx 2^{32.17}$ | $2^{25} \cdot (2^7-7) \cdot 2^7$ | 107.55 | 138.94 | 16441.68 | | |
| $2^{21}$ | $2^{28}$ | $2^{21}$ | $2^{10}$ | $402664864 \approx 2^{28.59}$ | $2^{21} \cdot (2^7-7) \cdot 2^7$ | 80.66 | 34.74 | 1027.61 | 0.68 | |
| $2^{17}$ | $2^{24}$ | $2^{17}$ | | $83897760 \approx 2^{26.33}$ | $2^{17} \cdot (2^7-7) \cdot 2^7$ | 24.19 | 17.37 | 64.23 | | |
| $2^{13}$ | $2^{20}$ | $2^{13}$ | | $68169120 \approx 2^{26.03}$ | $2^{13} \cdot (2^7-7) \cdot 2^7$ | 1.86 | 17.37 | 4.02 | | 0.000014 |
| $2^{29}$ | $2^{36}$ | $2^{29}$ | | $71940721016 \approx 2^{36.07}$ | $2^{29} \cdot (2^7-7) \cdot 2^7$ | 115.58 | 3827.32 | 263066.75 | | |
| $2^{25}$ | $2^{32}$ | $2^{25}$ | | $5100292472 \approx 2^{32.25}$ | $2^{25} \cdot (2^7-7) \cdot 2^7$ | 101.89 | 956.84 | 16441.68 | | |
| $2^{21}$ | $2^{28}$ | $2^{21}$ | $2^{11}$ | $469780856 \approx 2^{28.81}$ | $2^{21} \cdot (2^7-7) \cdot 2^7$ | 69.14 | 239.22 | 1027.61 | 4.68 | |
| $2^{17}$ | $2^{24}$ | $2^{17}$ | | $117459320 \approx 2^{26.81}$ | $2^{17} \cdot (2^7-7) \cdot 2^7$ | 17.28 | 119.62 | 64.23 | | |
| $2^{13}$ | $2^{20}$ | $2^{13}$ | | $101730680 \approx 2^{26.61}$ | $2^{13} \cdot (2^7-7) \cdot 2^7$ | 1.24 | 119.62 | 4.02 | | |

Table 2: **Comparing Key-length Size and Runtime** between our 1-bit output BDPF and the naive approach, where $m$ is the batch size, $q = m \cdot n$, $\ell = 3$, $\tau = q/n$ is the noise parameter of RHW and $\lambda$ is the asymmetric security parameter. We used $n = 128$ and $\kappa = 128$ for all estimates in this table. The key sizes are measured in bits and the runtimes are measured in seconds.

We choose different values for the variables, calculate the approximate bit-length of keys for both constructions, and proceed to compare their lengths. Specifically, we set the standard security parameters as $\lambda \in \{1024, 2048\}, \kappa = 128$ and $\varepsilon = 40$. We set $n = 128$, which is greater than $3(\varepsilon+1)/2$ as needed for the correctness proof in Theorem 10. We select different $m \in \{2^{13}, 2^{17}, 2^{21}, 2^{25}, 2^{29}\}$ to demonstrate the compression ratio. Given the number of samples $q = mn$ in our HSSIC construction (utilizing the LPN assumption variant in Section 2.1), we need to choose suitable values for the dimension parameter $k$ and noise parameter $\tau$ to ensure robust concrete security. For simplicity, we set $k = 2^{13}$ and utilize the estimator tool provided in [40] to validate that all the parameters used in our key-size calculations guarantee sufficient security. Table 1 presents the concrete security values for each combination of LPN parameters we employed against various attacks discussed in [40].

Table 2 presents the key sizes of our BDPF and the naive solution, along with the compression ratio. As expected, as the number of DPF instances increases, so does the compression ratio. Note that, in applications of BFSS such as oblivious transfer extension [10], private set intersection [23], and privacy-preserving machine learning [35], they usually require billions of instances of FSS. This motivates the contribution of our construction.

| Parameters | | LAN (or 10Gbps) | | WAN 100Mbps | | WAN 80Mbps | | WAN 50Mbps | | WAN 30Mbps | | WAN 10Mbps | | WAN 1Mbps | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $m$ | $\lambda$ | Ours | Naive | Ours | Naive | Ours | Naive | Ours | Naive | Ours | Naive | Ours | Naive | Ours | Naive |
| $2^{29}$ | | 563.19 | 263859.74 | 1232.28 | 342365.31 | 1401.24 | 362189.95 | 1908.12 | 421663.87 | 2809.24 | 527395.29 | 7314.84 | 1056052.35 | 68140.44 | 8192922.75 |
| $2^{25}$ | | 140.09 | 16491.24 | 185.71 | 21397.84 | 197.23 | 22636.88 | 231.79 | 26354 | 293.23 | 32962.21 | 600.43 | 66003.28 | 4747.63 | 512057.68 |
| $2^{21}$ | $2^{10}$ | 35.46 | 1030.71 | 39.27 | 1337.37 | 40.23 | 1414.81 | 43.11 | 1647.13 | 48.23 | 2060.14 | 73.83 | 4125 | 419.43 | 32003 |
| $2^{17}$ | | 18.06 | 64.42 | 18.86 | 83.59 | 19.06 | 88.43 | 19.66 | 102.95 | 20.72 | 128.76 | 26.06 | 257.83 | 98.06 | 2000.23 |
| $2^{13}$ | | 18.06 | 4.03 | 18.71 | 5.23 | 18.87 | 5.53 | 19.36 | 6.44 | 20.22 | 8.05 | 24.56 | 16.12 | 83.06 | 125.02 |
| $2^{29}$ | | 3838.87 | 263859.74 | 4518.09 | 342365.31 | 4689.61 | 362189.95 | 5204.17 | 421663.87 | 6118.94 | 527395.29 | 10692.81 | 1056052.35 | 72440.02 | 8192922.75 |
| $2^{25}$ | | 962.01 | 16491.24 | 1010.17 | 21397.84 | 1022.33 | 22636.88 | 1058.81 | 26354 | 1123.66 | 32962.21 | 1447.93 | 66003.28 | 5825.54 | 512057.68 |
| $2^{21}$ | $2^{11}$ | 243.95 | 1030.71 | 248.39 | 1337.37 | 249.51 | 1414.81 | 252.87 | 1647.13 | 258.84 | 2060.14 | 288.71 | 4125 | 691.92 | 32003 |
| $2^{17}$ | | 124.32 | 64.42 | 125.43 | 83.59 | 125.71 | 88.43 | 126.55 | 102.95 | 128.04 | 128.76 | 135.51 | 257.83 | 236.32 | 2000.23 |
| $2^{13}$ | | 124.31 | 4.03 | 125.28 | 5.23 | 125.52 | 5.53 | 126.25 | 6.44 | 127.54 | 8.05 | 134.01 | 16.12 | 221.32 | 125.02 |

Table 3: **Comparing Runtime** between our 1-bit output BDPF and the naive approach, where $m$ is the batch size, $q = m \cdot n$, $\ell = 3$,$\tau = m/n$ is the noise parameter of $\mathsf{RHW}$ and $\lambda$ is the asymmetric security parameter. The runtimes are measured in seconds.

## 5.2 Computational Efficiency

### 5.2.1 Asymptotic Complexity

Given that our BDPF constructions use an HSS scheme in a black-box way and execute RMS programs by using such a scheme, our analysis measures the computational complexity in a not-so-conventional way. We measure the computational complexity by the computational complexity of the RMS programs executed by using the HSS scheme, plus the operations performed by the algorithm. So, at the end of our analysis of a specific algorithm that runs an RMS program through $\mathsf{HSS.Eval}$, we end up with two complexity classes that jointly describe the efficiency of the algorithm.

The description for $\mathsf{HSSIC.Expand}$ starts by sampling $A, \boldsymbol{e}$ and $\boldsymbol{s}$. We assume that $\mathbf{C}_{\mathsf{L}}$ samples a seed to a pseudorandom generator with polynomial stretch that serves as a representation to $A$ and that $A$ can then be sampled executing this PRG $O(d \cdot n)$ times since $A$ has $d \cdot n$ non-zero entries, which leads to a complexity of $O(d \cdot n \cdot \kappa)$. We assume that sampling $\boldsymbol{e}$ and $\boldsymbol{s}$ is done in constant time. Next, we compute vectors $\boldsymbol{w}$ and $\boldsymbol{z}$. Since $A$ is a sparse matrix with $d$ non-zero entries per row, we claim that computing these vectors can be done in $O(d \cdot n)$, where $d$ is the locality factor of $\mathbf{C}_{\mathsf{L}}$. Then, as the last step, we execute $\mathsf{HSS.Input}$ $k$ times to encode the vector $\boldsymbol{s}$ as input. Since we don't make any assumption about the HSS scheme, we say that encoding a single input takes $t_{\mathsf{in}}$ to encode a single input and $O(k \cdot t_{\mathsf{in}})$ to encode the whole vector $\boldsymbol{s}$. This means $\mathsf{HSSIC.BIn}$ has its computational complexity in $O(d \cdot n \cdot \kappa + k \cdot t_{\mathsf{in}})$.

The execution of $\mathsf{HSSIC.Expand}$ involves three steps, parsing the compressed structure $\rho_\sigma$, encoding $w_i$ as an HSS Input, and then running the program $P^{\mathsf{Ex}}$ using $\mathsf{HSS.Eval}$. Assuming parsing is done in constant time, it is easy to see that the first 2 steps can be done in $O(t_{\mathsf{in}})$. We measure the last step by the number of arithmetic operations performed by the RMS program $P^{\mathsf{Ex}}$, which is in $O(d)$. So, we say that $\mathsf{HSSIC.Expand}$ takes $O(t_{\mathsf{in}})$ normal operations, plus $O(d)$ HSS program operations.

We start algorithm $\mathsf{GMCC.Comp}$ by parsing the public key $\mathsf{pk}$, sampling a key $\mathsf{k_H}$, and then executing the $H$ $n$ times. Assuming parsing and sampling $\mathsf{k_H}$ is done in constant time, and executing $H$ a single time takes $O(t \cdot \lambda^2)$ (since it needs to determine the Jabobi symbols as part of its execution), we have that all these three operations can be done in $O(n \cdot t \cdot \lambda^2)$ time. After these operations, we then decrypt $t \cdot n$ GM ciphertexts and do $n$ bitwise xors of $t$-length binary words, so we claim that performing these two operations takes $O(n \cdot t \cdot \lambda)$ time. This leads to a total asymptotic time of $O(n \cdot t \cdot \lambda^2)$ for $\mathsf{GMCC.Comp}$. Algorithm $\mathsf{GMCC.Expand}$ simply parses public key $\mathsf{pk} = N$ and compressed structure $\mathsf{cc}$, executes $H$ once, and performs $t$ exponentiations and multiplications over $N$. Assuming again the parsing is done in constant time and that executing $H$ takes $O(t \cdot \lambda^2)$ time, we conclude that $\mathsf{GMCC.Expand}$ takes $O(t \cdot \lambda^2)$ time in total.

Now, by inspecting our 1-bit and $t$-bit DPF constructions, and given our previous asymptotic

computational analysis of HSSIC.BIn and GMCC.Comp, it is easy to see that the key generation Gen of both the 1-bit construction and the $t$-bit construction have the following computational complexity, respectively:

- For 1-bit output BDPF: $O(\lambda + d \cdot m \cdot n \cdot \kappa + k \cdot t_{\mathsf{in}})$
- For $t$-bit output BDPF: $O(\lambda + d \cdot m \cdot n \cdot \kappa + k \cdot t_{\mathsf{in}} + m \cdot t \cdot \lambda^2)$

The Eval algorithm for the 1-bit construction starts by parsing the DPF key, executing HSSIC.BIn $n$ times, and executing $H$ a single time. Assuming parsing is done in constant time and $H$ takes $O(n)$ time, we claim that these three steps take $O(n \cdot t_{\mathsf{in}})$ normal time plus $O(n \cdot d)$ HSS program execution time. After that, the algorithm encodes $n$ bits as HSS Inputs, executes the RMS program $P^{\mathsf{DPF}}$ using HSS.Eval, and performs a constant amount of bit-xors. By simply inspecting $P^{\mathsf{DPF}}$ it is easy to see that it performs $O(n)$ many arithmetic operations. So, we can say that these last three steps can be executed in $O(n \cdot t_{\mathsf{in}})$ normal time plus $O(n)$ HSS program execution time. We thus conclude the Eval algorithm for the 1-bit construction takes $O(n \cdot t_{\mathsf{in}})$ normal time plus $O(n \cdot d)$ HSS program execution time.

The Eval algorithm for the $t$-bit construction mirrors the operations of the 1-bit construction while incorporating additional steps. In addition to the operations carried out by the 1-bit construction, the $t$-bit construction also involves computing $h_\sigma$ through a fixed number of additions, executing GMCC.Expand once, performing $t$ exponentiations modulo $N = \mathsf{pk}_{\mathsf{GM}}$, conducting $t$ executions of $\mathsf{DDL}^{\mathsf{GM}}$, and completing $t$ bit concatenations. We claim that performing the $t$ exponentiations and $t$ executions of $\mathsf{DDL}^{\mathsf{GM}}$ take $O(t\lambda)$ time, while the other 3 operations take $O(t\lambda^2)$ given our previous analysis of GMCC.Expand. This leaves us with the Eval algorithm of the $t$-bit construction having normal time $O(n \cdot t_{\mathsf{in}} + t\lambda^2)$ plus $O(n \cdot d)$ HSS program execution time. Note that the RMS program of the $t$-bit construction is different from that of the 1-bit one, but it only performs a constant amount more of arithmetic operations, so it does not affect its asymptotic computational efficiency.

Hence the computational complexity of both the proposed batch DPF Eval functions is linear in the number of input bits.

### 5.2.2 Concrete Cost

Starting our analysis with the Paillier-based HSS scheme [42], we observe that executing HSS.Setup and HSS.Input algorithms involves $\ell$ and $\ell + 1$ Paillier encryptions, respectively. Consequently, this requires $\ell$ and $\ell + 1$ exponentiations. Moreover, during the homomorphic evaluation of an RMS program, only multiplication operations necessitate exponentiations to be computed, with each multiplication requiring $\ell + 1$ exponentiations. Given this analysis, we evaluate the concrete efficiency of our proposed HSSIC scheme as follow. Computing HSS.Input entails computing $k \cdot (\ell+1)$ exponentiations since HSS.Input is executed $k$ times and no other exponentiation is executed. On the other hand, HSSIC.Expand consists of a single execution of HSS.Input and a single execution of HSS.Eval: the HSS.Eval does not have multiplications, and HSS.Input requires $\ell + 1$ multiplications to be computed. Thus, HSSIC.Expand necessitates $\ell + 1$ exponentiations.

The key generation algorithm Gen involves invoking HSS.Setup and HSSIC.BIn, alongside a call to a random oracle $H$. Following our analysis of HSS.Setup and HSSIC.BIn, we can infer that executing this algorithm requires $k \cdot (\ell + 1) + \ell$ exponentiations. On the other hand, Eval consists of $n$ calls to HSSIC.Expand and HSS.Input, along with a single call to HSS.Eval. Additionally, it involves executing $H$ once and a small constant number of XOR operations. Upon inspection of the program evaluated by HSS.Eval, we observe that $2n$ multiplications are executed, indicating a

requirement for $2n$ exponentiations to evaluate this program. Therefore, drawing from our previous analyses of HSSIC.Expand and HSS.Input, we ascertain that this algorithm will necessitate $2 \cdot n \cdot (\ell+2)$ exponentiations.

Note that from the LPN assumption, we have $k \in O(\sqrt{m \cdot n})$ (given the quadratic stretch of primal-LPN), however, for small values of $m \cdot n$ the assumption is insecure. Because of this, when estimating the efficiency of our scheme for small values of $m \cdot n$, we select $k = 2^{13}$. This is the case when estimating the efficiency of our scheme for $m \in \{2^{17}, 2^{13}\}$. For this reason, the key generation runtime shown in Table 2 for $m \in \{2^{17}, 2^{13}\}$ are the same, given the number of exponentiations executed by this procedure is defined only by $k$ and $\ell$.

Building upon this analysis, we estimate the total runtime of our proposed 1-bit BDPF construction by assuming the runtime of approximately 0.53 and 3.65 milliseconds for single exponentiation of elements of bit-length 2048 and 4096, respectively. These two runtimes resulted from a benchmark produced by the command line tool OpenSSL when running the command "speed rsa". From this benchmark, we picked the runtime of a single 2048 and 4096-bit private RSA operation as the runtime of our exponentiations. This benchmark was run on a Ubuntu 20.04.3 LTS server (AMD EPYC 74F3 24-Core Processor at 3.96 GHz, 257GB RAM at 3.200 GHz).

Table 2 shows the estimated running times of our construction and Naive BDPF's key generation and evaluation algorithms for different parameter settings. We estimate the runtime of the naive batched solution using libfss [31], which was implemented based on [12] and [14].

It is clear that in many cases our approach is orders of magnitude slower than the naive one since the naive solution only performs symmetric operations, however, we believe our solution offers an interesting tradeoff between key size and running time, especially when considering settings where keys need to be transmitted over slower networks. To show this, we report the estimation of concrete key sizes and running times of batched approaches in Table 3. The time presents the total time of generating keys for a batch of $m$ point function, transmitting one of the keys, and evaluating a single function in a point, in sequence.

## References


[1] M. Alekhnovich. More on average case vs approximation complexity. In *44th Annual IEEE Symposium on Foundations of Computer Science, 2003. Proceedings.*, pages 298–307, 2003.

[2] Benny Applebaum. Pseudorandom generators with long stretch and low locality from random local one-way functions. In *Proceedings of the forty-fourth annual ACM symposium on Theory of computing*, pages 805–816, 2012.

[3] Benny Applebaum, Ivan Damgård, Yuval Ishai, Michael Nielsen, and Lior Zichron. Secure arithmetic computation with constant computational overhead. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology – CRYPTO 2017*, pages 223–254, Cham, 2017. Springer International Publishing.

[4] Benny Applebaum and Shachar Lovett. Algebraic attacks against random local functions and their countermeasures. In *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing*, pages 1087–1100, 2016.

[5] Anja Becker, Antoine Joux, Alexander May, and Alexander Meurer. Decoding random binary linear codes in 2n/20: How $1 + 1 = 0$ improves information set decoding. In David Pointcheval and Thomas Johansson, editors, *Advances in Cryptology – EUROCRYPT 2012*, pages 520–536, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[6] Avrim Blum, Merrick L. Furst, Michael J. Kearns, and Richard J. Lipton. Cryptographic primitives based on hard learning problems. In Douglas R. Stinson, editor, *CRYPTO'93*, volume 773 of *LNCS*, pages 278–291. Springer, Heidelberg, August 1994.

[7] 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 Anne Canteaut and François-Xavier Standaert, editors, *EUROCRYPT 2021, Part II*, volume 12697 of *LNCS*, pages 871–900. Springer, Heidelberg, October 2021.

[8] Elette Boyle, Geoffroy Couteau, Niv Gilboa, and Yuval Ishai. Compressing vector OLE. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 896–912. ACM Press, October 2018.

[9] 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*, CCS '19, page 291–308, New York, NY, USA, 2019. Association for Computing Machinery.

[10] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Efficient pseudorandom correlation generators: Silent OT extension and more. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 489–518. Springer, Heidelberg, August 2019.

[11] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 337–367. Springer, Heidelberg, April 2015.

[12] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015*, pages 337–367, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[13] Elette Boyle, Niv Gilboa, and Yuval Ishai. Breaking the circuit size barrier for secure computation under DDH. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part I*, volume 9814 of *LNCS*, pages 509–539. Springer, Heidelberg, August 2016.

[14] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing: Improvements and extensions. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 1292–1303. ACM Press, October 2016.

[15] Elette Boyle, Niv Gilboa, and Yuval Ishai. Secure computation with preprocessing via function secret sharing. In Dennis Hofheinz and Alon Rosen, editors, *TCC 2019, Part I*, volume 11891 of *LNCS*, pages 341–371. Springer, Heidelberg, December 2019.

[16] Elette Boyle, Lisa Kohl, and Peter Scholl. Homomorphic secret sharing from lattices without FHE. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part II*, volume 11477 of *LNCS*, pages 3–33. Springer, Heidelberg, May 2019.

[17] Zvika Brakerski, Pedro Branco, Nico Döttling, and Sihang Pu. Batch-ot with optimal rate. In Orr Dunkelman and Stefan Dziembowski, editors, *Advances in Cryptology – EUROCRYPT 2022*, pages 157–186, Cham, 2022. Springer International Publishing.

[18] Kévin Carrier, Thomas Debris-Alazard, Charles Meyer-Hilfiger, and Jean-Pierre Tillich. Statistical decoding 2.0: Reducing decoding to lpn. In *Advances in Cryptology – ASIACRYPT 2022: 28th International Conference on the Theory and Application of Cryptology and Information Security, Taipei, Taiwan, December 5–9, 2022, Proceedings, Part IV*, page 477–507, Berlin, Heidelberg, 2023. Springer-Verlag.

[19] 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.

[20] 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 Computer Society Press, May 2015.

[21] Leo de Castro and Antigoni Polychroniadou. Lightweight, maliciously secure verifiable function secret sharing. In Orr Dunkelman and Stefan Dziembowski, editors, *EUROCRYPT 2022, Part I*, volume 13275 of *LNCS*, pages 150–179. Springer, Heidelberg, May / June 2022.

[22] Thomas Debris-Alazard and Jean-Pierre Tillich. Statistical decoding. In *2017 IEEE International Symposium on Information Theory (ISIT)*, pages 1798–1802, 2017.

[23] Daniel Demmler, Peter Rindal, Mike Rosulek, and Ni Trieu. PIR-PSI: Scaling Private Contact Discovery. In *Proceedings on Privacy Enhancing Technologies (PETS)*, 2018.

[24] Yevgeniy Dodis, Shai Halevi, Ron D. Rothblum, and Daniel Wichs. Spooky encryption and its applications. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part III*, volume 9816 of *LNCS*, pages 93–122. Springer, Heidelberg, August 2016.

[25] Jack Doerner and abhi shelat. Scaling ORAM for secure computation. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 523–535. ACM Press, October / November 2017.

[26] Nico Döttling, Sanjam Garg, Yuval Ishai, Giulio Malavolta, Tamer Mour, and Rafail Ostrovsky. Trapdoor hash functions and their applications. In Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology – CRYPTO 2019*, pages 3–32, Cham, 2019. Springer International Publishing.

[27] Andre Esser, Robert Kübler, and Alexander May. Lpn decoded. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology – CRYPTO 2017*, pages 486–514, Cham, 2017. Springer International Publishing.

[28] Nils Fleischhacker, Kasper Green Larsen, and Mark Simkin. Compressing encrypted data over small fields. Cryptology ePrint Archive, Paper 2023/946, 2023. `https://eprint.iacr.org/2023/946`.

[29] Nils Fleischhacker, Kasper Green Larsen, and Mark Simkin. How to compress encrypted data. In Carmit Hazay and Martijn Stam, editors, *Advances in Cryptology – EUROCRYPT 2023*, pages 551–577, Cham, 2023. Springer Nature Switzerland.

[30] M. P. C. Fossorier, K. Kobara, and H. Imai. Modeling bit flipping decoding based on nonorthogonal check sums with application to iterative decoding attack of mceliece cryptosystem. *IEEE Trans. Inf. Theor.*, 53(1):402–411, jan 2007.

[31] Ph.D Frank Wang. Function secret sharing (fss) library. `https://github.com/frankw2/libfss`, 2018.

[32] Gayathri Garimella, Mike Rosulek, and Jaspal Singh. Structure-aware private set intersection, with applications to fuzzy matching. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part I*, volume 13507 of *LNCS*, pages 323–352. Springer, Heidelberg, August 2022.

[33] Niv Gilboa and Yuval Ishai. Distributed point functions and their applications. In Phong Q. Nguyen and Elisabeth Oswald, editors, *Advances in Cryptology – EUROCRYPT 2014*, pages 640–658, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[34] Shafi Goldwasser and Silvio Micali. Probabilistic encryption & how to play mental poker keeping secret all partial information. In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, STOC '82, page 365–377, New York, NY, USA, 1982. Association for Computing Machinery.

[35] 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, Paper 2023/1269, 2023. `https://eprint.iacr.org/2023/1269`.

[36] Yann Hamdaoui and Nicolas Sendrier. A non asymptotic analysis of information set decoding. Cryptology ePrint Archive, Paper 2013/162, 2013. `https://eprint.iacr.org/2013/162`.

[37] Piotr Indyk and David P. Woodruff. Polylogarithmic private approximations and efficient matching. In Shai Halevi and Tal Rabin, editors, *TCC 2006*, volume 3876 of *LNCS*, pages 245–264. Springer, Heidelberg, March 2006.

[38] A. Al Jabri. A statistical decoding algorithm for general linear block codes. In Bahram Honary, editor, *Cryptography and Coding*, pages 1–8, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[39] Jonathan Katz and Moti Yung. Threshold cryptosystems based on factoring. In Yuliang Zheng, editor, *Advances in Cryptology — ASIACRYPT 2002*, pages 192–205, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[40] Hanlin Liu, Xiao Wang, Kang Yang, and Yu Yu. The hardness of lpn over any integer ring and field for pcg applications. Cryptology ePrint Archive, Paper 2022/712, 2022. `https://eprint.iacr.org/2022/712`.

[41] Claudio Orlandi, Peter Scholl, and Sophia Yakoubov. The rise of paillier: Homomorphic secret sharing and public-key silent OT. In Anne Canteaut and François-Xavier Standaert, editors, *EUROCRYPT 2021, Part I*, volume 12696 of *LNCS*, pages 678–708. Springer, Heidelberg, October 2021.

[42] Claudio Orlandi, Peter Scholl, and Sophia Yakoubov. The rise of paillier: Homomorphic secret sharing and public-key silent ot. In Anne Canteaut and François-Xavier Standaert, editors, *Advances in Cryptology – EUROCRYPT 2021*, pages 678–708, Cham, 2021. Springer International Publishing.

[43] R. Overbeck. Statistical decoding revisited. In Lynn Margaret Batten and Reihaneh Safavi-Naini, editors, *Information Security and Privacy*, pages 283–294, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[44] Lawrence Roy and Jaspal Singh. Large message homomorphic secret sharing from DCR and applications. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part III*, volume 12827 of *LNCS*, pages 687–717, Virtual Event, August 2021. Springer, Heidelberg.

[45] Kang Yang, Chenkai Weng, Xiao Lan, Jiang Zhang, and Xiao Wang. Ferret: Fast extension for correlated ot with small communication. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 1607–1626, New York, NY, USA, 2020. Association for Computing Machinery.

# A Limitations and Open Problems

We design a new batch DPF scheme using a novel HSS ciphertext compression primitive along with designing RMS programs that can evaluate a point function given HSS inputs with occasional errors (as generated by $\mathsf{HSSIC}$ scheme). We leave it as future work, to explore how to use the $\mathsf{HSSIC}$ primitive and proposed techniques to construct batch FSS for other interesting classes of functions including comparison, internal and $d$-dimensional interval. Our batch DPF construction based on public key assumptions still remains impractical for many applications using DPFs due to the high computational complexity of the FSS evaluation function - which requires a constant number of public key operations per input bit. It remains open if one could design more computationally efficient batch DPF constructions from weaker assumptions like one-way functions, with better key size than naive batching.

$\underline{\mathsf{GM.Gen}(1^\lambda)}$:
Execute $(N, p, q) \leftarrow \mathsf{GenModulus}(1^\lambda)$.
Output $\mathsf{pk} = N$ and $\mathsf{sk} = \varphi(N)/4$, where $\varphi = N - p - q + 1$
$\underline{\mathsf{GM.Enc}(\mathsf{pk}, x \in \{0,1\})}$:
Parse $\mathsf{pk}$ as $N$.
Sample $r \in_R \mathbb{Z}_N$.
Output $\mathsf{ct} \equiv r^2 \cdot (-1)^x \pmod N$.
$\underline{\mathsf{GM.Dec}(\mathsf{sk}, \mathsf{ct})}$:
Compute $y \equiv \mathsf{ct}^{\mathsf{sk}} \pmod N$.
Output 0 if $y \equiv 1 \pmod N$, and 1 otherwise.

Figure 8: The Variant of the GM Cryptosystem presented in [39].

$\underline{\mathsf{DDL}^{\mathsf{GM}}(a \in \mathbb{Z}_N)}$:
Map $a$ to an integer in $\{0, \ldots, N-1\}$.
Output 1 if $a < N/2$, and 0 otherwise.

Figure 9: $\mathsf{DDL}^{\mathsf{GM}}$ Procedure as described in [41].

# B Mod 2 Addition and Integer Addition

We utilize the function Least Significant Bit ($\mathsf{LSB}$) in our batched FSS scheme description. As such, we list some straightforward equations/properties for the relationship between modulo 2 addition and integer addition with respect to the LSB function, which will be used in our correctness proof. By definition $\mathsf{LSB}(x) = 1 \iff x$ is odd.

- $\forall a, b \in \mathbb{Z}^+$, $\mathsf{LSB}(a+b) = \mathsf{LSB}(a) \oplus \mathsf{LSB}(b)$: The sum $a+b$ is odd if and only if either $a$ is odd or $b$ is odd.
- $\forall a, b \in \mathbb{Z}^+$, $\mathsf{LSB}(a \cdot b) = \mathsf{LSB}(a) \wedge \mathsf{LSB}(b)$: The product $a \cdot b$ is odd if and only if either $a$ is odd or $b$ is odd.
- $\forall a, b \in \mathbb{Z}_2$ $a + b \pmod 2 = \mathsf{LSB}(a+b)$.
- $\forall a_{i \in [n]} \in \mathbb{Z}_2$, $\sum_{i=1}^n a_i \pmod 2 = \mathsf{LSB}(\sum_{i=1}^n a_i)$.

Thus, $\forall a_{i\in[n]}, b_{i\in[n]} \in \mathbb{Z}_2$, $\sum_{i=1}^n a_i b_i \pmod 2 = \mathsf{LSB}(\sum_{i=1}^n a_i b_i)$;

- $\forall \mathbf{a}, \mathbf{b} \in \mathbb{Z}_2^n$, $\langle \mathbf{a}, \mathbf{b} \rangle \pmod 2 = \mathsf{LSB}(\langle \mathbf{a}, \mathbf{b} \rangle)$
- $\forall A \in \mathbb{Z}^{m \times n}, \mathbf{b} \in \mathbb{Z}_2^n$, $A \cdot \mathbf{b} \pmod 2 = \mathsf{LSB}(A \cdot \mathbf{b})$.

# C Proof of Theorem 9

*Proof.* We prove this theorem by showing that each security property defined in Definition 8 is satisfied.

1. **Last bit correctness**: From the description of $\mathsf{HSSIC.BIn}$, the component of vector $\boldsymbol{z}$ for $i \in [n]$ are computed as:

$$\begin{aligned} z_i &= A_{i,*}\boldsymbol{s} + w_i \\ \implies \mathsf{LSB}(z_i) &= A_{i,*}\boldsymbol{s} + (A_{i,*}\boldsymbol{s} + e_i + m_i) \pmod 2 \\ &= \boldsymbol{e}_i + \boldsymbol{m}_i \pmod 2 \\ \implies \mathsf{LSB}(\boldsymbol{z}) &= \boldsymbol{e} + \boldsymbol{m} \pmod 2 \end{aligned}$$

   We assume a structured noise distribution is $\mathsf{RHW} = \{\mathsf{RHW}_{\tau,n}\}_{\tau,n\in\mathbb{N}}$, hence $\boldsymbol{e}[j\lfloor n/\tau \rfloor, \ldots, (j+1)\lfloor n/\tau \rfloor - 1]$ is a unit vector for each $j$. Which proves $e = \mathsf{LSB}(\boldsymbol{z}) \oplus \boldsymbol{m} \in E^{\otimes n/\eta}$ .

2. **Bounded decompressed messages**: By construction, $z_i = A_{i,*}\boldsymbol{s} + w_i$. The LPN code has constant locality $d$ - the maximum number of non-zero elements in $A_{i,*}$, implying $z_i \leq d + 1$.

3. **Private input equivalence**: To prove this, we first analyze program $P^{\mathsf{Ex}}$, which outputs the input encoding of $z'_i$ computed as follows:

$$z'_i = w_i + A_{i,*}\boldsymbol{s} = z_i$$

   Hence, the output of $\mathsf{HSSIC.Expand}$ is exactly the HSS input encodings of $z_i$ - which is output by $\mathsf{HSSIC.BIn}$. We can now prove the private input equivalence property from the correctness guarantee of the underlying HSS scheme. By contrapositive, assume the private input equivalence property was violated for some program $P$, input $m$, and index $i \in [t]$. Hence, the corresponding HSS computation would give incorrect output for the program $P$ with non-negligible probability (let's say $p$). Now, we can construct an input and a program as follows, on which the HSS correctness would fail with probability $p$ as well:

   - **Inputs**: $\boldsymbol{s} \in_R \mathbb{Z}_2^k, \boldsymbol{w} \leftarrow A\boldsymbol{s} + e + m \pmod 2$ (where $A \xleftarrow{\$} \mathbf{C}_{\mathsf{L}}(k,n)$, $\boldsymbol{e} \xleftarrow{\$} \mathsf{RHW}_{\tau,n}$ )
   - **RMS program** $P'$: Execute RMS program $P^{\mathsf{Ex}}_{k,i,A}$ for each $i \in [n]$, where the input encodings given as output ($[\![z'_i]\!]$) are fed into the program $P$.

   Hence, if the private equivalence property is violated for program $P$ with input $m$, then we can construct a program $P'$ and inputs $\boldsymbol{s}, \boldsymbol{w}$ that violate the original correctness definition of HSS.

4. **Compressed private input indistinguishably**: We prove this property by constructing a sequence of hybrids $H_0, H_1, H_2$ played by a PPT adversary. $H_0$ is the security game exactly matching the security definition in Figure 3 for the proposed construction. We alter this hybrid incrementally, such that each hybrid is indistinguishable from the previous except with negligible probability, and in the last hybrid the adversary's advantage is $1/2$ - proving the compressed private input indistinguishably property. The hybrids are formally presented in Figure 10.

   - *Hybrid* $H_0$: This is the original compressed private input indistinguishably experiment for the proposed $\mathsf{HSSIC}$ construction.

Hybrid $H_0(1^\lambda)$:

$(\boldsymbol{m}^{(0)}, \boldsymbol{m}^{(1)}, \mathsf{state}) \leftarrow \mathcal{A}(1^\lambda)$

$(\mathsf{pk}, (\mathsf{ek}_0, \mathsf{ek}_1)) \leftarrow \mathsf{HSS.Setup}(1^\lambda)$

Sample $A \xleftarrow{\$} \mathbf{C}_{\mathsf{L}}(k, n)$, $\boldsymbol{e} \xleftarrow{\$} \mathsf{RHW}_{\tau,n}$, and $\boldsymbol{s} \in_R \mathbb{Z}_2^k$.

Compute $\boldsymbol{w} = A\boldsymbol{s} + \boldsymbol{e} + \boldsymbol{m}^{(b)} \pmod 2$.

Execute $(\llbracket \boldsymbol{s} \rrbracket_0, \llbracket \boldsymbol{s} \rrbracket_1) \leftarrow \mathsf{HSS.Input}(\mathsf{pk}, \boldsymbol{s})$.

$\rho_\sigma = (A, \boldsymbol{w}, \llbracket \boldsymbol{s} \rrbracket_\sigma)$

$b' \leftarrow \mathcal{A}(\mathsf{state}, \mathsf{pk}, \mathsf{ek}_\sigma, \rho_\sigma)$

**return** $b'$

Hybrid $H_1(1^\lambda)$:

$(\boldsymbol{m}^{(0)}, \boldsymbol{m}^{(1)}, \mathsf{state}) \leftarrow \mathcal{A}(1^\lambda)$

$(\mathsf{pk}, (\mathsf{ek}_0, \mathsf{ek}_1)) \leftarrow \mathsf{HSS.Setup}(1^\lambda)$

Sample $A \xleftarrow{\$} \mathbf{C}_{\mathsf{L}}(k, n)$, $\boldsymbol{e} \xleftarrow{\$} \mathsf{RHW}_{\tau,n}$, and $\boldsymbol{s} \in_R \mathbb{Z}_2^k$.

Compute $\boldsymbol{w} = A\boldsymbol{s} + \boldsymbol{e} + \boldsymbol{m}^{(b)} \pmod 2$.

$z \leftarrow 0^{|s|}$

Execute $(\llbracket \boldsymbol{z} \rrbracket_0, \llbracket \boldsymbol{z} \rrbracket_1) \leftarrow \mathsf{HSS.Input}(\mathsf{pk}, z)$.

$\rho_\sigma = (A, \boldsymbol{w}, \llbracket \boldsymbol{z} \rrbracket_\sigma)$

$b' \leftarrow \mathcal{A}(\mathsf{state}, \mathsf{pk}, \mathsf{ek}_\sigma, \rho_\sigma)$

**return** $b'$

Hybrid $H_2(1^\lambda)$:

$(\boldsymbol{m}^{(0)}, \boldsymbol{m}^{(1)}, \mathsf{state}) \leftarrow \mathcal{A}(1^\lambda)$

$(\mathsf{pk}, (\mathsf{ek}_0, \mathsf{ek}_1)) \leftarrow \mathsf{HSS.Setup}(1^\lambda)$

Sample $A \xleftarrow{\$} \mathbf{C}_{\mathsf{L}}(k, n), w \leftarrow \mathbb{Z}_2^n$

$z \leftarrow 0^{|s|}$

Execute $(\llbracket \boldsymbol{z} \rrbracket_0, \llbracket \boldsymbol{z} \rrbracket_1) \leftarrow \mathsf{HSS.Input}(\mathsf{pk}, z)$.

$\rho_\sigma = (A, \boldsymbol{w}, \llbracket \boldsymbol{z} \rrbracket_\sigma)$

$b' \leftarrow \mathcal{A}(\mathsf{state}, \mathsf{pk}, \mathsf{ek}_\sigma, \rho_\sigma)$

**return** $b'$

Figure 10: HSSIC Security Proof – Sequence of Hybrids

- *Hybrid $H_1$:* Here we replace the HSS input with $0^k$, and the rest of the hybrid is unchanged compared to $H_0$. An adversary that can distinguish between the two experiments $H_0$ and $H_1$ can be used to win the security game of the underlying HSS scheme with the same advantage, where the adversary can distinguish between input encodings of one party for messages $s$ and $0^k$. Hence the adversary has the same advantage for the modified hybrid $H_1$ as the hybrid in $H_0$, except for negligible probability.
- *Hybrid $H_2$:* Here the output vector $w$ is replaced with a random element of $\mathbb{Z}_2^n$. If the adversary in $H_2$ can guess bit $b$ with probability significantly different than in $H_1$, then it can be used to construct an adversary that can break the LPN assumption with the same advantage. Further, note the adversary in $H_2$ has advantage exactly $1/2$ of guessing $b$, completing the proof.

□

# D GM Ciphertext Compression

## D.1 GMCC Definitions

**Definition 12** (GM Ciphertext Compression: Syntax)**.** *A GM Ciphertext Compression scheme is a pair of algorithms* ($\mathsf{GMCC.Comp}$, $\mathsf{GMCC.Expand}$) *with the following syntax:*

- $\mathsf{cc} \leftarrow \mathsf{GMCC.Comp}_{n,t}(\mathsf{sk}, \boldsymbol{w} = (w_0, \ldots, w_{n-1}))$*: takes as input a private key* $\mathsf{sk}$ *sampled by* $\mathsf{GM.Gen}$ *and a vector* $\boldsymbol{w}$ *of length* $n$ *containing binary words* $w_i \in \{0,1\}^t$ *where* $n, t \in \mathbb{N}$*, and outputs a algebraic structure* $\mathsf{cc}$*.*
- $(\mathsf{ct}_0, \ldots, \mathsf{ct}_{t-1}) \leftarrow \mathsf{GMCC.Expand}_t(\mathsf{pk}, \mathsf{cc}, i)$*: takes as input a public key* $\mathsf{pk}$ *sampled by* $\mathsf{GM.Gen}$*, the algebraic structure outputted by* $\mathsf{GMCC.Comp}$ *and an index* $i \in [n]$*, and outputs a vector GM ciphertexts* $(\mathsf{ct}_0, \ldots, \mathsf{ct}_{t-1})$ *where* $t \in \mathbb{N}$*.*

**Definition 13** (GM Ciphertext Compression: Security)**.** *A GM Ciphertext Compression scheme is a pair of algorithms* ($\mathsf{GMCC.Comp}$, $\mathsf{GMCC.Expand}$)*, that satisfies the following requirements.*

- ***Correctness:*** *For* $t, n, \lambda \in \mathbb{N}$*, for all* $i \in [n]$*, where* $\boldsymbol{w} \in (\{0,1\}^t)^n$*,* $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{GM.Gen}(1^\lambda)$*,* $\mathsf{cc} \leftarrow \mathsf{GMCC.Comp}_t(\mathsf{sk}, \boldsymbol{w} = (w_0, \ldots, w_{n-1}))$*, for* $(\mathsf{ct}_0, \ldots, \mathsf{ct}_{t-1}) \leftarrow \mathsf{GMCC.Expand}_t(\mathsf{pk}, \mathsf{cc}, i)$*, the following holds*

$$\mathsf{GM.Dec}(\mathsf{sk}, \mathsf{ct}_0) \;||\; \ldots \;||\; \mathsf{GM.Dec}(\mathsf{sk}, \mathsf{ct}_{t-1}) = w_i$$

- ***Security:*** *For every non-uniform adversary* $\mathcal{A}$ *of size polynomial in the security parameter* $\lambda$*, it holds that*

$$\left| Pr\left[\mathsf{Exp}_{\mathcal{A},0}^{\mathsf{GMCC,sec}}(\lambda) = 1\right] - Pr\left[\mathsf{Exp}_{\mathcal{A},1}^{\mathsf{GMCC,sec}}(\lambda) = 1\right]\right| \leq \mathsf{negl}(\lambda)$$

*for all sufficiently large* $\lambda$*, where* $\mathsf{Exp}_{\mathcal{A},b}^{\mathsf{GMCC,sec}}(\lambda)$ *for* $b \in \{0,1\}$ *is as defined in Figure 11.*

$\underline{\mathsf{Exp}^{\mathsf{GMCC},\mathsf{sec}}_{\mathcal{A},b}(\lambda)}$:
$(\boldsymbol{w}^{(0)}, \boldsymbol{w}^{(1)}, \mathsf{state}) \leftarrow \mathcal{A}(1^\lambda)$
Sample $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{GM.Gen}(1^\lambda)$
$\mathsf{cc} \leftarrow \mathsf{GMCC.Comp}_t(\mathsf{sk}, \boldsymbol{w}^{(b)})$
$b' \leftarrow \mathcal{A}(\mathsf{state}, \mathsf{pk}, \mathsf{cc})$
**return** $b'$

Figure 11: GMCC Security Experiment

### D.2 GMCC Protocol:Correctness and Security Definitions

**Correctness.** Let parameters $t, n, \kappa, \lambda \in \mathbb{N}$, $i \in [n]$ and $\boldsymbol{w} \in (\{0,1\}^t)^n$. Let $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{GM.Gen}(1^\lambda)$, $\mathsf{cc} \leftarrow \mathsf{GMCC.Comp}_t(\mathsf{sk}, \boldsymbol{w})$, and $(\mathsf{ct}_0, \ldots, \mathsf{ct}_{t-1}) \leftarrow \mathsf{GMCC.Expand}_t(\mathsf{pk}, \mathsf{cc}, i)$. From the description of GMCC.Comp, we have $\mathsf{cc} = (\mathsf{k_H}, \boldsymbol{e})$, where $\mathsf{k_H}$ is a key to a keyed random oracle $H\colon \{0,1\}^\kappa \times \{0,1\}^* \to \mathbb{J}_N$ and $e_i = w_i \oplus r_i$. Thus, $\boldsymbol{c} \leftarrow H_{\mathsf{k_H}}(i)$ (computed when executing GMCC.Expand) and $C_{i,*} \leftarrow H_{\mathsf{k_H}}(i)$ (computed when executing GMCC.Comp) are the same, which in turn implies that $\mathsf{GM.Dec}(\mathsf{sk}, C_{i,*}) = \mathsf{GM.Dec}(\mathsf{sk}, \boldsymbol{c})$. Because GMCC.Expand outputs $(\mathsf{ct}_0, \ldots, \mathsf{ct}_{t-1})$, where $\mathsf{ct}_j = c_j \cdot (-1)^{e_{i,j}} = C_{i,j} \cdot (-1)^{e_{i,j}} \pmod N$, and $e_{i,j} = w_{i,j} \oplus r_{i,j} = w_{i,j} \oplus \mathsf{GM.Dec}(\mathsf{sk}, C_{i,j})$, we know by the homomorphic property of the GM cryptosystem that the following holds for all $j \in [t]$

$$\begin{aligned}
\mathsf{GM.Dec}(\mathsf{sk}, \mathsf{ct}_j) &= \mathsf{GM.Dec}(\mathsf{sk}, C_{i,j} \cdot (-1)^{e_{i,j}}) \\
&= \mathsf{GM.Dec}(\mathsf{sk}, C_{i,j}) \oplus e_{i,j} \\
&= \mathsf{GM.Dec}(\mathsf{sk}, C_{i,j}) \oplus (w_{i,j} \oplus r_{i,j}) \\
&= \mathsf{GM.Dec}(\mathsf{sk}, C_{i,j}) \oplus (w_{i,j} \oplus \mathsf{GM.Dec}(\mathsf{sk}, C_{i,j})) = w_{i,j}
\end{aligned}$$

It is easy to see that $\mathsf{GM.Dec}(\mathsf{sk}, \mathsf{ct}_0) \;||\; \ldots \;||\; \mathsf{GM.Dec}(\mathsf{sk}, \mathsf{ct}_{t-1}) = w_i$, thus concluding our correctness proof.

**Security.** The security proof of this compression scheme follows directly from the security of the GM cryptosystem. Consequently, we opt to omit it here.

## E Proof of Theorem 10

*Proof.* This proof involves demonstrating two aspects: Correctness and Secrecy, as defined in Definition 5.

**Correctness.** The Eval function in batch DPF protocol involves two steps. First it invokes HSSIC.Expand to generate HSS input encodings for the corresponding point function, and it generates HSS input encoding for the hash of the input $x^H = H(x)$. The second step of the algorithm executes HSS.Eval on an RMS program $P_n^{\mathsf{DPF}}$ (in Figure 6) given the input encodings of DPF input and the point function parameter. By the input equivalent property of HSSIC, the HSS.Eval function outputs secret shares of the correct output for any input RMS program with all but negligible probability in $\lambda$.

Note that no memory value in the RMS program $P_n^{\mathsf{DPF}}$ ever exceeds the bound $M$ when its execute HSS.Eval routine in the Eval function. Each HSS ciphertext output by HSSIC.Expand has its plaintext bounded by $\delta$ by definition. Each $f_i$ memory value in the RMS program is computed by performing multiplications of $2n/3$ $e_i$ values, each of which are bounded by $(\delta + 1)$. Hence, each

$f_i$ is bounded by $(\delta+1)^{2n/3} < M$, ensuring that the $\mathsf{HSS.Eval}$ does not output undefined values. The correctness of our 1-bit BDPF follows directly from the protocol description. We consider the $i^{th}$ DPF in the batched construction. As mentioned before, we have:

- The vector of HSS input encoding shares is

$$[\![\boldsymbol{\alpha}^\star]\!]_\sigma = \left([\![z_{n\cdot i}]\!]_\sigma, [\![z_{n\cdot i+1}]\!]_\sigma, \ldots, [\![z_{n\cdot i+n-1}]\!]\right)$$

- and $\mathsf{HD}(\mathsf{LSB}(\boldsymbol{\alpha}^\star), H(\alpha_i)) \leq 1$

To show the DPF gives the correct output for any $x \in \{0,1\}^n$ in the domain with all but negligible probability, we need to show that the RMS program $P_n^{\mathsf{DPF}}$ outputs 1 if and only if the two inputs satisfy $\mathsf{HD}(\mathsf{LSB}(\boldsymbol{\alpha}^\star), H(\alpha_i)) \leq 1$. We prove it by considering the following three cases:

- Case 1($x = \alpha_i$ and $\mathsf{LSB}(\boldsymbol{\alpha}^*) = H(\alpha_i)$): Let

$$h_0 = \Pi_{t=0\cdot n/3}^{(0+1)\cdot n/3-1} e_t, h_0 = \Pi_{t=0\cdot 2n/3}^{(1+1)\cdot n/3-1} e_t \text{ and } h_0 = \Pi_{t=0\cdot n}^{(2+1)\cdot n/3-1} e_t$$

  be products as computed in the RMS program. Then we have,

$$\mathsf{LSB}(h_0) = \bigwedge_{t=0\cdot n/3}^{(0+1)\cdot n/3-1} (\neg \mathbf{x}_t^H \oplus \boldsymbol{\alpha}_i[t])$$

$$\mathsf{LSB}(h_1) = \bigwedge_{t=1\cdot n/3}^{(1+1)\cdot n/3-1} (\neg \mathbf{x}_t^H \oplus \boldsymbol{\alpha}_i[t])$$

$$\mathsf{LSB}(h_2) = \bigwedge_{t=2\cdot n/3}^{(2+1)\cdot n/3-1} (\neg \mathbf{x}_t^H \oplus \boldsymbol{\alpha}_i[t])$$

$$\mathsf{LSB}(f_0) = \mathsf{LSB}(h_0 \cdot h_1) = \mathsf{LSB}(h_0) \wedge \mathsf{LSB}(h_1)$$

$$\mathsf{LSB}(f_1) = \mathsf{LSB}(h_1 \cdot h_2) = \mathsf{LSB}(h_1) \wedge \mathsf{LSB}(h_2)$$

$$\mathsf{LSB}(f_2) = \mathsf{LSB}(h_2 \cdot h_0) = \mathsf{LSB}(h_2) \wedge \mathsf{LSB}(h_0)$$

  Since $H(\alpha^*) = H(x)$,we have $\mathsf{LSB}(h_0) = \mathsf{LSB}(h_1) = \mathsf{LSB}(h_2) = 1$, which implies $z = \mathsf{LSB}(f_0) \oplus \mathsf{LSB}(f_1) \oplus \mathsf{LSB}(f_2) = 1$.

- Case 2 ($x = \alpha_i$ and $\mathsf{HD}(\mathsf{LSB}(\boldsymbol{\alpha}^\star), H(\alpha_i)) = 1$): Let $e^* = \mathsf{LSB}(\boldsymbol{\alpha}^\star) \oplus H(\alpha_i)$, which would be a unit vector. In this case, we have

$$\mathsf{LSB}(h_0) = \bigwedge_{t=0\cdot n/3}^{(0+1)\cdot n/3-1} (\neg \mathbf{x}_t^H \oplus \boldsymbol{\alpha}_i[t] \oplus e_t^*)$$

$$\mathsf{LSB}(h_1) = \bigwedge_{t=1\cdot n/3}^{(1+1)\cdot n/3-1} (\neg \mathbf{x}_t^H \oplus \boldsymbol{\alpha}_i[t] \oplus e_t^*)$$

Hybrid $H_0$:
Execute $(\mathsf{pk}, (\mathsf{ek}_0, \mathsf{ek}_1)) \leftarrow \mathsf{HSS.Setup}(1^\lambda)$.
$\boldsymbol{u} = H(\boldsymbol{\alpha}_0) \mid\mid H(\boldsymbol{\alpha}_1) \mid\mid \ldots \mid\mid H(\boldsymbol{\alpha}_{m-1}) \in \{0,1\}^{n \cdot m}$
$(\rho_0, \rho_1, \boldsymbol{z}) \leftarrow \mathsf{HSSIC.BIn}(\mathsf{pk}, \boldsymbol{u})$
Output $\mathsf{k}_c = (\mathsf{pk}, \mathsf{ek}_c, \rho_c)$

Hybrid $H_1$ (Simulator):
Execute $(\mathsf{pk}, (\mathsf{ek}_0, \mathsf{ek}_1)) \leftarrow \mathsf{HSS.Setup}(1^\lambda)$.
$(\rho_0, \rho_1, \boldsymbol{z}) \leftarrow \mathsf{HSSIC.BIn}(\mathsf{pk}, 0^{mn})$
Output $\mathsf{k}_c = (\mathsf{pk}, \mathsf{ek}_c, \rho_c)$

Figure 12: 1-bit Security Proof — Sequence of Hybrids

$$\mathsf{LSB}(h_2) = \bigwedge_{t=2\cdot n/3}^{(2+1)\cdot n/3-1} (\neg \mathbf{x}_t^H \oplus \boldsymbol{\alpha}_i[t] \oplus e_t^*)$$

Since $x = \alpha_i$ and exactly one position of $\mathbf{e}^\star$ is equal to 1, we have that exactly one of $\mathsf{LSB}(h_0)$, $\mathsf{LSB}(h_1), \mathsf{LSB}(h_2)$ is equal to zero, which implies that $z = \mathsf{LSB}(f_0) \oplus \mathsf{LSB}(f_1) \oplus \mathsf{LSB}(f_2) = 1$.

- Case 3 ($x \neq \alpha_i$): We have $z = \mathsf{LSB}(f_0) \oplus \mathsf{LSB}(f_1) \oplus \mathsf{LSB}(f_2) = 1$ only in one of the following two cases:

  (i) All bits $\mathsf{LSB}(f_0), \mathsf{LSB}(f_1), \mathsf{LSB}(f_2)$ are 1: This only happens when $\mathsf{LSB}(h_0) = \mathsf{LSB}(h_1) = \mathsf{LSB}(h_2) = 1$, which happens with probability $2^{-n}$ - since it requires $H(x) = H(\alpha_i)$.

  (ii) Exactly one of $\mathsf{LSB}(f_0), \mathsf{LSB}(f_1), \mathsf{LSB}(f_2)$ bits is 1: In this case, exactly one of $\mathsf{LSB}(h_0)$ , $\mathsf{LSB}(h_1)$, $\mathsf{LSB}(h_2)$ is equal to 0, which happens with probability $(1 - 2^{n/3}) \cdot 2^{-2n/3} \leq 2^{-2n/3}$.

  Hence, when $x \neq \alpha_i$, the DPF outputs shares of 0 with probability $< 2^{-2n/3} + 2^{-n} < 2^{-(2n/3-1)} \leq 2^{-\varepsilon}$.

**Secrecy.** For corrupt party $c \in \{0,1\}$, we construct two hybrids (see Figure 12), where the first hybrid is the corrupt party's DPF key, and the simulator is defined by the second hybrid. The second hybrid's output distribution can be simulated by a poly time algorithm given just the leakage function - containing the number of DPFs batched and the size of input domain. The two hybrids are indistinguishable given that the underlying $\mathsf{HSSIC}$ satisfies compressed-private input indistinguishably.

□