Distributed Vector-OLE: Improved Constructions and Implementation

Phillipp Schoppmann
Humboldt-Universität zu Berlin
schoppmann@informatik.hu-berlin.de

Adrià Gascón
Google
adriagascon@gmail.com

Leonie Reichert
Humboldt-Universität zu Berlin
leonie.reichert@hu-berlin.de

Mariana Raykova
Google
mpr2111@columbia.edu

ABSTRACT

We investigate concretely efficient protocols for distributed oblivious linear evaluation over vectors (Vector-OLE). Boyle et al. (CCS 2018) proposed a protocol for secure distributed pseudorandom Vector-OLE generation using *sublinear* communication, but they did not provide an implementation. Their construction is based on a variant of the LPN assumption and assumes a distributed key generation protocol for single-point Function Secret Sharing (FSS), as well as an efficient batching scheme to obtain multi-point FSS. We show that this requirement can be relaxed, resulting in a weaker variant of FSS, for which we give an efficient protocol. This allows us to use efficient probabilistic batch codes that were also recently used for batched PIR by Angel et al. (S&P 2018). We construct a full Vector-OLE generator from our protocols, and compare it experimentally with alternative approaches. Our implementation parallelizes very well, and has low communication overhead in practice. For generating a VOLE of size 2^{20}, our implementation only takes 0.52s on 32 cores.

1 INTRODUCTION

The ability to distribute correlated randomness between two parties in the absence of a trusted dealer is a central problem to cryptography. In the context of secure computation this ability enables splitting the computation in an offline phase which is input independent and can be executed in advance, and an online phase which is very efficient. Many previous works have focused on improving and optimizing methods for generation of correlated randomness in the context of oblivious transfer extension [3, 4, 6, 35], which provides offline precomputation for two party computation based on garbled circuits [56], Beaver multiplicative triples [5, 18, 20, 38, 39], which are at the core of offline computation for secure arithmetic computation, as well as oblivious linear evaluation (OLE) [9, 24, 25, 46], which can be viewed as the equivalent of OT extension in arithmetic setting and which can be used for multiplicative triple generation.

In this paper we focus on vector OLE (VOLE), which is the vectorized variant of an oblivious linear evaluation. More concretely, one party, the sender, holds vectors $\mathbf{u}, \mathbf{v}$ and a second party, the receiver has value x. The goal of the protocol is to enable the receiver to learn $\mathbf{w} = \mathbf{u}x + \mathbf{v}$ without revealing any further information to any of the parties. The concept of VOLE was introduced by Applebaum et al. [2]. In the recent work of Boyle et al. [9], the authors showed that VOLE is implied by a pseudorandom variant of the protocol where the vectors $\mathbf{u}, \mathbf{v}$ are pseudorandom and are generated during the execution of the protocol as outputs to the first party.

Succinctness is a crucial property for correlated randomness protocols, which aim to distribute long correlated outputs between the parties by communicating only short seeds. Boyle et al. [9] showed how to achieve succinctness in the setting of pseudorandom VOLE. The idea of their approach is to use a random linear code to extend short (sub-linear) seed vectors to long pseudorandom vectors. By masking the encoded vectors with a shared, sparse noise vector, they reduce security of their VOLE generator to the LPN assumption [7]. The authors leverage functions secret sharing (FSS) for multi-point functions as a way to distribute the LPN noise vector to the inputs of the parties in an oblivious manner with succinct communication. This requires a two-party computation protocol for distributed generation of the FSS keys for the underlying multi-point function. The proposed approach reduces the multi-point FSS to several executions of single point FSS by leveraging batching techniques. For distributed single-point FSS key generation, the authors suggest using the two party FSS key generation protocol of Doerner and shelat [23].

In this work we address pseudorandom VOLE generation from a *practical* perspective. In particular, we focus on the *primal* variant of the protocol proposed in [9]. This is because to the best of our knowledge, there is no practical (i.e., implemented) construction of the "LPN-friendly" codes required for the *dual* variant. While the primal variant has a lower bound of $\tilde{O}(\sqrt{n})$ on its communication overhead, our implementation is still very efficient in practice. This is due to several improvements we make to the construction in [9]. Our main observation is that the non-zero *indices* of the shared sparse noise vector needed for LPN are part of the output to one of the parties. We use this observation in two ways. First, it allows us to use a more efficient batching scheme than what is proposed in [9]. Similar to previous work [1, 21], we use cuckoo hashing to do *probabilistic batching*. This allows us to split up an instance of t-point FSS into $m = O(t)$ instances of single-point FSS, where m is in practice only slightly larger than t. Second, we modify the FSS

CCS '19, November 11–15, 2019, London, United Kingdom

ACM ISBN 978-1-4503-6747-9/19/11…$15.00
https://doi.org/10.1145/3319535.3363228

construction itself, which gives us a large constant-factor speedup in each FSS generation and evaluation. Our protocol is constant round, does not require secure PRG evaluations, has sub-linear communication, and like the distributed construction proposed by [9], provides security in the semi-honest model.

Our VOLE construction implies efficiency improvements in a wide range of applications such as secure linear algebra, sparse matrix multiplications and machine learning computations over sparse data, oblivious polynomial evaluation and private set intersection, and improved efficiency for semi-private data accessed in some ORAM constructions.

1.1 Contributions

As building blocks for our distributed VOLE protocol we develop constructions of several primitives of independent interest.

A protocol for $(n-1)$-out-of-n Random OT (Section 3). An important component of our solution is a novel protocol for $(n-1)$-out-of-n Random OT that requires one round and logarithmic communication. While any m-out-of-n OT protocol requires communication $\Omega(m)$, we show how to leverage the fact that all messages are random to compress $n-1$ messages in a logarithmic number of seeds in manner oblivious to the sender. In terms of computation, an execution involves (i) $2n$ *local* PRG evaluations per party, and (ii) $\log_k(n)$ *parallel* executions of an $(k-1)$-out-of-k OT protocol. In our implementation we choose $k=2$, and thus rely on 1-out-of-2 OT. Our $(n-1)$-out-of-n Random OT implies a construction private puncturable PRF [8], which enables a party to obtain a punctured PRF key at a location that remains secret to the full PRF key owner.

Known-Index SPFSS (Section 4). The VOLE generation in [9] uses SPFSS but assumes that one of the parties knows the input that evaluate to non-zero in the point function, while the function value us secret-shared. We propose a protocol for known-index SPFSS that outperforms the alternatives proposed in [9]. The protocol uses a reduction to $(n-1)$-out-of-n Random OT, and thus leverages the protocol mentioned above. While known-index SPFSS implies distributed VOLE, its relevance is not limited to this application. It can be also viewed as a type of "scatter" vector operation, which is a core component in secure protocols for machine learning tasks [51]. Our protocol outperforms the solutions presented in [51], resulting in immediate gains for tasks such as gradient descent model training on sparse data.

Efficient Known-Indices MPFSS from SPFSS (Section 5). To obtain a solution for distributed VOLE generation, we show an efficient reduction from known-indices MPFSS, where one party chooses the indices of the point function, to known-index SPFSS. Our reduction is based on Cuckoo hashing [47], and in practice it is very efficient, in particular when compared with the alternatives proposed by Boyle et al. [9].

Distributed VOLE (Section 6). We combine the above protocol building blocks with some further optimizations, to obtain a full protocol for distributed VOLE generation.

Applications (Section 7). We investigate several applications of our protocols, including linear algebra and matrix manipulation primitives commonly used in data analysis tasks. We show how our protocols yield concretely efficient secure two-party instantiations for Oblivious Polynomial Evaluation. We further show that our known-index SPFSS protocol can be used to improve the efficiency (both in asymptotic round complexity and concrete efficiency) of semi-private accesses in a recent FSS-based distributed ORAM construction [23].

Experimental Evaluation (Section 8). While Boyle et al. [9] provide estimates for runtime and communication, they do not provide an implementation or an experimental evaluation. We implement all of our protocols, both over finite fields and integer rings, as well as the *primal* variant of the VOLE protocol proposed by Boyle et al. [9]. Instantiated over a finite field, we can generate a random VOLE of length $n = 2^{20}$ in about 5s. For comparison, generating the same using standard Gilboa multiplication takes 70% longer and has a 160% higher communication overhead. At the same time, our construction parallelizes nicely: using 32 threads, we can get an additional speedup of 9.8x, reducing the time to compute the above triple to 0.52s. In order to have a comprehensive comparison with alternative approaches we implemented and optimized Gilboa's multiplication protocol [31] and presented a comparison between our random VOLE protocol and a instantiation leveraging Gilboa's multiplications.

Concurrent Work. In recent concurrent work [10], Boyle et al. present a two-round OT extension protocol based on Vector-OLE. As in our work, they observe that VOLE key generation can be performed in a constant number of rounds. Unlike us, they implement the more efficient dual VOLE generator and provide malicious security. However, their implementation is limited to binary fields.

2 PRELIMINARIES

2.1 Oblivious Transfer

Oblivious transfer (OT) [50] is a fundamental primitive in cryptography that allows a receiver to obliviously select one out of two messages held by a sender without revealing the selection bit to the sender and without learning anything about the second message. The OT functionality is sufficient to implement general secure computation [40, 43, 56]. Here, a major step towards practical efficiency was the development of OT extension [6, 35], which allows to compute many computationally efficient online OTs using a small number of expensive OTs that can be executed in an advance offline phase. A variant of OT extension is random OT (ROT) extension [48], where random messages are generated as output to the sender while the receiver obtains one of the messages based on the selection bit. The ROT functionality has been used as a PRF with a single oblivious evaluation in the context of private set intersection protocols [48, 49].

All of the above OT notions can be generalized also to a setting where the sender has multiple messages and the receiver selects multiple indices. We formalize this in the following definition.

Definition 2.1 (m-out-of-n Oblivious Transfer (OT)). An m-out-of-n oblivious transfer is a protocol between two parties, sender and a receiver, where the sender has n messages as input and the receiver has m selection indices. The receiver obtains the messages corresponding to its indices while learning nothing about the remaining messages, and the sender learns nothing. If the n messages

are random and generated during the execution of the protocol as output for the sender, the protocol is called random m-out-of-n OT, or m-out-of-n ROT.

The communication complexity of 1-out-of-2 OT constructions is linear in the number of messages that the sender has. Naor and Pinkas [45] showed a reduction of 1-out-of-n OT to $\log n$ instances of 1-out-of-2 OT, which yields logarithmic communication complexity. Observing that 1-out-of-n OT is equivalent functionality to symmetric private information retrieval [16] is another approach to obtain an OT protocol with logarithmic complexity. Considering the general m-out-of-n functionality the communication complexity naturally scales linearly in m because we need to transfer at least that many messages. In Section 3 we show that this is no longer the case when we consider the m-out-of-n ROT functionality and we present a protocol that requires only logarithmic communication. In the spirit of the use of random OT extension as a PRF with single oblivious evaluation, we can view $(n-1)$-out-of-n ROT as a privately punctured PRF [8] where we can generate a partial PRF key that enables evaluation of the PRF on all but one point, where the full PRF key holder does not know the punctured point.

2.1.1 *OT-based secure product a.k.a Gilboa multiplication.*

Gilboa [31] proposed a two-party secure multiplication protocol of two l-bit numbers. The protocol outputs additive shares, and requires l 1-out-of-2 OT that can be run in parallel (throughout this paper we assume l to be a constant, and set it to 64 in our experiments). Due to the practical efficiency of OT Extension protocols [3, 35], Gilboa multiplication is a common approach to secure multiplication. In particular, this approach has been considered in several works as a way to compute Beaver triples for secure multiplication in the preprocessing model of MPC [22, 28, 44]. In the context of our work, this protocol is used for scalar vector multiplications. In terms of practical considerations, one should note that Gilboa multiplication can be implemented from correlated OT [3], a more efficient particular case of OT. Moreover, for the problem of scalar-vector multiplication, one can employ optimizations based on batching for concrete efficiency (see [44] for details). We employ these optimizations in our implementation of secure scalar-vector multiplication based on Gilboa's protocol, which we use as a baseline.

2.2 Cuckoo Hashing

Cuckoo hashing [47] is an algorithm to build hash tables for (key, value) pairs with worst-case constant lookup. A cuckoo hash table is determined by κ hash functions, where the value corresponding to a key is guaranteed to reside in one of the κ locations determined by the hash function evaluations on the key. Hash collisions are resolved using the cuckoo approach: if a collision occurs when placing an item in the hash table, the item residing in the location is evicted and then placed in the table using a different hash function, potentially evicting another item in the case of collision. This process continues until all evicted items are placed, if possible. Due to possible cycles in this graph of evictions, the insertion algorithm for cuckoo hashing has a chance to fail. For two hash functions, it is known that inserting n items in a cuckoo tables of size $O(n)$ incurs more than s insertion failures with probability bounded by $O(n^{-s})$ [41]. The exact constants in this asymptotic bound are not known, but several papers have studied them empirically [1, 15, 21]. This is done by estimating, for any fixed statistical security parameter η, the number of hash functions and the cuckoo table size such that inserting n items in the table fails with probability at most $2^{-\eta}$.

In cryptography, cuckoo hashing has been used as a probabilistic bath code to optimize Private Set Intersection (PSI) [15, 42, 48, 49] and Private Information Retrieval (PIR) [1] protocols. We introduce these ideas in Section 5, where we apply cuckoo hashing to obtain an optimized multi-point function secret sharing protocol.

2.3 Function Secret Sharing

Function secret sharing [12, 13] is a primitive that allows a key generator to distribute the evaluation of a function between two parties in way that neither of the two parties learns anything about the evaluated function, but jointly the two parties can recover the evaluation at any point.

Definition 2.2 (Function Secret Sharing). Let $\mathcal{F} = \{f : I \to \mathbb{G}\}$ be a class of functions with input domain I and output group $\mathbb{G}$, and let $\lambda \in \mathbb{N}$ denote a security parameter. A function secret sharing scheme consists of the following two algorithms:

- $(K_1, K_2) \leftarrow \mathsf{FSS.Gen}(1^\lambda, f)$ – given a description of $f : I \to \mathbb{G}$, output two keys K_1, K_2.
- $f_b(x) \leftarrow \mathsf{FSS.Eval}(b, K_b, x)$ – given an evaluation key K_b for $b \in \{1, 2\}$ and an input x, output a share $f_b(x)$ of the value $f(x)$.

We require the following guarantees from the above algorithms:

Correctness. For any $f \in \mathcal{F}$, and any $x \in I$, when $(K_1, K_2) \leftarrow \mathsf{FSS.Gen}(1^\lambda, f)$, we have $\Pr\left[\sum_{b\in\{1,2\}} \mathsf{FSS.Eval}(b, K_b, x) = f(x)\right] = 1$.

Security. For any $b \in \{1, 2\}$, there exists a ppt simulator Sim_b such that for any polynomial-size function sequence $f_\lambda \in \mathcal{F}$,

$$\left\{K_b \,\middle|\, (K_1, K_2) \leftarrow \mathsf{FSS.Gen}(1^\lambda, f_\lambda)\right\} \overset{c}{\approx} \left\{K_b \leftarrow \mathsf{Sim}_b(1^\lambda, \mathsf{Leak}_b(f_\lambda))\right\}. \quad (1)$$

Note that the only difference between this definition and the one of Boyle et al. [13] is the leakage function is allowed to be different for each party. In the standard FSS construction, $\mathsf{Leak}_1(f_\lambda) = \mathsf{Leak}_2(f_\lambda) = (I, \mathbb{G})$, i.e., FSS keys must be simulated given only the input and output domains for f.

While FSS is defined for any function, an FSS instantiation is non-trivial if the length of the FSS keys is sub-linear in the size of the function domain. In this regime of operation we have single point FSS (SPFSS) constructions for point functions which evaluate to zero on all but one of their domain points. Boyle et al. [12] introduced an FSS constructions for point functions where the keys are of length logarithmic in the function domain size.

Multi-point FSS (MPFSS) is a generalization of FSS where the shared functions has a larger number of non-zero evaluations. However, for the purposes of Vector-OLE (cf. Section 2.4), we observe that it is enough to consider a relaxed variant of MPFSS, where one party knows the where f is nonzero in the clear. We call this variant *known-indices MPFSS*, and we provide a reduction to cuckoo hashing and known-index SPFSS in Section 5.

2.4 Vector OLE

Oblivious linear evaluation (OLE) is functionality that enables two parties to obtain correlated outputs. One party has input values u, v. The second party has input x and obtains as output $w = ux + v$. Similarly to the use of OT for garbled circuits, OLE is a basic building block for secure arithmetic computation enabling the generation of multiplicative triples. Vector OLE (VOLE) [2, 9] is a generalization of OLE to the setting of vector inputs, i.e., one party has input vectors $\mathbf{u}, \mathbf{v}$, the other party has input value x and obtains a vector $\mathbf{w} = \mathbf{u}x + \mathbf{v}$. Boyle et al. [9] present application of VOLE to secure computation and zero-knowledge constructions.

Analogously to OT there is a variant of VOLE referred to as pseudorandom VOLE, where the vectors $\mathbf{u}, \mathbf{v}$ are generated randomly during the protocol execution. They are then provided as output to the first party. This primitive suffices for the construction of VOLE as well as its applications [9]. In Section 6 we present a new pseudorandom VOLE construction that requires a weaker version of the distributed MPFSS functionality compared to the approach of Boyle et al. [9], which can be implemented efficiently as we demonstrate in Section 8.

Definition 2.3 (Pseudorandom VOLE). A pseudorandom VOLE consists of the following algorithms:

- $(\mathsf{seed}_1, \mathsf{seed}_2) \leftarrow \mathsf{VOLE.Setup}(1^\lambda, n, \mathbb{F}, x)$ – this algorithms takes vector length n, field $\mathbb{F}$ and value x and outputs two seeds.
- $\mathsf{VOLE.Expand}(\mathsf{b}, \mathsf{seed}_\mathsf{b})$ – if $b = 1$, output $(\mathbf{u}, \mathbf{v}) \in \mathbb{F}^n \times \mathbb{F}^n$, else if $b = 2$, then output $\mathbf{w} \in \mathbb{F}^n$.

The correctness of the protocol guarantees that $\mathbf{w} = \mathbf{u}x + \mathbf{v}$. The security property requires that seed_1 does not reveal any information about x and that seed_2 does not allow to distinguish $(\mathbf{u}, \mathbf{v})$ from random vectors subject to the correctness property, i.e., for any ppt algorithm $\mathcal{A}$ the following holds:

$$\Big| \Pr[b = b' \;\Big|\; b' \leftarrow \mathcal{A}(\mathsf{seed}_1), (\mathsf{seed}_1, \mathsf{seed}_2) \leftarrow \mathsf{VOLE.Setup}(1^\lambda, n, \mathbb{F}, x_b), (\mathbb{F}, n, x_1, x_2) \leftarrow \mathcal{A}(1^\lambda)] - 1/2 \Big| < \mathsf{negl}.$$

$$\Big| \Pr[b = b' \;\Big|\; b' \leftarrow \mathcal{A}(\mathbf{u}_b, \mathbf{v}_b, \mathsf{seed}_2), (\mathsf{seed}_1, \mathsf{seed}_2) \leftarrow \mathsf{VOLE.Setup}(1^\lambda, n, \mathbb{F}, x), (\mathbb{F}, n, x) \leftarrow \mathcal{A}(1^\lambda), (\mathbf{u}_1, \mathbf{v}_1) \leftarrow \mathsf{VOLE.Expand}(1, \mathsf{seed}_1), \mathbf{w} \leftarrow \mathsf{VOLE.Expand}(2, \mathsf{seed}_2), \mathbf{u}_2 \leftarrow_R \mathbb{F}^n, \mathbf{v}_2 \leftarrow \mathbf{w} - \mathbf{u}_2 x] - 1/2 \Big| < \mathsf{negl}.$$

2.5 LPN Assumption

The learning parity with noise (LPN) assumption [7] states that given the noisy dot product of many public binary vectors $\mathbf{a}_i$ with a secret binary vector $\mathbf{s}$ is indistinguishable from a string of random bits. Adding noise to a bit is equivalent the flipping the bit with a fixed probability. We use the following generalization of the LPN assumption to larger fields.

Definition 2.4 (LPN Assumption). Let $\mathbb{C}$ be a probabilistic code generation algorithm which given inputs values k, q and a field $\mathbb{F}$, outputs a matrix $\mathsf{A} \in \mathbb{F}^{k \times q}$. The LPN assumption with respect to $\mathbb{C}$ with dimension $k = k(\lambda)$, $q = q(\lambda)$ queries and noise rate $r = r(\lambda)$ states that for any PPT algorithm $\mathcal{A}$ the following holds:

$$\Pr[1 \leftarrow \mathcal{A}(\mathsf{A}, \mathbf{b}) \;\big|\; \mathbb{F} \leftarrow \mathcal{A}(1^\lambda), \mathsf{A} \leftarrow \mathbb{C}(k, q, \mathbb{F}), \mathbf{e} \leftarrow \mathsf{Ber}_r(\mathbb{F})^q, \mathbf{s} \leftarrow \mathbb{F}^k, \mathbf{b} \leftarrow \mathbf{s} \cdot \mathsf{A} + \mathbf{e}]$$
$$\approx \Pr[1 \leftarrow \mathcal{A}(\mathsf{A}, \mathbf{b}) \;\big|\; \mathbb{F} \leftarrow \mathcal{A}(1^\lambda), \mathsf{A} \leftarrow \mathbb{C}(k, q, \mathbb{F}), \mathbf{b} \leftarrow \mathbb{F}^q].$$

In our construction (Section 6) we use LPN instantiations with parapets settings as those described by Boyle et al. [9], i.e., high dimension k, low noise rate $1/k^\epsilon$ for a constant ϵ and a polynomial number of queries $q = k + o(k)$. Since we focus on the primal variant of VOLE, we can instantiate $\mathbb{C}$ using a *local linear code*, where LPN is assumed to hold [2, 9].

Ring-LPN is a variant of the LPN assumption defined over rings rather than fields. The security of this assumption is less studied but there are works that explore its use in the context of protocols for the purposes of efficiency [19, 34]. In our implementation we evaluate the performance of our protocol both in the setting of a field and a ring which rely on the two variants of the LPN assumption.

Functionality 1: $(n-1)$-out-of-n-ROT

Parties: P_1, P_2
Input: P_2: Index $i \in [n]$
Output (for P_1): Pseudorandom vector $\mathbf{u} \in \mathbb{F}^n$
Output (for P_2): vector $\mathbf{v} = (\mathbf{u}_j)_{j \neq i}$

2.6 Definitions, Functionalities, and Secure Two-Party Protocols

All the constructions in this paper describe communication efficient two-party protocols for computing correlated vectors, for different types of correlations. This notion has recently been formalized as a *Pseudorandom Correlation Generator (PCG)* [11]. As observed there, communication-efficient PCGs don't lend themselves to direct simulation-based security proofs. Intuitively, this stems from the fact that any simulator that takes as input the ideal pseudorandom output and produces succinct messages for the protocol would be able to compress pseudorandom strings, which is impossible. However, it was shown that in many applications (including all the applications of vector OLE we consider), a weaker security definition is sufficient [9, 11]. We will therefore use the same approach as Boyle et al. and split up our protocols in two phases, namely *setup* (or *generation*), and *expansion* (or *evaluation*). This allows us to use the following structure in our security proofs: First, (i) we define correctness and security requirements of the generation and expansion algorithms. Then, (ii) we define ideal functionalities for the two phases and show that they satisfy our definition. And finally (iii), we show that our protocols securely (and efficiently) implement the key generation functionality. We focus on presenting our two-party protocols in the main paper, and giving the intuition behind for both security and efficiency. Nevertheless, detailed definitions, functionalities, and security proofs for all our novel constructions are given in the appendix.

Protocol 2: $(n-1)$-out-of-n Random OT

Public Params: PRG G of stretch $k > 1$ and security parameter λ, integer $n = k^c$ with $c > 0$

Inputs: P_1: $\perp$; P_2: index $i \in [n]$

Outputs:

P_1: n random values $(r_j)_{j\in[n]}$

P_2: $n-1$ random values $(r_j)_{j\in[n],j\neq i}$

Key Generation ($\mathsf{ROT.Gen}(1^\lambda, n, i)$**):**

(1) P_1 generates a PRG seed $s_0 \xleftarrow{R} \{0,1\}^\lambda$.

(2) P_1 computes a k-ary GGM tree of depth $\alpha = \log_k(n)$, denoted $T = T(s_0, \alpha)$, by associating s_0 to T and, if $\alpha > 1$, constructing the k children of T recursively as $T(s_j, \alpha - 1)$, with $j \in [k]$ and seeds $s_1, \ldots, s_k$ computed as

$$(\ s_1 \mid s_2 \mid \ \ldots \ \mid s_k\ ) := \mathrm{G}(s_0)$$

(3) P_2 computes $(b_1, \ldots, b_\alpha)$, the k-ary encoding of $i-1$.

(4) The parties execute α instances of $(k-1)$-out-of-k OTs:

- P_1 acts as sender. For the lth OT, let $(p_1, \ldots, p_{k^l})$ be the seeds of the lth level of T. The jth message in the OT is set to be

$$m_j := \bigoplus_{s\in\{p_x\,:\,x\,\equiv\,j\ \bmod k\}} s$$

  (the jth message is the XOR of the seeds of the jth children of trees at level $l-1$).
- P_2 acts as the chooser and inputs, in the lth OT, the set $\{0, \ldots, k-1\} \setminus \{b_l\}$, and obtains $k-1$ seeds $q_{l,j}$ with $j \in [k] \setminus \{b_l\}$.

(5) P_1 outputs $K_1 \leftarrow s_0$

(6) P_2 outputs $K_2 \leftarrow (q_{l,j})_{l\in[\alpha],j\in[k]\setminus\{b_l\}}$.

Expansion ($\mathsf{ROT.Expand}(b, K_b)$**):**

(i) If $b = 1$: P_1 returns the list of leaves of T.

(ii) If $b = 2$: P_2 uses the seeds $q_{l,j}$ to reconstruct T, except for the path to the ith leaf (recall that $(b_1, \ldots, b_\alpha)$ is the k-ary encoding of $i-1$).

- For the first level, P_2 constructs trees $T_j = T(q_{1,j}, \alpha - 1)$ with $j \in [k] \setminus \{b_1\}$.
- For each level $l \in [\alpha]$, let $T_1, \ldots, T_{k^l}$ be the sub-trees of T at level l. In previous iterations P_2 has computed all such sub-trees except for T_{i_j}, with $i_l = \sum_{x\in[l]} k^{x-1} \cdot b_x + 1$. P_2 then collects the seeds of the direct children of each T_j as $\{s_{j,1}, \ldots, s_{j,k}\}_{j\in[k]\setminus i_l}$. Then, additional seeds $\{s_{b_l,j}\}_{j\neq i_{l+1}}$ can be obtained from $(q'_{l,j})_{j\neq b_{l+1}}$ as

$$s_{b_l,j} := \bigoplus_{s\in\{s_{j,x}\,:\,x\,\equiv\,j\ \bmod k\}} s \oplus q_{l,j}$$

  By expanding those seeds using G, P_2 computes all sub-trees of T at level $l+1$, except for the one at position $i_{l+1} = \sum_{x\in[l+1]} k^{x-1} \cdot b_x + 1$.

P_2 returns the list of seeds of leaves of T, except for the one at position $i = \sum_{x\in[\alpha]} k^{x-1} \cdot b_x + 1$.

3 $(n-1)$-OUT-OF-n RANDOM OT

In this section we consider the question of oblivious selection of $n-1$ items out of n in the case when all items are pseudorandom. This corresponds to Functionality 1, namely $(n-1)$-out-of-n random oblivious transfer. If we allow linear communication, a protocol for Functionality 1 can be easily obtained using oblivious selection techniques. We instead propose a protocol with sub-linear communication and linear computation. Our protocol consists of a key generation phase where P_1 learns a key K_1 consisting of a single PRG seed s_0, and P_2 learns a key K_2 consisting of $\log_k(n)$ PRG seeds, via $\log_k(n)$ *parallel* executions of a $(k-1)$-out-of-k OT protocol, for parameter $k > 1$. Expanding the respective seeds to obtain their length-n outputs takes $O(n)$ PRG evaluations per party.

Key generation via a GGM tree. We crucially leverage the fact that values are generated pseudo-randomly in order to obtain a protocol with the above communication complexity. Let us assume, without loss of generality, that $\log_k(n)$ is an integer. The n values of $\mathbf{u}$ are generated from a single random seed s_0 using a GGM tree T [32] constructed using a PRG G of stretch k, i.e. $G : \{0,1\}^\lambda \mapsto \{0,1\}^{k\lambda}$, for security parameter λ. More concretely, T is an ordered complete k-ary tree of depth $\log_k(n)$ and n leaves, with its nodes labeled with seeds in $\{0,1\}^\lambda$ (we will refer to nodes and their seeds/labels indistinctly). The label of the root is s_0, and the label s_j of the jth child of a node v is obtained from the seed of v, by applying the PRG G and parsing the output as $(s_1|\cdots|s_j|\cdots|s_k)$.

The 2-party protocol. Our protocol is presented as Protocol 2. First, P_1, the sender, computes the tree T locally from a seed s_0 (note that this can be done with $2n-1$ calls to G) and sets s_0 to be its key K_1. The rest of the protocol allows P_2, the receiver, to recover all the seeds of T, except for the ones in the path to the ith leaf. This is done in a way that does not leak i to P_1, and requires only $\log(n)$ seeds, which will constitute P_2's key K_2, to be expanded locally. We now informally discuss the correctness and security of our protocol, as well as associated communication and computation costs.

Let $(i_1, \ldots, i_{\log_k(n)})$ be the path to the i-th leaf (this is a sequence of values in $\{0, \ldots, k-1\}$, indicating which children to follow at each level to reach the ith leaf from the root, and in fact corresponds to the k-ary encoding of the integer $i-1$). For example, Figure 1 shows how for $n = 8$ and $i = 3$, the path the receiver should not learn is 010. As mentioned above, our goal is that the receiver can reconstruct all the tree except for the nodes on this path.

Although it will become clear that the protocol can be parallelized across levels, for explanatory purposes it is useful to think of it as processing T level by level from the root guaranteeing that, for each level $l \in [\log_k(n)]$, the receiver can reconstruct T up to level l, except for the nodes in the path $(i_1, \ldots, i_l)$. This property obviously holds for $l = 0$ and, to argue the correctness of our protocol, we now argue inductively how to extend it from level l to level $l+1$.

By induction assume that the receiver can reconstruct all subtrees $T_1, \ldots T_{k^l}$ of depth $\alpha - l$ rooted at the nodes of level l except for exactly one: the one rooted at path $(i_1, \ldots, i_l)$. This is, precisely, $T_{(\sum_{x\in[l]} k^{x-1}\cdot i_x+1)}$, which we denote T^* for simplicity. Now, let us show how a single execution of $(k-1)$-out-of-k OT is enough to extend the above property to level $l+1$. Intuitively, we want

to ensure that the receiver learns all direct children of T^*, except for the i_{l+1}th one. As T^* has k direct children, this corresponds to a $(k-1)$-out-of-k-OT. However, for privacy, it is important that the sender never learns that T^* is in fact the sub-tree that the receiver cannot reconstruct at level l, as this reveals too much about the index i. This difficulty can be overcome by constructing the messages in the $(k-1)$-out-of-k-OT as follows.

Let $s_{j,0}, \ldots, s_{j,k-1}$ be the seeds of the nodes that are direct children of each tree T_j. As the receiver knows all the T_js except for T^*, she has all such seeds except for the ones with $j = (\sum_{x \in [l]} k^{x-1} \cdot i_x + 1)$, i.e., the children of T^*. The key idea to achieve the above goal is to have the sender compute k values $m_0 = \left(\bigoplus_{j=1}^{k^l} s_{j,0} \right), \ldots, m_{k-1} = \left(\bigoplus_{j=1}^{k^l} s_{j,k-1} \right)$. Here, m_0 is the XOR of all direct first children of nodes at level l, m_1 is the XOR of all second children, and so on. Now observe that, given any value m_y the receiver can compute the seed $s_{(\sum_{x \in [l]} k^{x-1} \cdot i_x + 1), j}$ (the yth child of T^*) since she knows all the other values XOR-ed into the m_j value. On the other hand, m_j does not reveal anything about the seeds $s_{k^l, x}$ with $x \neq j$. Thus, the sender and the receiver run $(k-1)$-out-of-k OT where the sender's inputs are $m_0, \ldots, m_{k-1}$ and the receiver's input is the set $\{0, \ldots, k-1\} \setminus \{i_l\}$. After running this sub-protocol the receiver can reconstruct T up to level $l+1$, except for the nodes in the path $(i_1, \ldots, i_{l+1})$. This shows how to extend the construction from level l to $l+1$, and the protocol finishes when $l = n$.

An important observation is that the instances of $(k-1)$-out-of-k OT used in the above construction can all be run in parallel. The correctness of our construction follows from the above discussion, and its security, stated in the next lemma, follows directly from the security of G, and the underlying protocol for $(k-1)$-out-of-k OT. A detailed proof can be found in Appendix A.1. In Section 8 we describe how G is instantiated in our implementation, as well as other practical considerations and optimizations.

Lemma 3.1. *For any constant $k > 1$, Protocol 2 is a secure two party computation protocol for the $(n-1)$-out-of-n ROT functionality in the $(k-1)$-out-of-k OT hybrid model assuming a secure PRG G. The protocol is one round, and requires $O(\lambda \log(n))$ communication and $O(\lambda n)$ computation per party, including $2n$ PRG evaluations, where λ is the length of the PRG seed.*

Proof Sketch. Showing the security of the above protocol consists of two steps: first, showing that the keys that the parties receive have the desired pseudorandom properties (Definition A.1), which follows from the pseudorandom properties of the GGM construction and which we formally prove in Theorem A.2. And second, showing that the key generation protocol is a secure two party computation protocol for the generation of the keys, which follows from the OT security and which we prove formally in Theorem A.3. The communication overhead follows from the fact that the parties execute $\log_k n$ OTs, which have linear communication in λ. The computation $O(\lambda n)$ for each comes from the execution of the $\log_k n$ OTs and the expansion of the keys which uses $2n$ PRG calls.

How to set k, and instantiations of $(k-1)$-out-of-k OT. The construction of Protocol 2 works for any integer $k > 1$. Choosing k constant results in logarithmic communication, and in fact in our implementation we use $k = 2$. In practice, this allows us to leverage very efficient implementations of 1-out-of-2 OT based on OT Extension. When instantiated with $k = 2$, our protocol resembles the Function Secret sharing construction by Boyle et al. [12].

Privately Punctured PRF. Our $n-1$-out-n random OT protocol also provides a construction for a privately punctured pseudorandom function, where one party has the PRF key and can evaluate the PRF on any input (in our case this is P_1 who has the GGM root) and the other party has a punctured key which allows it to evaluate the PRF on all but one inputs (P_2 in our case). The OT protocol enables P_2 to obtain its punctured PRF key without revealing the punctured point to P_1 (the punctured key is the output that P_2 has at the end of the KeyExchange phase of the OT protocol). We note the difference in the punctured key generation algorithm from the one defined in other contexts for privately puncturable PRFs [8], where the party who has the full PRF key generates the punctured key and knows the point at which it is punctured.

4 KNOWN-INDEX SPFSS

In this section we use our $n-1$-out-of-n random OT protocol to construct a 2-party computation protocol to jointly generate FSS keys for point functions. The setup for our distributed FSS protocol assumes that one party knows the non-zero evaluation point while the value at that point is shared between the two parties. Thus, it is not equivalent to a generic distributed FSS scheme for point functions, as for example described in [23]. However, this relaxed version suffices for our VOLE construction described in Section 6. We call our FSS variant *Known-Index SPFSS* to emphasize that one party knows the non-zero index.

Conceptually, the existing construction of point function FSS [12] generates two PRF keys K_1 and K_2 such that $\mathsf{PRF}_{K_1}(x) = -\mathsf{PRF}_{K_2}(x)$ for all values of x except the input with non-zero evaluation i. The values $\mathsf{PRF}_{K_1}(i)$ and $\mathsf{PRF}_{K_2}(i)$ are random shares of the function evaluation β at the input i. If two parties need to generate K_1 and K_2 in a distributed way, they can use secure general computation for this task, and Doerner and shelat [23] show a more efficient way to construct such an MPC protocol in the semi-honest setting.

When one of the parties, P_2, knows i, we can construct K_1 and K_2 in a distributed fashion as follows. First, P_1 and P_2 run a secure $(n-1)$-out-of-n-ROT key generation protocol (the construction from the previous section), for the parties to obtain keys K_1^{ROT} and K_2^{ROT}. Note that, if the parties compute $\mathbf{r}^b = \mathsf{ROT.Expand}(b, K_b^{\mathsf{ROT}})$, the vectors $\mathbf{r^1}, \mathbf{r^2}$ coincide at every position except for a position i known to P_2. As P_2 can negate its vector, we can think of $\mathbf{r^1}$ and $-\mathbf{r^2}$ as additive shares of a vector of all zeroes except for the ith position. Now all that remains is to modify $\mathbf{r^1}$ and $-\mathbf{r^2}$ to fix the ith position to be a share of a value β shared among P_1 and P_2 (see Protocol 3). Crucially, this needs to be done in a way that does not leak β to either party, and keeps i private from P_1. To do this we leverage the observation that P_1 and P_2 can compute sums $R = \sum_j r_j^1$ and $R' = \sum_{j \neq i} r_j^2$. The difference $R - R'$ will be the evaluation of $\mathsf{PRF}_{K_1}(j)$. Since the parties have shares β_1 and β_2 of the point function evaluation β at i, we complete the protocol by P_1 sending $R_\beta = R - \beta_1$ to P_2 (note that this hides β_1 because P_2 does not know r_i^1, which is a random mask) who

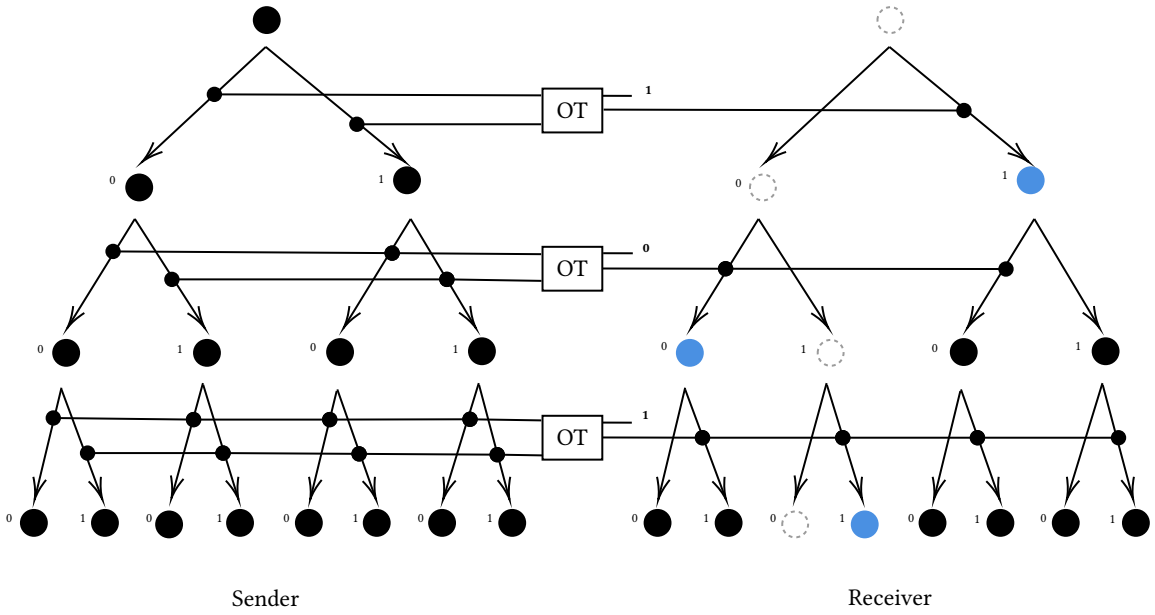

Figure 1: Example of the GGM tree generated by the sender and partially learnt by the receiver. Here, $k = 2$, $n = 8$, and $i = 3$. Thus, the path not learnt by the receiver is (010). For each level, the parties run an OT where the receiver learns an XOR of either the left children or the right children of that level. Using previously expanded sub-trees, this information allows the receiver to learn a new seed (nodes filled in blue) which can be expanded by repeatedly calling G (the nodes resulting from such expansions are filled in black).

can then appropriately set the ith entry of $\mathbf{r^2}$ so that $r_i^1 + r_i^2 = \beta$. Note that, as long as P_2 obtains R_β in the key generation phase, the corrections can be applied during expansion. Our construction is presented in Protocol 3 in terms of the key generation and expansion procedures for $(n-1)$-out-of-n-ROT from the previous section, which encompasses the steps from above.

Lemma 4.1. *Protocol 3 securely implements Known-Index SPFSS over a domain of size n in the $(n-1)$-out-of-n-ROT hybrid model. With $(n-1)$-out-of-n-ROT instantiated by the construction of Protocol 2, Protocol 3 requires $O(\lambda \log n)$ communication and $O(\lambda n)$ computation per party where λ is the security parameter of the ROT.*

Proof Sketch. The main argument in the security proof is that R_β is a one-time pad that masks β_1, given the property of $(n-1)$-out-of-n-ROT that the output of P_1 is a random vector. A detailed proof is given in Appendix A.2.

5 KNOWN-INDICES MPFSS VIA CUCKOO HASHING

In this section we present a reduction from known-index multi-point FSS to known index single point FSS. The multi-point setting is analogous to the SPFSS functionality of Protocol 3, but extended to functions that fix the value of $t \geq 1$ points. We formalize our *Known-Indices MPFSS* variant in Definition A.7 in the appendix. A naive reduction executes t independent instances of known-index SPFSS on the original database. However, as observed by Boyle et al. [9], this requires evaluating all t SPFSS instances on the whole domain, which results in an $\Omega(tn)$ computational overhead.

Protocol 3: Distributed Known-Index Single Point FSS

Params and Building Blocks: $(n-1)$-out-of-n-ROT;
Point function $f : [n] \to \mathbb{G}, f(i) = \beta, f(j) = 0 \; \forall \; j \neq i$
Random shares $\beta_1, \beta_2 : \beta_1 + \beta_2 = \beta$; $b \in \{0, 1\}$
Parties: P_1, P_2
Inputs: $\mathsf{P}_1 : \beta_1$, $\mathsf{P}_2 : \beta_2, i$
Key Generation ($\mathsf{SPFSS.Gen}(1^\lambda, f_{i,\beta})$)**:**

(1) The parties run a secure $\mathsf{ROT.Gen}(1^\lambda, n, i)$ protocol to obtain keys K_1^{ROT} and K_2^{ROT}.
(2) The parties execute locally ROT.Expand, from which P_1 gets n random values $\{r_i\}_{j\in[n]}$, and P_2 obtains $\{r_i\}_{j\in[n], j\neq i}$.
(3) Let $R = \sum_{j\in[n]} r_i$. P_1 sends to P_2 the value $R_\beta = R - \beta_1$.
(4) P_2 computes $\tilde{r} = \beta_2 - R_\beta + \sum_{j\in[n]\setminus\{i\}} r_i$.
(5) P_1 outputs $K_1 \leftarrow K_1^{\mathsf{ROT}}$
(6) P_2 outputs $K_2 \leftarrow (K_2^{\mathsf{ROT}}, \tilde{r})$.

Expansion ($\mathsf{SPFSS.Eval}(b, K_b, x)$)**:**

- If $b = 1$, compute $\mathbf{v}^1 \leftarrow \mathsf{ROT.Expand}(1, K_1)$ and output v_x^1.
- If $b = 2$, parse K_2 as $\left(K_2^{\mathsf{ROT}}, \tilde{r}\right)$. If $x = i$, output $\tilde{r}$. Otherwise, compute $\mathbf{v}^2 \leftarrow \mathsf{ROT.Expand}\left(2, K_2^{\mathsf{ROT}}\right)$ and output $-v_x^2$.

A general idea to improve on this baseline is to rely on batching schemes that split the domain of size n into m small parts in a way that allows to distribute the t SPFSS instances across the m smaller parts. One can instantiate this general idea using a combinatorial object called *batch codes* (see Ishai et al. [36] for an introduction).

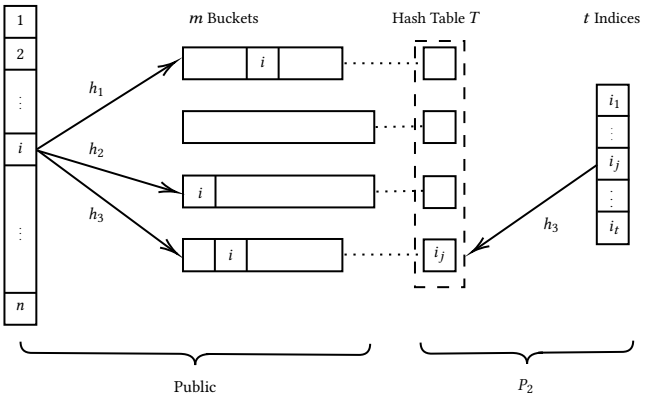

Figure 2: The domain of the MP function is hashed κ times into m Buckets using κ different hash functions (this arrangement is public). P_2 privately builds a cuckoo hash table of the indices of the MP function. Then, an instance of known-index SPFSS is executed for each bucket.

A batch code with parameters n, t, k, m gives a partition of a database of size n into m parts such that any t indices from $[n]$ can be recovered by reading at most k entries in each of the m parts. Although batch codes are attractive in that they offer very strong provable guarantees, they can be hard to instantiate in practice. This issue arises in the construction proposed by Boyle et al. [9], who explore Combinatorial Batch Codes (CBCs) for batching multiple FSS instances to obtain MPFSS. Since explicit constructions of the expander graphs required for instantiating a CBC do not satisfy their efficiency requirements, Boyle et al. propose a heuristic construction of a CBC. This leads to a small failure probability, which asymptotically depends on t and the expansion factor of the batch code. However, concrete parameters for the heuristic CBC construction are not given by Boyle et al., and in their running time estimates, the authors assume t SPFSS instances on disjoint subsets of $[n]$ instead of full MPFSS.

A second approach to baching is given by Angel et al. [1], who introduce a relaxed notion of Probabilistic Batch Codes (PBCs). Unlike the heuristic CBC construction of Boyle et al. [9], batching here may fail *on each insertion* of t indices with a certain probability (which can be made arbitrarily small). The PBC construction of Angel et al. [1] is inspired by many works in the PSI literature [15, 21, 26, 49], where cuckoo hashing [47] is commonly used to reduce PSI to private set membership queries. We follow this line of work, and base our MPFSS construction on probabilistic batching.

5.1 Batching Known-Index SPFSS

5.1.1 Cuckoo hashing as a PBC. Our approach to build MPFSS from single point FSS is to use Cuckoo hashing [47] and simple hashing in a similar manner as in PSI and PIR protocols [1, 15, 21, 26, 49]. Cuckoo hashing [47] is a multi-choice hashing scheme with eviction parameterized by κ universal hash functions $h_1, \ldots, h_\kappa$. Cuckoo hashing achieves the goal of distributing t items in a table T of size m in a manner that guarantees that each location in T is occupied by at most one item. The insertion algorithm puts the item to be inserted x at $T[h_1(x)]$ and, if this position is occupied, evicts the item in that position and relocates it using h_2, which may cause yet another eviction resolved using h_3, and so on. This insertion algorithm may fail when a cycle of evictions is found, and thus cuckoo hashing has a failure probability that depends on parameters κ, n, t and m. Several works [15, 21, 49] that use cuckoo hashing in secure computation protocols have empirically studied such parameters and how they relate to the failure probability. In our work we use the estimates of Demmler et al. [21], which leads to the same parameter choices used by Angel et al. [1] and Chen et al. [15].

While we present our concrete parameter choices in Section 8.2, we keep these symbolic in the protocol description for presentation purposes. We therefore introduce a statistical security parameter η, meaning that the probability of failing at hashing t items is bounded by $2^{-\eta}$. More specifically, we denote by $\mathsf{ParamGen}(n, t, \eta)$ the function that generates cuckoo hashing parameters, i.e., number of hash functions κ and cuckoo table size m, that guarantee this statistical bound on the insertion failure probability. Note that in the case of such an insertion failure, P_1 learns of it in Step (1) of the protocol and thus can handle this case in several ways in practice. For example, it could simply abort the protocol, or it could sample new hash functions until the hashing step succeeds or a maximum number of trials is reached. In these cases, hashing failures result in leakage, as the adversary can infer information about the indices from the fact that they failed (or did not fail) to hash. A second option is to sacrifice correctness instead, and simply ignore indices that failed to hash. This way, no information is leaked from the generated MPFSS keys, but the multi-point function changes with a small probability. As discussed in [1, 15], the strategy for handling hashing failures depends a lot on the exact use case. In the case of vector OLE, we choose to drop indices that fail to hash (cf. Section 6). This is also the approach suggested by Boyle et al. [9] for their heuristic batch code construction. Our protocol therefore achieves the same type of security guarantee, while at the same time being concretely efficient.

5.1.2 Our protocol. Our construction is shown in Protocol 4. We use a cuckoo hashing scheme with capacity t instantiated with κ hash functions mapping $[n]$ to $[m]$, where $(m, \kappa) = \mathsf{ParamGen}(n, t, \eta)$ as described above. In step (1), the party holding the t non-zero evaluation points of the multi-point function computes a cuckoo hash table T of size m that contains them. In step (2), the two parties use all of the κ hash functions to simple-hash the whole domain $[n]$. This results in m buckets $I_1, \ldots, I_m$, with κ copies of each integer in $[n]$ distributed across them uniformly at random. An important point is that this arrangement is public (see left side of Figure 2). The parties also fix an order within each bucket I_l, and compute the reverse mapping pos_l from items to positions.

After having assigned indices to buckets, our protocol securely runs an SPFSS key generation for each bucket I_l. First, in step (3), the parties obtain shares of the vector $\mathbf{v}$ of values to be fixed in each of the SPFSS instances (the value β in Protocol 3). This needs to be done in a secure computation because both parties share all β_i's, while only P_2 knows which β_i maps to which bucket. The secure computation can be implemented using permutation networks [53] in a garbled circuit, or using additive homomorphic encryption.

Protocol 4: Distributed Known-Indices MPFSS

Public Params: Input domain $[n]$, number of points t, statistical security parameter η,
Cuckoo hash parameters: table size m, and number of hash functions κ, $(m, \kappa) = \mathsf{ParamGen}(n, t, \eta)$
Point function $f_{\mathbf{i},\boldsymbol{\beta}} : [n] \to \mathbb{F}$, $f_{\mathbf{i},\boldsymbol{\beta}}(i_j) = \beta_j^1 + \beta_j^2$ for all $j \in [t]$, $f_{\mathbf{i},\boldsymbol{\beta}}(j') = 0$ for all other inputs.
Parties: P_1, P_2
Inputs: P_1: x, $\beta_1^1, \ldots, \beta_t^1$; P_2: $i_1, \ldots i_t$, $\beta_1^2, \ldots, \beta_t^2$
Key Generation ($\mathsf{MPFSS.Gen}(1^\lambda, f_{\mathbf{i},\boldsymbol{\beta}})$**):**

(1) P_2 randomly chooses κ hash functions $(h_j)_{j\in[\kappa]}$, with $h_j : [n] \to [m]$. P_2 inserts $i_1, \ldots i_t$ into a Cuckoo hash table T of size m using $h_1, \ldots, h_\kappa$, and it sends the κ hash functions to P_1. Let empty bins in T be denoted by $\perp$.
(2) P_1 and P_2 do simple hashing with all $k_1, \ldots, h_\kappa$ on the domain $[n]$, to independently build m buckets $I_1, \ldots, I_m$, i.e. $I_l = (x \in [n] \mid \exists p \in [\kappa] : h_p(x) = l)$, for $l \in [m]$, each sorted in some canonical order. The parties compute functions $\mathsf{pos}_l : I_l \to [|I_l|]$ that map values to their position in the l-th bucket.
(3) Let $\mathbf{u} = \left((\beta_j^1 + \beta_j^2, l_j)\right)_{j\in[t]}$, where l_j is the location of i_j in T. The parties run a secure 2PC protocol to obtain random shares $\mathbf{v^1}, \mathbf{v^2}$ of the vector $\mathbf{v} \in \mathbb{F}^m$ defined as

$$\mathbf{v}_j = \begin{cases} a & \text{if } (a, j) \in \mathbf{u}, \\ 0 & \text{otherwise.} \end{cases}$$

(4) For all $l \in [m]$, P_1 and P_2 run $SPFSS.Gen(1^\lambda, g_l)$ (Protocol 3) to obtain seeds (K_1^l, K_2^l), with $g_l : [|I_l|] \to \mathbb{F}$ defined as

$$g_l(x) = \begin{cases} \mathbf{v^1}_l + \mathbf{v^2}_l & \text{if } T[l] \neq \perp \text{ and } x = \mathsf{pos}_l(T[l]), \\ 0 & \text{otherwise.} \end{cases}$$

(5) P_1 outputs $K_1 = \left(K_1^l\right)_{l\in[m]}$ and P_2 outputs $K_2 = \left(K_2^l\right)_{l\in[m]}$.

Expansion ($\mathsf{MPFSS.Eval}(b, K_b, x)$**):**
Output $\sum_{p=1}^{\kappa} \mathsf{SPFSS.Eval}\left(b, K_b^{h_p(x)}, \mathsf{pos}_{h_p(x)}(x)\right)$.

However, as we will see at the end of this section, this step can be omitted in the special case of Vector OLE.

In Steps (4) and (5), we generate and return SPFSS keys for each of the buckets, where the values are the shares obtained in the previous step, and the indexes are known to P_2. Note that, since $m \geq t$, some positions in T might be empty, so those instances have the zero function associated to them (which is known only to P_2).

Finally, the evaluation of an MPFSS key on an input x is the sum the evaluations of the SPFSS keys corresponding to the buckets into which x is mapped by the cuckoo hash functions.

Lemma 5.1. *Assume a secure Known-Index SPFSS scheme with a secure two-party key generation protocol, both with security parameter λ. Then Protocol 4 implements a secure two-party protocol for generating Known-Indices MPFSS keys in the semi-honest model with statistical security η. Using Yao garbled circuits to instantiate step (2), the MPFSS.Gen protocol is constant round, and requires $O(m\lambda \log n)$ communication and $O(\lambda\kappa n + \lambda m \log n)$ local computation per party, where $(m, \kappa) = \mathsf{ParamGen}(n, t, \eta)$ are cuckoo hashing parameters.*

Protocol 5: MPFSS Optimization for VOLE

Public Params: Input domain $[n]$, number of points t, hash table size $m = \tilde{O}(t)$, and number of hash functions κ.
Point function $f_{\mathbf{i},x\mathbf{y}} : [n] \to \mathbb{F}$, $f_{\mathbf{i},x\mathbf{y}}(i_j) = (x\mathbf{y})_j$ for all $j \in [t]$, $f_{\mathbf{i},x\mathbf{y}}(j') = 0$ for all other inputs.
Parties: P_1, P_2
Inputs: P_1: x; P_2: $i_1, \ldots i_t$, $y_1, \ldots, y_t$
Key Generation ($\mathsf{MPFSS.Gen}(1^\lambda, f_{\mathbf{i},x\mathbf{y}})$**):**

1,2 *These are the same as in Protocol 4.*
(3a) Let $\mathbf{u} = ((y_j, l_j))_{j\in[t]}$, where l_j is the location of i_j in T. P_2 locally computes the vector $\mathbf{w} \in \mathbb{F}^m$ defined as:

$$\mathbf{w}_j = \begin{cases} a & \text{if } (a, j) \in \mathbf{u}, \\ 0 & \text{otherwise.} \end{cases}$$

(3b) The parties run an MPC to compute shares of $\mathbf{v} = x\mathbf{w}$.
(4, 5) *The rest of the protocol is as Protocol 4.*

Proof Sketch. We outline the intuition for the proof of Lemma 5.1 here and provide the full proof in Appendix A.3. Proving the security of the MPFSS protocol involves two steps: first, proving that the keys generated from the generation algorithm satisfy the FSS security requirements, and second, proving the generation protocol is a secure two party computation protocol that reveals to each party only its corresponding key. The first claim follows directly from the security guarantee of the SPFSS construction used to generate a key for each bucket. We prove this formally in Theorem A.8. The second claim follows from the security of the two party protocol used for the SPFSS key generation, which we prove formally in Theorem A.9.

The communication and computation for the garbled circuit used for Step (2) is $O(\lambda m \log m)$ since it needs to implement an oblivious permutation protocol over m items. For each SPFSS instance in Step (3), we need $O(\lambda \log n)$ communication, since in the worst case each bucket has size $O(n)$. The computation that each party does includes simple hashing of all elements in $O(\lambda\kappa n)$, SPFSS distributed key generation for each bucket in $O(\lambda m \log n)$ and the MPFSS evaluation in $O(\lambda\kappa n)$.

An Optimization for Vector-OLE. We leverage another observation related to the use of MPFSS in the context of vector OLE, which allows us to construct a more efficient solution. In the VOLE generator of Boyle et al. [9], the non-zero values for t-point MPFSS are of the form $xy_1, \ldots, xy_t$, where one party knows the indices of the non-zero function values and $y_1, \ldots, y_t$, and the other party knows x. Thus, we can have a secure two party computation protocol where one party inputs $y_1, \ldots, y_t$ padded with zero up to the size of the cuckoo table, in the order in which they are mapped to the cuckoo bins, and the other party inputs x. The protocol multiplies x with the permuted vector and outputs shares of the result to the two parties. That way, we can generate the MPFSS keys needed for VOLE generation without the expensive secure permutation in Step (3), and instead use a cheap multiplication protocol such as Gilboa multiplication [31].

6 DISTRIBUTED PSEUDORANDOM VOLE FROM MULTI-POINT FSS

In this section we present a new construction for two party computation of pseudorandom vector OLE that relies on multi-point function secret sharing. The main difference between our construction and the reduction described in the work of Boyle et al. [9] is the observation that the multi-point function that the two parties evaluate does not need to be completely hidden from both of them, since one of the keys contains the non-zero points in the clear. Thus, it suffices to use our distributed *Known-Indices MPFSS* from the previous section. We present our construction in Protocol 6.

The goal of a pseudorandom VOLE is to enable two parties P_1 and P_2 to obtain the following correlated outputs: P_1 obtains vectors $\mathbf{u}$ and $\mathbf{v}$, and P_2 obtains integer value x and a vector $\mathbf{w}$ such that $\mathbf{u}x + \mathbf{v} = \mathbf{w}$. The requirements for these correlated outputs are that 1) $\mathbf{u}$ and $\mathbf{v}$ do not reveal information about x and 2) given $\mathbf{w}$, $\mathbf{u}$ and $\mathbf{v}$ are indistinguishable from random vectors generated subject to the above relation. Without any further efficiency constraints the above functionality can be realized using standard MPC techniques. However, the goal here is to generate a VOLE correlation with much less communication than the length of the vectors. In this case, distributed VOLE faces the same problems as other correlation generators (cf. Section 2.6 and Boyle et al. [11]), i.e., that protocol messages of sublinear size can't be simulated from an ideal uniform output. Hence, the VOLE functionality is divided into two parts: an interactive setup protocol VOLE.Setup that produces short seeds for each party, and an expansion protocol VOLE.Expand that involves only local computation in which each party expands the short seed it has obtained from the setup to generate its long output vectors. It was shown that if these two phases satisfy Definition 2.3, the resulting pseudorandom correlation can securely be used for various applications of VOLE, such as secure arithmetic computation [9, 11].

The idea of the construction of Boyle et al. [9] is to start from short vector $\mathbf{a}, \mathbf{b}$ and $\mathbf{c}$ of length $k < n$ that have the required correlation, i.e., $\mathbf{c} = \mathbf{a}x + \mathbf{b}$, which the two parties can generate efficiently using MPC, and to expand them to long pseudorandom vectors using the LPN assumption. This assumption states that for appropriate code generating matrix $\mathbf{C} \in \mathbb{F}^{k \times n}$, the vector $\mathbf{u} = \mathbf{a} \cdot \mathbf{C} + \boldsymbol{\mu}$ is pseudorandom, where $\boldsymbol{\mu}$ is a sparse random vector. Now if we compute $\mathbf{v} = \mathbf{b} \cdot \mathbf{C} - \boldsymbol{\nu}_1$ and $\mathbf{w} = \mathbf{c} \cdot \mathbf{C} + \boldsymbol{\nu}_2$, where $\boldsymbol{\nu}_1$ and $\boldsymbol{\nu}_2$ are shares of $\boldsymbol{\mu}x$, we will achieve the correctness property that $\mathbf{u}x + \mathbf{v} = \mathbf{w}$. Additionally, in order to get security, we need that $\boldsymbol{\nu}_1$ and $\boldsymbol{\nu}_2$ are pseudorandom and do not reveal any information about $\boldsymbol{\mu}x$. This guarantees the pseudorandom properties of $\mathbf{u}$ and $\mathbf{v}$ under the correlation and that $\boldsymbol{\nu}_1$ (and hence $\mathbf{u}$ and $\mathbf{v}$) does not reveal any information about x.

Given the above idea, the heart of the VOLE generation is obtaining the shares $\boldsymbol{\nu}_1$ and $\boldsymbol{\nu}_2$ in a communication efficient manner. Boyle et al. [9] propose using a distributed multi-point FSS protocol. Our observation is that this functionality is more than what is needed for the pseudorandom VOLE construction. More specifically, an FSS protocol will guarantee that both shares $\boldsymbol{\nu}_1$ and $\boldsymbol{\nu}_2$ do not reveal any information about the multi-point function defined by $\boldsymbol{\mu}x$. However, while x needs to remain hidden, $\boldsymbol{\mu}$ is revealed to P_1, which in turn reveals the non-zero indices of $\boldsymbol{\mu}x = \boldsymbol{\nu}_1 + \boldsymbol{\nu}_2$. This observation is what allows us to use our known index MPFSS from Section 5 to generate the shares $\boldsymbol{\nu}_1$ and $\boldsymbol{\nu}_2$ more efficiently.

Protocol 6: Distributed Vector OLE

Public Params: Vector length n, LPN parameters t, k, code generating matrix $\mathbf{C} \in \mathbb{F}^{k \times n}$.

Parties: P_1, P_2.

Inputs: None.

Outputs: $\mathsf{P}_1 : \mathbf{u}, \mathbf{v} \in \mathbb{F}^n$; $\mathsf{P}_2 : \mathbf{w} \in \mathbb{F}^n$, $x \in \mathbb{F}$, such that $\mathbf{u}x + \mathbf{v} = \mathbf{w}$.

Share Generation ($\mathsf{VOLE.Setup}(1^\lambda, \mathbb{F}, n)$**)**

(1) P_1 chooses a set of S random positions $S = \{s_1, \ldots, s_t\}$, with $s_i \in [n]$, t random values $\mathbf{y} = (y_1, \ldots, y_t) \in \mathbb{F}^t$, and a pair of random vectors $\mathbf{a}, \mathbf{b} \in \mathbb{F}^k$. P_2 chooses random $x \in \mathbb{F}$.
(2) P_1 and P_2 run MPFSS.Gen to obtain keys K_1, K_2 of the multi-point function $f_{S,x\mathbf{y}}$.
(3) P_1 and P_2 run an MPC with inputs $\mathbf{a}, \mathbf{b}$ and x respectively, from which P_2 obtains a vector $\mathbf{c} = \mathbf{a}x + \mathbf{b}$.
(4) P_1 outputs $\mathsf{seed}_1 \leftarrow (K_1, S, \mathbf{y}, \mathbf{a}, \mathbf{b})$ and P_2 outputs $\mathsf{seed}_2 \leftarrow (K_2, x, \mathbf{c})$.

Expansion ($\mathsf{VOLE.Expand}(b, \mathsf{seed}_b)$**)**

(i) If $b = 1$, P_1 runs $\boldsymbol{\nu}_1[i] \leftarrow \mathsf{MPFSS.Eval}(1, K_1, i)$ for $i \in [n]$ and defines a vector $\boldsymbol{\mu} \in \mathbb{F}^n$ such that $\boldsymbol{\mu}[s_i] = y_i$ for all $i \in [t]$ and $\boldsymbol{\mu}[s] = 0$ for all $s \notin S$. P_1 outputs $\mathbf{u} = \mathbf{a} \cdot \mathbf{C} + \boldsymbol{\mu}, \mathbf{v} = \mathbf{b} \cdot \mathbf{C} - \boldsymbol{\nu}_1$.
(ii) If $b = 2$, P_2 runs $\nu_2[i] \leftarrow \mathsf{MPFSS.Eval}(2, K_2, i)$ for $i \in [n]$ and outputs $\mathbf{w} = \mathbf{c} \cdot \mathbf{C} + \boldsymbol{\nu}_2$.

We note that as discussed in Section 5, our batching scheme introduces a small probability $2^{-\eta}$ of failing to batch all t non-zero indices. This is also the case for the heuristic batch code construction of Boyle et al. [9]. However, as also pointed out there, this only strengthens the required LPN assumption a little: If batching fails (which results in some elements of $\boldsymbol{\mu}x$ becoming zero instead of nonzero), the distribution of noise values will only slightly deviate from uniform, but LPN for such a distribution remains a very conservative assumption.

Theorem 6.1. *Protocol 6 implements a secure distributed vector OLE generator in the semi-honest model. With step (3) instantiated with OT-based Gilboa multiplication,* MPFSS *instantiated using Protocol 5, and* $\mathbf{C}$ *instantiated by a local linear code, the protocol is constant round, and requires* $O(\lambda m \log n + \lambda k)$ *communication and* $O(\lambda \kappa n + \lambda m \log n)$ *computation per party, where* $(m, \kappa) \leftarrow \mathsf{ParamGen}(n, t, \eta)$*,* λ *is a computational security parameter, and* η *is the statistical security parameter of the MPFSS scheme.*

Proof Sketch. As mentioned above, our protocol is obtained using a simple modification of the scheme of Boyle et al. [9], i.e., using known-index MPFSS instead of full MPFSS. Since the only additional information our variant reveals is already included in the VOLE keys, their proof [9, Section 3.2.2] can be trivially adapted to our protocol. We will give an overview here, but refer the reader to [9] for the full details.

Correctness follows from the observation that $\boldsymbol{\mu}x = \boldsymbol{\nu}_1 + \boldsymbol{\nu}_2$. It follows that

$$\begin{aligned}\mathbf{u}x + \mathbf{v} &= (\mathbf{a} \cdot \mathbf{C} + \mu)x + \mathbf{b} \cdot \mathbf{C} - \boldsymbol{\nu}_1 \\ &= (\mathbf{a}x + \mathbf{b})\mathbf{C} + \boldsymbol{\mu}x - \boldsymbol{\nu}_1 = \mathbf{c} \cdot \mathbf{C} + \nu_2 = \mathbf{w}.\end{aligned}$$

To prove security we need to show that the two security properties from Definition 2.3 hold. To show the first property we observe that the only part of seed_1 that depends on x is K_1. However, since it is generated using the distributed MPFSS construction, it follows by the security of known-index MPFSS (see Appendix A.3) that there is a simulator that can simulate K_1 without knowledge of x. Note that the non-zero indices needed to simulate K_1 are also included in seed_1.

To prove the second property we show a transition between the distributions $(\mathbf{u}_1, \mathbf{v}_1, \text{seed}_2)$ and $(\mathbf{u}_2, \mathbf{v}_2, \text{seed}_2)$ in two steps and argue that an adversary cannot distinguish the changes applied in each of them. In the first step the input to the adversary is the same but we replace the K_2 with the simulated MPFSS key, which is generated from $\mathbb{F}$ and n alone. Security of the MPFSS scheme guarantees that this simulated key is indistinguishable from the real one. In this distribution $\mathbf{u}_1 = \mathbf{a} \cdot \mathbf{C} + \boldsymbol{\mu}$ and $\mathbf{v}_1 = \mathbf{b} \cdot \mathbf{C} - \boldsymbol{\nu}_1 = \mathbf{b} \cdot \mathbf{C} + \boldsymbol{\nu}_2 - \boldsymbol{\mu}x = \mathbf{c} \cdot \mathbf{C} + \boldsymbol{\nu}_2 - (\mathbf{a} \cdot \mathbf{C} + \boldsymbol{\mu})x = \mathbf{c} \cdot \mathbf{C} + \boldsymbol{\nu}_2 - \mathbf{u}_1 x$. In the next step we replace $\mathbf{u}_1$ and $\mathbf{v}_1$ with $\mathbf{u}_2 \xleftarrow{R} \mathbb{F}^n$ and $\mathbf{v}_2 \leftarrow \mathbf{w} - \mathbf{u}_2 x = \mathbf{c} \cdot \mathbf{C} + \boldsymbol{\nu}_2 - \mathbf{u}_2 x$. By the LPN assumption, $\mathbf{u}_1$ and $\mathbf{u}_2$ are indistinguishable and since $\mathbf{v}_1$ and $\mathbf{v}_2$ are computed in the same way, the change in the second step in indistinguishable for the adversary.

The communication in the protocol consists of the execution of the distributed MPFSS key generation and the secure computation for $\mathbf{c}$, which have cost $O(\lambda m \log n)$ and $O(\lambda k)$, respectively. The computation overhead additionally consists of the expansion of the MPFSS, which is $O(\lambda \kappa n)$ and the vector matrix multiplications with the matrix C, which using a local linear code is in $O(n)$.

In our evaluation (Section 8), rely on previous work [21] to choose a constant κ and $m = O(t)$ such that $\eta \geq 40$ for the parameter ranges we're interested in. Together with the observation that t and k are both in $O(\sqrt{n})$ [9], this simplifies the communication overhead of our protocol to $O(\lambda \sqrt{n} \log n)$ and the computation to $O(\lambda n)$.

7 APPLICATIONS

Our distributed pseudorandom vector OLE protocol can be seen as a communication efficient precomputation that enables arbitrary secure two-party scalar-vector multiplications. This is thanks to a simple reduction from VOLE to pseudorandom vector OLE. The reduction is analogous to how a random multiplication triple can be exploited to compute extremely efficiently a secure multiplication in just a round of communication. The reduction from VOLE to pseudorandom VOLE is given in [9] (Proposition 10). The overhead of the reduction with respect to running pseudorandom VOLE generation and expanding the resulting seeds is just the cost of performing the scalar vector multiplication in the clear, and transmitting a vector of the same length as the input vector. For that reason, in the context of multi-party computation, distributed pseudorandom vector OLE should be considered as a data independent preprocessing step that enables fast secure distributed scalar-vector multiplication, aka vector OLE. In this section we overview some applications that fit in this paradigm and thus can benefit from our protocol for distributed vector-OLE, as well as applications of our sub-protocol for known-Indices MPFSS.

Generally speaking, vector OLE can be used to batch one-against-many OLE computations, and thus directly provides a way to batch applications that rely on OLE computations. Such applications include, for example, PSI [29], and keyword search [27]. The latter relies on Oblivious Polynomial Evaluation (OPE) for which, as we will discuss later in this section, an efficient reduction to vector OLE exists. From a general MPC perspective, vector OLE enables communication efficient evaluation of arithmetic circuits with multiplication gates with large fan-out. This includes several important settings, including protocols for secure distributed data analysis.

7.1 Secure Linear algebra

As mentioned above, vector OLE is directly applicable in settings were OLE computations, i.e., secure multiplications, can be *vectorized* and thus computed by invoking several instances of vector OLE. This is the case, for example, in matrix-vector multiplication, as this operation can be computed, for a matrix of dimensions $n \times m$, by m invocations of length n vector OLE. Hence interesting settings for our protocols are the ones where n is a lot larger than m. This corresponds to datasets with many records, and a limited number of features per record, which are natural in the context of training and evaluation of machine learning models, such as logistic regression. Similarly, matrix convolutions operations, the main ingredient of convolutional neural networks, rely on multiplying a small matrix called kernel (common kernel sizes are 3×3, 5×5, and 9×9) in a sliding fashion at each position of a input image (or layer input for intermediate layers). This corresponds to a small number of vector OLE computations of length the size of the image (which is commonly 255×255).

A natural approach to distributed vector OLE is (vectorized) Gilboa multiplication, as discussed in Section 2, and thus it has been used as a way to precompute multiplication triples for MPC in several works [22, 28, 44]. This approach requires linear communication and computation in the size of the matrix. In contrast our Protocol 6 has sub-linear communication. In Section 8 we compare these two approaches empirically, both in terms of communication and computation.

7.1.1 Sparse matrix manipulations. As mentioned in the introduction, known-index MPFSS can also be seen as a type of "scatter" vector operation. This functionality was presented by Schoppmann et al. [51] under the name of "Scatter-Init". In that setting, two parties hold a share of a sparse vector, represented as a list of index-value pairs for which one party knows the indices and the values are additively shared. The goal is to securely convert the vector into a dense representation, where it is represented as an array of shared values of length the size of the domain of indices. This conversion was used in the context of a sparse matrix multiplication protocol that enables an efficient protocol for two-party secure gradient descent on sparse training data. Schoppmann et al. [51] propose a protocol for known-Index MPFSS based on full-blown FSS that lacks an efficient batching strategy, and thus incurs $O(ln)$ for a length l sparse vector over a domain of indices of size n. The value of l in their applications is such that that cost is prohibitive, while

our MPFSS only requires $O(n)$ local computation and improves significantly the efficiency of these functionalities.

7.2 Oblivious Polynomial Evaluation

The problem of oblivious polynomial evaluation (OPE) considers the setting where one party, the server, has the coefficients of a polynomial $P(x)$ and a second party, the client, has an input z and the goal of the protocol is to enable the client to learn $P(z)$ without learning anything more about the polynomial and without the server learning anything about the input. OPE has applications to privacy preserving set operations and data comparison, anonymous initialization for metering and anonymous coupons [46]. The OPE setting can be viewed as a generalization of the OLE problem to a higher degree polynomial.

We show that we can implement the OPE protocol leveraging the VOLE functionality. In order to this we use the OPE construction introduced in the works of Naor and Pinkas [46] and Gilboa [30]. The idea of these constructions is to reduce the evaluation of a degree n polynomial to n evaluations of linear polynomials, which can be executed in parallel. Next we overview the main idea of the reduction. Let $P(x) = a_n x^n + \cdots + a_1 x + a_0$ be a degree n polynomial. It can be expressed as $P(x) = xQ(x) + b_0$ where $Q(x)$ is a degree $n-1$ polynomial. If the client and the server have obtained respectively additive shares q_C, q_S of the evaluation of $Q(x) = q_C + q_S$, then $P(x) = q_C x + q_S x + b_0$. If the server fixes its share q_S in advance, then the client's share $q_C = Q(x) - q_S = Q'(x)$ can be computed using oblivious polynomial evaluation of $Q'(x)$, which is of degree $n-1$ and its coefficients are known to the server. Now $P(x) = xQ'(x) + P'(x)$ where $P'(x) = q_S x + b_0$ is a linear polynomial. Therefore the OPE of $P(x)$ reduces to the oblivious evaluation of $Q'(x)$ and $P'(x)$, which can be done in parallel. By induction we obtain that the evaluation of $P(x)$ can be reduced to the parallel evaluation of n linear polynomials of the forms $w_i = P_i(x) = u_i x + v_i$ for $i \in [n]$ where the server knows the values $(u_i, v_i)_{i\in[n]}$ and the client knows x and obtains $\{w_i\}_{i\in[n]}$. These corresponds to n OLE evaluations, with the crucial aspect that one of the inputs is common to all of them. Hence an OPE of degree n can be implemented with a single vector OLE computation where the server has two vectors of length n: $\mathbf{u}$ and $\mathbf{v}$, which consist of the values $\{u_i\}_{i\in[n]}$ and $\{v_i\}_{i\in[n]}$ respectively, and the client obtains $\mathbf{w} = \mathbf{u}x + \mathbf{v}$, which contains the values $\{w_i\}_{i\in[n]}$.

7.3 Partially Private Distributed ORAM

Doerner and shelat [23] presented a distributed ORAM construction that has asymptotically linear access time but achieves practically very competitive concrete efficiency. This advantage is even more pronounced in the RAM secure computation setting where this ORAM construction is used for memory access and the access queries are executed jointly by the two parties. The authors also consider semi-private queries which consist of both data dependent and data independent queries. In the latter type the parties know the accessed index. For these types of queries the FLORAM construction enables access in constant time.

We consider semi-private queries where the query index is known only to one of the parties. This corresponds to situations when data held by one party is indexed at private locations by the other party. We show that in this setting we can use our SPFSS construction and avoid having a Write-Only ORAM structure in the overall construction.

First, we briefly overview the FLORAM construction [23]. The ORAM in this construction consists of a Read-only ORAM, a Write-Only ORAM and a stash. The Read-Only ORAM consists of encryptions of the data under a key shared among the two parties. Each party has a copy of the Read-Only ORAM. The two parties execute an access query using a two server PIR construction based on SPFSS to retrieve the corresponding data item. They generate the distributed query running the distributed FSS key generation. The Write-Only ORAM consists of two XOR shares of the database, where each party holds one of the shares. It is updated with a write for a new item again using an SPFSS which evaluates to a non-zero value at the location of the write and this evaluation there is the XOR of the old value and the new value. The stash contains all the items that are currently in Write-Only ORAM. An ORAM access that hides read and writes consists of one Read-Only ORAM access, and one addition to the stash of the item that is written. Periodically all the content of the Write-Only ORAM is moved to the Read-Only ORAM using a special protocol with linear communication.

We observe that in setting of partially private queries where one of the parties knows the access index we can use our distributed only shared value FSS key generation presented in Section 4. This results in an improvement in terms of round communication, as the general SPFSS construction by Doerner and Shelat requires a logarithmic number of rounds. In Section 8 we show empirically the benefits of using our variant in the specific setting of semi-private queries by comparing two implementations of these protocols. Our results show improvements of up to an order of magnitude.

8 EXPERIMENTAL EVALUATION

8.1 Implementation and Setup

We implement all the protocols needed for Vector-OLE (Protocols 2, 3, 5, 6). Our implementation[1] is written in C++. For OT extension we use EMP [54], for finite field computations we use NTL [52], and for matrix multiplications needed in Protocol 6 we rely on Eigen [33]. We use AES to implement the PRG needed for Protocol 2. Just as the FLORAM implementation of Doerner and shelat [23], we rely on the Davies-Meyer construction [55] to avoid repeated expansions of AES keys. We further interleave the setup and expansion phases in our implementation, and therefore only report the total time in each of our experiments.

All our experiments are done on Azure Dsv3 machines in the same region, using 2.4 GHz Intel Xeon E5-2673 v3 CPUs. For our comparisons against other protocols, we used a single thread. Note that this does not penalize any protocol in particular, since their local computations all parallelize well. To show the scalability of our protocol, we also implement a parallel version of it using OpenMP [17].

8.2 Parameter Selection

In our experiments, we use $\lambda = 128$ as the computational security parameter. Following the analysis in [9, Section 5.1], we choose

[1] Source code available at https://github.com/schoppmp/distributed-vector-ole.

n	2^{14}	2^{16}	2^{18}	2^{20}	2^{22}	2^{24}
t	192	382	741	1422	2735	5205
k	3482	7391	15336	32771	67440	139959

Table 1: Vector-OLE parameters we chose in our evaluation. These were computed by Boyle et al. such that solving the corresponding LPN instance requires at least 2^{80} operations using either low-weight parity check, Gaussian elimination, or Information Set Decoding [9].

the parameters for Vector-OLE (i.e., number of noise indices t and number of rows in the code matrix k) such that known attacks on LPN require at least 2^{80} arithmetic operations. The concrete parameters depending on the vector size n are given in Table 1. To instantiate the code generator $C \in \mathbb{F}^{k \times n}$, we choose a local linear code with $d = 10$ non-zeros per column, which is also suggested by previous work on Vector-OLE from LPN [2, 9]. Finally, we rely on the estimates in [21, Appendix B] to choose cuckoo hashing parameters such that hashing of the t random indices fails with probability at most 2^{-40}, i.e., $\eta = 40$. For the values of t in Table 1 and $\kappa = 3$, this yields $m = 1.5t$. Those exact parameters have also been used in a previous work that uses cuckoo hashing for batching [1].

8.3 Results

8.3.1 Comparison of Known-Index SPFSS with FLORAM. First, we compare our distributed Known-Index SPFSS variant (Protocol 3) with the SPFSS implementation of Doerner and Shelat [23] in order to demonstrate the efficiency gain we can obtain in settings where the index might be known to one of the parties, e.g., semi-private accesses. The results are shown in Figure 3. Our implementation performs better for all vector lengths we tested. For short vectors, this is not surprising given that our protocol does not require expensive garbled circuits, but only $\log(n)$ oblivious transfers. Even for large vectors, where both protocols take time approximately linear in n, our implementation remains very efficient, which is made possible by the simplicity of our construction.

8.3.2 Vector-OLE computation. We also measure the time it takes to generate a full Vector-OLE. Here, we compare our implementation of Protocol 6 against three baselines. First, the approach proposed in [9], i.e., using FLORAM's FSS implementation in each bucket, but using our probabilistic batch codes. Second, our variant of MPFSS, but using naive batching by repeatedly evaluating over the whole domain. And third, our own implementation of Gilboa's multiplication protocol [31]. We already heavily optimized this second baseline. In particular, we employ all of the optimizations from [44], and our time per single-element multiplication is lower than the one reported in [44].

Figure 4 (left) shows a comparison of wall-clock running times of all three approaches. First, it becomes obvious that using FLORAM is not practical when compared to either of the alternatives. While its asymptotic running time matches ours, it is consistently slower by a large factor. Our second baseline, known-indices MPFSS with naive batching, outperforms FLORAM for small vectors, but it is

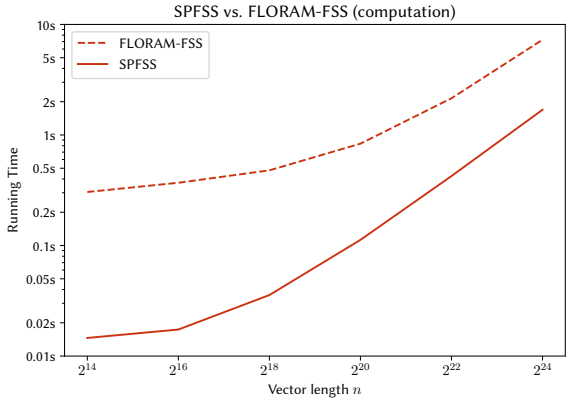

Figure 3: Comparison of our single-point FSS variant (Section 4) with the implementation of [23].

worse than both Gilboa and our VOLE in terms of asymptotics as well as concrete efficiency. As for the third baseline, Gilboa's multiplication is faster than our protocol for vector lengths below 300k (finite fields) and 2M (integer rings). The large discrepancy here is due to the fact that in addition to the reduced computational overhead from the lack of modular reductions, using 64-bit integers directly allows us to use correlated OT [3].

8.3.3 Communication Experiments. We also investigate the communication overhead of both our VOLE implementation and Gilboa multiplication. To that end, we artificially limit the bandwidth of our machine to 100Mbit/s, which is about the download bandwidth of a consumer household connection. We measure running time and the number of bytes sent by both parties during the protocol execution. The results are shown in Figure 4 (middle) and (right). Compared to Figure 4 (left), the cutoff point where our protocol outperforms Gilboa is lower, at about 200k for finite fields, and 400k for integer rings.

8.3.4 Parallelization of VOLE. Finally, we investigate the effect of parallelization on our VOLE protocol. Figure 5 shows the results for $n = 2^{20}$. With 8 threads we observe a speedup of about 5x, with 32 threads this increases to over 10x. While we did not run experiments on more than 32 cores, the slope of the plot suggests that the running time can be further reduced with additional hardware parallelism.

9 CONCLUSION

Our work presents a new protocol for shared randomness generation in the form of a random vector oblivious linear evaluation, which generates vectors with linear correlations. On the way to our final construction we also developed several new protocols, which are of independent interest, in the areas of random OT, private puncturable PRFs, and function secret sharing for single and multi-point functions with known indices. We showed how our VOLE construction can be leveraged in the context of several secure computation constructions, and compare them experimentally with two alternatives.

A possible improvement can be found in our lowest-level primitive, $(n-1)$-out-of-n-ROT. While our construction is based on

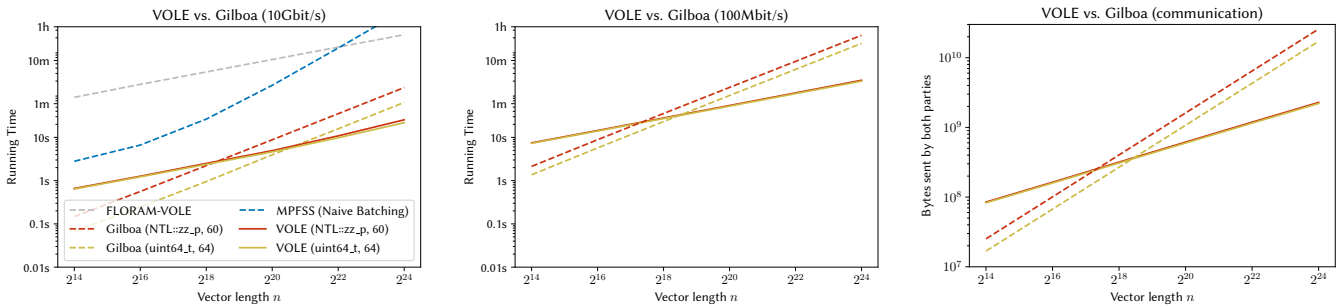

Figure 4: (Left) running time of our Vector-OLE implementation (*VOLE*) for generating a single random Vector-OLE. We compare against three baselines: A variant using our batching techniques, but the DPF implementation proposed by Boyle et al. [9], which is also used in FLORAM [23]; our known-indices MPFSS, but using naive batching (cf. Section 5); and Gilboa multiplication [31], which is also commonly used in the literature to implement two-party multiplications [22, 44]. We also compare two multiplication types: a 60-bit finite field (`NTL::zz_p`), and a 64-bit integer ring (`uint64_t`). It can be seen that VOLE outperforms the first two baselines, and is faster than Gilboa for vector lengths above 300k (prime field) and 2M (integer ring). (Middle) running time of our VOLE and Gilboa multiplication in a *bandwidth-constrained* setting. Here, our implementation is already more efficient for much smaller vectors, at 200k in the prime field, and 400k in the integer ring. (Right) communication overhead of our Vector-OLE and Gilboa multiplication. The cutoff points where our implementation outperforms the baseline are similar to the running time plot in the middle.

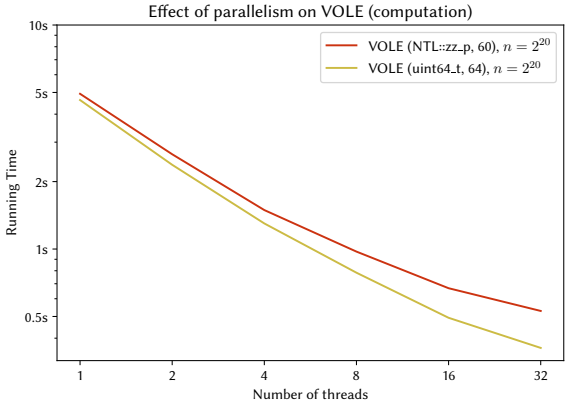

Figure 5: Running time of a Vector-OLE with varying number of threads.

GGM trees with arbitrary arity, our implementation is limited to binary trees and 1-out-of-2-OT. We believe that using efficient $(k-1)$-out-of-k-OT sub-protocols from homomorphic encryption for larger k, we can gain additional concrete efficiency.

In terms of asymptotics, to our knowledge, ours is the first implementation of Vector OLE with sub-linear communication. However, it does not reach the asymptotical guarantees that alternative constructions (in particular the dual version by Boyle et al.) provide, namely poly-logarithmic communication. This is due to the lack of concretely efficient, LPN-friendly encoding schemes, and we believe that if such encoding schemes become available, our implementation can yield poly-logarithmic communication complexity while staying concretely efficient.

ACKNOWLEDGMENTS

We would like to give special thanks to Benny Pinkas for his help during the development of this paper. We also thank Geoffroy Couteau and Mike Rosulek for helpful discussions. Adrià Gascón's work on this paper was done while at The Alan Turing Institute, supported by EPSRC grant EP/N510129/1, and funding from the UK Government's Defence & Security Programme in support of the Alan Turing Institute.

REFERENCES

[1] Sebastian Angel, Hao Chen, Kim Laine, and Srinath T. V. Setty. 2018. PIR with Compressed Queries and Amortized Query Processing. In *IEEE Symposium on Security and Privacy*. IEEE, 962–979.
[2] Benny Applebaum, Ivan Damgård, Yuval Ishai, Michael Nielsen, and Lior Zichron. 2017. Secure Arithmetic Computation with Constant Computational Overhead. In *CRYPTO (1)*. Springer, 223–254.
[3] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. 2013. More efficient oblivious transfer and extensions for faster secure computation. In *CCS*. ACM, 535–548.
[4] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. 2015. More Efficient Oblivious Transfer Extensions with Security for Malicious Adversaries. In *EUROCRYPT (1)*. Springer, 673–701.
[5] Donald Beaver. 1991. Efficient Multiparty Protocols Using Circuit Randomization. In *CRYPTO*. Springer, 420–432.
[6] Donald Beaver. 1996. Correlated Pseudorandomness and the Complexity of Private Computations. In *STOC*. ACM, 479–488.
[7] Avrim Blum, Adam Kalai, and Hal Wasserman. 2003. Noise-tolerant learning, the parity problem, and the statistical query model. *J. ACM* 50, 4 (2003), 506–519.
[8] Dan Boneh, Kevin Lewi, and David J. Wu. 2017. Constraining Pseudorandom Functions Privately. In *Public Key Cryptography (2)*. Springer, 494–524.
[9] Elette Boyle, Geoffroy Couteau, Niv Gilboa, and Yuval Ishai. 2018. Compressing Vector OLE. In *CCS*. ACM, 896–912.
[10] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, Peter Rindal, and Peter Scholl. 2019. Efficient Two-Round OT Extension and Silent Non-Interactive Secure Computation. In *CCS*. ACM, 291–308.
[11] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. 2019. Efficient Pseudorandom Correlation Generators: Silent OT Extension and More. In *CRYPTO (3)*. Springer, 489–518.
[12] Elette Boyle, Niv Gilboa, and Yuval Ishai. 2015. Function Secret Sharing. In *EUROCRYPT (2)*. Springer, 337–367.
[13] Elette Boyle, Niv Gilboa, and Yuval Ishai. 2016. Function Secret Sharing: Improvements and Extensions. In *CCS*. ACM, 1292–1303.

[14] Ran Canetti. 2000. Security and Composition of Multiparty Cryptographic Protocols. *J. Cryptology* 13, 1 (2000), 143–202.
[15] Hao Chen, Kim Laine, and Peter Rindal. 2017. Fast Private Set Intersection from Homomorphic Encryption. In *CCS*. ACM, 1243–1255.
[16] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. 1995. Private Information Retrieval. In *FOCS*. IEEE Computer Society, 41–50.
[17] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: An industry-standard API for shared-memory programming. *Computing in Science & Engineering* 1 (1998), 46–55.
[18] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. 2013. Practical Covertly Secure MPC for Dishonest Majority - Or: Breaking the SPDZ Limits. In *ESORICS*. Springer, 1–18.
[19] Ivan Damgård and Sunoo Park. 2012. Is Public-Key Encryption Based on LPN Practical? *IACR Cryptology ePrint Archive* (2012), 699.
[20] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. 2012. Multiparty Computation from Somewhat Homomorphic Encryption. In *CRYPTO*. Springer, 643–662.
[21] Daniel Demmler, Peter Rindal, Mike Rosulek, and Ni Trieu. 2018. PIR-PSI: Scaling Private Contact Discovery. *PoPETs* 2018, 4 (2018), 159–178.
[22] Daniel Demmler, Thomas Schneider, and Michael Zohner. 2015. ABY - A Framework for Efficient Mixed-Protocol Secure Two-Party Computation. In *NDSS*. The Internet Society.
[23] Jack Doerner and abhi shelat. 2017. Scaling ORAM for Secure Computation. In *CCS*. ACM, 523–535.
[24] Nico Döttling, Satrajit Ghosh, Jesper Buus Nielsen, Tobias Nilges, and Roberto Trifiletti. 2017. TinyOLE: Efficient Actively Secure Two-Party Computation from Oblivious Linear Function Evaluation. In *CCS*. ACM, 2263–2276.
[25] Nico Döttling, Daniel Kraschewski, and Jörn Müller-Quade. 2012. David & Goliath Oblivious Affine Function Evaluation - Asymptotically Optimal Building Blocks for Universally Composable Two-Party Computation from a Single Untrusted Stateful Tamper-Proof Hardware Token. *IACR Cryptology ePrint Archive* (2012), 135.
[26] Michael J. Freedman, Carmit Hazay, Kobbi Nissim, and Benny Pinkas. 2016. Efficient Set Intersection with Simulation-Based Security. *J. Cryptology* 29, 1 (2016), 115–155.
[27] Michael J. Freedman, Yuval Ishai, Benny Pinkas, and Omer Reingold. 2005. Keyword Search and Oblivious Pseudorandom Functions. In *TCC*. Springer, 303–324.
[28] Adrià Gascón, Phillipp Schoppmann, Borja Balle, Mariana Raykova, Jack Doerner, Samee Zahur, and David Evans. 2017. Privacy-Preserving Distributed Linear Regression on High-Dimensional Data. *PoPETs* 2017, 4 (2017), 345–364.
[29] Satrajit Ghosh and Tobias Nilges. 2019. An Algebraic Approach to Maliciously Secure Private Set Intersection. In *EUROCRYPT (3)*. Springer, 154–185.
[30] Niv Gilboa. [n. d.]. Private Communication.
[31] Niv Gilboa. 1999. Two Party RSA Key Generation. In *CRYPTO*. Springer, 116–129.
[32] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. 1986. How to construct random functions. *J. ACM* 33, 4 (1986), 792–807.
[33] Gaël Guennebaud, Benoît Jacob, et al. 2010. Eigen v3. http://eigen.tuxfamily.org.
[34] Stefan Heyse, Eike Kiltz, Vadim Lyubashevsky, Christof Paar, and Krzysztof Pietrzak. 2012. Lapin: An Efficient Authentication Protocol Based on Ring-LPN. In *FSE*. Springer, 346–365.
[35] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. 2003. Extending Oblivious Transfers Efficiently. In *CRYPTO*. Springer, 145–161.
[36] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. 2004. Batch codes and their applications. In *STOC*. ACM, 262–271.
[37] Jonathan Katz and Yehuda Lindell. 2014. *Introduction to Modern Cryptography* (2 ed.). Chapman and Hall/CRC Press.
[38] Marcel Keller, Emmanuela Orsini, and Peter Scholl. 2016. In *CCS*. ACM, 830–842.
[39] Marcel Keller, Valerio Pastro, and Dragos Rotaru. 2018. Overdrive: Making SPDZ Great Again. In *EUROCRYPT (3)*. Springer, 158–189.
[40] Joe Kilian. 1988. Founding Cryptography on Oblivious Transfer. In *STOC*. ACM, 20–31.
[41] Adam Kirsch, Michael Mitzenmacher, and Udi Wieder. 2009. More Robust Hashing: Cuckoo Hashing with a Stash. *SIAM J. Comput.* 39, 4 (2009), 1543–1561.
[42] Vladimir Kolesnikov, Ranjit Kumaresan, Mike Rosulek, and Ni Trieu. 2016. Efficient Batched Oblivious PRF with Applications to Private Set Intersection. In *CCS*. 818–829.
[43] Yehuda Lindell and Benny Pinkas. 2009. A Proof of Security of Yao's Protocol for Two-Party Computation. *J. Cryptology* 22, 2 (2009), 161–188.
[44] Payman Mohassel and Yupeng Zhang. 2017. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 19–38.
[45] Moni Naor and Benny Pinkas. 1999. Oblivious Transfer and Polynomial Evaluation. In *STOC*. ACM, 245–254.
[46] Moni Naor and Benny Pinkas. 2006. Oblivious Polynomial Evaluation. *SIAM J. Comput.* 35, 5 (2006), 1254–1281.
[47] Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo hashing. *J. Algorithms* 51, 2 (2004), 122–144.
[48] Benny Pinkas, Thomas Schneider, and Michael Zohner. 2014. Faster Private Set Intersection Based on OT Extension. In *USENIX Security Symposium*. USENIX Association, 797–812.
[49] Benny Pinkas, Thomas Schneider, and Michael Zohner. 2018. Scalable Private Set Intersection Based on OT Extension. *ACM Trans. Priv. Secur.* 21, 2 (2018).
[50] Michael O. Rabin. 1981. How To Exchange Secrets with Oblivious Transfer. *TR-81 edition, Aiken Computation Lab, Harvard University* (1981).
[51] Phillipp Schoppmann, Adrià Gascón, Mariana Raykova, and Benny Pinkas. 2019. Make Some ROOM for the Zeros: Data Sparsity in Secure Distributed Machine Learning. In *CCS*. ACM.
[52] Victor Shoup et al. 2001. NTL: A library for doing number theory. https://www.shoup.net/ntl.
[53] Abraham Waksman. 1968. A Permutation Network. *J. ACM* 15, 1 (1968), 159–163.
[54] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. 2016. EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit.
[55] Robert S. Winternitz. 1984. A Secure One-Way Hash Function Built from DES. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 88–90.
[56] Andrew Chi-Chih Yao. 1986. How to Generate and Exchange Secrets (Extended Abstract). In *FOCS*. IEEE Computer Society, 162–167.

A SECURITY PROOFS

In this section, we will prove security of all our main constructions, that is, Protocols 2, 3, and 4. We do not provide a full proof for Protocol 6, but as we discuss in Section 6, this proof can be obtained directly by taking the one given by Boyle et al. [9] and replacing their MPFSS construction by ours. As described in Section 2.6, we split our proofs in three phases, i.e., we (i) define correctness and security requirements, (ii) define ideal functionalities that satisfy these requirements, and (iii) prove our key generation protocols securely implement the ideal functionalities.

We note that our definitions are also closely related to the Pseudorandom Correlation Generators (PCGs) of Boyle et al. [11]. However, as our key generators take additional arguments beyond the security parameter, we cannot use their definition out-of-the-box. Still, our $(n-1)$-out-of-n-ROT is defined in a similar way as PCGs. For our FSS variants, we stick to pure simulation-based proofs using Definition 2.2, which ensures they cen be used as a drop-in replacement for the constructions of Boyle et al. [9].

A.1 $(n-1)$-out-of-n-ROT

Definition A.1 (Pseudorandom $(n-1)$-out-of-n-OT Generator). A pseudorandom $(n-1)$-out-of-n-OT generator for a group $\mathbb{G}$ consists of the following two algorithms:

- $(K_1, K_2) \leftarrow \mathsf{ROT.Gen}(1^\lambda, n, i)$ - Outputs two keys when given an output size n and a single index $i \in [n]$.
- $\mathbf{v}^b \leftarrow \mathsf{ROT.Expand}(b, K_b)$ - Given an evaluation key K_b for $b \in \{1, 2\}$, outputs a vector of length n.

Here, $\lambda \in \mathbb{N}$ denotes a security parameter. Additionally, the following properties must hold:

Correctness. For any $n \in \mathbb{N}$ and $i \in [n]$, any pair (K_1, K_2) in the image of $\mathsf{ROT.Gen}(1^\lambda, n, i)$, and $\mathbf{v}^b \leftarrow \mathsf{ROT.Expand}(b, K_b)$ for $b \in \{1, 2\}$, we have that $\mathbf{v}^1$ is computationally indistinguishable from a uniformly random vector from $\mathbb{G}^n$, and $v^1_j = v^2_j$ for all $j \in [n] \setminus \{i\}$.

Functionality 7: $n-1$-out-of-n-ROT

Public Parameter: k

Key Generation ($\mathsf{ROT.Gen}(1^\lambda, n, i)$**):**

(1) Run steps (1) and (2) from Protocol 2 as P_1 to obtain a k-ary GGM tree T with root s_0 and depth $\alpha = \log_k(n)$, using seeds of size λ.

(2) For each level $l \in [\alpha]$, let $(p_1, \ldots, p_{k^l})$ be the seeds of the lth level of T, and for each $j \in [k] \setminus \{b_l\}$, compute

$$q_{l,j} \leftarrow \bigoplus_{s \in \{p_x \,:\, x \equiv j \bmod k\}} s.$$

(3) Let $(b_1, \ldots, b_\alpha)$ be a k-ary encoding of $i-1$. Return $K_1 \leftarrow s_0$ and $K_2 \leftarrow \left(i, (q_{l,j})_{l \in [\alpha], j \in [k] \setminus \{b_l\}}\right)$.

Expansion ($\mathsf{ROT.Expand}(b, K_b)$**):**

Let $\alpha = \log_k(n)$.

- If $b = 1$, compute the GGM tree $T = T(K_b, \alpha)$ and output the n leaves of T.
- If $b = 2$, parse K_b as $\left(i, (q_{l,j})_{l \in [\alpha], j \in [k] \setminus \{b_l\}}\right)$, where $(b_1, \ldots, b_\alpha)$ is a k-ary encoding of $i-1$. Then run steps (5) and (7) of Protocol 2 as P_2.

Security. There are ppt simulators Sim_b for $b \in \{1, 2\}$ such that for any $n \in \mathbb{N}$ and $i \in [n]$,

$$\left\{ K_1 \;\middle|\; (K_1, K_2) \xleftarrow{R} \mathsf{ROT.Gen}(1^\lambda, n, i), \right\} \overset{c}{\approx} \left\{ K_1 \;\middle|\; K_1 \xleftarrow{R} \mathsf{Sim}_1(1^\lambda, n) \right\}, \quad (2)$$

and

$$\left\{ K_2, v_i^1 \;\middle|\; \begin{array}{l} (K_1, K_2) \xleftarrow{R} \mathsf{ROT.Gen}(1^\lambda, n, i), \\ \mathbf{v}^1 \leftarrow \mathsf{ROT.Expand}(1, K_1) \end{array} \right\} \overset{c}{\approx} \left\{ K_2, v_i^1 \;\middle|\; K_2 \xleftarrow{R} \mathsf{Sim}_2(1^\lambda, n, i), v_i^1 \xleftarrow{R} \mathbb{G} \right\}. \quad (3)$$

Informally, the above security definition ensures that P_1 does not learn anything about i, while P_2 does not learn anything about v_i^1, i.e., the random value it chooses not to receive, beyond the fact that it is random.

Theorem A.2. *Functionality 7 is a pseudorandom generator for $(n-1)$-out-of-n-OT.*

Proof. *Correctness.* First, observe that a GGM tree T with n leaves and initial seed s_0 implements a PRF $F_{s_0} : [n] \to \{0,1\}^\lambda$ with key s_0, where $F_{s_0}(j)$ is the j-th leaf of T [32, 37]. Since s_0 is chosen uniformly at random, $\mathbf{v}^1 \leftarrow \mathsf{ROT.Expand}(1, K_1) = \left(F_{s_0}(j)\right)_{j \in [n]}$ is indistinguishable from a vector drawn uniformly at random from $\mathbb{G}^n$. Second, observe that in $\mathsf{ROT.Expand}(2, K_2)$ in Functionality 7, all seeds of sub-trees of T that do not lie on the path to the i-th leaf are recovered. Since the expansion of G is deterministic, all leaves of these sub-trees are equal to the corresponding leaves in T, and therefore $v_j^1 = v_j^2$ for all $j \in [n] \setminus \{i\}$.

Security. We construct simulators Sim_b for $b \in \{1, 2\}$ as follows.

$b = 1$. Sample a random seed $s_0' \in \{0,1\}^\lambda$ and output s_0'. Indistinguishability of the two sides in Eq. (2) follows immediately as K_1 on the left hand side is also sampled uniformly from $\{0,1\}^\lambda$.

$b = 2$. Let $\alpha = \log_k(n)$, and let $(b_1, \ldots, b_\alpha)$ be a k-ary encoding of $i-1$. Construct a partial GGM tree by following the path from the root to the i-th leaf, sampling uniformly random seeds for all siblings of nodes on that path, and expanding them using the GGM construction. Now, for each level $l \in [\alpha]$ and each $j \in [k] \setminus \{b_l\}$, compute $q'_{l,j}$ as in Step (2) of Functionality 7, and output $K_2' \leftarrow \left(i, (q'_{l,j})_{l \in [\alpha], j \in [k] \setminus \{b_l\}}\right)$.
We will now show the indistinguishability of the two sides of Eq. (3) using a hybrid argument. We construct $\alpha + 1$ hybrid distributions by successively modifying ROT.Gen as follows. Let $\mathcal{H}^0$ be the left-hand side of Eq. (3), and let $(p_1, \ldots, p_\alpha)$ denote the nodes on the path from the root to the i-th leaf of the GGM tree generated in ROT.Gen. Now, for each $l \in [\alpha]$, construct $\mathcal{H}^l$ from $\mathcal{H}^{l-1}$ by replacing the result of the PRG expansion of p_l by k uniformly random strings from $\{0,1\}^\lambda$, and proceeding normally from then on to compute K_2'. Note that neither H^{l-1} nor H^l contain p_l, but both contain at least one of the children. Thus, any distinguisher between $\mathcal{H}^{l-1}$ and $\mathcal{H}^l$ could be used to distinguish the output of a PRG from random. Now, by construction of Sim_2 above, $\mathcal{H}^\alpha$ is precisely the right-hand side of Eq. 3 which concludes the security proof.

□

What remains to be shown is that the key generation of Protocol 2 securely implements Functionality 7. We reduce this to the security of the $(k-1)$-out-of-k-OT sub-protocol used in Protocol 2.

Theorem A.3. *Steps (1) – (4) of Protocol 2 implement Functionality 7 in the $(k-1)$-out-of-k-OT-hybrid model with security against semi-honest adversaries.*

Proof. For each $b \in \{1, 2\}$, we construct a simulator Sim_b for the view of P_b in the $(k-1)$-out-of-k-OT-hybrid model.

$b = 1$. Since P_1 does not receive any messages in Protocol 2, Sim_1 is the identity function. Since the computation performed is the same in Protocol 2 and Functionality 7, the simulated and real views are identically distributed.

$b = 2$. Here, in addition to the outputs of the ideal functionality, P_2 receives the outputs of the OTs in Step (4). However, note that these are directly passed through to P_2's output and are therefore trivially simulatable. Since the values computed in Step (2) of Functionality 7 are precisely the ones selected by the OT functionality, the two views are again identically distributed.

□

We can now compose Protocol 2 with any $(k-1)$-out-of-k-OT protocol using a standard modular composition theorem, as for example given by Canetti [14], thus obtaining a secure protocol in the plain model.

Functionality 8: Known-Index SPFSS

Key Generation ($\mathsf{SPFSS.Gen}(1^\lambda, f_{i,\beta})$**):**
Let $[n]$ denote the domain of $f_{i,\beta}$.

(1) Generate keys for a $(n-1)$-out-of-n-ROT scheme $\left(K_1^{\mathsf{ROT}}, K_2^{\mathsf{ROT}}\right) \leftarrow \mathsf{ROT.Gen}(1^\lambda, n, i)$.
(2) Compute $\mathbf{v}^1 = \mathsf{ROT.Expand}(1, K_1)$ and $\tilde{r} = \beta - v_i^1$.
(3) Output $K_1 = K_1^{\mathsf{ROT}}$ and $K_2 = \left(K_2^{\mathsf{ROT}}, \tilde{r}\right)$.

Expansion ($\mathsf{SPFSS.Eval}(b, K_b, x)$**):**
Let $\mathbb{G}$ denote the image of $f_{i,\beta}$.

- If $b = 1$, compute $\mathbf{v}^1 \leftarrow \mathsf{ROT.Expand}(1, K_1)$ and output v_x^1.
- If $b = 2$, parse K_2 as $\left(K_2^{\mathsf{ROT}}, \tilde{r}\right)$. Note that K_2^{ROT} contains the non-zero index i. If $x = i$, output $\tilde{r}$. Otherwise, compute $\mathbf{v}^2 \leftarrow \mathsf{ROT.Expand}\left(2, K_2^{\mathsf{ROT}}\right)$ and output $-v_x^2$.

A.2 Known-Index SPFSS

Here, we define out Known-Index SPFSS as an instance of Definition 2.2 from the preliminaries section.

Definition A.4 (Known-Index SPFSS). Let $\mathcal{F} = \{f_{i,\beta} : [n] \to \mathbb{G}\}$ denote a class of point functions, where for all $x \in [n]$,

$$f_{i,\beta} = \begin{cases} \beta & \text{if } x = i, \\ 0 & \text{otherwise.} \end{cases}$$

A *Known-Index Single-Point Function Secret Sharing (Known-Index SPFSS)* scheme is a FSS scheme for $\mathcal{F}$, where $\mathsf{Leak}_1(f_{i,\beta}) = (I, \mathbb{G})$ and $\mathsf{Leak}_2(f_{i,\beta}) = (I, \mathbb{G}, i)$, i.e., we allow the recipient of K_2 to additionally learn the non-zero index i (but not the value β).

In Functionality 8, we define key generation and evaluation procedures for our known-index FSS scheme. We will now prove that this functionality indeed satisfies Definition A.4, and that Protocol 3 implements the key generation phase securely.

Theorem A.5. *Functionality 8 is a Known-Index Single-Point Function Secret Sharing scheme.*

Proof. *Correctness.* For any $j \in [n] \setminus \{i\}$, the correctness of the ROT scheme guarantees that $v_j^1 = v_j^2$, and hence $\mathsf{SPFSS.Eval}(1, K_1, j) + \mathsf{SPFSS.Eval}(2, K_2, j) = v_j^1 - v_j^2 = 0$. On the other hand, for $j = i$, we have $\mathsf{SPFSS.Eval}(1, K_1, j) + \mathsf{SPFSS.Eval}(2, K_2, j) = v_i^1 + \tilde{r} = v_i^1 + \beta - v_i^1 = \beta$.

Security. We construct the following simulators Sim_b for $b \in \{1, 2\}$, assuming simulators $\mathsf{Sim}_b^{\mathsf{ROT}}$ for the random OT scheme used.

$b = 1$. Output $\mathsf{Sim}_1^{\mathsf{ROT}}(1^\lambda, n)$. Indistinguishability follows from Eq. (2) in Definition A.1.

$b = 2$. Sample $r \xleftarrow{R} \mathbb{G}$ and output $\left(\mathsf{Sim}_2^{\mathsf{ROT}}(1^\lambda, n, i), r\right)$. Note that this distribution is the same as the right side of Eq. (3). Therefore, any distinguisher of the two sides of Eq. (1) could be used to distinguish the distributions in Eq. (3) by choosing a $\beta \leftarrow \mathbb{G}$ and replacing v_i^1 in Eq. (3) by $\beta - v_i^1$.

□

Theorem A.6. *Steps (1)–(6) in Protocol 3 together implement* $\mathsf{SPFSS.Gen}(1^\lambda, f_{i,\beta})$ *from Functionality 8 with security against semi-honest adversaries, where i is input by P_2 and β is secret-shared between the two parties.*

Proof. We first prove that Protocol 3 is secure in the $(n-1)$-out-of-n-ROT-hybrid model when all calls to ROT.Gen are performed by the ideal Functionality 7. We construct simulators Sim_b for $b \in \{1, 2\}$ for the views of both parties in the ideal model.

$b = 1$. The only messages received by P_1 come from the execution of ROT.Gen, and thus Sim_1 is the identity function.

$b = 2$. Here, in addition to the output of ROT.Gen, P_2 receives R_β. Simulate this with $\beta_2 - \tilde{r} + \sum_{j \in [n] \setminus \{i\}} \mathsf{SPFSS.Eval}(2, K_2, j)$. In the $(n-1)$-out-of-n-ROT-hybrid model, this simulated view is distributed identically to the real view.

To prove security in the plain model, we again use the modular composition theorem for semi-honest security together with a secure protocol for ROT.Gen, as proven in Theorem A.3 □

A.3 Known-Indices MPFSS

We will now prove security of our batched FSS implementation. However, as discussed in Section 5, there is a small probability that the batching fails (note that this is also the case for the heuristic batch code construction suggested by Boyle et al. [9]). Here we have two options if batching fails: We could abort the key generation, sacrificing security as this leaks some information about the non-zero indices that failed to be batched; or we could sacrifice correctness by returning keys that will result in shares of zeros for some indices that should be non-zero. Both are valid approaches depending on the concrete application, as also discussed in [1, 15]. For our VOLE construction, we will opt for the second choice, since this will allow us to achieve the same security guarantee as Boyle et al. [9], i.e., our scheme is either secure under standard LPN (if batching succeeds), or under a slightly stronger variant of LPN (if batching fails). See also the discussion in Section 6. We will not mention this explicitly in the following definitions and proofs, but whenever cuckoo hashing is performed, we assume that failures are handled by dropping indices that would result in a hashing failure.

Definition A.7 (Known-Indices MPFSS). For any $t, n \in \mathbb{N}$, let $\mathcal{F} = \left\{f_{\mathbf{i},\boldsymbol{\beta}} : [n] \to \mathbb{G}\right\}$ be a class of multi-point functions, where $\mathbf{i} \in [n]^t$, $\boldsymbol{\beta} \in \mathbb{G}^t$, and

$$f_{\mathbf{i},\boldsymbol{\beta}}(x) = \begin{cases} \beta_j & \text{if } x = i_j \text{ for some } j \in [t], \\ 0 & \text{otherwise.} \end{cases}$$

Let further $\eta, \lambda \in \mathbb{N}$ denote statistical and computational security parameters, respectively. A *Known-Indices Multi-Point Function Secret Sharing (Known-Indices MPFSS)* scheme consists of the following two algorithms:

- $(K_1, K_2) \leftarrow \mathsf{MPFSS.Gen}(1^\lambda, \eta, f)$ – given a description of $f \in \mathcal{F}$, outputs two keys.
- $f_b(x) \leftarrow \mathsf{MPFSS.Eval}(b, K_b, x)$ – given a key for party $b \in \{1, 2\}$ and an input $x \in [n]$, return a share of $f(x)$.

Where the following properties have to be satisfied:

Functionality 9: Known-Indices MPFSS

Key Generation ($\mathsf{MPFSS.Gen}(1^\lambda, \eta, f_{\mathbf{i},\boldsymbol{\beta}})$**):**
Let $[n]$ denote the domain of $f_{\mathbf{i},\boldsymbol{\beta}}$, and t the number of non-zero points.

(1) Choose parameters $(\kappa, m) \leftarrow \mathsf{ParamGen}(n, t, \eta)$ for a cuckoo hashing scheme such that hashing any t indices from $[n]$ fails with probability at most $2^{-\eta}$.
(2) Perform Steps (1) and (2) from Protocol 4, i.e., choose κ random hash functions and use them to insert $(i_1, \ldots, i_t)$ into a cuckoo hash table T, and simple-hash the domain $[n]$. Let pos_l be defined as in Protocol 4.
(3) Let $\mathbf{u} = \left((\beta_j, l_j)\right)_{j\in[t]}$, where l_j is the location of i_j in T. Compute $\mathbf{v} \in \mathbb{G}^m$, where
$$v_j = \begin{cases} a & \text{if } (a, j) \in \mathbf{u}, \\ 0 & \text{otherwise.} \end{cases}$$
(4) Call $\mathsf{SPFSS.Gen}$ m times as in Step (4) from Protocol 4 to obtain m sets of keys $\left((K_1^l, K_2^l)\right)_{l\in[m]}$
(5) Output $K_b = \left((h_p)_{p\in[\kappa]}, (K_b^l)_{l\in[m]}\right)$ for $b \in \{1, 2\}$.

Expansion ($\mathsf{MPFSS.Eval}(b, K_b, x)$**):**
Parse K_b as $\left((h_p)_{p\in[\kappa]}, (K_b^l)_{l\in[m]}\right)$ and output $\sum_{p=1}^{\kappa} \mathsf{SPFSS.Eval}\big(b, K_b^{h_p(x)}, \mathsf{pos}_{h_p(x)}(x)\big)$.

Correctness. For any $f \in \mathcal{F}$, and any $x \in I$, when $(K_1, K_2) \leftarrow \mathsf{MPFSS.Gen}(1^\lambda, \eta, f)$, we have
$$\Pr\left[\sum_{b\in\{1,2\}} \mathsf{MPFSS.Eval}(b, K_b, x) = f(x)\right] \geq 1 - 2^{-\eta}.$$

Security. For any $b \in \{1, 2\}$, there exists a ppt simulator Sim_b such that for any polynomial-size function sequence $f_\lambda \in \mathcal{F}$,
$$\left\{K_b \,\middle|\, (K_1, K_2) \leftarrow \mathsf{MPFSS.Gen}(1^\lambda, \eta, f_\lambda)\right\} \overset{c}{\approx} \left\{K_b \leftarrow \mathsf{Sim}_b(1^\lambda, \eta, \mathsf{Leak}_b(f_\lambda))\right\}, \quad (4)$$
where $\mathsf{Leak}_1(f_{\mathbf{i},\boldsymbol{\beta}}) = ([n], \mathbb{G})$ and $\mathsf{Leak}_2(f_{\mathbf{i},\boldsymbol{\beta}}) = ([n, \mathbb{G}], \mathbf{i})$.

Note that the security guarantee of Definition A.7 is the same as in Definition 2.2. The main difference is in the correctness guarantee, where we allow the output to be incorrect with a small probability depending on the statistical security parameter η.

Functionality 9 describes our MPFSS procedure. We will now prove its correctness and security guarantees according to Definition A.7.

Theorem A.8. *Functionality 9 is a Known-Index MPFSS scheme.*

Proof. *Correctness.* First, observe that the parameters for cuckoo hashing are chosen in Step (1) such that insertion fails with probability of at most $2^{-\eta}$. Thus, it remains to show in the case that cuckoo hashing succeeds,
$$\begin{aligned} f_{\mathbf{i},\boldsymbol{\beta}}(x) &= \sum_{b\in\{1,2\}} \mathsf{MPFSS.Eval}(b, K_b, x) \\ &= \sum_{b\in\{1,2\}} \sum_{p=1}^{\kappa} \mathsf{SPFSS.Eval}\big(b, K_b^{h_p(x)}, \mathsf{pos}_{h_p(x)}(x)\big) \\ &= \sum_{p=1}^{\kappa} g_{h_p(x)}(\mathsf{pos}_{h_p(x)}(x)) \end{aligned}$$
where $g_l(x)$ is defined as in Step (4) of Protocol 4. There are two cases.

(1) $x = i_j$ for some $j \in [t]$. Then, since cuckoo hashing was successful, for exactly one $p^* \in [\kappa]$, $T\left[h_{p^*}(x)\right] = x$. Let $l^* = h_{p^*}$ be the location of x in T. Then $g_{l^*}(\mathsf{pos}_{l^*}(x)) = \beta_j$, while $g_l(\mathsf{pos}_l(x)) = 0$ for all $l \in \left\{h_p(x) \,\middle|\, p \in [\kappa] \setminus \{p^*\}\right\}$.
(2) $x \notin \mathbf{i}$. Then for all possible locations $l \in \left\{h_p(x) \,\middle|\, p \in [\kappa]\right\}$, $T[l] \neq x$ and thus $g_l(\mathsf{pos}_l(x)) = 0$.

Security. We construct simulators Sim_b for $b \in \{1, 2\}$ by calling simulators $\mathsf{Sim}_b^{\mathsf{SPFSS.Gen}}$ for the SPFSS key generation algorithm used in Step (4) of Functionality 9.

Both simulators start by computing $(\kappa, m) \leftarrow \mathsf{ParamGen}(n, t, \eta)$ and sampling κ random hash functions $(h_p)_{p\in[\kappa]}$. They then simple-hash the domain $[n]$, resulting in m buckets of sizes $(I_l)_{l\in[m]}$.

$b = 1$. For each bucket $l \in [m]$, call $\mathsf{Sim}_1^{\mathsf{SPFSS.Gen}}(1^\lambda, I_l, \mathbb{G})$ to obtain keys (K_1^l). Output $\left((h_p)_{p\in[\kappa]}, (K_1^l)_{l\in[m]}\right)$. Indistinguishability of the distributions in Eq. (4) follows from the fact that the h_p (and therefore the bucket sizes I_l) are identically distributed, and for each bucket the simulated keys are indistinguishable from the real ones due to the security of the $\mathsf{SPFSS.Gen}$ procedure (Eq. (1)).

$b = 2$. Construct a cuckoo hash table T of size m using the hash functions $(h_p)_{p\in[\kappa]}$ and $i_1, \ldots, i_t$ as in Step (2) of Functionality 9. Now for each bucket $l \in [m]$, compute $K_2^l \leftarrow \mathsf{Sim}_1^{\mathsf{SPFSS.Gen}}\big(1^\lambda, I_l, \mathbb{G}, \mathsf{pos}_l(T[l])\big)$ and output $\left((h_p)_{p\in[\kappa]}, (K_2^l)_{l\in[m]}\right)$. Again, indistinguishability follows from the fact that both views are identically distributed up to and including the creation of T, and then from the fact that the simulated and real keys for each bucket are indistinguishable by Eq. (1).

□

Theorem A.9. *Protocol 4 implements* $\mathsf{MPFSS.Gen}(1^\lambda, \eta, f_{\mathbf{i},\boldsymbol{\beta}})$ *from Functionality 9 with security against semi-honest adversaries, where* $\mathbf{i}$ *is input by* P_2 *and* $\boldsymbol{\beta}$ *is secret-shared element-wise between the parties.*

Proof. We will first prove security assuming an ideal functionalities SPFSS.Gen (Functionality 8) for SPFSS key generation, and $\mathcal{F}^{\mathsf{2PC}}$ for generic two-party computation. Then we again rely on modular composition to obtain a protocol in the plain model. We construct simulators Sim_b for the the views of both parties $b \in \{1, 2\}$. Both simulators perform simple hashing to obtain bucket sizes consistent with the keys from the ideal output. Then, the simulation depends on b:

$b = 1$. The only messages Sim_1 needs to simulate are the outputs of $\mathcal{F}^{\mathsf{2PC}}$, which by construction are equal to the inputs to the calls to SPFSS.Gen, since all other messages received by P_1 are part of the output. Since by definition, $\mathbf{v^1}$ in Step (3) of Protocol 4 is a random share, this can be simulated by sampling $\mathbf{v^1} \xleftarrow{R} \mathbb{G}$. The resulting view is identical to the one in the (SPFSS.Gen, $\mathcal{F}^{\mathsf{2PC}}$)-hybrid model.

$b = 2$. Sim_2 needs to first perform cuckoo hashing to generate a hash table T consistent with the input indices $\mathbf{i}$ and hash functions from the ideal output. It can then call $\mathcal{F}^{\mathsf{2PC}}$ with a uniform vector $\mathbf{v^1} \xleftarrow{R} \mathbb{G}$ as above. The inputs to each SPFSS.Gen call are computed from T as in Step (4) of Protocol 4. The resulting view is again identical to the one in the (SPFSS.Gen, $\mathcal{F}^{\mathsf{2PC}}$)-hybrid model.

□