Lightweight, Verifiable Function Secret Sharing and its Applications

Leo de Castro
MIT CSAIL
ldec@mit.edu

Antigoni Polychroniadou
J.P. Moran AI Research
antigoni.poly@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. We use these verifiable FSS schemes to construct two-server private information retrieval and private set intersection that are secure & verifiable in the face of any one malicious corruption.**

Our verifiability method is lightweight in two ways. Firstly, it is concretely very efficient, making use of only symmetric key operations and no MPC or linear PCP techniques. For security parameter λ, our verification procedure is simply to check if two 2λ-bit strings match. 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 depended on at least one and often all three of these factors.

In addition, we give a novel method for packing DPFs into shares of a multi-point function that allows for the number of nonzero points in the multi-point function to grow without growing the evaluation time. We also show how our verification scheme carries over to the multi-point setting.

We give an implementation of our verifiable distributed point functions and our verifiable distributed multi-point function.

I. Introduction

Function secret sharing (FSS), first introduced by Boyle et al. [3], 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-sharing a function $f\colon \mathcal{D} \to \mathbb{G}$ and produce function shares k_0 and k_1. These shares have several useful properties. Firstly, viewing either share alone computationally hides the function f. Second, 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 $\mathsf{k}_0(x) + \mathsf{k}_1(x) = f(x)$.

Boyle et al. [4] gave an efficient construction of a distributed point function (DPF), which is a special case of FSS that supports point functions. A point function $f\colon \mathcal{D} \to \mathbb{G}$ 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 $f_{\alpha,\beta}$.

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) [4], private set intersection (PSI) [10], Oblivious-RAM [12], contact-tracing [11], and many more [1], [23]. These two-server protocols often have a similar structure. For example, the PIR construction from a DPF begins with a client generating DPF shares for the function $f_{i,1}$, where i is the query index. The client then sends one function share to each server; the servers hold an identical copy of the database of size N. 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 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. 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. [4] also constructed VDPFs. These VDPF schemes relied on some form of multi-party computation (MPC) between the servers. In order to efficiently run this MPC, the servers relied on correlated randomness provided by the client. However, this verification procedure is only relevant when the client is potentially malicious, and hence may send malformed correlated randomness. Therefore, the MPC that is used in these verification techniques must be resilient against malformed randomness, which limits the usefulness of the techniques in several ways. Firstly, the output field must always be at least as large as the security parameter, which prevents applications where a smaller field is necessary. Second, the verification procedure often strictly limits the values of β that will pass the verification. In particular, there is often only two options for β, such as $\beta \in \{0, 1\}$ or $\beta \in \{-1, 1\}$. This prevents applications with many valid β values. Finally, as discussed in more detail in Boyle et al., it is often the case that only some subset of the domain can be verified to be a valid point function, rather than the full domain, in order to maintain the MPC security for malformed client randomness. While there has been recent improvements in the verification method for $\beta \in \{0, 1\}$ in the work of Boneh et al. [1], the persistence of these issues raises the need for better DPF verification techniques.

Distributed Multi-Point Functions: While DPFs result in a surprisingly rich class of two-server protocols, in many applications [10], [11] it is desirable to have FSS for functions with more than one nonzero point, which we call multi-point functions (MPFs) (see section IV-B for more details). Naively, this requires constructing a DPF for each nonzero point. To evaluate the 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.

A. Our Contribution & Technical Overview

This paper improves the state-of-the-art in DPF constructions in two main ways.

Lightweight Verifiable DPF: We give a lightweight construction of a verifiable DPF, which admits a very efficient way to verify that the 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, as we show in section VI. Unlike all other DPF verification methods, we do not make use of any public key operations or 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 share is a well-formed DPF share regardless of output field size, regardless of the value of the non-zero output element, and our approach also works for any set of evaluations. 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. This is in stark contrast to prior works, which depend on at least one and often all of these constraints.

As a brief technical overview, our VDPF construction takes a non-black-box view of the GGM-style DPF construction of Boyle et al. [4]. For readers not familiar with this construction, we provide a more thorough review in section III. At a high level, this tree-based construction punctures a single path through a pair of GGM PRFs by applying a *correction word* at each level. Our observation is that a correction word can correct at most one difference at each level. By extending the GGM tree by one additional level, we can map the true output to the left leaves and then check that all of the right leaves are the same. Since the correction word can only correct one difference, if all the right leaves are the same, then all but one of the left leaves must also be the same. For more details and intuition about this approach, see section III.

Efficient Batched DPFs: We give an efficient technique to batch DPFs into a distributed multi-point function (DMPF). Our batching technique can be viewed orthogonally to our verification technique, and we believe it is of independent interest. However, we give a construction of our DMPF that is also efficiently verifiable, obtaining a verifiable DMPF (VDMPF). Our verification guarantee is a bit weaker than in the DPF. Namely, if a multi-point function has t non-zero points, our verification procedure will guarantee to the servers that they hold well-formed share for some multi-point function with no more that $2t$ evaluation points. More details on the exact tightness of this relation in section IV-A.

The main efficiency improvement of our VDMPF construction is that the evaluation time is significantly faster than the naive, or "textbook," DMPF evaluation time. In the textbook DMPF construction, evaluation of a share with t nonzero points requires evaluating t DPF shares. In our batched construction, only three DPF shares must be evaluated, *regardless* of the number of nonzero points in the point function. We use a novel PRP-based Cuckoo-hashing scheme to achieve this evaluation speed-up while still maintaining an efficient share generation time. As we show below, share generation time increases by a factor of 4-5$\times$ over the textbook version, but overall this is only a few additional milliseconds of client work that saves orders of magnitude (as shown in figure 6) on the server side for functions with many nonzero points.

As a brief technical overview, our VDMPF construction makes use of cuckoo hashing with $\kappa = 3$ hash functions to pack the t nonzero points of the MPF into a cuckoo hash table of size $m < 2t$. For each bucket in the table, the client creates a VDPF, and this is the VDMPF share. Now, the server only needs to evaluate the DPF in three buckets instead of all t. The full construction and further performance optimizations are given in section IV.

Ultimately, we will show the following two theorems.

Theorem I.1 (Verifiable DPF (informal))**.** *There exists a secure verifiable DPF for any point function $f\colon \{0,1\}^n \to \mathbb{G}$. 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 $O(n\lambda)$. The proof size is 2λ and the verification time is also 2λ, simply checking that the two proofs match.*

Theorem I.2 (Verifiable DMPF (informal))**.** *There is a secure verifiable DMPF construction for multi-point functions $f\colon [N] \to \mathbb{G}$ with at most t non-zero evaluation points. For any MPF f, security parameter λ, and $m = O(t\lambda + t\log(t))$, the runtime of share generation is $O\left(m\lambda \log(N/m)\right)$. The runtime of share evaluation is $O(\lambda \log(N/m))$. The proof size is 2λ and the verification time is also 2λ, simply checking that the two proofs match.*

We also provide an implementation of our VDPF and VDMPF and present benchmarks in section VI.

Malicious Two-server Protocols: As an immediate consequence, we obtain several concretely efficient and maliciously secure protocols in the two-server model. We present two in this work to illustrate the usefulness of our construction. All protocols we present are secure against any one malicious

corruption (i.e. either the client or one of the two servers) and are verifiable by all parties.

1) A malicious two-server PIR protocol, where the honest servers are guaranteed that at most one database entry is revealed per query. The honest client is guaranteed that the protocol is run correctly on the database, otherwise the protocol outputs $\perp$. In other words, a malicious server cannot, except with negligible probability, make the honest client output an incorrect result to its query. The ideal functionality of this protocol is defined in figure 7.
2) A malicious two-server PSI protocol, where if the honest set size is t then the servers are guaranteed that their answer can be simulated by the ideal functionality with an input set of size at most $2t$. As in the PIR protocol, a malicious server cannot alter the client's output (i.e. the intersection) from the correct output without being detected with overwhelming probability. The ideal functionality of this protocol is defined in figure 8.

B. Related Work

The most relevant related work is the verifiable DPF constructions of Boyle et al. [4] and subsequent works of Boneh et al. [1]. As discussed in the introduction, the constraints on these techniques make then incomparable to ours, although we note that all of the techniques in these works make use of some MPC, while our VDPF verification uses no MPC.

For other relevant works on two-server protocols, we briefly mention the work of Demmler et al. [10], which is a two-server PSI protocol that uses both distributed point functions and cuckoo hashing, although in very different ways than in our construction. They cite Boyle et al. [4] as a method to achieve malicious security, but they do not provide a maliciously secure implementation. As with any other protocol that uses DPFs, our verifiable DPF can be used in this protocol to immediately obtain meaningful malicious security guarantees. Furthermore, our VDMPF construction could be used here to likely give a performance improvement along with malicious security.

In general, our two-server protocol setting differs from the majority of prior works in this area because we explicitly consider malicious security. We therefore choose to compare directly to DPF constructions, which we believe best showcases the improvements in this work.

There are many works that focus on malicious security for multiparty PSI [5], [7], [13], [14], [16], [17], [19], [20]. However, these protocols consider multi-party PSI, which is distinct from our setting. Our setting is essentially two-party PSI where one party is split into two servers, so these protocols are not comparable to ours.

There is a recent line of work [9], [22] on two-server PIR protocols in the preprocessing model with online computation time that is sublinear in the database size. This line of work differs from ours in two ways. We do not consider preprocessing, and these preprocessing works all focus on achieving only semi-honest security.

II. Background

A. 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 $[n] := \{1, \ldots, n\}$.

B. Function Secret Sharing

In this section, we give a high level definition of functions 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.

Definition II.1 (Function Secret Sharing, Syntax & Correctness [3], [4])**.** *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 II.2 (FSS Security: Function Privacy [3], [4])**.** *Let* FSS *be a function secret sharing scheme for the function class $\mathcal{F}$, as defined in Theorem II.1. For any $f, f' \in \mathcal{F}$, the following should hold:*

$$\{\mathsf{k}_b \mid (\mathsf{k}_0, \mathsf{k}_1) \leftarrow \mathsf{FSS.Gen}(\{0,1\}, f)\} \approx_c \{\mathsf{k}'_b \mid (\mathsf{k}'_0, \mathsf{k}'_1) \leftarrow \mathsf{FSS.Gen}(\{0,1\}, f')\}, \text{ for } b \in \{0,1\}$$

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 as well as background on the concrete DPF construction that is the starting point for our protocol. We begin by defining a point function.

Definition II.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 II.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 $\mathcal{F}$.*

III. Lightweight, Verifiable DPF

In this section, we present our lightweight, verifiable distributed point function. Our approach follows the construction of Boyle et al. [4] and the verification approach follows the punctured PRF verification method of Boyle et al. [2].

A. Definitions

We begin by defining a verifiable DPF.

Definition III.1 (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* $\mathsf{VerDPF} := \mathsf{VerDPF}_{n,\mathbb{G}}$.

- $\mathsf{VerDPF.Gen}(1^\lambda, f_{\alpha,\beta}) \to (\mathsf{k}_0, \mathsf{k}_1)$
  Takes in a function $f_{\alpha,\beta}$ and generates two shares k_0 and k_1.
- $\mathsf{VerDPF.Eval}(b, \mathsf{k}_b, x \in \mathcal{D}) \to y_b$
  Same as DPF.Eval*, as defined in definition II.1.*
- $\mathsf{VerDPF.FDEval}(b, \mathsf{k}_b) \to \left(\{y_b^{(i)}\}_{i=1}^N, \pi_b\right)$
  The VerDPF.FDEval *algorithm outputs additive shares $\{y_b^{(i)}\}_{i=1}^N \in \mathbb{G}^N$ for $N = 2^n$ such that $y_0^{(i)} + y_1^{(i)} = f(i)$. The* FDEval *algorithm also outputs 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* VerDPF.Verify *algorithm takes in the proofs π_0 and π_1 from $(\mathsf{y}_b, \pi_b) \leftarrow \mathsf{FDEval}(b, \mathsf{k}_b)$ 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.*

Correctness for a verifiable DPF is defined in the same way as correctness for any FSS scheme, as in definition II.1.

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 III.2 (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. Let $(\{y_b^{(i)}\}_{i=1}^n, \pi_b) \leftarrow \mathsf{VerDPF.FDEval}(b, \mathsf{k}_b)$. We say that* VerDPF *is secure against malicious share generation if the the following holds. If $\mathsf{VerDPF.Verify}(\pi_0, \pi_1)$ outputs* Accept*, then the values $y_0^{(i)} + y_1^{(i)}$ must define the truth table of some $f_{\alpha,\beta} \in \mathcal{F}$.*

Definition III.3 (Verifiable DPF Function Privacy, or Security Against a Malicious Evaluator)**.** *Let* $\mathsf{VerDPF} := \mathsf{VerDPF}_{n,\mathbb{G}}$ *support point function class $\mathcal{F}$. Define the distribution representing the view of server $\mathcal{S}_b$ for a fixed function $f \in \mathcal{F}$.*

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

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

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

B. Our Construction

We will now review the DPF construction of Boyle et al. [4], the starting point for our protocol, which is a construction of two-server functions secret sharing scheme for distributed point functions. Recall that a distributed point function scheme allows a party to run an algorithm $(\mathsf{k}_0, \mathsf{k}_1) \leftarrow \mathsf{Gen}(1^\lambda, f_{\alpha,\beta})$, where $f_{\alpha,\beta}\colon \{0,1\}^n \to \mathbb{G}$ is a point function. This party 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 of the non-zero point of f, but not the fact that it's 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 $\mathsf{y}_0 + \mathsf{y}_1 = f_{\alpha,\beta}(x)$.

This DPF scheme is based on the PRF construction of Goldreich, Goldwasser, & Micali [15]. At a high level, the idea behind this DPF scheme is that a the zero function can be secret-shared by giving each server an identical copy of a PRF along with an additional bit that indicates if the output should be negated. Each server $\mathcal{S}_0$ and $\mathcal{S}_1$ is each given the same PRF seed σ. The servers can then evaluate their PRF on the same input x, then server $\mathcal{S}_1$ negates the output. The result will be that outputs of the same input x will sum to zero, making this a secret sharing of the zero function.

We can then instantiate this PRF using the GGM construction, where the output of the PRF is a leaf of the GGM 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. 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 arrange operations at the final level to turn this one specific mismatched pair into additive shares of the non-zero output. The main idea will be that as soon as we diverge from this path, the seeds will match again, bringing us back to the zero-function state. To achieve this, a correction operation is applied each GGM node as we traverse the tree.

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 III.4 (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:*

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

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 on place in which the two trees differ, which is the node in that level corresponding to 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 III.4, we perform a correction operation at each node as we traverse the tree. Each level of the tree is associated with a correction word, which is defined in defintion III.6. At each node, we perform the PRG expansion defined in definition III.5, then apply the correction operation define in definition III.8 to compute the seeds and control bits for the left and right children.

Definition III.5 (VerDPF PRG Expansion [4])**.** *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 III.6 (VerDPF Correction Word [4])**.** *A* VerDPF *correction word consist of a seed* $s_c \in \{0,1\}^\lambda$ *and two correction bits* t_c^L *and* t_c^R*, where* t_c^L *corresponds to the left direction and* t_c^R *corresponds to the right direction. Denote the full correction word as* $\mathsf{cw} = (s_c, t_c^L, t_c^R)$*.*

Definition III.7 (VerDPF Output Correction Word [4])**.** *For a* DPF *supporting point functions with output group* $\mathbb{G}$*, an output correction word is a single element of the group* $\mathbb{G}$*. Denote this group element as* ocw*.*

Definition III.8 (VerDPF Correction Operation [4])**.** *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*.*

Algorithm 1 $\mathsf{VerDPF}_{n,\mathbb{G}}$ Node Expansion, denoted 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: $(s'_0, t'_0), (s'_1, t'_1)$

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.

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,

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.

1: Expand $(s_b^L, t_b^L, s_b^R, t_b^R) \leftarrow \mathcal{G}(s_b)$ for $b \in \{0,1\}$.
2: **if** $x = 0$ **then** $\mathsf{Diff} \leftarrow L$, $\mathsf{Same} \leftarrow R$ ▷ Set the right children to be equal.
3: **else** $\mathsf{Diff} \leftarrow R$, $\mathsf{Same} \leftarrow L$ ▷ Set the left children to be equal.
4: $s_c \leftarrow s_0^{\mathsf{Same}} \oplus s_1^{\mathsf{Same}}$
5: $t_c^L \leftarrow t_0^L \oplus t_1^L \oplus 1 \oplus x$ ▷ Ensure that the left control bits are not equal iff $x = 0$.
6: $t_c^R \leftarrow t_0^R \oplus t_1^R \oplus x$ ▷ Ensure that the right control bits are not equal iff $x = 1$.
7: $\mathsf{cw} \leftarrow s_c || t_c^L || t_c^R$
8: $s'_b \leftarrow \mathsf{correct}(s_b^{\mathsf{Diff}}, s_c, t_b^{(i-1)})$ for $b \in \{0,1\}$.
9: $t'_b \leftarrow \mathsf{correct}(t_b^{\mathsf{Diff}}, t_c^{\mathsf{Diff}}, t_b)$ for $b \in \{0,1\}$.

Output: $\mathsf{cw}, (s'_0, t'_0), (s'_1, t'_1)$

at the final level we replace the PRG with a hash function $\mathcal{H}$ sampled from a family 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 our 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 our 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 III.9 (XOR-Collision Resistance)**.** *We say a function family $\mathcal{F}$ such that $f \in \mathcal{F}$ is XOR-collision resistant if no adversary given a randomly sampled $f \in \mathcal{F}$ running in time less than $O(2^\lambda)$ with space less than $O(2^\lambda)$ 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 $2^{-\lambda}$.*

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. 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 defintion III.10. The full verifiable DPF construction is given in figure 1.

Definition III.10 (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 $s^{(0)} \in \{0,1\}^\lambda$.*
- *n correction words $\mathsf{cw}_1, \ldots, \mathsf{cw}_n$, as defined in definition III.6.*
- *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 III.11 (VerDPF Correctness)**.** *The* VerDPF *scheme defined in figure 1 defines a correct verifiable DPF scheme.*

Proof. See appendix B. □

C. 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 III.12 (Detection of Malicious Function Shares)**.** *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. Let $(\mathsf{y}_b, \pi_b) \leftarrow \mathsf{VerDPF.FDEval}(b, \mathsf{k}_b^*)$ such that $\mathsf{Accept} \leftarrow \mathsf{VerDPF.Verify}(\pi_0, \pi_1)$ passes but $\mathsf{y}_0 + \mathsf{y}_1$ is not the truth table of a point function.*

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 $\{s_0^{(i)}\}_{i=1}^N$ and $\{s_1^{(i)}\}_{i=1}^N$. The servers also have the *same* correction seed cs. Let $\tilde{\pi}_b^{(i)} \leftarrow \mathsf{H}(i || s_b^{(i)})$, let $t_b^{(i)} \leftarrow \mathsf{LSB}(s_b^{(i)})$, and let

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.

We now give the share generation and and evaluation algorithm for our batched DPF scheme. The VerDPF.Verify algorithm simply checks if the two input proofs are equal.

VerDPF.Gen

Input: Security parameter 1^λ
Point function $f_{\alpha,\beta}\colon \{0,1\}^n \to \mathbb{G}$

Sample $s_0^{(0)} \leftarrow \{0,1\}^\lambda$ and $s_1^{(0)} \leftarrow \{0,1\}^\lambda$.
Set $t_0^{(0)} = 0$ and $t_1^{(0)} = 1$.
Let $\alpha_1, \dots \alpha_n$ be the bits of α.
for i from 1 to n **do**
  $\mathsf{vals} \leftarrow \mathsf{CWGen}(\mathcal{G}, s_0^{(i-1)}, t_0^{(i-1)}, s_1^{(i-1)}, t_1^{(i-1)}, \alpha_i)$
  Parse $\mathsf{cw}_i, (s_0^{(i)}, t_0^{(i)}), (s_1^{(i)}, t_1^{(i)}) \leftarrow \mathsf{vals}$
$\tilde{\pi}_b \leftarrow \mathsf{H}(\alpha || s_b^{(n)})$ for $b \in \{0,1\}$.
$\mathsf{cs} \leftarrow \tilde{\pi}_0 \oplus \tilde{\pi}_1$.
$s_b^{(n+1)} \leftarrow s_b^{(n)}$ ▷ True output always extends to left child.
$t_b^{(n+1)} \leftarrow \mathsf{LSB}(s_b^{(n+1)})$
if $t_0^{(n+1)} = t_1^{(n+1)}$ **then** `goto 1`
Compute output correction word:

$$\mathsf{ocw} \leftarrow (-1)^{t_1^{(n+1)}}[\beta - \mathsf{convert}(s_0^{(n+1)}) + \mathsf{convert}(s_1^{(n+1)})]$$

Set $\mathsf{k}_b \leftarrow (s_b^{(0)}, \{\mathsf{cw}_i\}_{i=1}^n, \mathsf{csocw})$ for $b \in \{0,1\}$

Output: $(\mathsf{k}_0, \mathsf{k}_1)$

VerDPF.FDEval

Input: $b \in \{0,1\}$ and VerDPF key 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 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: (y, π)

Fig. 1: Verifiable Distributed Point Function $\mathsf{VerDPF}_{n,\mathbb{G}}$. For brevity, VerDPF.Eval is omitted, but the algorithm is clear from the FDEval definition: simply follow one path of the GGM tree and skip the proof generation.

$\pi_b^{(i)} \leftarrow \mathsf{correct}(\tilde{\pi}_b^{(i)}, \mathsf{cs}, t_b^{(i)})$. The bulk of this proof is covered by the following lemma.

Lemma III.13. *Suppose there exists two indices $i^*, j^* \in [N]$ such that $s_0^{(i^*)} \neq s_1^{(i^*)}$ and $s_0^{(j^*)} \neq s_1^{(j^*)}$. 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 $i \in [N]$ we will have $\pi_0^{(i)} = \pi_1^{(i)}$.*

Proof. Suppose for contradiction that there exists two indices $i^*, j^* \in [N]$ such that $s_0^{(i^*)} \neq s_1^{(i^*)}$ and $s_0^{(j^*)} \neq s_1^{(j^*)}$ and for all $i \in [N]$ we will have $\pi_0^{(i)} = \pi_1^{(i)}$. By collision-resistance, we have that $\tilde{\pi}_0^{(i^*)} \neq \tilde{\pi}_1^{(i^*)}$ and $\tilde{\pi}_0^{(i^*)} \neq \tilde{\pi}_1^{(i^*)}$. In order to get $\pi_0^{(i^*)} = \pi_1^{(i^*)}$ and $\pi_0^{(i^*)} = \pi_1^{(i^*)}$, we need the following:

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

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

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

Therefore, no value of cs will result in $\pi_0^{(i)} = \pi_1^{(i)}$ for all $i \in [N]$. □

From the collision resistance of H', we have that if the proofs produced by the FDEval algorithm match, then $\pi_0^{(i)} = \pi_1^{(i)}$ for all $i \in [N]$. From lemma III.13, we have that this implies that there is at most one $i^* \in [N]$ such that $s_0^{(i^*)} \neq s_1^{(i^*)}$, and for all $i \in [N]$ such that $i \neq i^*$, we have $s_0^{(i)} = s_1^{(i)}$.

Define $\alpha = i^*$ for the unique i^* such that $s_0^{(i^*)} \neq s_1^{(i^*)}$. If

no such i^* exists, set $i^* = 0$. Define

$$\begin{aligned}\beta =&\mathsf{correct}_{\mathbb{G}}\left(\mathsf{convert}(s_0^{(i^*)}), \mathsf{ocw}, t_0^{(i^*)}\right)\\ &- \mathsf{correct}_{\mathbb{G}}\left(\mathsf{convert}(s_1^{(i^*)}), \mathsf{ocw}, t_1^{(i^*)}\right)\end{aligned}$$

Note that this β is well-defined for any $s_b^{(i^*)}$ and $t_b^{(i^*)}$. Since $t_b^{(i)} = \mathsf{LSB}(s_b^{(i)})$, we have the following implication:

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

Since $s_0^{(i)} = s_1^{(i)}$ for all $i \in i^*$, $\mathsf{y}_0 + \mathsf{y}_1$ defines the truth table of $f_{\alpha,\beta}$. Therefore, the construction in figure 1 satisfies definition III.2. □

Lemma III.14 (VerDPF Function Privacy)**.** *The VDPF construction* VerDPF *satisfies defintion III.3.*

Proof. See appendix B. □

Combining lemmas III.12 and III.14, we get the proof of the following theorem.

Theorem III.15 (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 $\mathsf{VerDPF.FDEval}(b, \mathsf{k}_b)$ is $O(2^n\lambda)$.*

D. Verifiable Evaluation of a Domain Subset

We will now show that we can get a similar verifiability guarantee without evaluating the full VDPF domain. We observe that lemma III.13 generalizes to cases where we are only concerned with a subset of seeds in the final level. Therefore, we can apply the same verification technique to show that for any set of evaluation outputs of the VDPF, at most one will be nonzero. Let BVEval denote this batched verifiable evaluation. This algorithm is defined in algorithm 3. At a high level, this algorithm evaluates the verifiable DPF at L distinct points, then produces a proof that, if verified, guarantees that of the L output shares produced by the VDPF evaluation, at most one pair of shares will sum to a non-zero value. In other words, since any subset of evaluations of a point function can also be viewed as a truth table of a point function, our evaluation procedure works just as well on a subset of evaluations.

We now give the formal security statement for this algorithm.

Lemma III.16 (Detection of Malicious Function Shares for Batched Evaluation)**.** *No PPT adversary $\mathcal{A}$ can generate* VerDPF *keys $(\mathsf{k}_0^*, \mathsf{k}_1^*) \leftarrow \mathcal{A}(1^\lambda)$ along with $L \geq 1$ distinct evaluation points $x_1, \ldots, x_L \in \{0,1\}^n$ 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. 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 not the truth table of a point function.*

Algorithm 3 VerDPF.BVEval. The hash functions H and H′ are as in figure 1

Input: $b \in \{0,1\}$ and VerDPF key k_b.
Set of L distinct evaluation points $x_1, \ldots, x_L$.

Parse the VerDPF key $(s^{(0)}, \{\mathsf{cw}_i\}_{i=1}^n, \mathsf{cs}, \mathsf{ocw}) \leftarrow \mathsf{k}_b$.
Define $\mathsf{y} \leftarrow \{\}$ and $\pi \leftarrow \mathsf{cs}$
for ℓ from 1 to L **do**
  Let $s \leftarrow s^{(0)}$ and $t \leftarrow b$
  Let $\beta_1 = \mathsf{MSB}(x_\ell), \ldots, \beta_n$ be the bits of x_ℓ
  for i from 1 to n **do**
    $(s'_0, t'_0), (s'_1, t'_1) \leftarrow \mathsf{NodeExpand}(\mathcal{G}, s, t)$
    if $\beta_i = 0$ **then** $(s, t) \leftarrow (s'_0, t'_0)$
    else $(s, t) \leftarrow (s'_1, t'_1)$
  $\tilde{\pi} \leftarrow \mathsf{H}(x_i || s)$
  $t \leftarrow \mathsf{LSB}(s)$
  $\mathsf{y}.\mathsf{append}\left((-1)^b \cdot \mathsf{correct}_{\mathbb{G}}(\mathsf{convert}(s), \mathsf{ocw}, t)\right)$
  $\pi \leftarrow \pi \oplus \mathsf{H}'(\pi \oplus \mathsf{correct}(\tilde{\pi}, \mathsf{cs}, t))$

Output: (y, π)

Proof. This lemma follows directly from lemma III.13, since checking that the right leaves match ensure that at most one pair of left leaves differ. This one pair of differing leaves defines the nonzero point in the point function. This holds for any set of distinct evaluation points, even when the set is adversarially chosen. □

IV. 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 III. Our construction is based on a novel Cuckoo-hashing scheme described below.

A. Cuckoo-hashing from PRPs

Our technique is inspired by the use of Cuckoo-hashing schemes that are common throughout the PSI [6], [10] and DPF [21] 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 $\mathsf{CHCompact}$, is given in algorithm 4.

Algorithm 4 $\mathsf{CHCompact}$ 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 ω from 1 to t **do**
  Set $\beta \leftarrow \alpha_\omega$ and set $\mathsf{success} \leftarrow \mathsf{False}$
  while $\mathsf{success}$ is False **do**
    Sample $k \xleftarrow{\$} [\kappa]$
    $i \leftarrow h_k(\beta)$.
    if $\mathsf{Table}[i] = \perp$ **then**
      $\mathsf{Table}[i] = \beta$ and $\mathsf{success} \leftarrow \mathsf{True}$
    else Swap β and $\mathsf{Table}[i]$

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 t elements. More specifically, from the empirical analysis of Demmler et al. [10], we have the following lemma.

Lemma IV.1 (Cuckcoo-hashing Failure Probability [10])**.** *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*

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

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 IV.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 IV.1 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 n elements and produces a matrix of dimension $m \times B$ that contains $\kappa \cdot n$ elements. This mode is produced by hashing all n elements with each of the κ hash functions, then inserting each of the n 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 $A_{i,j} := \mathsf{PRP}(\sigma, i \cdot m + j)$. 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:

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

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:

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

With these functions $\mathsf{index}_1, \ldots, \mathsf{index}_\kappa$ in addition to the hash functions $h_1, \ldots, h_\kappa$, 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 $(i, j)_k = (h_k(x), \mathsf{index}_k(x))$.

B. 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 $\alpha_1, \ldots, \alpha_t$

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. 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 $\alpha_1, \ldots, \alpha_t$ 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 $(\mathsf{k}_0^{(i)}, \mathsf{k}_1^{(i)}) \leftarrow \mathsf{VerDPF.Gen}(1^\lambda, g_{0,0})$. The verifiable distributed MPF (VDMPF) share has the form $\mathsf{mpk}_b = (\sigma, \mathsf{k}_b^{(1)}, \ldots, \mathsf{k}_b^{(m)})$ 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 $i_1, \ldots, i_k$ evaluated at $j_1, \ldots, j_k$. This gives the output

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

In addition, this VDMPF inherits all of the features of the VDPF construction from section III, 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 each bucket VDPF. 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 IV-A, we will essentially always have $m \leq 2t$ (see remark IV.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 III. We leave for future work the challenge of closing this gap while maintaining similar performance.

The proofs of the following lemmas are in appendix C.

Lemma IV.2 (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 $\mathcal{F}$.*

Lemma IV.3 (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 II.2*

Lemma IV.4 (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.*

Lemmas IV.2, IV.3, and IV.4 combine to give the following theorem.

Theorem IV.5. *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* VerDMPF.Gen *is $O(m\lambda\log(N/m))$. For η inputs, the runtime of* 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 IV.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 VerDMPF.Gen time of $O(m\lambda\log(N))$ and a VerDMPF.BVEval time of $O(\eta\lambda\log(N))$.

In appendix C-A, 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 used in the PSI construction in appendix E.

V. Malicious Two-Server PIR Protocol

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

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 V.1 (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* dbState*.*
- $\mathsf{Accept/Reject} \leftarrow \mathsf{PIR.QueryVerify}(\pi_0, \pi_1)$
  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 D-A.

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 III. The construction is given in figure 3.

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 D-B.

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

VI. 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 & Experimental Details: We implemented[1] our VDPF and VDMPF constructions in C++. We follows the approach of Wang et al. [23] by using a fixed-key AES cipher to construct a Matyas-Meyer-Oseas [18] 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. [4] and Wang et al. [23], 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.

We note again that we did not comapre against the verifiable constructions of Boyle et al. because of the incomparable constraints placed on the DPF by their techniques, namely that β must be of specific values.

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 evaluation. The runtime reported is then the time per evaluation point. Benchmarks are given in figure 5.

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 slower, 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.

[1]For anonymity, we omit the link to our code. We intend to include the link in the published version.

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.
We now give the share generation and and evaluation algorithm for our VDMPF scheme. As with the VerDPF scheme, the VerDMPF.Verify algorithm simply checks that the two input proofs are equal.

VerDMPF.Gen

Input: Security parameter 1^λ
t point functions $\{f_{\alpha_i,\beta_i}\}_{i=1}^t$

$m \leftarrow \mathsf{CHBucket}(t, 3, \lambda)$, where $\kappa = 3$.
Sample a random PRP seed $\sigma \leftarrow \{0,1\}^\lambda$.
Let $B \leftarrow \lceil N\kappa/m \rceil$.
From σ, m, B, define $h_1, \ldots, h_\kappa\colon [N\kappa] \to [m]$ as in eq. 1 and $\mathsf{index}_1, \ldots, \mathsf{index}_\kappa\colon [N\kappa] \to B$ in eq. 2.
$\mathsf{Table} \leftarrow \mathsf{CHCompact}\left(\{\alpha_i\}_{i=1}^t, \{h_k\}_{k=1}^\kappa, m\right)$
Let $n' = \lceil \log(B) \rceil$ and let $\mathsf{VerDPF} := \mathsf{VerDPF}_{n',\mathbb{G}}$.
Let $\mathsf{k}_0 \leftarrow \{\sigma\}$ and $\mathsf{k}_1 \leftarrow \{\sigma\}$
for i from 1 to m **do**
  if $\mathsf{Table}[i] = \perp$ **then** Define $\alpha' \leftarrow 0$ and $\beta' \leftarrow 0$
  else
    Let $\alpha_j = \mathsf{Table}[i]$, for $j \in [t]$
    Let $k \in [\kappa]$ be such that $h_k(\alpha_j) = i$.
    Let $\alpha' \leftarrow \mathsf{index}_k(\alpha_j)$ and $\beta' \leftarrow \beta_j$
  Define the point function $f := f_{\alpha',\beta'}$
  $(\mathsf{k}_0^{(i)}, \mathsf{k}_1^{(i)}) \leftarrow \mathsf{VerDPF.Gen}(1^\lambda, f)$
  Append $\mathsf{k}_0^{(i)}$ to k_0 and $\mathsf{k}_1^{(i)}$ to k_1.

Output: $(\mathsf{k}_0, \mathsf{k}_1)$

VerDMPF.BVEval

Input: Bit b and VerDMPF key k_b
η 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 inputs of length m.
for ω from 1 to η **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, ω) to $\mathsf{inputs}[i_k]$ for each $k \in [\kappa]$, ignoring duplicates.
Initialize an array $\mathsf{outputs}$ of length η to all zeros.
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.

Overall, our comparisons show that our technique 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 4.

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 $2\times$ 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 duo 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. These benchmarks are given in figure 5.

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 IV, 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 verifiability proof, even for a small number of nonzero points (e.g. 10 points). These results are displayed in figure 6.

Two-server MPC: We do not present explicit benchmarks for our two-server PIR and two-server PSI constructions since they are direct applications of the VDPF and VDMPF. Their performance is dominated by the underlying FSS.

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.

$\mathsf{PIR.Setup}(1^\lambda, N, \mathbb{G})$

1: 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	**Server Algorithms**
$\mathsf{PIR.QueryGen}(1^\lambda, i)$ 1: Sample $r \xleftarrow{\$} \mathbb{G}$ 2: Define the point function $f_{i,r}$. 3: $(\mathsf{k}_0, \mathsf{k}_1) \leftarrow \mathsf{VerDPF.Gen}\left(1^\lambda, f_{i,r}\right)$ **Output:** $(\mathsf{q}_0, \mathsf{q}_1, \mathsf{state}) \leftarrow (\mathsf{k}_0, \mathsf{k}_1, r)$	$\mathsf{PIR.QueryEval}(\mathsf{q}_b, \mathcal{D}, \mathsf{dbState})$ 1: $(\mathsf{y}_b, \pi_b) \leftarrow \mathsf{VerDPF.FDEval}(b, \mathsf{q}_b)$ 2: $z \leftarrow \langle \mathsf{y}_b, \mathcal{D} \rangle$ ▷ Inner product. 3: $(g, s') \leftarrow \mathcal{G}(\mathsf{dbState})$ ▷ dbState is just a PRG seed. 4: **if** $b = 0$ **then** $z \leftarrow z + g$ 5: **else** $z \leftarrow z - g$ **Output:** $(\mathsf{res}_b, \pi_b, \mathsf{dbState}') \leftarrow (z, \pi_b, s')$
$\mathsf{PIR.Reconstruct}(\mathsf{res}_0, \mathsf{res}_1, \mathsf{state})$ 1: $g \leftarrow \mathsf{res}_0 + \mathsf{res}_1$. 2: **if** $g = \mathsf{state}$ **then** $\omega \leftarrow 1$ 3: **else if** $g = 0$ **then** $\omega \leftarrow 0$ 4: **else** $\omega \leftarrow \perp$ **Output:** ω	$\mathsf{PIR.QueryVerify}(\pi_0, \pi_1)$ 1: **if** $\pi_0 = \pi_1$ **then** $\nu \leftarrow \mathsf{Accept}$ 2: **else** $\nu \leftarrow \mathsf{Reject}$ **Output:** ν

Fig. 3: Our PIR construction.

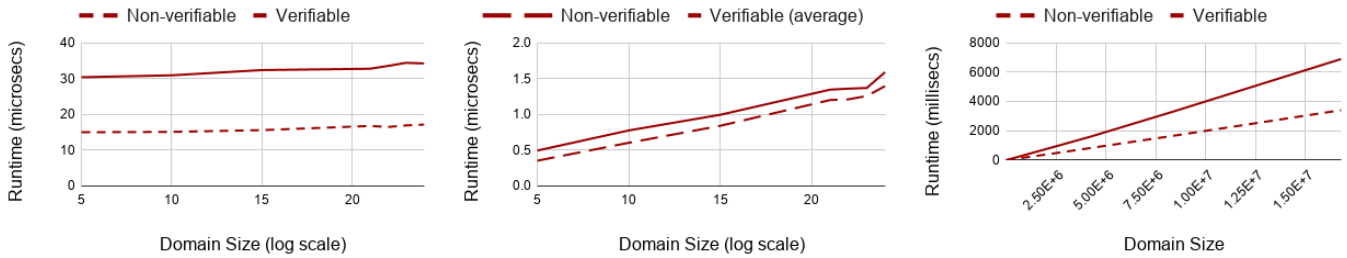

Fig. 4: In this figure, we present the benchmarks of the textbook DPF and the verifiable DPF presented in this work. The left graph plots runtimes for the share generation time. As can be seen, the slowdown for verifiability is roughly $2\times$. The middle graph plots the runtimes for the share evaluation. As discussed in section VI, the verifiable runtime was computed by taking the runtime of the verifiable batch evaluation procedure (algorithm 3) for 100 random points and dividing it by 100. The right graph plots the runtimes for the full domain evaluation operation.

Acknowledgments

We would like to thank Vinod Vaikuntanathan for his helpful conversations and insights.

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

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.

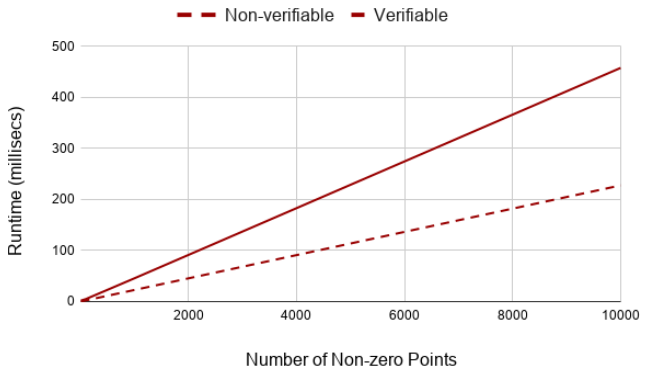

Fig. 5: This graph plots the share generation time for the textbook DMPF and the batched, verifiable DMPF presented in this work.

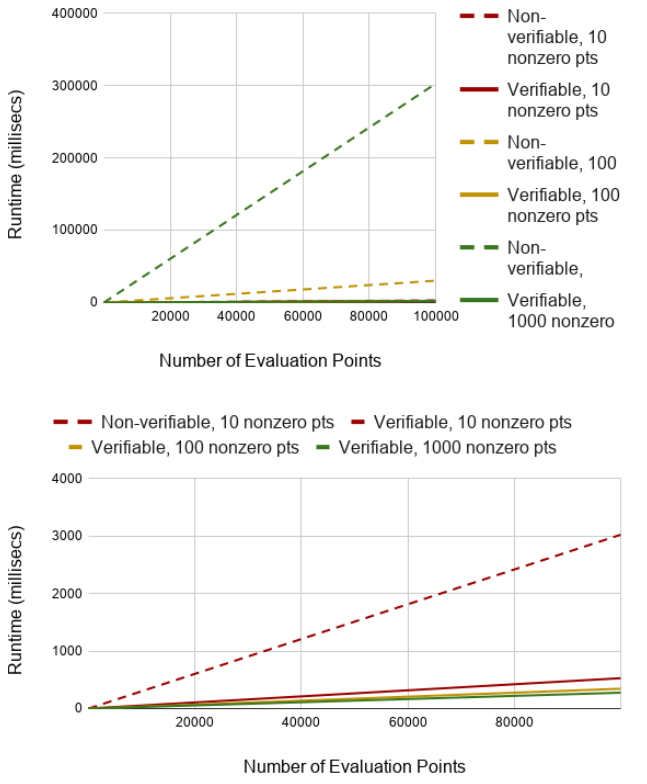

Fig. 6: This figure plots the evaluation times for the textbook DMPF and the batched, verifiable DMPF presented in this work. Both graphs in this figure plot the same data; the top graph shows all plots while the bottom graph is only a plot of the smallest four lines in so that the batched VDMPF runtimes 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 bottom graph that the evaluation time decreases as the number of nonzero points on in the MPF grows.

[2] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, Peter Rindal, and Peter Scholl. Efficient two-round ot extension and silent non-interactive secure computation. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 291–308, New York, NY, USA, 2019. Association for Computing Machinery.

[3] 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.

[4] 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.

[5] Nishanth Chandran, Nishka Dasgupta, Divya Gupta, Sai Lakshmi Bhavana Obbattu, Sruthi Sekar, and Akash Shah. Efficient linear multiparty psi and extensions to circuit/quorum psi. Cryptology ePrint Archive, Report 2021/172, 2021. https://eprint.iacr.org/2021/172.

[6] 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.

[7] Jung Cheon, Stanislaw Jarecki, and Jae Hong Seo. Multi-party privacy-preserving set intersection with quasi-linear complexity. *IACR Cryptology ePrint Archive*, 2010:512, 01 2010.

[8] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, FOCS '95, page 41, USA, 1995. IEEE Computer Society.

[9] Henry Corrigan-Gibbs and Dmitry Kogan. Private information retrieval with sublinear online time. In Anne Canteaut and Yuval Ishai, editors, *Advances in Cryptology – EUROCRYPT 2020*, pages 44–75, Cham, 2020. Springer International Publishing.

[10] 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.

[11] 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.

[12] 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.

[13] Aner Ben Efraim, Olga Nissenbaum, Eran Omri, and Anat Paskin-Cherniavsky. Psimple: Practical multiparty maliciously-secure private set intersection. Cryptology ePrint Archive, Report 2021/122, 2021. https://eprint.iacr.org/2021/122.

[14] Michael J. Freedman, Kobbi Nissim, and Benny Pinkas. Efficient private matching and set intersection. In Christian Cachin and Jan L. Camenisch, editors, *Advances in Cryptology - EUROCRYPT 2004*, pages 1–19, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

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

[16] Carmit Hazay and Muthuramakrishnan Venkitasubramaniam. Scalable multi-party private set-intersection. In Serge Fehr, editor, *Public-Key Cryptography – PKC 2017*, pages 175–203, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg.

[17] Vladimir Kolesnikov, Ranjit Kumaresan, Mike Rosulek, and Ni Trieu. Efficient batched oblivious prf with applications to private set intersection. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 818–829, New York, NY, USA, 2016. Association for Computing Machinery.

[18] 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.

[19] Yingpeng Sang and Hong Shen. Privacy preserving set intersection based on bilinear groups. In *Proceedings of the Thirty-First Australasian Conference on Computer Science - Volume 74*, ACSC '08, page 47–54, AUS, 2008. Australian Computer Society, Inc.

[20] Yingpeng Sang and Hong Shen. Privacy preserving set intersection protocol secure against malicious behaviors. pages 461 – 468, 01 2008.

Two-Server PIR.: This protocol is between three parties: a client $\mathcal{C}$ and two servers $\mathcal{S}_0$ and $\mathcal{S}_1$. The servers each begin with identical copies of a database $\mathcal{D}$ of size N, and the client begins with an index $i \in [N]$. At the end of the protocol, the client should get the i^{th} entry $\mathcal{D}[i]$ of the database, and the servers should get nothing. This functionality is detailed below.

1) Receive $\mathcal{D}^{(0)}$ from $\mathcal{S}_0$ and $\mathcal{D}^{(1)}$ from $\mathcal{S}_1$. If $\mathcal{D}^{(0)} \neq \mathcal{D}^{(1)}$ or $|\mathcal{D}^{(0)}| \neq N$, abort. Let $\mathcal{D} \leftarrow \mathcal{D}^{(0)}$.
2) Receive an index i from the client. If $i \notin [N]$, abort.
3) Sent the database element $\mathcal{D}[i]$.

Fig. 7: Two-Server PIR Ideal Functionality

Two-Server PSI.: This protocol is between three parties: a client $\mathcal{C}$ and two servers $\mathcal{S}_0$ and $\mathcal{S}_1$. The protocol is parametrized by a domain $\mathcal{D}$ of size N. The client begins with as set $X \subset \mathcal{D}$, and the servers begin with identical copies of a subset $Y \subset \mathcal{D}$. At the end of the protocol, the servers will receive nothing, and the client receives $X \cap Y$. The functionality is detailed below.

1) Receive $Y^{(0)}$ from $\mathcal{S}_0$ and $Y^{(1)}$ from $\mathcal{S}_1$. If $Y^{(0)} \neq Y^{(1)}$ or $Y^{(0)} \not\subset \mathcal{D}$, abort. Let $Y \leftarrow Y^{(0)}$.
2) Receive X from the client. If $X \not\subset \mathcal{D}$, abort.
3) Sent the database elements $X \cap Y$.

Fig. 8: Two-Server PSI Ideal Functionality

[21] 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.

[22] Elaine Shi, Waqar Aqeel, Balakrishnan Chandrasekaran, and Bruce Maggs. Puncturable pseudorandom sets and private information retrieval with polylogarithmic bandwidth and sublinear time. Cryptology ePrint Archive, Report 2020/1592, 2020. https://eprint.iacr.org/2020/1592.

[23] 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.

Appendix A
PIR & PSI Background

Private information retrieval was first introduced by Chor et al. [8]. We give the ideal PIR functionality in figure 7 and the ideal PSI functionality in figure 8.

Appendix B
Verifiable DPF Proofs

In this section, we give the proofs for some lemmas from section III. These lemmas are restated for convenience.

Lemma B.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. [4]. 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 correct operation is deterministic, the nodes 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 $t_0 \neq t_1$. □

Lemma B.2 (VerDPF Function Privacy)**.** *The VDPF construction* VerDPF *satisfies defintion III.3.*

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 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 compute the correct proof π_{1-b}, since $\pi_{1-b} = \pi_b$. Therefore, the simulator Sim can set all elements of the key k^* to be randomly sampled elements, then compute $(\_, \pi*) \leftarrow \mathsf{VerDPF.FDEval}(\mathsf{k}^*)$ to output $(\mathsf{k}^*, \pi^*) \approx_c (\mathsf{k}_b, \pi_{1-b})$. □

Appendix C
Verifiable DMPF Proofs

In this section, we give proofs of lemmas that are omitted in section IV. The lemmas are restated for convenience.

Lemma C.1 (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 from the low statistical failure probability of the Cuckoo-hashing scheme and the correctness of the VerDPF for each Cuckoo-hashing bucket. We consider correctness only when the share generation succeeds, which occurs with overwhelming probability. Therefore, this scheme achieves perfect correctness, and the case where share generation fails is handled in the function privacy proof. □

Lemma C.2 (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 II.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 IV.1 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 C.3 (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. □

A. Match-Mode VDMPF: Point Matching

In this section, we present an alternative evaluation mode for our VDMPF scheme that will be useful in constructing protocols below. 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 will produce one output for each of the cuckoo-hash buckets in the VDMPF key. The purpose of this evaluation mode will be to determine if one of the server's input elements match 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.

Lemma C.4 (VerDMPF Match-Mode Evaluation Integrity)**.** *For a VDMPF scheme* $\mathsf{VerDMPF} := \mathsf{VerDMPF}_{N,\mathbb{G}}$*, algorithm 5 defines an evaluation algorithm that takes time $O(m\lambda \log(N/m))$ such that no PPT adversary $\mathcal{A}$ can produce* VerDMPF *shares* $\mathsf{k}_0^*, \mathsf{k}_1^*$ *along with η evaluation points $x_1, \ldots, x_\eta$ such that the following holds. Let* $(\mathsf{y}_b, \pi_b) \leftarrow \mathsf{VerDMPF.MatchEval}(b, \mathsf{k}_b^*, \{x_i\}_{i=1}^L)$ *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 point-matching evaluation contains more than m non-zero outputs.*

Algorithm 5 VDMPF Match-Mode Evaluation, denoted VerDMPF.MatchEval. The setting for this algorithm is the same as the VDMPF construction in figure 2.

Input: bit b and VerDMPF key k_b
η 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 inputs of length m.
for ω from 1 to η **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 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: outputs, π

APPENDIX D
PIR SECURITY AND PRIVACY

A. PIR Security and Privacy Definitions

We will now define security for this PIR protocol.

Definition D.1 (PIR Privacy Against a Malicious Server)**.** *Let* $\mathsf{PIR} := \mathsf{PIR}_{N,\mathbb{G}}$ *be a PIR protocol as defined in definition V.1.*

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

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.*

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

Definition D.2 (PIR Security Against a Malicious Server)**.** *Let* $\mathsf{PIR} := \mathsf{PIR}_{N,\mathbb{G}}$ *be a PIR protocol as defined in definition V.1. 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* $\{\mathcal{D}[i], \perp\}$*. In other words, if* PIR.Reconstruct *does not output* $\perp$*, then the output should be* $x = \mathcal{D}[i]$*, where* $\mathcal{D}$ *is the honest server's copy of the database.*

Definition D.3 (PIR Security Against a Malicious Client)**.** *Let* $\mathsf{PIR} := \mathsf{PIR}_{N,\mathbb{G}}$ *be a PIR protocol as defined in definition V.1. 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* $b \in \{0, 1\}$*.*

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

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.*

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

B. PIR Security and Privacy Proofs

We now argue the security of the PIR protocol in figure 3.

Lemma D.4 (PIR Privacy Against a Malicious Server)**.** *The PIR construction given in figure 3 satisfies definition D.1.*

Proof. This lemma follows from the function hiding of the underlying VDPF scheme, shown in lemma III.14. □

Lemma D.5 (PIR Security Against a Malicious Server)**.** *The PIR construction given in figure 3 satisfies definition D.2.*

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 $\perp$), 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 D.6 (PIR Security Against a Malicious Client)**.** *The PIR construction given in figure 3 satisfies definition D.3.*

Proof. This lemma follows from the verifiability of the DPF. From lemma III.12, 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. □

APPENDIX E
MALICIOUS TWO-SERVER PSI PROTOCOL

We will now define the API for our protocol.

Definition E.1 (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 E-A.

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 IV. The construction is

given in figure 9. The proofs of security are given in appendix E-B.

A. PSI Security and Privacy Definitions

Definition E.2 (PSI Privacy Against a Malicious Server)**.** *Let* $\mathsf{PSI} := \mathsf{PSI}_{N,\mathbb{G}}$ *be a PSI protocol as defined in definition E.1.*

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

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 E.3 (PSI Security Against a Malicious Server)**.** *Let* $\mathsf{PSI} := \mathsf{PSI}_{N,\mathbb{G}}$ *be a PSI protocol as defined in definition E.1. 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* 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 E.4 (PSI Security Against a Malicious Client)**.** *Let* $\mathsf{PSI} := \mathsf{PSI}_{N,\mathbb{G}}$ *be a PSI protocol as defined in definition E.1. 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.*

B. PSI Security & Privacy Proofs

Lemma E.5 (PSI Privacy Against a Malicious Server)**.** *The PSI construction given in figure 9 satisfies definition E.2.*

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

Lemma E.6 (PSI Security Against a Malicious Server)**.** *The PSI construction given in figure 9 satisfies definition E.3.*

Proof. This follows from a parallel argument to lemma D.5; namely, it follows from the size of the output group $\mathbb{G}$. □

Lemma E.7 (PSI Security Against a Malicious Client)**.** *The PSI construction given in figure 9 satisfies definition E.4.*

Proof. This follows from the verifiability of the VDMPF. □

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

$\mathsf{PSI.QueryGen}(1^\lambda, X)$

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

Output: $(\mathsf{q}_0, \mathsf{q}_1, \mathsf{state}) \leftarrow (\mathsf{k}_0, \mathsf{k}_1, \{(x_i, r_i)\}_{i=1}^t)$

$\mathsf{PSI.Reconstruct}(\mathsf{res}_0, \mathsf{res}_1, \mathsf{state})$

1: Parse $(x_1, r_1), \ldots, (x_t, r_t) \leftarrow \mathsf{state}$
2: Initialize $Z \leftarrow \emptyset$
3: **for** (z_1, z_2) in $(\mathsf{res}_0, \mathsf{res}_1)$ **do**
4: $\quad z \leftarrow z_0 + z_1$.
5: $\quad$ **if** $z = r_i$ **then** Insert x_i in Z

Output: Z

Server Algorithms

$\mathsf{PSI.Setup}(1^\lambda, N, \mathbb{G})$

1: Sample a random seed s for $\mathcal{G}$.

Output: $\mathsf{dbState} \leftarrow s$

$\mathsf{PSI.QueryEval}(\mathsf{q}_b, Y, \mathsf{dbState})$

1: $(\mathsf{y}_b, \pi_b) \leftarrow \mathsf{VerDMPF.MatchEval}(b, \mathsf{q}_b, Y)$ ▷ Algorithm 5
2: **for** $\mathsf{y}_b^{(i)}$ in y_b **do**
3: $\quad (g, \mathsf{dbState}) \leftarrow \mathcal{G}(\mathsf{dbState})$
4: $\quad$ **if** $b = 0$ **then** $\mathsf{y}_b^{(i)} \leftarrow \mathsf{y}_b^{(i)} + g$
5: $\quad$ **else** $\mathsf{y}_b^{(i)} \leftarrow \mathsf{y}_b^{(i)} - g$

Output: $(\mathsf{res}_b, \pi_b, \mathsf{dbState}) \leftarrow (\mathsf{y}_b, \pi_b, \mathsf{dbState})$

$\mathsf{PSI.QueryVerify}(\pi_0, \pi_1)$

1: **if** $\pi_0 = \pi_1$ **then** $\nu \leftarrow \mathsf{Accept}$
2: **else** $\nu \leftarrow \mathsf{Reject}$

Output: ν

Fig. 9: Our PSI construction.