Lightweight, Maliciously Secure Verifiable Function Secret Sharing[⋆]

Leo de Castro[1] and Anitgoni Polychroniadou[2]

[1] MIT, `ldec@mit.edu`
[2] J.P. Morgan AI Research, `antigoni.polychroniadou@jpmorgan.com`

Abstract. In this work, we present a lightweight construction of verifiable two-party function secret sharing (FSS) for point functions and multi-point functions. Our verifiability method is lightweight in two ways. Firstly, it is concretely efficient, making use of only symmetric key operations and no public key or MPC techniques are involved. Our performance is comparable with the state-of-the-art *non-verifiable* DPF constructions, and we outperform all prior DPF verification techniques in both computation and communication complexity, which we demonstrate with an implementation of our scheme. Secondly, our verification procedure is essentially unconstrained. It will verify that distributed point function (DPF) shares correspond to some point function irrespective of the output group size, the structure of the DPF output, or the set of points on which the DPF must be evaluated. This is in stark contrast with prior works, which depend on at least one and often all three of these constraints. In addition, our construction is the first DPF verification protocol that can verify general DPFs while remaining secure even if one server is malicious. Prior work on maliciously secure DPF verification could only verify DPFs where the non-zero output is binary. As an additional feature, our verification procedure can be batched so that verifying a polynomial number of DPF shares requires the *exact* same amount of communication as verifying one pair of DPF shares. We combine this packed DPF verification with a novel method for packing DPFs into shares of a multi-point function where the evaluation time, verification time, and verification communication are *independent* of the number of non-zero points in the function.

An immediate corollary of our results are two-server protocols for PIR and PSI that remain secure when any one of the three parties is malicious (either the client or one of the servers).

1 Introduction

Function secret sharing (FSS), first introduced by Boyle, Gilboa, and Ishai [2], is a cryptographic primitive that extends the classical notion of secret-sharing a scalar value to secret sharing a function. FSS allows a party to secret-share a function $f\colon \mathcal{D} \to \mathbb{G}$ and produce function shares k_0 and k_1. These shares

[⋆] This is the full version of [6].

have several useful properties. Firstly, viewing either share alone computationally hides the function f. Secondly, the function shares can be evaluated at points in the domain $\mathcal{D}$ to produce additive shares of the output of f. In other words, for $x \in \mathcal{D}$, we have $\mathsf{k}_0(x) + \mathsf{k}_1(x) = f(x)$.

A point function $f : \mathcal{D} \to \mathbb{G}$ is defined by a single point $(\alpha, \beta) \in \mathcal{D} \times \mathbb{G}$ such that $f(\alpha) = \beta$ and for all $\gamma \neq \alpha$ we have $f(\gamma) = 0$. We will often denote the point function f defined by (α, β) as $f_{\alpha,\beta}$. Distributed point functions (DPFs), first introduced by Gilboa and Ishai [10], are a special case of FSS that supports point functions. Boyle, Gilboa, and Ishai [3] gave an efficient construction of a distributed point function.

An FSS construction is immediately applicable to the problem of constructing two-server protocols, where a client interacts with two servers that are assumed to not collude. Despite the simplicity of point functions, DPFs give rise to a rich class of two-server protocols, including private information retrieval (PIR) [3], private set intersection (PSI) [7], Oblivious-RAM [9], contact-tracing [8], and many more [1, 15]. These two-server protocols often have a similar structure. For example, a simple, semi-honest PIR construction from a DPF begins with a client generating DPF shares for the function $f_{i,1}$, where i is the query index, and the servers begin with identical copies of a database of size N. The client sends one function share to each server, and the servers evaluate the share on each index $i \in [N]$ to obtain a secret sharing of a one-hot vector. The servers then take the inner product with their copy of the database to obtain an additive share of the i^{th} element, which is returned to the client.

Verifiable DPF. A crucial barrier that must be overcome in order for many applications to be deployed in the real world is achieving some form of malicious security. For the two-server model, this often means verifying that the client's inputs are well-formed in order to ensure that the client does not learn unauthorized information about the servers' database or modify the database in an unauthorized way. A DPF scheme that supports this well-formedness check is called a verifiable DPF (VDPF).

In addition to constructing DPFs, the work of Boyle et al. [3] also constructs VDPFs that are secure when the servers are semi-honest; a malicious server is able to learn non-trivial information about the client's chosen point (α, β) through the verification procedure. Even to achieve semi-honest security, the VDPF protocol of [3] requires a constant-sized MPC protocol (consisting of several OLEs) to be run between the servers to verify the DPF. The recent work of Boneh, Boyle, Corrigan-Gibbs, Gilboa, and Ishai [1] achieves maliciously secure VDPFs when $\beta \in \{0, 1\}$, but they do not extend their protocol to support general β values. They also require a constant sized MPC (also a few OLEs) to be run between the servers to verify a single DPF share. More detail on these protocols is given in Section 1.2. These works leave open the problem of constructing a maliciously-secure VDPF for general β values, which we solve in this work.

Distributed Multi-Point Functions. While DPFs result in a surprisingly rich class of two-server protocols, in many applications [7,8] it is desirable to have FSS for functions with more than one nonzero point. We call these multi-point functions (MPFs) (see Section 4.2 for more details). Naively, this requires constructing a DPF for each nonzero point. To evaluate the naive MPF share the servers must perform a DPF evaluation for each nonzero point in the function. In other words, if a function contains t non-zero points and the servers wish to evaluate the MPF share at η points, then they must perform $t \cdot \eta$ DPF evaluations. This clearly wastes a tremendous amount of work, since we know that for each of the t DPF shares, at most one of the η evaluation points will map to a non-zero value, and yet each share is evaluated the full η times. To maintain efficient FSS for these multi-point functions, it is clear that a more efficient manner of batching DPF shares is required. Furthermore, the naive verifiable DMPF construction is simply a concatenation of many verifiable DPF shares, which means the complexity of the verification procedure grows linearly in t. Prior works have left open the problem of constructing DMPF shares with evaluation time and verification complexity sublinear in the number of nonzero points, which we solve in this work.

1.1 Our Contributions

Lightweight Verifiable DPF. We give a lightweight construction of a verifiable DPF, which admits a very efficient way to verify that DPF shares are well-formed. This construction is light-weight in two ways. First, it's performance is comparable with the state-of-the-art non-verifiable DPF constructions (within a factor of 2 in both communication and computation), as we show in Section 5. In addition, we strictly outperform all prior DPF verification methods in both communication and computation. These verification methods often have strictly stronger or incomparable constraints, such as remaining secure when the servers are semi-honest or only verifying if $\beta \in \{0, 1\}$. Unlike all other DPF verification methods [1,3], we do not make use of arithmetic MPC; for a security parameter λ, our verification procedure is a simple exchange of 2λ bits.

Second, the constraints on the verification procedure are essentially non-existent. We can verify that a DPF share is well-formed regardless of output field size, regardless of the value of the non-zero output element, and regardless of the set of evaluation points the servers choose. This is in stark contrast to prior works [1,3], which depend on at least one and often all of these constraints, as we describe in Section 1.2. Our method is able to verify that there is at most one non-zero value in any set of outputs of the DPF share, even if the set is adversarially chosen.

Efficient Batched Verification. Another novel feature of our verification procedure is efficient batching. When verifying any polynomial number of shares in our VDPF scheme, the communication for the verification procedure remains only an exchange of 2λ bits. This is because our verification procedure for a single pair of VDPF shares is to check if two 2λ-bit strings are equal (we explain how these strings are generated in Section 3), so the two servers are able to check if many pairs of 2λ-bit strings are equal by simply hashing all strings down into a single pair of 2λ-bit strings. This batching requires no additional computational overhead beyond hashing the strings together. Furthermore, this means that the communication to verify VDPF shares from *many different clients* is bounded at 2λ bits, and, to our knowledge, this is the first efficient cross-client batched VDPF verification procedure of its kind.

Verifiable FSS for Multi-Point Functions. Another immediate consequence of our batched verification procedure is verifiable FSS for multi-point functions. Our VDMPF scheme goes beyond the naive construction, which is to simply generate a pair of VDPF shares for each non-zero point in the multi-point function. As mentioned above, when using this naive method for an MPF with t non-zero points, a sever evaluating the MPF at η points needs to perform $t \cdot \eta$ VDPF evaluations. We show how a simple application of Cuckoo-hashing can reduce the number of VDPF evaluations to 3η *regardless* of the value of t. This is at the cost of the client needing to produce and send roughly 2× the number of VDPF shares as in the naive case, where these VDPF shares are at most the same size as in the naive case. For even moderately sized t (e.g. $t > 30$) this provides a tremendous savings in the overall computation time. Due to our batched verification, the communication between the two servers never grows beyond 2λ bits.

Ultimately, we will show the following two theorems.

Theorem 1 (Verifiable DPF (informal)). *There exists a verifiable DPF for any point function $f\colon \{0,1\}^n \to \mathbb{G}$ that remains secure even when one server is malicious. For security parameter λ, the runtime of share generation is $O(n\lambda)$, and the size of a function share is $O(n\lambda)$. For any $x \in \{0,1\}^n$, the runtime of share evaluation $O(n\lambda)$. For the verification procedure with η outputs, additional runtime is $O(\eta\lambda)$ and the communication between the two servers is $O(\lambda)$.*

Theorem 2 (Verifiable DMPF (informal)). *There exists is a verifiable DMPF for multi-point functions $f\colon [N] \to \mathbb{G}$ with at most t non-zero evaluation points that remains secure even when one server is malicious. For security parameter λ and a number of hash table buckets $m = O(t\lambda + t\log(t))$, the runtime of share generation is $O\left(m\lambda \log(N/m)\right)$. The runtime of share evaluation is $O(\lambda \log(N/m))$. For the verification procedure with η outputs, the additional runtime is $O(\eta\lambda)$ and the communication between the two servers is $O(\lambda)$.*

We implement our VDPF and VDMPF schemes and present benchmarks in Section 5.

Applications As a direct result of our verifiable FSS construction, we obtain several protocols in the two-server model that are secure against any one malicious corruption. More specifically, our verifiable DPF directly results in a maliciously-secure two-server PIR scheme and our verifiable DMPF directly results in a maliciously-secure two-server PSI scheme. These protocol follow immediately from our constructions, and we give these constructions in Appendix C and Appendix D.

1.2 Related Work

The most relevant related work is the verifiable DPF constructions of Boyle, Gilboa, and Ishai [3] and the subsequent work of Boneh, Boyle, Corrigan-Gibbs, Gilboa, and Ishai [1]. We begin with an overview of the DPF construction of [3], then discuss the semi-honest verification protocols presented in [3]. We also briefly discuss the malicious verification procedure in [1], which handles the case where $\beta \in \{0,1\}$.

Overview of the Boyle et al. [3] construction. The DPF construction of Boyle et al. [3] is a function secret sharing scheme for point functions in the two-server model. As described in Section 2.2, a distributed point function scheme allows a client to run an algorithm $(\mathsf{k}_0, \mathsf{k}_1) \leftarrow \mathsf{Gen}(1^\lambda, f_{\alpha,\beta})$, where $f_{\alpha,\beta} \colon \{0,1\}^n \rightarrow \mathbb{G}$ is a point function. This client can then send k_0 to a server $\mathcal{S}_0$ and send k_1 to a server $\mathcal{S}_1$. A single share k_b completely hides the function $f_{\alpha,\beta}$; it completely hides the location and value of the non-zero point, but not necessarily the fact that $f_{\alpha,\beta}$ is a point function. For any $x \in \{0,1\}^n$, the servers are able to compute $\mathsf{y}_b = \mathsf{Eval}(b, \mathsf{k}_b, x)$, such that $\mathsf{y}_0 + \mathsf{y}_1 = f_{\alpha,\beta}(x)$.

The construction of [3] begins with the observation that a point function differs from the zero function (the function that outputs zero on every input) on at most a single point. Therefore, they begin with the following protocol to share the zero function. The zero function can be shared by giving each server an identical copy of a PRF along with an additional bit that indicates if the output should be negated. The servers $\mathcal{S}_0$ and $\mathcal{S}_1$ can then evaluate their PRF on the same input x, then server $\mathcal{S}_1$ negates the output. The scheme will produce outputs from the same input x that sum to zero, making this a secret sharing of the zero function.

We can then instantiate this PRF using the GGM [11] construction, where the PRF is evaluated by expanding a tree of PRGs, and the output of the PRF is a leaf of this tree. Each input to the PRF will arrive at a unique leaf and will have a distinct path through the tree. To turn this zero function into a point function, we need to puncture a single path in this tree. In other words, we need to ensure that there is exactly one path in the tree where the values at the GGM nodes differ. For a point function $f_{\alpha,\beta}$, the path through the tree and the location of the leaf corresponds to α, and the value at the leaf corresponds to β. Since all other paths will have matching nodes, they will result in matching leaves, which become additive shares of zero. The GGM nodes along the punctured path will differ, which will result in this path terminating in leaves that do not match. We will discuss in Section 3 how to arrange operations at the final level to turn this one specific mismatched pair into additive shares of the desired non-zero output β.

As we traverse the tree, we maintain the following invariant. If we are along the punctured path (the path leading to the leaf at position α), the PRG seeds should differ. If we are not along the punctured path, then the PRG seeds should be the same. At each level in the tree, we need to ensure that as soon as we diverge from the punctured path, the seeds will match again, bringing us back to the zero-function state. To achieve this, a *correction operation* is applied at each GGM node as we traverse the tree. One of the core contributions of [3] is a method to achieve this correction operation. We make use of the same correction operation and exploit its useful properties to obtain our VDPF construction. More details are provided in Section 1.3.

Verification procedure of [3]. We will now provide a high-level overview of the verification procedures for the DPF construction of [3]. One of the main features of these verification procedures is that they view the DPF as a black-box. With this view, the task becomes taking a secret-shared vector $\mathsf{y} = \mathsf{y}_0 + \mathsf{y}_1$ of length N and verifying that the shared values are non-zero in at-most one location. There are several different protocols presented in [3] to achieve this, although all follow a basic template. These verification procedures all begin by sampling a linear sketching matrix L from a distribution $\mathcal{L}$ with N columns and a small constant number of rows. Each server $\mathcal{S}_b$ multiplies their share y_b by L to obtain a short vector z_b. The servers then run a simple MPC procedure is run to verify that z_0 and z_1 are well-formed. These verification procedures are only secure when the servers are semi-honest.

To give an example, the verification for $\beta \in \{0,1\}$ with an output field $\mathbb{F}$ begins by defining a matrix $L \in F^{3\times N}$. Each column j of L is defined to be $L_{1,j} = r_j$, $L_{2,j} = s_j$, and $L_{3,j} = r_j s_j$, where r_j and s_j are sampled uniformly at random over $\mathbb{F}$. The two servers begin with PRG seeds to locally generate this L matrix. The servers then locally compute $L \cdot \mathsf{y}_b = \mathsf{z}_b$, which is a secret sharing of three elements z_1, z_2, and z_3. Finally, the servers run an MPC protocol to check if $z_3 = z_1 z_2$. In [3], it is shown that the probability this check passes if y is not the zero-vector or a unit vector is at most $2/|\mathbb{F}|$. For security parameter λ, this means that $|\mathbb{F}|$ must be at least $2^{\lambda+1}$; all verification procedures given in [3] require that $|\mathbb{F}|$ must be $O(2^\lambda)$.

To obtain a verification for a general β, we can simply take the protocol for $\beta \in \{0,1\}$ and slightly modify the verification procedure to account for a $\beta \neq \beta^2$. In particular, the servers run the same protocol as above, but the final MPC check now verifies that $\beta \cdot z_3 = z_1 z_2$. The client provides a secret-sharing of β to the servers to allow them to compute shares of the product $\beta \cdot z_3$. We conclude this description by noting that this construction is vulnerable to additive attacks, where a malicious server can learn non-trivial information about the client's point (α, β).

Verification procedure of [1]. We now briefly describe the maliciously secure verification procedure of [1]. Recall that this approach verifies if DPF shares are well formed and if $\beta \in \{0,1\}$. At a high level, this approach is an extension of the check for binary β described above with checksum values added to defend against additive attacks from a malicious server. Instead of sending a single DPF share corresponding to the point (α, β), this VDPF scheme consists of two DPF shares: one defined by the original point (α, β) and the other defined by the point $(\alpha, \kappa \cdot \beta)$, where κ is a uniformly random element of the output field $\mathbb{F}$. Intuitively, the purpose of this κ value is to defend against a malicious server learning information about the non-zero point by applying additive shifts to candidate α locations. However, the value κ must also be included in the sketching checks in order to verify the consistency of the two DPF shares. This task is prone to error. For example, if the servers were simply given shares of κ directly, then a malicious server could learn if β was 0 if it applied an additive shift to κ and the verification still passed. This would occur because both β and $\kappa \cdot \beta$ are 0 when β is 0. The overcome this, the method of [1] embeds the κ value in an OLE correlation that the client sends to the servers. We omit the details, but we note that, at the time of this writing, there is no published method to extend this binary check to a general check in the same way as in [3]. We conclude this description by summarizing the complexity of this approach. The computational overhead of the verification includes evaluating the second DPF share that encodes the checksum, as well as sampling the sketching matrix. Then the sketching matrix must be multiplied by the output DPF vectors, which is a constant number of length N inner-products. The communication of the verification procedure consists of 4 elements of $\mathbb{F}$ sent by each server over two rounds of communication.

1.3 Technical Overview

We now give a brief technical overview of our verification methods.

Our DPF verification procedure. In our work, we observe that the correction operation of [3] is limited in a way that is useful for us. In particular, the correction operation is designed to correct *at most* one difference per level. With this observation, we can construct a simple verification procedure by extending the GGM tree by one level. This level takes each leaf in the original tree and produces two children: a left leaf and a right leaf. The left leaves can all be equivalent to the parent, while the right leaves should all correspond to a zero output. This means that all pairs of right leaves in the two DPF shares should be *equal.* Since the correction word can only correct one difference, if all the right leaves are the same, then there can be at most one difference in the previous level, meaning that all but one of the left leaves must be the same. The location of the differing left leaf is the α value, and the value produced by the differing leaves is the β value. All other left leaves will be equal, thus corresponding to the zero output. Since the DPF evaluation is completely deterministic, the α and β values that define the point function can be deterministically extracted from any pair of DPF shares that pass this verification check, meaning that regardless of the method a malicious client uses to generate the DPF shares, if the verification check passes then the servers have the guarantee that the shares encode a DPF defined by this (α, β) pair.

Furthermore, the servers need only check that all of their right leaves (which they can hash down into a single string of length 2λ) are equal, meaning that if the DPF shares are well formed the servers' messages to each other are perfectly simulatable. Therefore, this verification method introduces zero additional privacy risk to an honest client even when one of the servers is malicious. This security extends to verifying a polynomial number of DPF shares from the same client or different clients, since an equality check that will always pass for honest clients will not leak information about the honest clients' choices of α and β.

For more details and intuition about this approach, see Section 3.

Multi-point function packing. A natural application of our packed verification technique is to verify multi-point functions, since the communication for the verification will not increase as the number of non-zero points grows. As discussed above, the naive construction of a multi-point function with t non-zero points requires time linear in t for each evaluation. This is prohibitively expensive for applications where the servers must evaluate the DMPF at many points. To avoid this linear scaling of the evaluation time, we observe that in the list of tuples $(\alpha_1, \beta_1), \ldots, (\alpha_t, \beta_t)$ all of the α values are unique. Therefore, for each evaluation point x, the output of at most one of the DPFs will be nonzero, which occurs if $x = \alpha_i$. This means that our DMPF evaluation algorithm only needs to guarantee that the evaluation point x will be evaluated on the DPF with $\alpha_i = x$ if this point is nonzero in the MPF.

Towards this goal, we have the client insert the values $\alpha_1, \ldots, \alpha_t$ into a Cuckoo-hash table. At a high level, a Cuckoo-hash table is defined by a list of m buckets and $\kappa = O(1)$ of hash functions $h_1, \ldots, h_\kappa$ (in our case we use $\kappa = 3$). Each hash function has an output space that is $\{1, \ldots, m\}$, and each element α inserted in the table is at an index $i = h_j(\alpha)$ for some $1 \leq j \leq \kappa$.

The client constructs a DMPF share from this Cuckoo-hash table by creating a DPF share for each bucket in the table. The empty buckets will hold a DPF that shares the zero function, and the buckets that hold an index α_i hold a DPF that shares f_{α_i,β_i}. The domain of this DPF can be the same as the domain of the MPF, so the index of this non-zero point is simply α_i. The client then sends these m DPF shares to the server along with the hash functions defining the Cuckoo-hash table. To evaluate a point x on these shares, the servers simply hash x with each of the κ hash functions to get κ candidate buckets. If $x = \alpha_i$ for one of the non-zero points in the MPF, the guarantee is that α_i is in one of these buckets, so the servers evaluate only these κ DPFs at x then sum the result. This is an MPF evaluation procedure with a runtime that does not grow with the number of non-zero points in the MPF. In Section 4, we discuss a variant of this method that allows the domain of the DPFs for Cuckoo-hash buckets to *shrink* as the number of nonzero points grows, further speeding up evaluation time.

To maintain client privacy, we must ensure that the Cuckoo-hash table does not leak information about the client's choice of $\alpha_1, \ldots, \alpha_t$ values. The only information that is leaked from the Cuckoo-hash table is that the client's choice of non-zero indices did not fail to be inserted in this Cuckoo-hash table. Therefore, we must choose parameters for the Cuckoo-hash table such that the probability of failure is at most $2^{-\lambda}$ for security parameter λ. For a fixed κ, the failure probability of a Cuckoo-hash table shrinks as the number of buckets m increases. For $\kappa = 3$ and $\lambda = 80$, we get an upper bound on the number of buckets as $m \leq 2t$ for nearly all practical values of t. Therefore, for less than a 2× growth in the client computation and the client-server communication, we can achieve this significant improvement in the servers' evaluation time.

To verify that these DMPF shares are well-formed, we can simply put verifiable DPF shares in the buckets. Since the DPF verification method can be packed, the communication between the two servers never grows beyond the 2λ equality check, and the security against a malicious server is maintained. Our only sacrifice is in the verification of the number of nonzero points in the DMPF. The naive approach would allow the servers to verify that there are at most t nonzero points in the MPF, while the Cuckoo-hashing approach only allows the servers to verify that there are at most m nonzero points in the MPF. However, as we mentioned, for nearly all settings of t we get $m \leq 2t$, and we believe this gap is acceptable for many applications.

2 Background

2.1 Notation

Let $\mathcal{T}$ be a complete binary tree with 2^n leaves. If we index each leaf from 0 to $2^n - 1$, let $v_\alpha^{(n)}$ be the leaf at index $\alpha \in \{0,1\}^n$. Let $v_\alpha^{(i)}$ be the node at the i^{th} level of $\mathcal{T}$ such that $v_\alpha^{(n)}$ is in the subtree rooted at $v_\alpha^{(i)}$. We will sometimes refer to $v_\alpha^{(i)}$ as the i^{th} node along the path to α.

For a finite set S, we will denote sampling a uniformly random element x as $x \xleftarrow{\$} S$.

For $n \in \mathbb{N}$, we denote the set $[n] := \{1, \ldots, n\}$.

We will denote a set of n-bit strings as either $\{x_i\}_{i=1}^{L}$ or simply as $\mathbf{x}$ if the length is not relevant or clear from context.

For a parameter λ, we say that a function $\mathsf{negl}(\lambda)$ is *negligible* in λ if it shrinks faster than all polynomials in λ. In other words, for all polynomials $\mathsf{poly}(\lambda)$, there exists a λ' such that for all $\lambda > \lambda'$ we have $\mathsf{negl}(\lambda) < \mathsf{poly}(\lambda)$.

2.2 Function Secret Sharing

In this section, we give a high level definition of function secret sharing and distributed point functions. A function secret sharing scheme takes a function $f\colon \mathcal{D} \to \mathcal{R}$ and generates two *function shares* f_0 and f_1. These function shares can be evaluated a points $x \in \mathcal{D}$ such that $f_b(x) = y_b$ and $y_0 + y_1 = y = f(x)$. In other words, when evaluated at an input x the function shares produce additive secret shares of the function output. It is currently an open problem to construct an efficient FSS scheme where the function is split into more than two shares [4].

Definition 1 (Function Secret Sharing, Syntax & Correctness [2, 3]). *A function secret sharing scheme is defined by two PPT algorithms. These algorithms are parametrized by a function class $\mathcal{F}$ of functions between a domain $\mathcal{D}$ and a range $\mathcal{R}$.*

- $\mathsf{FSS.Gen}(1^\lambda, f \in \mathcal{F}) \to (\mathsf{k}_0, \mathsf{k}_1)$
  The $\mathsf{FSS.Gen}$ *algorithm takes in a function $f \in \mathcal{F}$ and generates two FSS keys k_0 and k_1.*
- $\mathsf{FSS.Eval}(b, \mathsf{k}_b, x \in \mathcal{D}) \to y_b$
  The $\mathsf{FSS.Eval}$ *algorithm takes in an $x \in \mathcal{D}$ and outputs an additive share $y_b \in \mathcal{R}$ of the value $y = f(x)$. In other words, $y_0 + y_1 = y = f(x)$.*

We now give the basic security property that an FSS scheme must satisfy.

Definition 2 (FSS Security: Function Privacy [2,3]). *Let* FSS *be a function secret sharing scheme for the function class $\mathcal{F}$, as defined in Definition 1. For any $f, f' \in \mathcal{F}$, the following should hold:*

$$\begin{aligned}&\{\mathsf{k}_b \mid (\mathsf{k}_0, \mathsf{k}_1) \leftarrow \mathsf{FSS.Gen}(\{0,1\}, f)\} \approx_c \\ &\qquad \{\mathsf{k}'_b \mid (\mathsf{k}'_0, \mathsf{k}'_1) \leftarrow \mathsf{FSS.Gen}(\{0,1\}, f')\}, \quad \textit{for } b \in \{0,1\}\end{aligned}$$

In words, the marginal distribution of one of the FSS keys computationally hides the function used to compute the share.

We now give the definition of a distributed point function (DPF) in terms of the FSS definitions above. We begin by defining a point function.

Definition 3 (Point Function). *A function $f\colon \mathcal{D} \to \mathcal{R}$ is a point function if there is $\alpha \in \mathcal{D}$ and $\beta \in \mathcal{R}$ such that the following holds:*

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

Throughout this work, we will be interested in point functions with domain $\mathcal{D} = \{0,1\}^n$ and range $\mathcal{R} = \mathbb{G}$ for a group $\mathbb{G}$.

Definition 4 (Distributed Point Function). *Let $\mathcal{F}_{n,\mathcal{G}}$ be the class of point functions with domain $\mathcal{D} = \{0,1\}^n$ and range $\mathcal{R} = \mathbb{G}$. We call an FSS scheme a Distributed Point Function scheme if it supports the function class $\mathcal{F}$.*

3 Lightweight, Verifiable DPF

3.1 Definitions

We begin by defining a verifiable DPF. We define correctness and security for a batched evaluation, since the verification procedure operates is defined over a set of outputs. The procedure ensures that at-most one of these outputs is non-zero.

Definition 5 (Verifiable DPF, denoted $\mathsf{VerDPF}_{n,\mathbb{G}}$). *A verifiable distributed point function scheme* $\mathsf{VerDPF}_{n,\mathbb{G}}$ *supports the function class* $\mathcal{F}$ *of point functions* $f\colon \{0,1\}^n \to \mathbb{G}$*. It is defined by three PPT algorithms. Define* $\mathsf{VerDPF} := \mathsf{VerDPF}_{n,\mathbb{G}}$*.*

- $\mathsf{VerDPF.Gen}(1^\lambda, f_{\alpha,\beta}) \to (\mathsf{k}_0, \mathsf{k}_1)$
  This is the FSS share generation algorithm. It takes in a function $f_{\alpha,\beta}$ *and generates two shares* k_0 *and* k_1*.*
- $\mathsf{VerDPF.BVEval}(b, \mathsf{k}_b, \{x_i\}_{i=1}^L) \to \left(\{y_b^{(x_i)}\}_{i=1}^L, \pi_b\right)$
  This is the verifiable evaluation algorithm, denoted BVEval *for batch verifiable evaluation. It takes in a set of* L *inputs* $\{x_i\}_{i=1}^L$*, where each* $x_i \in \{0,1\}^n$*, and outputs a tuple of values. The first set of values are the FSS outputs, which take the form* $y_b^{(x_i)}$ *for* $i \in [L]$ *satisfying* $y_0^{(x_i)} + y_1^{(x_i)} = f(x_i)$*. The second output is a proof* π_b *that is used to verify the well-formedness of the output.*
- $\mathsf{VerDPF.Verify}(\pi_0, \pi_1) \to \mathsf{Accept}/\mathsf{Reject}$
  For some pair of VerDPF *keys* $(\mathsf{k}_0, \mathsf{k}_1)$*, the* $\mathsf{VerDPF.Verify}$ *algorithm takes in the proofs* π_0 *and* π_1 *from* $(\mathsf{y}_b, \pi_b) \leftarrow \mathsf{BVEval}(b, \mathsf{k}_b, \{x_i\}_{i=1}^L)$ *and outputs either* Accept *or* Reject*. The output should only be* Accept *if* $\mathsf{y}_0 + \mathsf{y}_1$ *defines the truth table of some point function, which occurs if it is non-zero in at most one location.*

Correctness for a verifiable DPF is defined in the same way as correctness for any FSS scheme, as in Definition 1. To verify that the entire share is well-formed, the BVEval algorithm can be run on the whole domain. We give a more efficient algorithm for evaluating our verifiable DPF on the whole domain in Algorithm 3, which uses techniques from Boyle et al. [3] to save a factor of n on the overall runtime.

We now define security for a verifiable DPF. Note that we are only interested in detecting a malformed share when the evaluators are semi-honest. However, we do require that even a malicious evaluator does not learn any information about the shared function; in other words, we require that the verification process does not compromise the function privacy of an honestly generated DPF share if one of the evaluators is malicious.

Definition 6 (Verifiable DPF Share Integrity, or Security Against Malicious Share Generation). *Let* $\mathsf{VerDPF} := \mathsf{VerDPF}_{n,\mathbb{G}}$*, and let* k_b *be the (possibly maliciously generated) share received by server* S_b*. For an adversarially chosen set of inputs* $\{x_i\}_{i=1}^\eta$*, let* $(\{y_b^{(x_i)}\}_{i=1}^\eta, \pi_b) \leftarrow \mathsf{VerDPF.BVEval}(b, \mathsf{k}_b, \{x_i\}_{i=1}^\eta)$*. We say that* VerDPF *is secure against malicious share generation if the the following holds with all but negligible probability over the adversary's choice of randomness. If* $\mathsf{VerDPF.Verify}(\pi_0, \pi_1)$ *outputs* Accept*, then the values* $y_0^{(x_i)} + y_1^{(x_i)}$ *must be non-zero in at most one location.*

Definition 7 (Verifiable DPF Function Privacy, or Security Against a Malicious Evaluator). *Let* $\mathsf{VerDPF} := \mathsf{VerDPF}_{n,\mathbb{G}}$ *support the class of point* *functions $\mathcal{F}$ with domain $\{0,1\}^n$ and range $\mathbb{G}$. For a set of function inputs $\mathbf{x}$, define the distribution representing the view of server $\mathcal{S}_b$ for a fixed function $f \in \mathcal{F}$.*

$$\mathsf{View}_{\mathsf{VerDPF}}(b, f, \mathbf{x}) := \left\{ (\mathsf{k}_b, \pi_{1-b}) \;\middle|\; \begin{array}{c} (\mathsf{k}_0, \mathsf{k}_1) \leftarrow \mathsf{VerDPF.Gen}(1^\lambda, f), \\ (\_, \pi_{1-b}) \leftarrow \mathsf{VerDPF.BVEval}(1-b, \mathsf{k}_{1-b}, \mathbf{x}) \end{array} \right\}$$

We say that VerDPF *maintains function privacy if there exists a PPT simulator* Sim *such that for any adversarially chosen* $\mathbf{x}$ *the following two distributions are computationally indistinguishable for any $f \in \mathcal{F}$.*

$$\left\{(\mathsf{k}_b, \pi_{1-b}) \mid (\mathsf{k}_b, \pi_{1-b}) \leftarrow \mathsf{View}_{\mathsf{VerDPF}}(b, f, \mathbf{x})\right\} \approx_c \left\{(\mathsf{k}^*, \pi^*) \mid (\mathsf{k}^*, \pi^*) \leftarrow \mathsf{Sim}(1^\lambda, b, n, \mathbb{G}, \mathbf{x})\right\}$$

3.2 Our Construction

In this DPF scheme, the shares of the point function are k_0 and k_1. Each key k_b contains a starting seed $s_b^{(0)}$ that defines the root of a GGM-style binary tree, where at each node there is a PRG seed that is expanded into two seeds that comprise the left and the right child of that node. However, the seeds that define the left and right children are not the direct output of the PRG; instead, we apply a correction operation to the PRG output in order to maintain the required property of these trees, which we call the "DPF invariant."

In addition to the PRG seed, each node is associated with a *control bit*, which is one additional bit of information that is updated along with the seed during the correction operation. This control bit is used in the correction operation, and its purpose is to maintain the DPF invariant.

Definition 8 (DPF **Invariant**). *Let* $\mathsf{DPF} = \mathsf{DPF}_{n,\mathbb{G}}$, *and let*

$$(\mathsf{k}_0, \mathsf{k}_1) \leftarrow \mathsf{DPF.Gen}(1^\lambda, f_{\alpha,\beta})$$

for $\alpha \in \{0,1\}^n$. Each key k_b defines a binary tree $\mathcal{T}_b$ with 2^n leaves, and each node in the tree is associated with a PRG seed and a control bit.

For a fixed node location, let s_0, t_0 be the seed and control bit associate with the node in $\mathcal{T}_0$, and let s_1, t_1 be the seed and control bit associated with the node in $\mathcal{T}_1$. The DPF invariant is defined as the following:

$$\begin{array}{rl} s_0 = s_1 \text{ and } t_0 = t_1 & \text{if the node is not along the path to } \alpha. \\ t_0 \neq t_1 & \text{if the node is along the path to } \alpha. \end{array}$$

In our construction, it is also very likely that $s_0 \neq s_1$ if the node is along the path to α, but this requirement is not necessary for the invariant.

From this invariant, we maintain that at each level there is exactly one place in which the two trees differ, which is the node in that level corresponding to

Algorithm 1 $\mathsf{VerDPF}_{n,\mathbb{G}}$ node expansion, denoted $\mathsf{NodeExpand}$. This algorithm describes generating the child nodes from the parent node in the DPF tree.

```
Input: PRG $\mathcal{G}\colon \{0,1\}^\lambda \to \{0,1\}^{2\lambda+2}$
       Seed $s \in \{0,1\}^\lambda$, control bit $t \in \{0,1\}$.
       Correction word $\mathsf{cw} = (s_c, t_c^L, t_c^R)$.
1: Expand $(s^L, t^L, s^R, t^R) \leftarrow \mathcal{G}(s)$
2: $s_0' \leftarrow \mathsf{correct}(s^L, s_c, t)$ and $t_0' \leftarrow \mathsf{correct}(t^L, t_c^L, t)$
3: $s_1' \leftarrow \mathsf{correct}(s^R, s_c, t)$ and $t_1' \leftarrow \mathsf{correct}(t^R, t_c^R, t)$
Output: $(s_0', t_0'), (s_1', t_1')$
```

the path to α. At the final level, all of the 2^n leaves in both trees will be the same, except at position α. We can define deterministic transformations on the values at the leaves such that leaves with the same value produce additive shares of zero. These transformations are each determined by the control bit, and symmetry in the control bits results in symmetric application of these deterministic operations. At the leaf where the values differ, the invariant tells us that the control bits will differ, and we can take advantage of this asymmetry to produce additive shares of β at this pair of leaves.

In order to maintain the invariant in Definition 8, we perform a correction operation at each node as we traverse the tree. Each level of the tree is associated with a correction word. At each node, we perform the PRG expansion defined in Definition 9, then apply the correction operation define in Definition 10 to compute the seeds and control bits for the left and right children.

Definition 9 (VerDPF **PRG Expansion** [3]). *Let $s \in \{0,1\}$ be a seed for the PRG $\mathcal{G}\colon \{0,1\}^\lambda \to \{0,1\}^{2\lambda+2}$. Define the PRG expansion of the seed s as follows:*

$$s^L || t^L || s^R || t^R \leftarrow \mathcal{G}(s)$$

where $s^L, s^R \in \{0,1\}^\lambda$ and $t^L, t^R \in \{0,1\}$.

Definition 10 (VerDPF **Correction Operation** [3]). *The* VerDPF *correction operation*

$$\mathsf{correct}_{\mathbb{G}}\colon \mathbb{G} \times \mathbb{G} \times \{0,1\} \to \mathbb{G}$$

is defined as follows:

$$\mathsf{correct}_{\mathbb{G}}(\xi_0, \xi_1, t) = \begin{cases} \xi_0 & \text{if} \quad t = 0 \\ \xi_0 + \xi_1 & \text{if} \quad t = 1 \end{cases}$$

When $\mathbb{G}$ is not defined, the group $\mathbb{G}$ is taken to be $\mathbb{Z}_2^\ell$ for some positive integer ℓ. In particular, this makes the group addition operation the component-wise XOR of ξ_0 and ξ_1.

From the node expansion described in Algorithm 1, it becomes clear what the correction word must be in order to maintain that only one pair of nodes differ at each level of the tree. In particular, if the bit x_i disagrees with α_i, the corresponding bit of α, then the correction word must ensure that seeds and controls bits in the next level match. We define the correction word generation algorithm in Algorithm 2.

Algorithm 2 $\mathsf{VerDPF}_{n,\mathbb{G}}$ correction word generation, denoted CWGen.

```
Input: PRG $\mathcal{G}\colon \{0,1\}^\lambda \to \{0,1\}^{2\lambda+2}$
       Left seed $s_0$, left control bit $t_0$.
       Right seed $s_1$, right control bit $t_1$.
       Bit $x$ of the input.
Expand $(s_b^L, t_b^L, s_b^R, t_b^R) \leftarrow \mathcal{G}(s_b)$ for $b \in \{0,1\}$.
if $x = 0$ then $\mathsf{Diff} \leftarrow L$, $\mathsf{Same} \leftarrow R$        ▷ Set the right children to be equal.
else $\mathsf{Diff} \leftarrow R$, $\mathsf{Same} \leftarrow L$        ▷ Set the left children to be equal.
$s_c \leftarrow s_0^{\mathsf{Same}} \oplus s_1^{\mathsf{Same}}$
$t_c^L \leftarrow t_0^L \oplus t_1^L \oplus 1 \oplus x$        ▷ Ensure that the left control bits are not equal iff $x = 0$.
$t_c^R \leftarrow t_0^R \oplus t_1^R \oplus x$        ▷ Ensure that the right control bits are not equal iff $x = 1$.
$\mathsf{cw} \leftarrow s_c || t_c^L || t_c^R$
$s_b' \leftarrow \mathsf{correct}(s_b^{\mathsf{Diff}}, s_c, t_b^{(i-1)})$ for $b \in \{0,1\}$.
$t_b' \leftarrow \mathsf{correct}(t_b^{\mathsf{Diff}}, t_c^{\mathsf{Diff}}, t_b)$ for $b \in \{0,1\}$.
Output: $\mathsf{cw}, (s_0', t_0'), (s_1', t_1')$
```

Intuitively, our construction takes advantage of the fact that the correction words in the DPF construction of Boyle et al. can only correct at most one difference in each level. In our construction, we extend the GGM tree by one level, extending the DPF evaluation to all of the left children. In addition, at the final level we replace the PRG with a hash function H sampled from a family $\mathcal{H}$ that is collision-resistant and correlation-intractable for an XOR correlation defined below. We then have the servers check that all of their right children are the same by hashing all right children and exchanging the hash value.

In an honest pair of function shares, the trees should only differ at one node at each level, and in the final level the only difference should be in one of the left children. The collision resistance of the hash function ensures that any difference in the second-to-last level will result in a difference in the right children. This forces the correction word to correct these right children in order for the consistency check to pass. Since the correction word can correct at most one difference in the right children, this will guarantee that all other right children are the same because their parents are the same, which, in turn, implies that all corresponding left children are the same.

As discussed above, it is straightforward to turn matching leaf nodes into additive shares of zero, although we will have to generate the final control bit slightly differently in this final level to ensure that this conversion is performed correctly. In particular, we generate these control bits deterministically from the seeds, which ensures that matching seeds will result in matching control bits. For the non-zero output, we will have the honest client generate the function shares until the control bits at the non-zero point are different. If a malicious client samples shares such that these bits are the same, this will simply correspond to a different choice of β.

To securely instantiate the final level of the tree, our hash function family must be collision resistant, which will ensure that a difference in the previous level will translate to a difference in the children. We will also require the hash function to be secure against a similar, but incomparable, correlation, which we call *XOR-collision resistance.* Intuitively, satisfying this definition will ensure that each correction seed will only be able to correct one difference in the right children.

Definition 11 (XOR-Collision Resistance). *We say a function family $\mathcal{F}$ is XOR-collision resistant if no PPT adversary given a randomly sampled $f \in \mathcal{F}$ can find four values $x_0, x_1, x_2, x_3 \in \{0,1\}^\lambda$ such that $(x_0, x_1) \neq (x_2, x_3)$, $(x_0, x_1) \neq (x_3, x_2)$, and $f(x_0) \oplus f(x_1) = f(x_2) \oplus f(x_3) \neq 0$ with probability better than some function $\mathsf{negl}(\lambda)$ that is negligible in λ.*

To satisfy this definition, our hash function output has length 4λ, since we must defend against a birthday-attack where the adversary is searching for a colliding 4-tuple. This is explained in more detail in Appendix A. With a hash function satisfying this definition, we will be able to argue that if an adversary can construct invalid VerDPF keys that pass the consistency check, then this adversary has found either a collision or an XOR-collision in the hash function.

We define the VerDPF key in Definition 12. The full verifiable DPF construction is given in Figure 1.

Definition 12 (VerDPF Function Share). *Let $\mathsf{VerDPF}_{n,\mathbb{G}}$ be our verifiable DPF scheme. Let λ be the security parameter. A function share contains the following elements.*

- *Starting seed $s^{(0)} \in \{0,1\}^\lambda$.*
- *Correction words $\mathsf{cw}_1, \ldots, \mathsf{cw}_n$, where each $\mathsf{cw}_i \in \{0,1\}^\lambda \times \{0,1\} \times \{0,1\}$.*
- *One additional correction seed $\mathsf{cs} \in \{0,1\}^{4\lambda}$, which corrects differences in the final level. Corrections to the control bits are not necessary at the final level.*
- *A final output correction group element $\mathsf{ocw} \in \mathbb{G}$.*

Lemma 1 (VerDPF Correctness). *The VerDPF scheme defined in Figure 1 defines a correct verifiable DPF scheme.*

Proof. If we ignore the last level of the DPF expansion, our DPF is essentially the same as the DPF construction of Boyle et al. [3]. The only difference is the way the final control bits are generated. The control bits for the nodes that correspond to zero outputs will be the same, since the seeds for these leaves will also be the same. In the key generation, the seeds are sampled such that the control bits for the leaf at position α will differ, allowing the selective XOR of the final correction word. Since the $\mathsf{correct}$ operation is deterministic, the nodes

Verifiable Distributed Point Function $\mathsf{VerDPF}_{n,\mathbb{G}}$**.**
Let $\mathsf{VerDPF} := \mathsf{VerDPF}_{n,\mathbb{G}}$. Let $\mathcal{G}\colon \{0,1\}^\lambda \to \{0,1\}^{2\lambda+2}$ be a PRG. Let $\mathsf{H}\colon \{0,1\}^{n+\lambda} \to \{0,1\}^{4\lambda}$ be a hash function sampled from a family $\mathcal{H}$ that is both collision-resistant and XOR-collision-resistant. Let $\mathsf{H}'\colon \{0,1\}^{4\lambda} \to \{0,1\}^{2\lambda}$ be a hash function sampled from a family $\mathcal{H}'$ that is collision-resistant. Let $\mathsf{convert}\colon \{0,1\}^\lambda \to \mathbb{G}$ be a map converting a random λ-bit string to a pseudorandom element of $\mathbb{G}$. Let $\mathsf{LSB}\{0,1\}^\ell \to \{0,1\}$ be the function that takes any bit-string and extracts the least significant bit.
The $\mathsf{VerDPF.Verify}$ algorithm simply checks if the two input proofs are equal.

$\mathsf{VerDPF.Gen}$

```
Input: Security parameter $1^\lambda$ and point function $f_{\alpha,\beta}\colon \{0,1\}^n \to \mathbb{G}$
 1: Sample $s_0^{(0)} \leftarrow \{0,1\}^\lambda$ and $s_1^{(0)} \leftarrow \{0,1\}^\lambda$. Set $t_0^{(0)} = 0$ and $t_1^{(0)} = 1$.
 2: Let $\alpha_1, \ldots \alpha_n$ be the bits of $\alpha$.
 3: for $i$ from 1 to $n$ do
 4:     $\mathsf{vals} \leftarrow \mathsf{CWGen}(\mathcal{G}, s_0^{(i-1)}, t_0^{(i-1)}, s_1^{(i-1)}, t_1^{(i-1)}, \alpha_i)$
 5:     Parse $\mathsf{cw}_i, (s_0^{(i)}, t_0^{(i)}), (s_1^{(i)}, t_1^{(i)}) \leftarrow \mathsf{vals}$
 6: $\tilde{\pi}_b \leftarrow \mathsf{H}(\alpha || s_b^{(n)})$ for $b \in \{0,1\}$.
 7: $\mathsf{cs} \leftarrow \tilde{\pi}_0 \oplus \tilde{\pi}_1$.
 8: $s_b^{(n+1)} \leftarrow s_b^{(n)}$                    ▷ True output always extends to left child.
 9: $t_b^{(n+1)} \leftarrow \mathsf{LSB}(s_b^{(n+1)})$
10: if $t_0^{(n+1)} = t_1^{(n+1)}$ then goto 1
11: Compute output correction word: $\mathsf{ocw} \leftarrow (-1)^{t_1^{(n+1)}}[\beta - \mathsf{convert}(s_0^{(n+1)}) + \mathsf{convert}(s_1^{(n+1)})]$
12: Set $\mathsf{k}_b \leftarrow (s_b^{(0)}, \{\mathsf{cw}_i\}_{i=1}^n, \mathsf{cs}, \mathsf{ocw})$ for $b \in \{0,1\}$
Output: $(\mathsf{k}_0, \mathsf{k}_1)$
```

$\mathsf{VerDPF.BVEval}$

```
Input: $b \in \{0,1\}$ and VerDPF key $\mathsf{k}_b$.
       Set of $L$ distinct evaluation points $x_1, \ldots, x_L$.
 1: Parse the VerDPF key $(s^{(0)}, \{\mathsf{cw}_i\}_{i=1}^n, \mathsf{cs}, \mathsf{ocw}) \leftarrow \mathsf{k}_b$.
 2: Define $\mathsf{y} \leftarrow \{\}$ and $\pi \leftarrow \mathsf{cs}$
 3: for $\ell$ from 1 to $L$ do
 4:     Let $s \leftarrow s^{(0)}$ and $t \leftarrow b$.
 5:     Let $\beta_1 = \mathsf{MSB}(x_\ell), \ldots, \beta_n = \mathsf{LSB}(x_\ell)$ be the bits of $x_\ell$.
 6:     for $i$ from 1 to $n$ do
 7:         $(s_0', t_0'), (s_1', t_1') \leftarrow \mathsf{NodeExpand}(\mathcal{G}, s, t)$
 8:         if $\beta_i = 0$ then $(s, t) \leftarrow (s_0', t_0')$
 9:         else $(s, t) \leftarrow (s_1', t_1')$
10:     $\tilde{\pi} \leftarrow \mathsf{H}(x_\ell || s)$ and $t \leftarrow \mathsf{LSB}(s)$
11:     $\mathsf{y}.\mathsf{append}\left((-1)^b \cdot \mathsf{correct}_{\mathbb{G}}(\mathsf{convert}(s), \mathsf{ocw}, t)\right)$
12:     $\pi \leftarrow \pi \oplus \mathsf{H}'(\pi \oplus \mathsf{correct}(\tilde{\pi}, \mathsf{cs}, t))$
Output: $(\mathsf{y}, \pi)$
```

Fig. 1. Verifiable Distributed Point Function $\mathsf{VerDPF}_{n,\mathbb{G}}$.

Algorithm 3 $\mathsf{VerDPF.FDEval}$. The verifiable full-domain evaluation function for our verifiable DPF construction. The hash functions H and H' are as in Figure 1.

```
Input: $b \in \{0,1\}$ and VerDPF key $\mathsf{k}_b$.
Parse the VerDPF key $(s^{(0)}, \{\mathsf{cw}_i\}_{i=1}^{n}, \mathsf{cs}, \mathsf{ocw}) \leftarrow \mathsf{k}_b$.
Let $s \leftarrow s^{(0)}$ and $t \leftarrow b$
Define $\mathsf{nodes} \leftarrow \{(s,t)\}$
for $i$ from 1 to $n$ do
    Define $\mathsf{nodes}' \leftarrow \{\}$
    for $(s,t)$ in $\mathsf{nodes}$ do
        $(s'_0, t'_0), (s'_1, t'_1) \leftarrow \mathsf{NodeExpand}(\mathcal{G}, s, t)$
        $\mathsf{nodes}'.\mathsf{append}((s'_0, t'_0))$
        $\mathsf{nodes}'.\mathsf{append}((s'_1, t'_1))$
    $\mathsf{nodes} \leftarrow \mathsf{nodes}'$
Define $\mathsf{y} \leftarrow \{\}$ and $\pi \leftarrow \mathsf{cs}$
for $i$ from 1 to $N$ do
    $(s, \_) \leftarrow \mathsf{nodes}[i]$.
    $\tilde{\pi} \leftarrow \mathsf{H}(i || s)$
    $t \leftarrow \mathsf{LSB}(s)$
    $\mathsf{y}.\mathsf{append}\left((-1)^b \cdot \mathsf{correct}_{\mathbb{G}}(\mathsf{convert}(s), \mathsf{ocw}, t)\right)$
    $\pi \leftarrow \pi \oplus \mathsf{H}'(\pi \oplus \mathsf{correct}(\tilde{\pi}, \mathsf{cs}, t))$
Output: $(\mathsf{y}, \pi)$
```

with matching seeds and control bits will produce shares of zero. This can be seen below, where we set $s_0 = s_1$ and $t_0 = t_1$.

$$\mathsf{y}_b = (-1)^b \cdot \mathsf{correct}_{\mathbb{G}}(\mathsf{convert}(s_b), \mathsf{ocw}, t_b) = -1 \cdot \mathsf{y}_{1-b}$$

For the leaf at position α, we have that $t_0 \neq t_1$. Here, the output values will be a secret sharing of β. For simplicity, we write $g_b = \mathsf{convert}(s_b)$.

$$\begin{aligned} \mathsf{ocw} &= (-1)^{t_1} \left[\beta - g_0 + g_1\right] \\ \mathsf{y}_0 + \mathsf{y}_1 &= \mathsf{correct}_{\mathbb{G}}(g_0, \mathsf{ocw}, t_0) + \mathsf{correct}_{\mathbb{G}}(g_1, \mathsf{ocw}, t_1) \\ &= g_0 - g_1 + (-1)^{t_0} \cdot \mathsf{ocw} = g_0 - g_1 + \beta - g_0 + g_1 = \beta \end{aligned}$$

where we get that $(-1)^{t_0} \cdot \mathsf{ocw} = \beta - g_0 + g_1$ from $t_0 \neq t_1$.

3.3 VDPF Security Proof

We will now prove that the verifiable DPF construction given in Figure 1 is secure. We will focus on proving the following theorem.

Lemma 2 (Detection of Malicious Function Shares). *Except with probability negligible in the security parameter λ, no PPT adversary $\mathcal{A}$ can generate* VerDPF *keys $(\mathsf{k}_0^*, \mathsf{k}_1^*) \leftarrow \mathcal{A}(1^\lambda)$ where the final level uses a hash function $\mathsf{H} \leftarrow \mathcal{H}$ sampled from a family $\mathcal{H}$ of collision-resistant and XOR-collision-resistant hash functions such that the following holds. For an adversarially chosen set of evaluation points $\{x_i\}_{i=1}^{L}$, let $(\mathsf{y}_b, \pi_b) \leftarrow \mathsf{VerDPF.BVEval}(b, \mathsf{k}_b^*, \{x_i\}_{i=1}^{L})$ such that*

$\mathsf{Accept} \leftarrow \mathsf{VerDPF.Verify}(\pi_0, \pi_1)$ *passes but* $\mathsf{y}_0 + \mathsf{y}_1$ *is nonzero in more than one location.*

Proof. The approach to proving this theorem will be to focus on the final level of the GGM tree. At the second-to-last level, each server has a set of seeds $\{s_0^{(x_i)}\}_{i=1}^L$ and $\{s_1^{(x_i)}\}_{i=1}^L$. The servers also have the *same* correction seed cs. Let $\tilde{\pi}_b^{(x)} \leftarrow \mathsf{H}(x||s_b^{(x)})$, let $t_b^{(x)} \leftarrow \mathsf{LSB}(s_b^{(x)})$, and let $\pi_b^{(x)} \leftarrow \mathsf{correct}(\tilde{\pi}_b^{(x)}, \mathsf{cs}, t_b^{(x)})$. The bulk of this proof is covered by the following lemma.

Lemma 3. *For $L \geq 2$, let $\mathbf{x} := \{x_i\}_{i=1}^L$. Suppose there exists two distinct inputs $u, v \in \mathbf{x}$ such that $s_0^{(u)} \neq s_1^{(u)}$ and $s_0^{(v)} \neq s_1^{(v)}$. If H is sampled from a collision-resistant and XOR-collision-resistant family, then no PPT adversary can find a correction seed cs such that for all $x \in \mathbf{x}$ we will have $\pi_0^{(x)} = \pi_1^{(x)}$.*

Proof. Suppose for contradiction that there exists two inputs $u, v \in \mathbf{x}$ such that $s_0^{(u)} \neq s_1^{(u)}$ and $s_0^{(v)} \neq s_1^{(v)}$ and for all $x \in \mathbf{x}$ we have $\pi_0^{(x)} = \pi_1^{(x)}$. By collision-resistance, we have that $\tilde{\pi}_0^{(u)} \neq \tilde{\pi}_1^{(u)}$ and $\tilde{\pi}_0^{(v)} \neq \tilde{\pi}_1^{(v)}$. In order to get $\pi_0^{(u)} = \pi_1^{(u)}$ and $\pi_0^{(v)} = \pi_1^{(v)}$, we need the following:

$$\mathsf{cs} = \tilde{\pi}_0^{(u)} \oplus \tilde{\pi}_1^{(u)} = \tilde{\pi}_0^{(v)} \oplus \tilde{\pi}_1^{(v)} \neq 0$$

From the XOR-collision-resistance of $\mathcal{H}$, in order to get this equality we must have one of the following two cases.

- Case (i): $\tilde{\pi}_0^{(u)} = \tilde{\pi}_0^{(v)}$ and $\tilde{\pi}_1^{(u)} = \tilde{\pi}_1^{(v)}$
- Case (ii): $\tilde{\pi}_0^{(u)} = \tilde{\pi}_1^{(v)}$ and $\tilde{\pi}_1^{(u)} = \tilde{\pi}_0^{(v)}$

We can show that any one of these four equalities violates the collision-resistance of H. Suppose we have $\mathsf{H}(u||s_b^{(u)}) = \tilde{\pi}_b^{(u)} = \tilde{\pi}_{b'}^{(v)} = \mathsf{H}(v||s_{b'}^{(v)})$ for any $b, b' \in \{0, 1\}$. Since $u \neq v$, any equality between these hash outputs violates the collision resistance of $\mathcal{H}$.

Therefore, no value of cs will result in $\pi_0^{(x)} = \pi_1^{(x)}$ for all $x \in \mathbf{x}$.

From the collision resistance of H', if the proofs produced by the BVEval algorithm match, then $\pi_0^{(x)} = \pi_1^{(x)}$ for all $x \in \mathbf{x}$. From Lemma 3, this implies that there is at most one $u \in \mathbf{x}$ such that $s_0^{(u)} \neq s_1^{(u)}$, and for all $x \in \mathbf{x}$ such that $x \neq u$, we have $s_0^{(x)} = s_1^{(x)}$.

Define $\alpha = u$ for the unique u such that $s_0^{(u)} \neq s_1^{(u)}$. If no such u exists (which occurs if all outputs are zero), set $u = x_1$. Define

$$\beta = \mathsf{correct}_{\mathbb{G}}\left(\mathsf{convert}(s_0^{(u)}), \mathsf{ocw}, t_0^{(u)}\right) - \mathsf{correct}_{\mathbb{G}}\left(\mathsf{convert}(s_1^{(u)}), \mathsf{ocw}, t_1^{(u)}\right)$$

Note that this β is well-defined for any $s_b^{(u)}$ and $t_b^{(u)}$. For all other $x \neq u$, observing that $t_b^{(x)} = \mathsf{LSB}(s_b^{(x)})$ implies the following:

$$\begin{aligned} s_0^{(x)} = s_1^{(x)} \implies t_0^{(x)} = t_1^{(x)} \implies &\mathsf{correct}_{\mathbb{G}}\left(\mathsf{convert}(s_0^{(x)}), \mathsf{ocw}, t_0^{(x)}\right) \\ &= \mathsf{correct}_{\mathbb{G}}\left(\mathsf{convert}(s_1^{(x)}), \mathsf{ocw}, t_1^{(x)}\right) \\ \implies \mathsf{y}_0^{(x)} + \mathsf{y}_1^{(x)} = 0 \end{aligned}$$

Since $s_0^{(x)} = s_1^{(x)}$ for all $x \neq u$, $\mathsf{y}_0 + \mathsf{y}_1$ defines the truth table of $f_{\alpha,\beta}$. Therefore, the construction in Figure 1 satisfies Definition 6.

Lemma 4 (VerDPF **Function Privacy**). *The VDPF construction* VerDPF *satisfies Definition 7.*

Proof. All elements of a VerDPF key are computationally indistinguishable from random elements. The starting seed is randomly sampled from $\{0,1\}^\lambda$. Each correction word is XOR'd with the output of a PRG where the seed is not known to the evaluator, and hence is also indistinguishable from random. Finally, the inclusion of the correct proof from the other party does not add any information, since the evaluator holding the share k_b can locally compute the correct proof $\pi_{1-b} = \pi_b$. Therefore, the simulator Sim can set all elements of the key k^* to be randomly sampled elements, then compute $(\_, \pi^*) \leftarrow \mathsf{VerDPF.BVEval}(b, \mathsf{k}^*, \mathbf{x})$ to output $(\mathsf{k}^*, \pi^*) \approx_c (\mathsf{k}_b, \pi_{1-b})$.

Combining Lemma 2 and Lemma 4 gives the proof of the following theorem.

Theorem 3 (Verifiable Distributed Point Function). *The construction in Figure 1 is a secure verifiable DPF for the class of point functions $\mathcal{F}_{n,\mathbb{G}}$. For any $f \in \mathcal{F}_{n,\mathbb{G}}$, the runtime of $(\mathsf{k}_0, \mathsf{k}_1) \leftarrow \mathsf{VerDPF.Gen}(1^\lambda, f)$ is $O(n\lambda)$, and the size of a function share is $O(n\lambda)$. For any $x \in \{0,1\}^n$, the runtime of $\mathsf{VerDPF.Eval}(b, \mathsf{k}_b, x)$ is $O(n\lambda)$, and the runtime of $\mathsf{VerDPF.BVEval}(b, \mathsf{k}_b, \mathbf{x})$ is $O(n \cdot \lambda \cdot |\mathbf{x}|)$.*

4 Verifiable Distributed Multi-Point Function

In this section, we present a novel method for efficiently batching many verifiable DPF queries to obtain a verifiable FSS scheme for multi-point functions (MPFs). Multi-point functions are defined as the sum of several point functions. While any function can be viewed as an MPF, we will focus here on MPFs that have a small number of non-zero points relatively to the domain size. This scheme will also be verifiable in a similar, although more relaxed, manner as in the verifiable DPF from Section 3. Our construction is based on a novel Cuckoo-hashing scheme described below.

4.1 Cuckoo-hashing from PRPs

Our technique is inspired by the use of Cuckoo-hashing schemes that are common throughout the PSI [5, 7] and DPF [13] literature. In particular, it is common for the Cuckoo-hashing scheme to have two modes: a *compact* mode and an *expanded* mode. Both modes are parameterized by m buckets and κ hash functions $h_1, \ldots, h_\kappa \colon \{0,1\}^* \to [m]$.

Compact Cuckoo-hashing mode. In the compact mode, the input is t elements $x_1, \ldots, x_t$ to be inserted into a table of m buckets. To insert an element x_i, an index $k \in [\kappa]$ is randomly sampled and x_i is inserted at index $h_k(x_i)$. If this index is already occupied by some other element x_j, then x_j is replaced by x_i and x_j is reinserted using this same method. After some limit on the number of trials, the insertion process is deemed to have failed. The purpose of the compact mode is to efficiently pack t elements into the table of size m. This algorithm, denoted CHCompact, is given in Algorithm 4.

Algorithm 4 CHCompact Compact Cuckoo-hashing scheme. The algorithm is given a fixed time to run before it is deemed to have failed.

```
Input: Domain elements $\alpha_1, \ldots, \alpha_t$
       Hash functions $h_1, \ldots, h_\kappa \colon \{0,1\}^\star \to m$
       Number of buckets $m \geq t$
Define an empty array of $m$ elements Table where each entry is initialized to $\perp$.
for $\omega$ from 1 to $t$ do
    Set $\beta \leftarrow \alpha_\omega$ and set success $\leftarrow$ False
    while success is False do
        Sample $k \xleftarrow{\$} [\kappa]$
        $i \leftarrow h_k(\beta)$.
        if Table$[i] = \perp$ then
            Table$[i] = \beta$ and success $\leftarrow$ True
        else Swap $\beta$ and Table$[i]$
Output: Table
```

We consider $m = e \cdot t$ for $e > 1$, where the size of e determines the probability over the choice of hash functions of failing to insert any set of t elements. More specifically, from the empirical analysis of Demmler et al. [7], we have the following lemma.

Lemma 5 (Cuckoo-hashing Failure Probability [7]). *Let $\kappa = 3$ and $t \geq 4$. Let $m = e \cdot t$ for $e > 1$. Let $\mathcal{H}$ be a family of collision-resistant hash functions, and let $h_1, \ldots, h_\kappa \leftarrow \mathcal{H}$ be randomly sampled from $\mathcal{H}$. We have that t elements will fail to be inserted into a table of size m with probability $2^{-\lambda}$, where*

$$\begin{aligned}\lambda &= a_t \cdot e - b_t - \log_2(t)\\ a_t &= 123.5 \cdot \mathsf{CDF}_{\mathsf{Normal}}(x = t, \mu = 6.3, \sigma = 2.3)\\ b_t &= 130 \cdot \mathsf{CDF}_{\mathsf{Normal}}(x = t, \mu = 6.45, \sigma = 2.18)\end{aligned}$$

Here, $\mathsf{CDF}_{\mathsf{Normal}}(x, \mu, \sigma)$ *refers to the cumulative density function of the normal distribution with mean* μ *and standard deviation* σ *up to the point* x*.*

Remark 1 (Cuckoo-hash parameters). Asymptotically, we have the number of Cuckoo-hash buckets as $m = O(t\lambda + t\log(t))$; however, concretely, the picture is much nicer than the asymptotics suggest. For sufficiently large t (i.e. $t \geq 30$), we can simplify Lemma 5 to be $\lambda = 123.5 \cdot e - 130 - \log_2(t)$, since the $\mathsf{CDF}_{\mathsf{Normal}}$ factors become effectively one. Then, for $\lambda = 80$, we have that $m \leq 2t$ for all $30 \leq t \leq 2^{37}$, which we believe captures nearly all practical use cases.

Expanded Cuckoo-hashing mode. In the expanded Cuckoo-hashing mode, the hashing scheme takes as input t elements and produces a matrix of dimension $m \times B$ that contains $\kappa \cdot t$ elements. This mode is produced by hashing all t elements with each of the κ hash functions, then inserting each of the t elements in all κ buckets as indicated by the hash functions. The parameter B is the maximum size of these buckets.

Our PRP Cuckoo-hashing. In the Cuckoo-hashing schemes from the prior literature, the design of the scheme is focused on the compact mode, and the extended mode is added without much change to the overall design. In our Cuckoo-hashing scheme, we begin with an efficient construction of the expanded mode, then show how we maintain efficiency of the compact mode. For a domain of elements $\mathcal{D}$ of size $n = |\mathcal{D}|$, we define the expanded mode of our Cuckoo-hashing scheme with a PRP of domain size $n\kappa$. Let m be the number of bins in the Cuckoo-hash table. Define $B := \lceil n\kappa/m \rceil$. The PRP then defines an expanded Cuckoo-hash table of dimension $m \times B$ by simply arranging the $n\kappa$ outputs of the PRP into the entries of an $m \times B$ matrix. More specifically, let $\mathsf{PRP}\colon \{0,1\}^\lambda \times [n\kappa] \to [n\kappa]$ be the PRP. Let $\sigma \leftarrow \{0,1\}^\lambda$ be the seed of the PRP. Define entry (i, j) of the $m \times B$ matrix A to be $A_{i,j} := \mathsf{PRP}(\sigma, (i-1) \cdot m + j)$. Note that the last row of the matrix may have some empty entries, but this turns out to have little consequence on the overall scheme.

To define the compact mode of this Cuckoo-hashing scheme, we explicitly define the hash functions in terms of the PRP. As above, let $\mathsf{PRP}\colon \{0,1\}^\lambda \times [n\kappa] \to [n\kappa]$ be the PRP, and let $\sigma \leftarrow \{0,1\}^\lambda$ be the seed of the PRP. For $i \in [\kappa]$, define the hash function $h_i\colon [n] \to [m]$ as follows:

$$h_i(x) := \lfloor \mathsf{PRP}(\sigma, x + n \cdot (i-1))/B \rfloor \tag{1}$$

The hash functions $h_1, \ldots, h_\kappa$ can then be used in the original compact Cuckoo-hashing scheme with m buckets. The main benefit of our construction comes with the next feature, which allows a party to learn the location of an element within a specific bucket of the expanded Cuckoo-hash table without directly constructing the expanded table. More specifically, for $i \in [\kappa]$, we define the function $\mathsf{index}_i\colon [n] \to [B]$ as follows:

$$\mathsf{index}_i(x) := \mathsf{PRP}(\sigma, x + n \cdot (i-1)) \mod B \tag{2}$$

With these functions $\mathsf{index}_1, \dots, \mathsf{index}_\kappa$ in addition to the hash functions h_1, $\dots$, h_κ, we can compute the locations $\{(i,j)_k \in [m] \times [B]\}_{k=1}^{\kappa}$ for each of the κ locations of an element $x \in [n]$ in the expanded Cuckoo-hash table. In particular, we have $(i,j)_k = (h_k(x), \mathsf{index}_k(x))$.

4.2 Verifiable Distributed MPFs via PRP Hashing

We now present our verifiable MPF scheme that makes use of the Cuckoo-hashing scheme described in the previous section. Let N be the MPF domain size. Our input will be an MPF f defined by t point functions $f_{\alpha_i,\beta_i} \colon [N] \to \mathbb{G}$ for $i \in [t]$. Without loss of generality, we consider $\alpha_1, \dots, \alpha_t$ as distinct points. We would like to efficiently support an FSS scheme for the function $f \colon [N] \to \mathbb{G}$ that is defined as follows:

$$f(x) = \sum_{i=1}^{t} f_{\alpha_i,\beta_i}(x)$$

Naively, we would generate t different DPF shares, one for each point function. Evaluation of this naive distributed MPF (DMPF) share at a single point would require t DPF share evaluations.

To improve over this naive construction, the idea is to pack our point functions into a Cuckoo-hash table with m buckets. We begin by instantiating our PRP-based Cuckoo-hashing scheme with a PRP of domain size $N\kappa$ and define $B = \lceil N\kappa/m \rceil$. The client can then use the compact mode to pack the values $\alpha_1, \dots, \alpha_t$ into a Cuckoo-hash table of size m. For each bucket at index $i \in [m]$, let α'_i be the value in the bucket. We can either have $\alpha'_i = \alpha_j$ for one of the input α_j, or $\alpha'_i = \perp$ if the bucket is empty. If $\alpha'_i = \alpha_j$, let $k \in [\kappa]$ be the index of the hash function used to insert α_j to bucket i. In other words, $h_k(\alpha_j) = i$. Define the index $\gamma_i = \mathsf{index}_k(\alpha_j)$, which is the index of α_j in the i^{th} bucket in the expanded Cuckoo-hash mode. Next, define the point function $g_{\gamma_i,\beta_j} \colon [B] \to \mathbb{G}$, which evaluates to β_j at the index of α_j within the i^{th} bucket. This point function is then shared to create $(\mathsf{k}_0^{(i)}, \mathsf{k}_1^{(i)}) \leftarrow \mathsf{VerDPF.Gen}(1^\lambda, g_{\gamma_i,\beta_j})$. In the case where $\alpha'_i = \perp$, the shared function is set to be the zero function. The verifiable distributed MPF (VDMPF) share has the form $\mathsf{mpk}_b = (\sigma, \mathsf{k}_b^{(1)}, \dots, \mathsf{k}_b^{(m)})$ where σ is the PRP seed.

To evaluate this multi-point function share at a point $x \in [N]$, the evaluator first computes the κ possible buckets in which x could lie, denoted $i_k = h_k(x)$ for $k \in [\kappa]$. Next, the evaluator computes the index of x in each bucket, denoted $j_k = \mathsf{index}_k(x)$ for $k \in [\kappa]$. Finally, the evaluator computes the sum of the VDPF in each of the buckets at $i_1, \dots, i_k$ evaluated at $j_1, \dots, j_k$. This gives the output

$$\mathsf{y}_b = \mathsf{VerDMPF.Eval}(b, \mathsf{mpk}_b, x) = \sum_{k \in [\kappa]} \mathsf{VerDPF.Eval}(b, \mathsf{k}_b^{(i_k)}, j_k)$$

In addition, this VDMPF inherits all of the features of the VDPF construction from Section 3, including the $O(\log(B))$ savings when evaluating the full domain (via tree traversal), as well as verifiability of share well-formedness. We note that the verifiability is a bit weaker than the definition achieved for point functions. More specifically, for point functions we showed how the servers can ensure that at most one evaluation point is nonzero when evaluating any subset of the domain. For this VDMPF construction, we can show that there are no more than m non-zero points in any subset of evaluations by showing there is no more than one non-zero point in the VDPF in each bucket. This is slightly weaker than the best-possible guarantee, which would be that there are no more than t non-zero points in any set of evaluations. However, as discussed in Section 4.1, we will essentially always have $m \leq 2t$ (see Remark 1), so we consider this gap acceptable for most applications. In addition, we can achieve an exact guarantee by reverting to the naive construction using the VDPFs from Section 3. We leave for future work the challenge of closing this gap while maintaining similar performance.

Our VDMPF construction is given in Figure 2.

Lemma 6 (VerDMPF **Correctness**). *Let $\mathcal{F}$ be the function class of multi-point functions with at most t non-zero points. Figure 2 gives a correct function secret sharing scheme for $\mathcal{F}$.*

Proof. This follows directly from the correctness of the VerDPF shares in each bucket and the low statistical failure probability of the Cuckoo-hashing scheme.

Lemma 7 (VerDMPF **Function Privacy**). *Let $\mathcal{F}$ be the function class of multi-point functions with at most t non-zero points. Figure 2 gives a function-private FSS scheme for $\mathcal{F}$, as defined in Definition 2*

Proof. The VerDPF shares in this construction computationally hide all information regarding the non-zero evaluation points. The only additional leakage is that these t evaluation points fit into a Cuckoo-hash table with the hash functions specified by the PRP seed σ. Lemma 5 gives us a way to set the number of buckets so that any t inputs will fail to hash with $2^{-\lambda}$ probability. Setting λ to be the computational security parameter maintains the adversary's negligible distinguishing advantage.

Lemma 8 (VerDMPF **Share Integrity**). *Let* VerDPF *be a secure verifiable point function scheme. Let* $\mathsf{VerDMPF} := \mathsf{VerDMPF}_{N,\mathbb{G}}$ *be a verifiable multi-point function scheme as defined in Figure 2 that uses* VerDPF *for the Cuckoo-hash buckets. No PPT adversary $\mathcal{A}$ can generate* VerDMPF *keys* $(\mathsf{k}_0^*, \mathsf{k}_1^*) \leftarrow \mathcal{A}(1^\lambda)$ *along with $L \geq 1$ distinct evaluation points $x_1, \ldots, x_L \in [N]$ such that the following holds. Let* $(\mathsf{y}_b, \pi_b) \leftarrow \mathsf{VerDMPF.BVEval}(b, \mathsf{k}_b^*, \{x_i\}_{i=1}^L)$ *such that* $\mathsf{Accept} \leftarrow \mathsf{VerDMPF.Verify}(\pi_0, \pi_1)$ *but there are $\omega > m$ indices $i_1, \ldots, i_\omega$ such that* $\mathsf{y}_0^{(i_j)} + \mathsf{y}_1^{(i_j)} \neq 0$ *for $j \in [\omega]$. In other words, the output of the batched evaluation contains more than m non-zero outputs.*

Proof. This follows directly from the verifiability of the VerDPF shares, which guarantees that there is at most one non-zero evaluation for each of the m buckets.

Lemma 6, Lemma 7, and Lemma 8 combine to give the following theorem.

Theorem 4. *The construction in Figure 2 is a secure verifiable DMPF for the class $\mathcal{F}$ of multi-point functions $f\colon [N] \to \mathbb{G}$ with at most t non-zero evaluation points. For any $f \in \mathcal{F}$ and $m = O(t\lambda + t\log(t))$, the runtime of* $\mathsf{VerDMPF.Gen}$ *is $O\left(m\lambda \log(N/m)\right)$. For η inputs, the runtime of* $\mathsf{VerDMPF.BVEval}$ *is $O(\eta\lambda \log(N/m))$.*

Proof. The asymptotics follow from the fact that generating a single VerDPF share in this scheme takes time $O(\lambda \log(N/m))$, and evaluation of a VerDPF share at one point is also $O(\lambda \log(N/m))$, where we take the PRP and PRG evaluations to be $O(\lambda)$.

Remark 2. We note briefly that if a PRP for the domain κN is not available, our method will work just as well utilizing a generic Cuckoo-hashing scheme and setting all $\mathsf{index}_j(i) = i$. The difference will be that the domain size of the DPF in each Cuckoo-hash bucket will not shrink as the number of nonzero points grows, resulting in a $\mathsf{VerDMPF.Gen}$ time of $O\left(m\lambda \log(N)\right)$ and a $\mathsf{VerDMPF.BVEval}$ time of $O(\eta\lambda \log(N))$.

In Appendix B, we give an alternate evaluation mode of our VDMPF, which we call “match-mode” evaluation. This mode has identical performance to the regular batch verifiable evaluation mode with the same verification guarantee. The difference is that for each of the m buckets, match-mode evaluation computes additive shares of whether or not any of the inputs matched with the nonzero point in that bucket. This mode is useful in two-server PSI protocols, among others.

5 Implementation & Performance

In this section, we present an implementation of our verifiable DPF and verifiable MPF constructions and compare them to their non-verifiable and non-batched counterparts.

Implementation Details. We implemented our VDPF and VDMPF constructions in C++. We follow the approach of Wang et al. [15] by using a fixed-key AES cipher to construct a Matyas-Meyer-Oseas [12] one-way compression function. We use AES-based PRFs to construct our PRGs, our hash functions, and our PRP. Using an AES-based PRP implicitly fixes our DMPF domain size to be 128 bits, and we leave for future work the task of implementing an efficient small-domain PRP. Our implementation is accelerated with the Intel AES-NI instruction, and all benchmarks were run on a single thread on an Intel i7-8650U CPU. For comparison, we also implemented a non-verifiable DPF following the constructions of Boyle et al. [3] and Wang et al. [15], which we refer to as the “textbook” DPF. We implement the “textbook” distributed MPF by naively applying the textbook DPF; namely, our textbook DMPF share contains one DPF share per non-zero point, and evaluating the share requires evaluation all DPF shares and summing their results.

Verifiable Distributed Multi-Point Function $\mathsf{VerDMPF}_{N,\mathbb{G}}$.
For a domain $\mathcal{D}$ of size N and output group $\mathbb{G}$, let $\mathsf{VerDMPF} := \mathsf{VerDMPF}_{N,\mathbb{G}}$. Let $\mathsf{domain}\colon \mathcal{D} \to [N]$ be an injective function mapping domain elements to indices in $[N]$. Unless otherwise specified, we will consider a domain element as its index. For $\kappa = 3$, let $\mathsf{PRP}\colon \{0,1\}^\lambda \times [N\kappa] \to [N\kappa]$ be a pseudorandom permutation. Let $\mathsf{CHBucket}(t, \kappa, \lambda) \to \mathbb{N}$ be the function that outputs the number of cuckoo hash buckets required so that inserting t elements with κ hash functions fails with probability at most $2^{-\lambda}$. The hash function H' is as in Figure 1.
The $\mathsf{VerDMPF.Verify}$ algorithm simply checks that the two input proofs are equal.

$\mathsf{VerDMPF.Gen}$

```
Input: Security parameter $1^\lambda$ and $t$ point functions $\{f_{\alpha_i,\beta_i}\}_{i=1}^t$
1: $m \leftarrow \mathsf{CHBucket}(t, 3, \lambda)$, where $\kappa = 3$.
2: Sample a random PRP seed $\sigma \leftarrow \{0,1\}^\lambda$ and let $B \leftarrow \lceil N\kappa/m \rceil$.
3: From $\sigma, m, B$, define $h_1, \ldots, h_\kappa\colon [N\kappa] \to [m]$ as in Equation (1) and
   $\mathsf{index}_1, \ldots, \mathsf{index}_\kappa\colon [N\kappa] \to B$ in Equation (2).
4: $\mathsf{Table} \leftarrow \mathsf{CHCompact}\left(\{\alpha_i\}_{i=1}^t, \{h_k\}_{k=1}^\kappa, m\right)$. If this algorithm fails, return to
   step 2 to sample a fresh PRP seed.
5: Let $n' = \lceil \log(B) \rceil$ and let $\mathsf{VerDPF} := \mathsf{VerDPF}_{n',\mathbb{G}}$. Let $\mathsf{k}_0 \leftarrow \{\sigma\}$ and $\mathsf{k}_1 \leftarrow \{\sigma\}$.
6: for $i$ from 1 to $m$ do
7:     if $\mathsf{Table}[i] = \perp$ then Define $\alpha' \leftarrow 0$ and $\beta' \leftarrow 0$
8:     else
9:         Let $\alpha_j = \mathsf{Table}[i]$, for $j \in [t]$, and let $k \in [\kappa]$ be such that $h_k(\alpha_j) = i$.
10:        Let $\alpha' \leftarrow \mathsf{index}_k(\alpha_j)$ and $\beta' \leftarrow \beta_j$
11:    Sample $(\mathsf{k}_0^{(i)}, \mathsf{k}_1^{(i)}) \leftarrow \mathsf{VerDPF.Gen}(1^\lambda, f_{\alpha',\beta'})$
12:    Append $\mathsf{k}_0^{(i)}$ to $\mathsf{k}_0$ and $\mathsf{k}_1^{(i)}$ to $\mathsf{k}_1$.
Output: $(\mathsf{k}_0, \mathsf{k}_1)$
```

$\mathsf{VerDMPF.BVEval}$

```
Input: Bit $b$ and $\mathsf{VerDMPF}$ key $\mathsf{k}_b$ and $\eta$ inputs $x_1, \ldots, x_\eta \in \mathcal{D}$.
Parse $\sigma, \mathsf{k}_b^{(1)}, \ldots, \mathsf{k}_b^{(m)} \leftarrow \mathsf{k}_b$ and define $B \leftarrow \lceil N\kappa/m \rceil$, $n' \leftarrow \lceil \log(B) \rceil$.
Let $\mathsf{VerDPF} := \mathsf{VerDPF}_{n',\mathbb{G}}$ and initialize an array $\mathsf{inputs}$ of length $m$.
for $\omega$ from 1 to $\eta$ do
    Let $i_1, \ldots, i_\kappa \leftarrow h_1(x_\omega), \ldots, h_\kappa(x_\omega)$
    Let $j_1, \ldots, j_\kappa \leftarrow \mathsf{index}_1(x_\omega), \ldots, \mathsf{index}_\kappa(x_\omega)$
    Append $(j_k, \omega)$ to $\mathsf{inputs}[i_k]$ for each $k \in [\kappa]$, ignoring duplicates.
Initialize an array $\mathsf{outputs}$ of length $\eta$ to all zeros and Initialize a proof $\pi \leftarrow 0$.
for $i$ from 1 to $m$ do
    Parse $(j_1, \omega_1), \ldots, (j_L, \omega_L) \leftarrow \mathsf{inputs}[i]$
    $\{\mathsf{y}_\ell\}_{\ell=1}^L, \pi^{(i)} \leftarrow \mathsf{VerDPF.BVEval}(b, \mathsf{k}_b^{(i)}, \{j_\ell\}_{\ell=1}^L)$
    $\mathsf{outputs}[\omega_\ell] \leftarrow \mathsf{outputs}[\omega_\ell] + \mathsf{y}_\ell$ for $\ell \in [L]$
    $\pi \leftarrow \pi \oplus \mathsf{H}'(\pi \oplus \pi^{(i)})$
Output: $\mathsf{outputs}, \pi$
```

Fig. 2. Verifiable Distributed Multi-Point Function.

DPF Comparisons. We now present the results of our DPF comparisons. For various domain sizes 2^n, we benchmarked the share generation time, the evaluation time, and the full-domain evaluation time for the textbook DPF and the verifiable DPF. All benchmarks of the verifiable DPF include the generation of the verification proof. The share evaluation comparison runs the verifiable DPF at 100 random points in $\{0,1\}^n$ and generates the proof verifying this set of evaluations. The runtime reported is the time per evaluation point. Benchmarks are given in Figure 4.

The slowdown for the verifiable evaluation time is quite small, as it essentially only requires evaluating one additional level of the GGM tree. The slowdown for the share generation time is a bit greater, since the verifiable share generation has a 50% chance of failure, at which point it must be restarted. This can be seen by the roughly factor of 2 slowdown in the runtime of the verifiable share generation.

Overall, our comparisons show that our techniques introduce relatively little overhead to the textbook DPF procedures. We view these results as an affirmation of our claim that our verifiable DPF can replace the textbook DPF in any application to provide a meaningful & robust malicious security claim without seriously impacting performance. Our results are displayed in Figure 3.

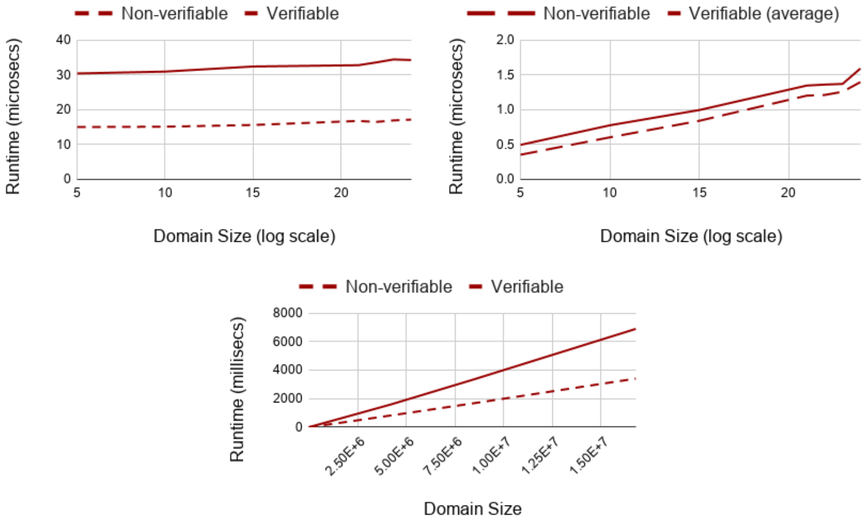

Fig. 3. In this figure, we present the benchmarks of the textbook DPF and the verifiable DPF presented in this work. The top-left graph plots runtimes for the share generation time. As can be seen, the slowdown for verifiability is roughly 2×. The top-right graph plots the runtimes for the share evaluation. As discussed in Section 5, the verifiable runtime was computed by taking the runtime of the verifiable batch evaluation procedure (Figure 1) for 100 random points and dividing it by 100. The bottom graph plots the runtimes for the full domain evaluation operation.

DMPF Comparisons. We now present the results of our DMPF comparisons. We benchmarked the share generation and evaluation time for MPFs with various numbers of nonzero points t. As with the DPF comparisons, all benchmarks of the VDMPFs include the time required to generate the verification proof. Recall that our "textbook" benchmark uses neither the batching nor the verification techniques presented in this work. The batched, verifiable share generation time is about 2× slower than the textbook share generation time. This is a balancing between the increased runtime due to the overhead of the verifiable share generation, the overhead due to the number of buckets being greater than the nonzero values, and the savings due to the domain size shrinking thanks to the PRP savings. To display benchmarks that demonstrate this optimization, we chose a domain size of $N = 2^{126}$. This is so that the $\kappa \cdot N$ elements of the permutation fit in the 128-bit domain of AES PRP. These benchmarks are given in Figure 4.

The real savings, and what we view as one of the main results of this section, comes in the share evaluation. As discussed in Section 4, the performance of the batched VDMPF evaluation effectively does not grow with the number of nonzero points t in the shared multi-point function. This is in stark contrast to the textbook version, where evaluation time grows linearly with the number of nonzero points t in the shared multi-point function. This leads to a dramatic difference in the evaluation times, even when considering the time to generate the verification proof, even for a small number of nonzero points (e.g. 10 points). These results are displayed in Figure 5.

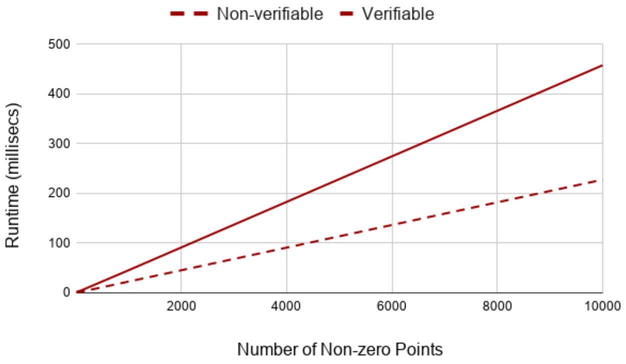

Fig. 4. This graph plots the share generation time for the textbook DMPF and the batched, verifiable DMPF presented in this work.

Acknowledgments

We would like to thank Vinod Vaikuntanathan, Henry Corrigan-Gibbs, and Yuval Ishai for helpful conversations and insights.

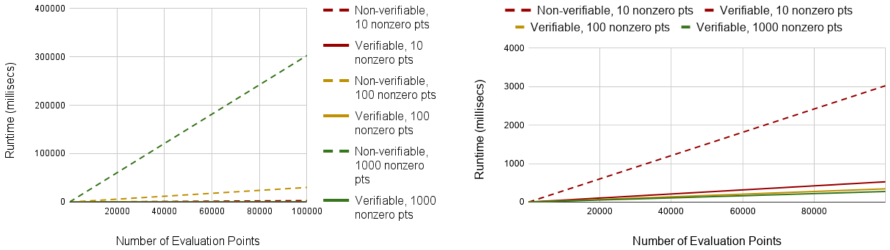

Fig. 5. This figure plots the evaluation times for the textbook DMPF and the batched, verifiable DMPF presented in this work. The domain sizes of these functions were 126 bits. Both graphs in this figure plot the same data; the first graph shows all plots while the second graph is only a plot of the smallest four lines so that the batched VDMPF runtimes can be viewed. The x-axis for these graphs is the number of points η on which the shares are evaluated, and the colors of each line represent the number of nonzero points t in the shared multi-point functions. The number of points is indicated in the legends of the graphs. Note in the second graph that the evaluation time decreases as the number of nonzero points on in the MPF grows.

Leo de Castro was supported by a JP Morgan AI Research PhD Fellowship.

This paper was prepared for informational purposes by the Artificial Intelligence Research group of JPMorgan Chase & Co. and its affiliates ("JP Morgan"), and is not a product of the Research Department of JP Morgan. JP Morgan makes no representation and warranty whatsoever and disclaims all liability, for the completeness, accuracy or reliability of the information contained herein. This document is not intended as investment research or investment advice, or a recommendation, offer or solicitation for the purchase or sale of any security, financial instrument, financial product or service, or to be used in any way for evaluating the merits of participating in any transaction, and shall not constitute a solicitation under any jurisdiction or to any person, if such solicitation under such jurisdiction or to such person would be unlawful. 2021 JPMorgan Chase & Co. All rights reserved.

This material is based upon work supported by DARPA under Agreement No. HR00112020023. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Government or DARPA.

References

1. Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Lightweight techniques for private heavy hitters. Cryptology ePrint Archive, Report 2021/017, 2021. https://eprint.iacr.org/2021/017.
2. 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.

3. Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing: Improvements and extensions. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 1292–1303, New York, NY, USA, 2016. Association for Computing Machinery.

4. Paul Bunn, Eyal Kushilevitz, and Rafail Ostrovsky. CNF-FSS and its applications. *IACR Cryptol. ePrint Arch.*, page 163, 2021.

5. Hao Chen, Kim Laine, and Peter Rindal. Fast private set intersection from homomorphic encryption. In *CCS '17 Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1243–1255. ACM New York, NY, USA ©2017, October 2017.

6. Leo de Castro and Anitgoni Polychroniadou. Lightweight, maliciously secure verifiable function secret sharing. In Orr Dunkelman and Stefan Dziembowski, editors, *Advances in Cryptology – EUROCRYPT 2022*, pages 150–179, Cham, 2022. Springer International Publishing.

7. Daniel Demmler, Peter Rindal, Mike Rosulek, and Ni Trieu. Pir-psi: Scaling private contact discovery. *Proceedings on Privacy Enhancing Technologies*, 2018:159–178, 10 2018.

8. Samuel Dittmer, Yuval Ishai, Steve Lu, Rafail Ostrovsky, Mohamed Elsabagh, Nikolaos Kiourtis, Brian Schulte, and Angelos Stavrou. Function secret sharing for psi-ca: With applications to private contact tracing. Cryptology ePrint Archive, Report 2020/1599, 2020. https://eprint.iacr.org/2020/1599.

9. Jack Doerner and Abhi Shelat. Scaling oram for secure computation. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 523–535, New York, NY, USA, 2017. Association for Computing Machinery.

10. 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.

11. Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *J. ACM*, 33(4):792–807, August 1986.

12. S. M. MATYAS, C.H. Meyer, and J. Oseas. Generating strong one-way functions with cryptographic algorithm. *IBM Technical Disclosure Bulletin*, 27:5658–5659, 1985.

13. Phillipp Schoppmann, Adrià Gascón, Leonie Reichert, and Mariana Raykova. Distributed vector-ole: Improved constructions and implementation. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 1055–1072, New York, NY, USA, 2019. Association for Computing Machinery.

14. David Wagner. A generalized birthday problem. In Moti Yung, editor, *Advances in Cryptology — CRYPTO 2002*, pages 288–304, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

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

A XOR-Collision Resistance

We briefly give the justification for why a hash output of 4λ bits is sufficient to achieve XOR-collision resistance. These parameters are based on a birthday-attack analysis, which gives more conservative parameters than an analysis based on Wagner's attack [14]. Defending against Wagner's attack would only require an output of 3λ bits.

Let $\mathcal{H} := \{\mathsf{H} \mid \mathsf{H}\colon D \to \{0,1\}^n\}$ be a hash function family with output space R such that for any fixed input $x \in D$ and output $y \in \{0,1\}^n$ we have $\Pr_{\mathsf{H} \overset{\$}{\leftarrow} \mathcal{H}}[\mathsf{H}(x) = y] \leq 2^{-n}$. The XOR-collision resistance problem asks to find four inputs $x_1, x_2, x_3, x_4 \in D$ to the hash function such that $\mathsf{H}(x_1) \oplus \mathsf{H}(x_2) = \mathsf{H}(x_3) \oplus \mathsf{H}(x_4) \neq 0$. For simplicity, we will not consider the requirement that the XOR'd values are not zero; this only makes the adversary's task easier. For any 4-tuple of inputs $x_1, x_2, x_3, x_4 \in D$ such that $x_1 \neq x_2 \neq x_3 \neq x_4$, it follows from distribution of the hash function output that $\Pr_{\mathsf{H} \overset{\$}{\leftarrow} \mathcal{H}}[\mathsf{H}(x_1) \oplus \mathsf{H}(x_2) = \mathsf{H}(x_3) \oplus \mathsf{H}(x_4)] \leq 2^{-n}$.

For a randomly sampled $\mathsf{H} \overset{\$}{\leftarrow} \mathcal{H}$, let q be the number of times that the adversary must evaluate H before an XOR-colliding 4-tuple is found. The number of 4-tuples after q evaluations is $\binom{q}{4} \leq q^4$. We want q to be 2^λ before the adversary is expected to find a colliding 4-tuple, which gives us the following

$$\begin{aligned}\mathbb{E}[\#\text{ of XOR-colliding tuples after } q \text{ evaluations}] &\leq q^4/2^n \\ &= 2^{4\lambda}/2^n = 1\end{aligned}$$

Therefore, setting $n = 4\lambda$ will require 2^λ evaluations of H before the adversary is expected to find an XOR-colliding tuple.

B Match-Mode VDMPF: Point Matching

In this section, we present an alternative evaluation mode for our VDMPF scheme that is be useful in various applications. In the "main" evaluation mode, which was presented in Figure 2, the servers produce one output for each input element to the batched evaluation algorithm. In the "match" evaluation mode discussed in this section, the servers produce one output for each of the cuckoo-hash buckets in the VDMPF key. The purpose of this evaluation mode is to determine if one of the server's input elements matches one of the non-zero points of the multi-point function.

In more detail, during the evaluation algorithm the servers still produce a set of inputs for each of the m buckets and evaluate the corresponding VDPF keys on these inputs. Instead of summing the VDPF outputs according to a matching input, the servers sum the outputs of each VDPF to create a single output for each of the m buckets. From the verifiability of the point function share in each bucket, the servers can easily ensure that the evaluation of at most one of their inputs is being revealed for each bucket. The algorithm is given in Algorithm 5.

Algorithm 5 VDMPF Match-Mode Evaluation, denoted $\mathsf{VerDMPF.MatchEval}$. The setting for this algorithm is the same as the VDMPF construction in Figure 2.

```
Input: bit $b$ and $\mathsf{VerDMPF}$ key $\mathsf{k}_b$
       $\eta$ inputs $x_1, \ldots, x_\eta$
Parse $\sigma, \mathsf{k}_b^{(1)}, \ldots, \mathsf{k}_b^{(m)} \leftarrow \mathsf{k}_b$
Define $B \leftarrow \lceil N\kappa/m \rceil$, $n' \leftarrow \lceil \log(B) \rceil$.
Let $\mathsf{VerDPF} := \mathsf{VerDPF}_{n',\mathbb{G}}$.
Initialize an array $\mathsf{inputs}$ of length $m$.
for $\omega$ from 1 to $\eta$ do
    Let $i_1, \ldots, i_\kappa \leftarrow h_1(x_\omega), \ldots, h_\kappa(x_\omega)$
    Let $j_1, \ldots, j_\kappa \leftarrow \mathsf{index}_1(x_\omega), \ldots, \mathsf{index}_\kappa(x_\omega)$
    Append $j_k$ to $\mathsf{inputs}[i_k]$ for each $k \in [\kappa]$, ignoring duplicates.
Initialize an array $\mathsf{outputs}$ of length $m$ to all zeros.
Initialize a proof $\pi \leftarrow 0$
for $i$ from 1 to $m$ do
    $\{\mathsf{y}_\ell\}_{\ell=1}^{L}, \pi^{(i)} \leftarrow \mathsf{VerDPF.BVEval}(b, \mathsf{k}_b^{(i)}, \mathsf{inputs}[i])$
    $\mathsf{outputs}[i] \leftarrow \mathsf{outputs}[i] + \mathsf{y}_\ell$ for $\ell \in [L]$
    $\pi \leftarrow \pi \oplus \mathsf{H}'(\pi \oplus \pi^{(i)})$
Output: $\mathsf{outputs}, \pi$
```

C Malicious Two-Server PIR Protocol

We will now present our malicious PIR protocol. Our construction relies heavily on the VerDPF scheme described in section 3.

Network Topology. Our two-server PIR protocol is a three-party protocol between a client $\mathcal{C}$ and two servers $\mathcal{S}_0$ and $\mathcal{S}_1$. All three parties are connected to one another.

Threat model. Our two-server PIR protocol will be secure against any one adversarial corruption. This corruption can be either semi-honest or malicious.

We will now define the API for our protocol.

Definition 13 (Two-Server PIR API). *A two-server PIR protocol* $\mathsf{PIR} := \mathsf{PIR}_{N,\mathbb{G}}$ *is parametrized by a database size* N *and a group* $\mathbb{G}$*, where each database element is a single bit. This protocol is between a client* $\mathcal{C}$ *and two servers* $\mathcal{S}_0$ *and* $\mathcal{S}_1$*. The servers begin the protocol with identical copies of a static database* $\mathcal{D} \in \mathbb{G}^N$*. Our PIR protocol consists of the following three PPT algorithms.*

- $\mathsf{dbState} \leftarrow \mathsf{PIR.Setup}(1^\lambda, N, \mathbb{G})$
  Takes as input the public parameters of the PIR *scheme and outputs state parameters that are used by the servers when evaluating queries.*
- $(\mathsf{q}_0, \mathsf{q}_1, \mathsf{state}) \leftarrow \mathsf{PIR.QueryGen}(1^\lambda, i)$
  Takes as input a query index i *and a database size* N *and outputs query values* $\mathsf{q}_0, \mathsf{q}_1$ *as well as a query state* state*. Intended to be run by the client,*

where q_b *is intended to be sent to* $\mathcal{S}_b$*. The query state* state *is held by the client to verify the response.*

- $(\mathsf{res}_b, \pi_b, \mathsf{dbState}') \leftarrow \mathsf{PIR.QueryEval}(\mathsf{q}_b, \mathcal{D}, \mathsf{dbState})$
  Takes an input a query value q_b *and a database* $\mathcal{D}$*. Outputs a response share* res_b *as well as a proof* π_b *that the query was well-formed. Also outputs updated state variables* $\mathsf{dbState}$.
- $\mathsf{Accept}/\mathsf{Reject} \leftarrow \mathsf{PIR.QueryVerify}(\pi_0, \pi_1)$
  Takes as input two proofs π_0, π_1 *of query-wellformedness and checks if the proofs are consistent with each other, indicating that the query values correspond to a single database index* $i \in [N]$*. Outputs* Accept *if this check passes, and* Reject *otherwise.*
- x *or* $\perp \leftarrow \mathsf{PIR.Reconstruct}(\mathsf{res}_0, \mathsf{res}_1, \mathsf{state})$
  Takes as input two response shares $\mathsf{res}_0, \mathsf{res}_1$ *as well as a query state* state*. If the response shares are not consistent with the query state, output* $\perp$*. Otherwise, reconstruct the response* $x \in \{0, 1\}$*, which should equal the* i^{th} *entry in the database* $\mathcal{D}[i]$*.*

We give formal definitions for security and privacy for a malicious and verifiable PIR protocol in appendix C.1.

We will now present our construction of a malicious two-server PIR protocol. This protocol will rely heavily on the verifiable DPF described in section 3. The construction is given in figure 6.

Security against a malicious client follows from the verifiability of the DPF. Security against a malicious server follows from the size of the output group. Since β is a random element of the output group, a malicious server cannot flip the output bit without guess β. The proofs of security are given in appendix C.2.

Due to space constraints, we present our PSI construction in appendix D. Intuitively, this construction is mirrors the PIR construction, except with use a VDMPF to batch the queries of the client set.

C.1 PIR Security and Privacy Definitions

We will now define security for this PIR protocol.

Definition 14 (PIR Privacy Against a Malicious Server). *Let* $\mathsf{PIR} := \mathsf{PIR}_{N,\mathbb{G}}$ *be a PIR protocol as defined in definition 13.*

$$\mathsf{View}_{\mathsf{PIR}}(b, i; \mathcal{D}, \mathsf{dbState}) :=$$
$$\left\{ (\mathsf{q}_b, \pi_{1-b}) \;\middle|\; \begin{array}{c} (\mathsf{q}_0, \mathsf{q}_1) \leftarrow \mathsf{PIR.QueryGen}(1^\lambda, i), \\ (\_, \pi_{1-b}) \leftarrow \mathsf{PIR.QueryEval}(\mathsf{q}_{1-b}, \mathcal{D}, \mathsf{dbState} \end{array} \right\}$$

We say that this PIR protocol maintains privacy against a malicious server if there exists a simulator Sim *for any* $i \in [N]$ *the following two distributions are computationally indistinguishable.*

$$\{(\mathsf{q}_b, \pi_{1-b}) \mid (\mathsf{q}_b, \pi_{1-b}) \leftarrow \mathsf{View}_{\mathsf{PIR}}(b, i; \mathcal{D}, \mathsf{dbState})\}$$
$$\approx_c \{(\mathsf{q}^*, \pi^*) \mid (\mathsf{q}^*, \pi^*) \leftarrow \mathsf{Sim}(1^\lambda, \mathcal{D}, \mathsf{dbState})\}$$

Let λ be a security parameter. For a group $\mathbb{G}$ with size at least 2^λ, let $\mathsf{VerDPF} := \mathsf{VerDPF}_{n,\mathbb{G}}$ be a verifiable distributed point function secret sharing scheme. Let $N = 2^n$. Let $\mathsf{PIR} := \mathsf{PIR}_{N,\mathbb{G}}$. Let $\mathcal{G}\colon \{0,1\}^\lambda \to \mathbb{G} \times \{0,1\}^\lambda$ be a secure PRG.

```
$\mathsf{PIR.Setup}(1^\lambda, N, \mathbb{G})$
  Sample a random seed $s$ for $\mathcal{G}$.
Output: $\mathsf{dbState} \leftarrow s$
```

We assume that the two servers have the same value of $\mathsf{dbState}$. We now give the algorithms to process a database query.

Client Algorithms

```
$\mathsf{PIR.QueryGen}(1^\lambda, i)$
  Sample $r \xleftarrow{\$} \mathbb{G}$
  Define the point function $f_{i,r}$.
  $(\mathsf{k}_0, \mathsf{k}_1) \leftarrow \mathsf{VerDPF.Gen}\left(1^\lambda, f_{i,r}\right)$
Output: $(\mathsf{q}_0, \mathsf{q}_1, \mathsf{state}) \leftarrow (\mathsf{k}_0, \mathsf{k}_1, r)$

$\mathsf{PIR.Reconstruct}(\mathsf{res}_0, \mathsf{res}_1, \mathsf{state})$
  $g \leftarrow \mathsf{res}_0 + \mathsf{res}_1$.
  if $g = \mathsf{state}$ then $\omega \leftarrow 1$
  else if $g = 0$ then $\omega \leftarrow 0$
  else $\omega \leftarrow \perp$
Output: $\omega$
```

Server Algorithms

```
$\mathsf{PIR.QueryEval}(\mathsf{q}_b, \mathcal{D}, \mathsf{dbState})$
  $(\mathsf{y}_b, \pi_b) \leftarrow \mathsf{VerDPF.FDEval}(b, \mathsf{q}_b)$
  $z \leftarrow \langle \mathsf{y}_b, \mathcal{D} \rangle$          ▷ Inner product.
  $(g, s') \leftarrow \mathcal{G}(\mathsf{dbState})$ ▷ $\mathsf{dbState}$ is just
      a PRG seed.
  if $b = 0$ then $z \leftarrow z + g$
  else $z \leftarrow z - g$
Output: $(\mathsf{res}_b, \pi_b, \mathsf{dbState}') \leftarrow (z, \pi_b, s')$

$\mathsf{PIR.QueryVerify}(\pi_0, \pi_1)$
  if $\pi_0 = \pi_1$ then $\nu \leftarrow \mathsf{Accept}$
  else $\nu \leftarrow \mathsf{Reject}$
Output: $\nu$
```

Fig. 6. Our PIR construction.

Definition 15 (PIR Security Against a Malicious Server). *Let* $\mathsf{PIR} := \mathsf{PIR}_{N,\mathbb{G}}$ *be a PIR protocol as defined in definition 13. For* $b \in \{0,1\}$*, let* $\mathcal{S}_b^*$ *be the malicious server and let* $\mathcal{S}_{1-b}$ *be the honest server. Let* $\mathcal{D}$ *be the database copy held by the honest server. For* $i \in [N]$*, let* $(\mathsf{q}_0, \mathsf{q}_1, \mathsf{state}) \leftarrow \mathsf{PIR.QueryGen}(1^\lambda, i, N)$ *be the client's query shares, let* res_{1-b} *be the honest server's response, and let* res_b^* *be the response from the malicious server. We say that* PIR *is secure against a malicious server if no PPT malicious server* $\mathcal{S}_b^*$ *can produce a response* res_b^* *such that the output* $x^* \leftarrow \mathsf{PIR.Reconstruct}(\mathsf{res}_0, \mathsf{res}_1, \mathsf{state})$ *is not in the set* $\{\mathcal{D}[i], \bot\}$*. In other words, if* PIR.Reconstruct *does not output* $\bot$*, then the output should be* $x = \mathcal{D}[i]$*, where* $\mathcal{D}$ *is the honest server's copy of the database.*

Definition 16 (PIR Security Against a Malicious Client). *Let* $\mathsf{PIR} := \mathsf{PIR}_{N,\mathbb{G}}$ *be a PIR protocol as defined in definition 13. For any query shares* q_0 *and* q_1*, define the following distribution of outputs of a pair of honest servers. Let* $(\mathsf{res}_b, \pi_b) \leftarrow \mathsf{PIR.QueryEval}(\mathsf{q}_b, \mathcal{D})$ *for* $b \in \{0,1\}$*.*

$$\mathsf{Out}(\mathsf{q}_0, \mathsf{q}_1; \mathcal{D}) := \begin{cases} \{\mathsf{res}_0, \mathsf{res}_1\} \\ \quad \textit{if } \mathsf{Accept} \leftarrow \mathsf{PIR.QueryVerify}(\pi_0, \pi_1) \\ \bot \quad \textit{otherwise.} \end{cases}$$

We say that the PIR *scheme is secure against a malicious client if there exists a PPT simulator* Sim *such that for any PPT adversary* $\mathcal{A}$ *and the following holds.*

$$\left\{\mathsf{Out}(\mathsf{q}_0, \mathsf{q}_1; \mathcal{D}) \mid (\mathsf{q}_0, \mathsf{q}_1) \leftarrow \mathcal{A}(1^\lambda, N, \mathbb{G})\right\} \approx_c$$
$$\left\{\mathsf{Sim}(\mathsf{q}_0, \mathsf{q}_1; \mathcal{D}[i]) \mid (\mathsf{q}_0, \mathsf{q}_1) \leftarrow \mathcal{A}(1^\lambda, N, \mathbb{G})\right\}$$

C.2 PIR Security and Privacy Proofs

We now argue the security of the PIR protocol in figure 6.

Lemma 9 (PIR Privacy Against a Malicious Server). *The PIR construction given in figure 6 satisfies definition 14.*

Proof. This lemma follows from the function hiding of the underlying VDPF scheme, shown in lemma 4.

Lemma 10 (PIR Security Against a Malicious Server). *The PIR construction given in figure 6 satisfies definition 15.*

Proof. This lemma follows from the size of the group $\mathbb{G}$. The output group element $r \xleftarrow{\$} \mathbb{G}$ is sampled uniformly at random. In order to fool the client into outputting the wrong value (as opposed to $\bot$), a malicious server $\mathcal{S}_b^*$ must output an additive share y_b^* that shifts the true output by exactly r. Let $g = \mathsf{y}_0 + \mathsf{y}_1$ be the true output where both output shares are computed honestly, and let $g^* = \mathsf{y}_b^* + \mathsf{y}_{1-b}$ be the output the client receives when using the malicious output share. Since we assume the client's share is honestly generated, the malicious server is able to compute the honest share y_b. If $g = 0$, then we must have $g^* = g + r$, so $g^* - g = \mathsf{y}_b^* - \mathsf{y}_b = r$. If $g = r$, then we have $g^* = g - r$, so $g - g^* = \mathsf{y}_b - \mathsf{y}_b^* = r$. In both cases, the output r is implicit in the view of $\mathcal{S}_b^*$, so either the malicious server has broken the function hiding property of the VDPF or has correctly guessed the value of a uniformly random element of $\mathbb{G}$, which succeeds with probability $1/|\mathbb{G}| \leq 2^{-\lambda}$.

Lemma 11 (PIR Security Against a Malicious Client). *The PIR construction given in figure 6 satisfies definition 16.*

Proof. This lemma follows from the verifiability of the DPF. From lemma 2, if the output proofs of the VDPF evaluation algorithm match, then the output is a secret sharing of the truth table of some point function. This point function, by definition, must have at most one non-zero element, and the index of this non-zero element is exactly what the simulator will use as the index of the database element to return. The evaluation algorithms are completely deterministic, so this index, if it exists, is completely determined by the input query. Since the real shares are also masked with a pseudorandom secret sharing of zero, the simulator can simply sample a truly random share of the database element at the correct index.

D Malicious Two-Server PSI Protocol

We will now define the API for our protocol.

Definition 17 (Two-Server PSI API). *A two-server PSI protocol* $\mathsf{PSI} := \mathsf{PSI}_{N,\mathbb{G}}$ *is parametrized by a domain* $\mathcal{D}$ *of size* N *and a group* $\mathbb{G}$*. This protocol is between a client* $\mathcal{C}$ *and two servers* $\mathcal{S}_0$ *and* $\mathcal{S}_1$*. The servers begin the protocol with identical copies of a set* $Y \subset \mathcal{D}$*, and the client begins with a set* $X \subset \mathcal{D}$*. Our PSI protocol consists of the following three PPT algorithms.*

- $\mathsf{dbState} \leftarrow \mathsf{PSI.Setup}(1^\lambda, N, \mathbb{G})$
  Takes as input the public parameters of the PSI *scheme and outputs state parameters that are used by the servers when evaluating queries.*
- $(\mathsf{q}_0, \mathsf{q}_1, \mathsf{state}) \leftarrow \mathsf{PSI.QueryGen}(1^\lambda, X)$
  Takes as input a set $X \subset \mathcal{D}$ *and outputs query values* $\mathsf{q}_0, \mathsf{q}_1$ *as well as a query state* state*. Intended to be run by the client, where* q_b *is intended to be sent to* $\mathcal{S}_b$*. The query state* state *is held by the client to verify the response.*
- $(\mathsf{res}_b, \pi_b, \mathsf{dbState}') \leftarrow \mathsf{PSI.QueryEval}(\mathsf{q}_b, Y, \mathsf{dbState})$
  Takes an input a query value q_b *and a set* $Y \subset \mathcal{D}$*. Outputs a response share* res_b *as well as a proof* π_b *that the query was well-formed. Also outputs updated state variables* dbState*.*
- $\mathsf{Accept}/\mathsf{Reject} \leftarrow \mathsf{PSI.QueryVerify}(\pi_0, \pi_1)$
  Takes as input two proofs π_0, π_1 *of query-wellformedness and checks if the proofs are consistent with each other, indicating that the query values correspond to a set of at most some size that is a deterministic function of the security parameter and the honest number of nonzero points. Outputs* Accept *if this check passes, and* Reject *otherwise.*

- Z *or* $\perp \leftarrow \mathsf{PSI.Reconstruct}(\mathsf{res}_0, \mathsf{res}_1, \mathsf{state})$
  Takes as input two response shares $\mathsf{res}_0, \mathsf{res}_1$ *as well as a query state* state*. If the response shares are not consistent with the query state, output* $\perp$*. Otherwise, reconstruct the response* $Z = X \cap Y$*.*

We give formal definitions for security and privacy for a malicious and verifiable PSI protocol in appendix D.1.

We will now present our construction of a malicious two-server PSI protocol. This protocol will rely heavily on the verifiable DMPF described in section 4. The construction is given in figure 7. The proofs of security are given in appendix D.2.

Let λ be a security parameter. For a domain size $\mathcal{D}$ with size N and group $\mathbb{G}$, let $\mathsf{PSI} := \mathsf{PSI}_{N,\mathbb{G}}$. Let $\mathsf{VerDMPF} := \mathsf{VerDMPF}_{N,\mathbb{G}}$ be the verifiable distributed multi-point function secret sharing scheme as defined in figure 2. Let $\mathcal{G} \colon \{0,1\}^\lambda \to \mathbb{G} \times \{0,1\}^\lambda$ be a secure PRG.
We assume that the two servers have the same value of $\mathsf{dbState}$. We now give the algorithms to process a database query.

Client Algorithms

```
PSI.QueryGen($1^\lambda, X$)
  Define $t \leftarrow |X|$.
  For each $x_i \in X$, sample $r_i \xleftarrow{\$} \mathbb{G}$.
  Define the point functions $f_i := f_{x_i,r_i}$ for each $i \in [t]$.
  $(\mathsf{k}_0, \mathsf{k}_1) \leftarrow \mathsf{VerDMPF.Gen}\left(1^\lambda, \{f_i\}_{i=1}^t\right)$
Output: $(\mathsf{q}_0, \mathsf{q}_1, \mathsf{state}) \leftarrow (\mathsf{k}_0, \mathsf{k}_1, \{(x_i, r_i)\}_{i=1}^t)$

PSI.Reconstruct($\mathsf{res}_0, \mathsf{res}_1, \mathsf{state}$)
  Parse $(x_1, r_1), \dots, (x_t, r_t) \leftarrow \mathsf{state}$
  Initialize $Z \leftarrow \emptyset$
  for $(z_1, z_2)$ in $(\mathsf{res}_0, \mathsf{res}_1)$ do
      $z \leftarrow z_0 + z_1$.
      if $z = r_i$ then Insert $x_i$ in $Z$
Output: $Z$
```

Server Algorithms

```
PSI.Setup($1^\lambda, N, \mathbb{G}$)
  Sample a random seed $s$ for $\mathcal{G}$.
Output: $\mathsf{dbState} \leftarrow s$

PSI.QueryEval($\mathsf{q}_b, Y, \mathsf{dbState}$)
  $(\mathsf{y}_b, \pi_b) \leftarrow \mathsf{VerDMPF.MatchEval}(b, \mathsf{q}_b, Y)$   ▷ Algorithm 5
  for $\mathsf{y}_b^{(i)}$ in $\mathsf{y}_b$ do
      $(g, \mathsf{dbState}) \leftarrow \mathcal{G}(\mathsf{dbState})$
      if $b = 0$ then $\mathsf{y}_b^{(i)} \leftarrow \mathsf{y}_b^{(i)} + g$
      else $\mathsf{y}_b^{(i)} \leftarrow \mathsf{y}_b^{(i)} - g$
Output: $(\mathsf{res}_b, \pi_b, \mathsf{dbState}) \leftarrow (\mathsf{y}_b, \pi_b, \mathsf{dbState})$

PSI.QueryVerify($\pi_0, \pi_1$)
  if $\pi_0 = \pi_1$ then $\nu \leftarrow \mathsf{Accept}$
  else $\nu \leftarrow \mathsf{Reject}$
Output: $\nu$
```

Fig. 7. Our PSI construction.

D.1 PSI Security and Privacy Definitions

Definition 18 (PSI Privacy Against a Malicious Server). *Let* $\mathsf{PSI} := \mathsf{PSI}_{N,\mathbb{G}}$ *be a PSI protocol as defined in definition 17.*

$$\mathsf{View}_{\mathsf{PSI}}(b, X; \mathcal{D}, \mathsf{dbState}) :=$$
$$\left\{ (\mathsf{q}_b, \pi_{1-b}) \;\middle|\; \begin{array}{c} (\mathsf{q}_0, \mathsf{q}_1) \leftarrow \mathsf{PSI.QueryGen}(1^\lambda, X), \\ (\_, \pi_{1-b}) \leftarrow \mathsf{PSI.QueryEval}(\mathsf{q}_{1-b}, Y, \mathsf{dbState} \end{array} \right\}$$

We say that this PSI protocol maintains privacy against a malicious server if there exists a simulator Sim *for any* $i \in [N]$ *the following two distributions are computationally indistinguishable.*

$$\{(\mathsf{q}_b, \pi_{1-b}) \mid (\mathsf{q}_b, \pi_{1-b}) \leftarrow \mathsf{View}_{\mathsf{PSI}}(b, X; \mathcal{D}, \mathsf{dbState})\}$$
$$\approx_c \{(\mathsf{q}^*, \pi^*) \mid (\mathsf{q}^*, \pi^*) \leftarrow \mathsf{Sim}(1^\lambda, Y, \mathsf{dbState})\}$$

Definition 19 (PSI Security Against a Malicious Server). *Let* $\mathsf{PSI} := \mathsf{PSI}_{N,\mathbb{G}}$ *be a PSI protocol as defined in definition 17. For* $b \in \{0,1\}$*, let* $\mathcal{S}_b^*$ *be the malicious server and let* $\mathcal{S}_{1-b}$ *be the honest server. Let* $\mathcal{D}$ *be the database copy held by the honest server. For* $X \subset \mathcal{D}$*, let* $(\mathsf{q}_0, \mathsf{q}_1, \mathsf{state}) \leftarrow \mathsf{PSI.QueryGen}(1^\lambda, X, N)$ *be the client's query shares, let* res_{1-b} *be the honest server's response, and let* res_b^* *be the response from the malicious server. We say that* PSI *is secure against a malicious server if no PPT malicious server* $\mathcal{S}_b^*$ *can produce a response* res_b^* *such that the output* $x^* \leftarrow \mathsf{PSI.Reconstruct}(\mathsf{res}_0, \mathsf{res}_1, \mathsf{state})$ *is not in the set* $\{\mathcal{D}[i], \perp\}$*. In other words, if* $\mathsf{PSI.Reconstruct}$ *does not output* $\perp$*, then the output should be* $x = X \cap Y$*, where* Y *is the honest server's copy of the database.*

Definition 20 (PSI Security Against a Malicious Client). *Let* $\mathsf{PSI} := \mathsf{PSI}_{N,\mathbb{G}}$ *be a PSI protocol as defined in definition 17. For any query shares* q_0 *and* q_1*, define the following distribution of outputs of a pair of honest servers. Let* $(\mathsf{res}_b, \pi_b) \leftarrow \mathsf{PSI.QueryEval}(\mathsf{q}_b, Y)$ *for* $b \in \{0,1\}$*.*

$$\mathsf{Out}(\mathsf{q}_0, \mathsf{q}_1; Y) := \begin{cases} \{\mathsf{res}_0, \mathsf{res}_1\} \\ \quad \textit{if } \mathsf{Accept} \leftarrow \mathsf{PSI.QueryVerify}(\pi_0, \pi_1) \\ \perp \quad \textit{otherwise.} \end{cases}$$

We say that the PSI *scheme is secure against a malicious client if there exists a PPT simulator* Sim *such that for any PPT adversary* $\mathcal{A}$ *and the following holds.*

$$\{\mathsf{Out}(\mathsf{q}_0, \mathsf{q}_1; Y) \mid (\mathsf{q}_0, \mathsf{q}_1) \leftarrow \mathcal{A}(1^\lambda, N, \mathbb{G})\} \approx_c$$
$$\{\mathsf{Sim}(\mathsf{q}_0, \mathsf{q}_1; X' \cap Y) \mid (\mathsf{q}_0, \mathsf{q}_1) \leftarrow \mathcal{A}(1^\lambda, N, \mathbb{G})\}$$

where $|X'| \leq 2t$ *and* t *is the size of the honest set.*

D.2 PSI Security & Privacy Proofs

Lemma 12 (PSI Privacy Against a Malicious Server). *The PSI construction given in figure 7 satisfies definition 18.*

Proof. This follows from the function hiding of the VDMPF.

Lemma 13 (PSI Security Against a Malicious Server). *The PSI construction given in figure 7 satisfies definition 19.*

Proof. This follows from a parallel argument to lemma 10; namely, it follows from the size of the output group $\mathbb{G}$.

Lemma 14 (PSI Security Against a Malicious Client). *The PSI construction given in figure 7 satisfies definition 20.*

Proof. This follows from the verifiability of the VDMPF.