VOLE-PSI: Fast OPRF and Circuit-PSI from Vector-OLE

Peter Rindal Phillipp Schoppman

August 8, 2024

Abstract

In this work we present a new construction for a batched Oblivious Pseudorandom Function (OPRF) based on Vector-OLE and the PaXoS data structure. We then use it in the standard transformation for achieving Private Set Intersection (PSI) from an OPRF. Our overall construction is highly efficient with $O(n)$ communication and computation. We demonstrate that our protocol can achieve malicious security at only a very small overhead compared to the semi-honest variant. For input sizes $n = 2^{20}$, our malicious protocol needs 6.2 seconds and less than 59 MB communication. This corresponds to under 450 bits per element, which is the lowest number for any published PSI protocol (semi-honest or malicious) to date. Moreover, in theory our semi-honest (resp. malicious) protocol can achieve as low as 219 (resp. 260) bits per element for $n = 2^{20}$ at the added cost of interpolating a polynomial over n elements.

As a second contribution, we present an extension where the output of the PSI is secret-shared between the two parties. This functionality is generally referred to as Circuit-PSI. It allows the parties to perform a subsequent MPC protocol on the secret-shared outputs, e.g., train a machine learning model. Our circuit PSI protocol builds on our OPRF construction along with another application of the PaXoS data structure. It achieves semi-honest security and allows for a highly efficient implementation, up to 3x faster than previous work.

1 Introduction

We consider the problem of private set intersection (PSI) in a two-party setting. Here, two mutually distrusting parties, a receiver and a sender, each hold a set of identifiers X, Y respectively. The goal of the two parties is for the receiver to learn the intersection $X \cap Y$ without revealing any

additional information to the parties. In particular, the sender should not learn any information about X beyond the size of it. Similarly, the receiver should not learn anything about $Y \setminus X$ beyond the size of Y.

A common approach to PSI is based on oblivious pseudo-random functions (OPRFs). An OPRF allows the receiver to input x and learn $F_k(x)$, where F is a PRF, and k is known to the sender. A straight-forward PSI protocol can be obtained by running an OPRF protocol for each $x \in X$, and then having the sender send $\{F_k(y) \mid y \in Y\}$ to the receiver. The receiver can then locally compare the sender's OPRF values to her own to learn which elements of X are in the intersection. This is the basis of several PSI protocols (see Section 1.4), and our first contribution also follows this paradigm.

While PSI alone has interesting applications, such as private contact discovery [KLS+17, DRRT18, KRS+19], other variants of PSI are gaining traction from a practical perspective. For example, both Google [IKN+20] and Facebook [BKM+20] have implemented variants of PSI that allow them to compute functions of the intersection, where only the result of the function evaluation and the intersection size is revealed, but not the intersection itself.

A generalization of these PSI-with-computation protocols yields *circuit PSI*, where the output isn't revealed to either party, but instead is secret-shared between the parties. More precisely, the receiver learns a random vector $\vec{Q^0}$ and the sender learns $\vec{Q^1}$ such that $(q_i^0 \oplus q_i^1) = 1$ if i corresponds to an element $x \in X$ in the intersection, and $(q_i^0 \oplus q_i^1) = 0$ otherwise. Note that this means that not even the intersection size is revealed to either party. We additionally can allow the sender (resp. receiver) to input an "associated value" $\tilde{y}_j$ (resp. $\tilde{x}_i$) for each $y_j \in Y$ (resp. $x_i \in X$). In this case, the output also includes a random vector $\vec{Z^0}$ to the receiver and $\vec{Z^1}$ to the sender such that $(z_i^0 \oplus z_i^1) = (\tilde{y}_j || \tilde{x}_i)$ if $x_i = y_j$.

1.1 Contributions

PSI: We present a protocol for private set intersection (Section 4) based on two building blocks. The first building block is a protocol known as Vector OLE and presented in Figure 2. Multiple implementations of VOLE have recently been presented [BCG+19, SGRR19, WYKW20, YWL+20]. We use an improved version of [SGRR19] in this paper. The second building block is a linear system solver, e.g. PaXoS [PRTY20], which we adapt for our purposes as shown in Figure 1. Combining these two primitives in a novel way, we obtain an OPRF protocol (Figure 4). This construction is highly efficient, requiring an amortized 2.4κ bits of communication per

input in our computationally efficient version or just κ bits when optimized for communication. We also demonstrate that malicious security can be obtained with only a very small overhead.

From an OPRF it is easy to obtain an PSI protocol which is our final goal. This final step is shown in Figure 6. We show that the malicious variant of this well known transformation can be optimized which reduces its overhead by as much as 50% compared to prior art [PRTY20, CKT10]. Our final PSI protocol is secure against both semi-honest and malicious adversaries, and we provide an implementation for both threat models. It is also highly efficient, requiring just 5.4 (resp. 6.2) seconds and less than 54 (resp. 59) MB communication in the semi-honest (resp. malicious) setting.

Circuit PSI: Our second contribution is a protocol for circuit PSI. In Section 5, we show that using our variant of the PaXoS solver along with any OPRF protocol yields an Oblivious Programmable PRF (OPPRF) protocol. Given this, we then construct the final protocol in Section 6 with the additional help of data structure known as a cuckoo hash table. We also implement two variants of our circuit PSI protocol in the semi-honest model and show that they outperform the best previous approach [PSTY19].

1.2 Notation

We use κ as the computational security parameter and λ for statistical security. The receiver's set is detonated as X while the sender's is Y. Their respective sizes are $n_{\mathsf{x}}, n_{\mathsf{y}}$. Often we will just assume both set are of size n. $[a, b]$ denotes the set $\{a, a+1, ..., b\}$ and $[b]$ is shorthand for $[1, b]$. We denote row vectors $\vec{A} = (a_1, ..., a_n)$ using the arrow notation while the elements are indexed without it. A set $S = \{s_1, ..., s_n\}$ will use similar notation. For a matrix M, we use $\vec{M}_i$ to denote its i-th row vector, and $M_{i,j}$ for the element at row i and column j. $\langle \vec{A}, \vec{B} \rangle$ denotes the inner product of $\vec{A}, \vec{B}$. We use $=$ to denote the statement that the values are equal. Assignment is denoted as $:=$ and for some set S, the notation $s \leftarrow S$ means that s is assigned a uniformly random element from S. If a function F is deterministic then we write $y := F(x)$ while if F is randomized we use $y \leftarrow F(x)$ to denote $y := F(x; r)$ for $r \leftarrow \{0,1\}^*$.

1.3 Overview

OPRF. We now present a simplified version of our main protocols. Our core building block is a functionality known as (random) vector OLE which

allows the parties to sample random vectors $\vec{A}, \vec{B}, \vec{C} \in \mathbb{F}^m$ and element $\Delta \in \mathbb{F}$ such that $\vec{C} = \Delta\vec{A} + \vec{B}$. The PSI receiver will hold $\vec{A}, \vec{C}$ while the sender will hold $\vec{B}, \Delta$. We note that in the vector OLE literature, the sender/receiver roles are typically reversed.

The parties (implicitly) sample an exponentially large random matrix $M^* \in \{0,1\}^{|\mathbb{F}| \times m}$. The receiver defines $M \in \{0,1\}^{n \times m}$ which is the submatrix indexed by the rows $x \in X$. The receiver then solves the linear system

$$M\vec{P}^\intercal = (0, ..., 0)^\intercal$$

for the unknown $\vec{P} \in \mathbb{F}^m$. For now let us assume $\vec{P}$ is some random solution and not the trivial $(0, ..., 0)$ solution[1]. The protocol proceeds by having the receiver send $\vec{A} + \vec{P}$ to the sender who defines

$$\vec{K} := \vec{B} + \Delta(\vec{A} + \vec{P})$$

The crucial observation is that

$$\begin{aligned} M\vec{K}^\intercal &= M\vec{B}^\intercal + \Delta(M\vec{A}^\intercal + M\vec{P}^\intercal) \\ &= M\vec{B}^\intercal + \Delta M\vec{A}^\intercal \\ &= M\vec{C}^\intercal \end{aligned}$$

In particular, for each $x \in X$ it holds that $\langle \vec{M}^*_x, \vec{K} \rangle = \langle \vec{M}^*_x, \vec{C} \rangle$ where $\vec{M}^*_x$ is the x'th row of M^*. An OPRF can then be obtained by having the receiver apply a random oracle as

$$\mathsf{H}(\langle \vec{M}^*_x, \vec{C} \rangle), \qquad x \in X$$

while the sender computes the output at any y as

$$F_{\vec{K}}(y) := \mathsf{H}(\langle \vec{M}^*_y, \vec{K} \rangle)$$

To ensure efficiency we will require M^* to be of a special form such that solving $M\vec{P}^\intercal = (0, ..., 0)^\intercal$ is efficient while also computing $\langle \vec{M}^*_x, \vec{V} \rangle$ in $O(1)$ time. Specifically, we will use the PaXoS solver [PRTY20] to enable these properties.

To achieve security it is crucial that the receiver can not compute the OPRF F at any other point $x \notin X$. In the formulation above this effectively means that it is hard to find a $x \notin X$, such that $\langle \vec{M}^*_x, \vec{P} \rangle = 0$. We demonstrate how such a property can be obtained at little to no overhead.

[1] In our malicious OPRF construction (Section 3.2), we will instead use a random oracle H, and set $M\vec{P}^\intercal = (\mathsf{H}(x_0), \mathsf{H}(x_1), \ldots, \mathsf{H}(x_n))^\intercal$. We stick to the semi-honest variant here for ease of presentation.

PSI. We then employ our OPRF construction as a subroutine to obtain a PSI protocol. This traditional transformation instructs the receiver to input their set X into an OPRF protocol to obtain $F(x)$ for $x \in X$. The sender can then send $Y' = \{F(y) \mid y \in Y\}$ which allows the receiver to identify the common items. In the malicious setting, one must show how the simulator extracts the set Y from observing Y'. The traditional analysis [PRTY20, CKT10] effectively achieves this by requiring the OPRF F to be second-preimage resistant and as such each $y' \in Y'$ must be of length $2\kappa \approx 256$ bits. We demonstrate that in fact preimage resistance is sufficient which allows the OPRF to have κ bit output which reduces the communication overhead by approximately 33%, or as much as 50% when $|Y| \gg |X|$.

Programmable OPRF. We present extension of our OPRF protocol to achieve a functionality known as a Programmable OPRF (OPPRF) [PSTY19]. This building block will allow the sender to sample an OPPRF key k such that $F_k(y_i) = v_i$ for their choice of y_i, v_i. At all other locations the output of F_k will be random.

The parties first perform a normal OPRF protocol for an OPRF F', where the receiver inputs their set X and receive $F'(x)$ for $x \in X$. The sender solves the system

$$M\vec{P}^\intercal = (v_1 - F'(y_1), ..., v_n - F'(y_n))^\intercal$$

where $M \in \{0,1\}^{n \times m}$ the submatrix of M^* indexed by the rows y_i. The sender will send $\vec{P}$ to the receiver who outputs

$$x' := F'(x) + \langle \vec{M}^*_x, \vec{P} \rangle$$

for $x \in X$. Observe that at $x = y_i \in Y$

$$\begin{aligned} x' :=& F'(x) + \langle \vec{M}^*_x, \vec{P} \rangle \\ =& F'(y_i) + v_i - F'(y_i) \\ =& v_i \end{aligned}$$

as desired. It can be shown that at all other points $y \notin Y$, the output is completely random. One security concern is that $\vec{P}$ might leak information about Y. Indeed, the PaXoS solver requires m larger than n, therefore several solutions could exist, and which $\vec{P}$ is output by PaXoS may leak information. We show that this is the case for PaXoS and then present an extension which is uniformly distributed under some constraints. We call our extension XoPaXoS and present it in Section 2. Our full OPPRF protocol is presented in Section 5.

Circuit-PSI. Finally, we present our Circuit PSI extension which allows the output of the PSI to be secret shared between the two parties. Our protocol builds on the previous approach of Pinkas et al. [PSTY19] by replacing their OPPRF construction with ours. For completeness we present this construction in Section 6.

1.4 Related Work

Early PSI protocols based on OPRFs/Diffie–Hellman (DH) have been around since the 1980s [Mea86], and they still form the basis of many modern PSI protocols [CT10, IKN+20, BKM+20]. The advantage of DH-based protocols is their low communication cost and constant round complexity, which however comes at the cost of high computational overhead. A more computationally efficient protocol based on oblivious transfer extension [IKNP03] (as opposed to OPRF based) was presented by Schneider et al. [PSSZ15] along with many derivatives [PSZ14, KKRT16, RR17b, OOS17].

More recently, these two paradigms have begun to merge, and various OPRF constructions have been proposed [DCW13, KKRT16, RR17a, PRTY19, PRTY20, CM20] which more closely resemble [IKNP03]. All of these come with higher communication cost than [Mea86], but they significantly reduce computation. However, as the evaluation of [CM20] has shown, the optimal choice of protocol often depends on the network setting. Our work also follows the OPRF-based approach, building on the recent PSI protocol of [PRTY20], but significantly reducing communication. As our experiments in Section 7 show, our protocol works particularly well in settings with limited bandwidth and large input sizes. For an extended overview of the different approaches to PSI, see [IKN+20, Section 4.1] and [PSZ18, Section 1.2].

The first circuit PSI protocols were based entirely on generic techniques such as garbled circuits [HEK12] or GMW [PSSZ15, PSZ18]. Subsequent works improved computation and communication [CO18, PSWW18, PSTY19], and the linear-complexity protocol of [PSTY19] forms the current state of the art. Their protocol combines an oblivious *programmable* PRF (OPPRF) based on polynomial interpolation with a relatively small GMW circuit. Our circuit PSI protocol follows a similar approach, but uses our new OPRF construction, as well as a novel way to program it based on PaXoS [PRTY20].

Parameters:

- Statistical security parameter λ and computational security parameter κ.
- Input length n.
- A finite group $\mathbb{G}$.
- For $m' = 2.4n$, let $d = O(\lambda)$ upper bound the size of 2-core of a (m', n)-Cuckoo graph [PRTY20].
- Output length $m = m' + d + \lambda$.
- A random function $\mathsf{row} : \mathbb{G} \times \{0,1\}^\kappa \to \{0,1\}^m$ s.t. $\forall x$, the weight of the first m' bits of $\mathsf{row}(x)$ is 2.

Encode $((z_1, v_1), ..., (z_n, v_n); r)$:

1. Define $\mathsf{row}' : \mathbb{G} \to \{0,1\}^{m'}$ and $\tilde{\mathsf{row}}(z)$ s.t. $\mathsf{row}'(z) || \tilde{\mathsf{row}}(z) = \mathsf{row}(z, r)$ for all z. Let
$$M := \begin{bmatrix} \mathsf{row}(z_1, r) \\ ... \\ \mathsf{row}(z_n, r) \end{bmatrix} \in \{0,1\}^{n \times m}$$
and let $M' \in \{0,1\}^{n \times m'}, \tilde{M} \in \{0,1\}^{n \times d + \lambda}$ s.t. $M' || \tilde{M} = M$.
2. Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a graph with vertex set $\mathcal{V} = [m']$ and edge set $\mathcal{E} = \{(c_0, c_1) \mid i \in [n], M'_{i,c_0} = M'_{i,c_1} = 1\}$. Let $\tilde{\mathcal{G}} = (\tilde{\mathcal{V}}, \tilde{\mathcal{E}})$ be the 2-core of $\mathcal{G}$.
3. Let $R \subset [n]$ index the rows of M in the 2-core, i.e. $R = \{i \mid M'_{i,c_0} = M'_{i,c_1} = 1 \wedge (c_0, c_1) \in \tilde{\mathcal{E}}\}$. Let $\tilde{d} := |R|$ and abort if $\tilde{d} > d$.
4. Let $\tilde{M}' \in \{0,1\}^{\tilde{d} \times (d+\lambda)}$ be the submatrix of $\tilde{M}$ obtained by taking the row indexed by R. Abort if $\tilde{M}'$ does not contain an invertible $\tilde{d} \times \tilde{d}$ matrix. Otherwise let $\tilde{M}^*$ be one such matrix and $C \subset [d+\lambda]$ index the corresponding columns of $\tilde{M}'$.
5. Let $C' := \{j \mid i \in R, M'_{i,j} = 1\} \cup ([d + \lambda] \setminus C + m')$ and for $i \in C'$ assign $P_i \leftarrow \mathbb{G}$. For $i \in R$, define $v'_i := v_i - (M\vec{P}^\intercal)_i$ where P_i is assumed to be zero if unassigned.
6. Using Gaussian elimination solve the system $\tilde{M}^*(P_{m'+C_1}, ..., P_{m'+C_{\tilde{d}}})^\intercal = (v'_{R_1}, ..., v'_{R_{\tilde{d}}})^\intercal$.
7. Let $T \subset [m']$ such that each tree in $\mathcal{G}$ has a single vertex in T. For $i \in T$, assign $P_i \leftarrow \mathbb{G}$.
8. Let $I := \{j \mid i \in R, M'_{i,j} = 1\} \cup T$ and $\overline{I} := [m'] \setminus I$.
9. While $I \neq \emptyset$, select an $i \in I$ and do the following: Update $I := I \setminus \{i\}$ and $\overline{I} := \overline{I} \cup \{i\}$. For all $j \in \{j \mid (j, i) \in \mathcal{E} \wedge j \notin \overline{I}\}$. Identify k s.t. $\{h_0(z_k, r), h_1(z_k, r)\} = \{i, j\}$ and assign $P_j := v_k - P_i$.
10. Return $\vec{P}$.

Decode $(\vec{P}, z, r)$:

1. Return $\langle \mathsf{row}(z, r), \vec{P} \rangle$.

Figure 1: XoPaXoS algorithm.

2 Oblivious Key Value Stores & PaXoS

Our constructions makes use of a type of linear system solvers called an oblivious key value store (OKVS)[GPR+21]. As discussed before, we will use these solvers to encode our input sets $(z_1, ..., z_n) = Z$ and values $(v_1, ..., v_n) = V$ as a vector $\vec{P} \in \mathbb{G}^m$. There will exist a function Decode such that $\mathsf{Decode}(\vec{P}, z_i) = v_i$ for $i \in [n]$ and is linear with respect to $\vec{P}$. There are three main performance metrics that we are concerted with. The first is the rate $\rho = m/n$ which denotes how compact the encoding is, i.e. n items can be encoded as m element vector. The last two metrics are the running time of the encoder/solver and that of the decoder/matrix multiplier.

Each instance of a solver is parameterized by a finite group $\mathbb{G}$, integer $m \geq n$, security parameter λ and an implicit random matrix $M^* \in \mathbb{G}^{|\mathbb{G}| \times m}$. The instance is fixed by sampling $M^* \leftarrow \mathcal{M}$ from some set $\mathcal{M}$ which depends on the particular solver. For any set $Z \subset \mathbb{G}$ s.t. $|Z| = n$, the solver will output $\vec{P} \in \mathbb{G}^m$ s.t.

$$M\vec{P}^\intercal = (v_1, ..., v_n)^\intercal$$

where $M \in \{0,1\}^{n \times m}$ is the submatrix of M^* obtained by taking the rows indexed by $z \in Z$. The target values $v_1, ..., v_n \in \mathbb{G}$ can be arbitrary. Our application will require the solver to output a solution with probability $1 - O(2^{-\lambda})$.

Since M^* is exponential in size, it is more efficient to represent it as a random seed $r \in \{0,1\}^\kappa$ and define the i-th row as being the output of the random function $\mathsf{row}(i, r)$. Therefore we will have the property $\langle \mathsf{row}(x_i, r), \vec{P} \rangle = v_i$. For easy of presentation we will further abstract this via the Decode function defined as $\mathsf{Decode}(\vec{P}, x_i, r) := \langle \mathsf{row}(x_i, r), \vec{P} \rangle$. We note that this is a very general encoding framework and encompasses several schemes, e.g. PaXoS, interpolation, bloom filters, and many others.

The Vandermonde OKVS. One example of this general approach is polynomial interpolation. In this case we require $\mathbb{G}$ to also be a field and $\mathcal{M}$ contains only the Vandermonde matrix, i.e. $\mathsf{row}(i, r) = (1, i, i^2, ..., i^{n-1})$ for all r. As such it achieves an optimal rate of $\rho = 1$, i.e. $m = n$. In this case, solving the system requires $O(n \log^2 n)$ time using polynomial interpolation and decoding n points also requires $O(n \log^2 n)$ time [BM74]. For large n it is also possible to construct row in such a way that t smaller systems of size $O(\lambda)$ are constructed and solved independently [PSTY19]. This so called binning technique effectively results in a $O(n \log^2 \lambda)$ running time while also maintaining near optimal rate $\rho \approx 1$.

The PaXoS OKVS. The PaXoS solver [PRTY20] significantly improves on polynomial interpolation in that it achieves $O(n)$ running time. However, it comes at the cost of rate $\rho \approx 2.4$, i.e., $m \approx 2.4n$. The scheme of [PRTY20] defines row as outputting a binary vector s.t. the first $m' := 2.4n$ elements have weight 2 while the last $m - m' = O(\lambda)$ bits are distributed uniformly. There is also a PaXoS variant which achieves a slightly better rate of $\rho = 2$ but at an increased running time. In this paper, we only make use of the first scheme.

Other Solvers. Other solvers have also been considered in the context of PSI and OPRF. A garbled bloom filter [DCW13, RR17a] where $\mathsf{row}(i, r)$ is a random weight κ vector of length $m = 2\kappa n$. Another options is to let $\mathsf{row} : \mathbb{G} \times \{0,1\}^\kappa \to \mathbb{G}^m$ be a random function with $m = n + O(\lambda)$. The Bloom filter has a linear time solver but very poor rate while the latter requires $O(n^3)$ time (via Guassian elimination) and near optimal rate. Constructing more efficient solvers remains an open question. With the advent of PaXoS we believe significant progress can be made at achieving improved rates, i.e., $\rho < 2$, while at the same time maintaining a linear running time. Evidence of this is that PaXoS is based on cuckoo hashing which is known to achieve a significantly better rate when the matrix has weight 3 instead of weight 2 used by PaXoS [DRRT18, PSZ18]. Moreover, solvers for such systems have been presented [LM10, KS12], but it is unclear whether they can be made robust enough to succeed with probability $1 - O(2^\lambda)$. As we will see in Section 7, our communication overhead is dominated by $\rho\kappa n$, so the performance of the solver has a direct impact.

PaXoS Details. We now present the PaXoS solver [PRTY20] in detail. Let $M' \in \{0,1\}^{n \times m'}$ be the submatrix formed by the first $m' = 2.4n$ columns of M which itself consists of rows $\mathsf{row}(z_1, r), ..., \mathsf{row}(z_n, r)$. As such, each row of M' has weight 2. The solver first analyses the sparse system formed by M' as follows. Let $\mathcal{G}$ be the graph consisting of m' vertices $\mathcal{V} = [m']$ and the edge set $\mathcal{E} = \{(c_0, c_1) \mid i \in [n] \wedge M'_{i,c_0} = M'_{i,c_1} = 1\}$. That is, for each constraint $v_i = \langle \vec{P}, \mathsf{row}(z_i, r) \rangle = P_{c_0} + P_{c_1} + ...$ there is an edge between vertices $(c_0, c_1) = e_i$. $\mathcal{G}$ is called the cuckoo-graph [PRTY20].

First, let us assume that $\mathcal{G}$ has no cycles and therefore consists of one or more trees. This case can be solved by doing a linear pass over the nodes along tree edges, and assigning values on the way. In particular: (1) Initialize $P_i := 0$ for $i \in [m]$. (2) Let $I \subseteq \mathcal{V}$ s.t. each tree in $\mathcal{G}$ has a single vertex in I and $\overline{I} := \mathcal{V} \setminus I$. (3) Pick an $i \in I$ and for each edge $(j, i) \in \mathcal{E}$ such that $j \in \overline{I}$,

identify $e_k \in \mathcal{E}$, i.e. $M'_{k,i} = M'_{k,j} = 1$, and update $P_j := v_k - P_i$. Note that because $\mathcal{G}$ is acyclic, P_i will not change value later. Update $I := I \cup \{j\}$. Finally, define $I := I \setminus \{i\}$, $\overline{I} := \overline{I} \cup \{i\}$ and if $I \neq \emptyset$, go back to (3).

Observe that this algorithm does not work if $\mathcal{G}$ contains a cycle since at some point in step (3) P_j will have already been updated. To address this, the solver first identifies the so called 2-core graph $\tilde{\mathcal{G}}$ which is the subgraph of $\mathcal{G}$ which only contains the cycles along with any paths between these cycles. Observe that the graph formed by $\mathcal{G} \setminus \tilde{\mathcal{G}}$ is acyclic.

The solver uses Gaussian elimination to solve the constraints contained in $\tilde{\mathcal{G}} = (\tilde{\mathcal{V}}, \tilde{\mathcal{E}})$ with the use of the $m - m'$ additional columns of M. In particular, [PRTY20] show that for $m' = 2.4n$, the size of $\tilde{\mathcal{E}}$ is bounded by $d = O(\lambda)$ with overwhelming probability. Let the actual number of edges in $\tilde{\mathcal{G}}$ be $\tilde{d} < d$. They then consider the submatix $\tilde{M}$ formed by the last $m - m'$ columns of M and the $\tilde{d}$ rows corresponding to edges in $\tilde{\mathcal{G}}$. In their parameterization they set $m = d + \lambda + m'$. As such $\tilde{M}$ is a $(d+\lambda) \times \tilde{d}$ random binary matrix. With probability $1 - O(2^{-\lambda})$ there exists an invertible $\tilde{d} \times \tilde{d}$ submatrix $\tilde{M}^*$ within $\tilde{M}$ [PRTY20]. The $\tilde{d}$ constraints in $\tilde{\mathcal{G}}$ can then be solved for using Gaussian elimination on $\tilde{M}^*$ which requires $O(\tilde{d}^3) = O(\lambda^3)$ time. The remaining P_i values corresponding to $\tilde{\mathcal{G}}$ and $\tilde{M}$ are assigned the value zero, and the remaining constraints in $\mathcal{G} \setminus \tilde{\mathcal{G}}$ can then be solved using the linear time algorithm described above.

X-oblivious PaXoS. We now present a modified scheme detailed in Figure 1 which we denote as XoPaXoS. Looking forward our Circuit PSI protocol will require an additional simulation property of the encode algorithm. Informally, given that the v_i values are uniform, we require the the distribution of $\vec{P}$ is independent of the z_i values. More formally, we will require that the distributions

$$\begin{aligned}
\mathcal{D}_0(z_1, ..., z_n) &:= (\mathsf{Encode}((z_1, v_1), ..., (z_n, v_n), r), r) \\
&\qquad\qquad \text{where } r \leftarrow \{0,1\}^\kappa; v_i \leftarrow \mathbb{G}, \forall i \in [n] \\
\mathcal{D}_1(z_1, ..., z_n) &:= (\vec{P}, r), \qquad \text{where } r \leftarrow \{0,1\}^\kappa; \vec{P} \leftarrow \mathbb{G}^m
\end{aligned}$$

be indistinguishable for any PPT adversary except with probability $2^{-\lambda}$.

However, this does not hold for the [PRTY20] construction outlined above. In particular, the PaXoS algorithm assigns zero to P_i values in two locations. When solving the 2-core using Gaussian elimination some of the column of $\tilde{M}$ are not used and therefore the corresponding P_i are assigned zero. The XoPaXoS scheme rectifies this in Step 5 of Figure 1 by first assigning random values to the redundant P_i positions and then solving

the remaining (fully constrained) system using Gaussian elimination. It is easy to verify that the P_i values output by Gaussian elimination have the desired distribution.

Secondly, when performing the linear pass over the trees of $\mathcal{G}$, a vertex i from each tree is picked and P_i is assigned zero. In Step 7 of XoPaXoS, we again replace this assignment with sampling P_i uniformly from $\mathbb{G}$. Finally, the remaining assignments have the form $P_i := v_k + P_j + ...$ where each assignment contains a distinct uniform v_k value and therefore P_i is uniform as desired. These modifications make the Encode algorithm randomized even for a fixed r. In particular, we assume Encode takes an addition random tape as input from which the uniform P_i values are sampled. We note that the original PaXoS algorithm can be obtained by omitting these addition steps and instead initializing all P_i to zero.

OKVS Extraction. As with [PRTY20], our protocols will construct an OKVS $\vec{P} := \mathsf{Encode}((x_1, H(x_1)), ..., (x_n, H(x_n)))$ where $H : \{0,1\}^* \to \{0,1\}^{\mathsf{out}}$ is a random oracle. An important task in proving the security of our protocols will be for the simulator to extract the effective input for an OKVS generated using a random oracle. In particular, given a OKVS $\vec{P}$, the simulator will extract the effective input as $X' = \{x \mid \mathsf{Decode}(x, \vec{P}) = H(x)\}$. This is accomplished by the simulator checking for all x that have been queried to H.

For [semi] honest parties, this will correspond to a set of size n with overwhelming probability. However, it is possible for malicious parties to construct an OKVS with additional elements in it. Indeed, with good probability the M will be invertible and therefore it is easy to construct a $\vec{P}$ such that the effective X' is of size $n' = m$. It is also possible to construct an OKVS $\vec{P}$ of size m that results in a effective input size of more than m. [PRTY20] gives an information theoretic argument that if $\mathsf{out} = O(\kappa)$ then with overwhelming probability the effective input has size at most $n' = |X'| = O(n)$. Their argument crucially relies on H being a random oracle. They show that if X' was larger than this, then the effect is that H is compressable which contradicts that its a random function. In particular, [PRTY20] proves that

$$\Pr[|\{x \mid \mathsf{Decode}(x, \vec{P}) = H(x)\}| > n'] \leq \frac{\binom{q}{n'}}{2^{(n'-m)\mathsf{out}}} \tag{1}$$

where q is the number of H queries the adversary makes. For example, if $\mathsf{out} = 128$ and $q \leq 2^{100}$ then there is less than a 2^{-128} probability that there exists a set X' with $|X'| \geq 3.2m$.

Strong OKVS Extraction Conjecture. Although it is standard to allow some difference between n and n', we conjecture that is it infeasible to find a set X' with $n' > m$ when the OKVS row hash function row is a one-way-function.

Conjecture 2.1 (Strong OKVS Extraction)**.** *When the row hash function* row *for an OKVS scheme* (Encode, Decode) *is a one-way random oracle and* $H : \{0,1\}^* \to \{0,1\}^{\mathsf{out}}\}$ *is a random oracle with* $\mathsf{out} = O(\kappa)$*, it is computationally infeasible to construct a encoding* $\vec{P}$ *of size* m *such that* $|\{x \mid \mathsf{Decode}(x, \vec{P}) = H(x)\}| > m$*.*

We give some evidence for this conjecture. First, recall that for each $x \in X'$, by definition it holds that

$$\langle \mathsf{row}(x), \vec{P} \rangle = H(x).$$

For the sake of illustration, consider the case that $x \in \mathbb{G}^m$ and define $\mathsf{row}(x) = x$. That is, row is not one-way and the equation above simplifies to

$$\langle x, \vec{P} \rangle = H(x).$$

The astute reader might recognize this formulation as the infamous ROS (Random inhomogeneities in a Overdetermined Solvable system of linear equations) problem which stated that it is computational intractable find a $\vec{P}$ s.t. $|X'| > m$. This assumption was used to construct various blind signatures and was shown to be false[BLL+21]. In particular, [BLL+21] gives an algorithm for constructing a $\vec{P}$ with an effective input size of $n' = m + 1$. This attack on the ROS problem crucially relies on the fact that the attacker has full control over x and that it appears inside the inner product. This allows them to start with an arbitrary set X of size $2m$ which defines a system of equations. This system can be solved to determine a subset of X of size m along with an additional x^* value that will be correctly decoded. [BLL+21] only describes the attack for constructing set of size $m + 1$ but the attack can be extend to find a limited number of additional collisions.

We contrast this with our construction where row is a one-way function. This implies that although the attacker can find x_i and an additional hash value h such that $\langle \mathsf{row}(x_i), \vec{P} \rangle = H(x_i)$ for $i \in [m]$ and $\langle h, \vec{P} \rangle = h$, they are unable to determine the preimage of h since row is a one-way random oracle. We note that for row for the PaXoS scheme to be one-way, we require $\binom{m'}{2}(2^{m-m'}) = O(2^\kappa)$.

We were not able to formally prove this conjecture and leave analyzing it as an interesting open problem. We stress however that the information

theoretical bound of [PRTY20] holds unconditionally and this conjecture only tightens the bound.

We thank Seongkwang Kim and Yongha Son for bringing to our attention that the prior version of this paper did not correctly address the problem of extracting the effective input.

3 Vole Based OPRF

3.1 Vector OLE

The VOLE functionality $\mathcal{F}_{\mathsf{vole}}$ is presented in Figure 2. Let $\mathbb{F}$ be some finite field, e.g., $\mathbb{F} = GF(2^\kappa)$. The parties have no input. The Sender obtains a random value $\Delta \in \mathbb{F}$ and a random vector $\vec{B} \in \mathbb{F}^m$. The Receiver obtains a random vector $\vec{A'} \in \mathbb{F}^m$ and the vector

$$\vec{C} = \vec{A'}\Delta + \vec{B}.$$

That is, the i-th position of $\vec{C}$ is equal to $A'_i\Delta + B_i$. We note that several definitions of VOLE have been introduced in the literature, for both chosen-input and random variants [ADI+17, BCGI18, BCG+19, WYKW20]. In the context of these previous works, the functionality described here can be seen as *random reversed vector OLE*. We refer to it as VOLE for simplicity.

A naive implementation of a VOLE generator would be to run a two-party multiplication protocol (e.g., Gilboa multiplication [Gil99]) for each $i \in [m]$. The drawback here is that communication is linear in m. Recently, significant advances have been made in developing VOLE generators with *sub-linear* communication. [BCGI18] presented the first protocols in that direction based on the LPN assumption. Their two protocols, a *primal* and a *dual* variant, rely on two different flavors of LPN. While the primal variant can be instantiated from LPN with cheap local linear codes, its communication grows asymptotically with the square-root of the output size. The dual variant, on the other hand, allows for logarithmic communication, but requires more computation.

A first implementation of a primal VOLE generator was provided in [SGRR19], while concurrently, [BCG+19] provide an implementation of dual VOLE over binary fields. Recently, [YWL+20] improved on the protocols of [SGRR19], significantly reducing the communication overhead. Their main observation is that the primal VOLE generator works by expanding a size-$O(\sqrt{m})$ random seed correlation to a size-m pseudorandom correlation. Now by applying this expansion iteratively, they manage to get VOLE

Parameters: There are two parties, a Sender and a Receiver. Let $\mathbb{F}$ be a field. Let m denote the size of the output vectors.

Functionality: Upon receiving $(\mathsf{sender}, \mathsf{sid})$ from the Sender and $(\mathsf{receiver}, \mathsf{sid})$ from the Receiver.

- If the Receiver is malicious, wait for them to send $\vec{C}, \vec{A} \in \mathbb{F}^m$. Sample $\Delta \leftarrow \mathbb{F}$ and compute $\vec{B} := \vec{C} - \vec{A}\Delta$. Otherwise,
- If the Sender is malicious, wait for them to send $\vec{B} \in \mathbb{F}^m, \Delta \in \mathbb{F}$. Sample $\vec{A} \leftarrow \mathbb{F}^m$ and compute $\vec{C} := \vec{B} + \vec{A}\Delta$. Otherwise,
- Sample $\vec{A}, \vec{B} \leftarrow \mathbb{F}^m, \Delta \leftarrow \mathbb{F}$ and compute $\vec{C} := \vec{B} + \vec{A}\Delta$.

The functionality sends $\Delta, \vec{B}$ to the Sender and $\vec{C} := \vec{A}\Delta + \vec{B}, \vec{A}$ to the Receiver.

Figure 2: Ideal functionality $\mathcal{F}_{\mathsf{vole}}$ of random reversed Vector-OLE (vole).

correlations of size m from a much shorter seed. Each expansion still takes $O(\sqrt{m})$ communication, but as [YWL+20] show, the LPN security parameters can be optimized so that the concrete communication complexity is still far below the non-iterative approach. Since they focus on the application of VOLE to correlated OT, the implementation of [YWL+20] is limited to binary fields. However, [WYKW20] extend this paradigm to VOLE over general fields, for which they also provide a consistency check for malicious security. In our implementation (Section 7), we use an improved version of the library of [SGRR19], incorporating the iterative approach of [YWL+20] and the consistency check of [WYKW20].

3.2 Malicious Secure Oblivious PRF.

We now present our main (multi-input) OPRF construction in the $\mathcal{F}_{\mathsf{vole}}$-hybrid model. Our construction Π_{oprf} is detailed in Figure 4 and realizes the functionality $\mathcal{F}_{\mathsf{oprf}}$ from Figure 3 in the malicious setting. Our protocol will make use of two random oracles, $\mathsf{H} : \mathbb{F} \times \mathbb{F} \to \{0,1\}^{\mathsf{out}}, \mathsf{H}^{\mathbb{F}} : \mathbb{F} \to \mathbb{F}$.

First, the receiver will solve the system

$$\begin{bmatrix} \mathsf{row}(x_1) \\ \dots \\ \mathsf{row}(x_n) \end{bmatrix} \vec{P}^{\intercal} = (\mathsf{H}^{\mathbb{F}}(x_1), \dots, \mathsf{H}^{\mathbb{F}}(x_n))^{\intercal}$$

as a function of the set X. Depending on the choice of row this can correspond to polynomial interpolation, a bloom filter solver, PaXoS or some

other fast solver, see Section 2. Recall that for all $x \in X$ it holds that $\mathsf{Decode}(\vec{P}, x) = \langle \mathsf{row}(x), \vec{P} \rangle = \mathsf{H}^{\mathbb{F}}(x)$ and that Decode is a linear function in $\vec{P}$. Another important property is that $\mathsf{Decode}(\vec{P}, x) = \mathsf{H}^{\mathbb{F}}(x)$ only for the elements in the set X, except the negligible probability[2].

The parties first invoke $\mathcal{F}_{\mathsf{vole}}$ where the Receiver obtains $\vec{A'}, \vec{C} \in \mathbb{F}^m$ while the Sender obtains $\Delta \in \mathbb{F}, \vec{B} \in \mathbb{F}^m$. Recall that $\vec{C} = \vec{A'}\Delta + \vec{B}$. The Receiver computes $\vec{A} := \vec{P} + \vec{A'}$ and sends this to the Sender who computes $\vec{K} := \vec{B} + \vec{A}\Delta$. The parties will run a coin flipping protocol to then choose a random $w \leftarrow \mathbb{F}$.

The Sender defines their the PRF function as

$$F(x) = \mathsf{H}(\mathsf{Decode}(\vec{K}, x) - \Delta \mathsf{H}^{\mathbb{F}}(x) + w, x).$$

The Receiver outputs the values

$$X' := \{\mathsf{H}(\mathsf{Decode}(\vec{C}, x) + w, x) \mid x \in X\}.$$

To understand why $F(x) = \mathsf{H}(\mathsf{Decode}(\vec{C}, x) + w, x)$ for $x \in X$, observe that

$$\begin{aligned}
\mathsf{Decode}(\vec{K}, x) - \Delta \mathsf{H}^{\mathbb{F}}(x) =& \mathsf{Decode}(\vec{B} + \vec{P}\Delta + \vec{A'}\Delta, x) - \Delta \mathsf{H}^{\mathbb{F}}(x) \\
=& \langle \vec{B} + \vec{P}\Delta + \vec{A'}\Delta, \mathsf{row}(x) \rangle - \Delta \mathsf{H}^{\mathbb{F}}(x) \\
=& \langle \vec{B} + \vec{A'}\Delta, \mathsf{row}(x) \rangle + \langle \vec{P}\Delta, \mathsf{row}(x) \rangle - \Delta \mathsf{H}^{\mathbb{F}}(x) \\
=& \langle \vec{C}, \mathsf{row}(x) \rangle + \Delta \langle \vec{P}, \mathsf{row}(x) \rangle - \Delta \mathsf{H}^{\mathbb{F}}(x) \\
=& \langle \vec{C}, \mathsf{row}(x) \rangle + \Delta \mathsf{H}^{\mathbb{F}}(x) - \Delta \mathsf{H}^{\mathbb{F}}(x), \qquad \forall x \in X \\
=& \mathsf{Decode}(\vec{C}, x), \qquad \forall x \in X
\end{aligned}$$

When this is decoded at any $x \in X$ recall that $\mathsf{Decode}(\vec{P}, x) = \mathsf{H}^{\mathbb{F}}(x)$ and therefore the receiver will compute the correct value $\mathsf{Decode}(\vec{C}, x)$. Also recall that this encoding has the property that at all other locations $x' \notin X$ it holds that $\mathsf{Decode}(\vec{P}\Delta, x') \neq \mathsf{H}^{\mathbb{F}}(x')$ and therefore the outputs will disagree. Finally, we obtain an OPRF by hashing away the linear correlation using the hash function H.

The final random oracle H call also contains to x to facilitate extraction in the case of a malicious Sender. In particular, our functionality requires the OPRF to effectively behave like a random oracle for the Sender. This differs from a normal PRF where there is no security with respect to the party holding the secret key.

[2] In the case of a malicious Receiver and depending on the choice of row, it may be possible for $|X| > n$ with noticeable probability. However, for PaXoS this can be bounded as $|X| \leq m \approx 2.4n$ while interpolation ensures that $|X| \leq m = n$.

Parameters: There are two parties, a Sender and a Receiver. Let $n, n' \in \mathbb{Z}$ be parameters such that if Receiver is malicious then $|X| < n'$ and otherwise $|X| = n$. Let $\mathsf{out} \in \mathbb{Z}$ be the output bit length.

Functionality: Upon input $(\mathsf{sender}, \mathsf{sid})$ from the Sender and $(\mathsf{receiver}, \mathsf{sid}, X)$ from the Receiver, the functionality samples $F : \mathbb{F} \to \{0,1\}^{\mathsf{out}}$ and sends $X' := \{F(x) \mid x \in X\}$ to the Receiver.
Subsequently, upon input $(\mathsf{sender}, \mathsf{sid}, y)$ from the Sender, the functionality returns $F(y)$ to the Sender.

Figure 3: Ideal functionality $\mathcal{F}_{\mathsf{oprf}}$ batched Oblivious PRF.

Theorem 3.1. *The Protocol Π_{oprf} realizes the $\mathcal{F}_{oprf}$ functionality against a Malicious adversary in the random oracle, $\mathcal{F}_{vole}$-hybrid model.*

Proof. First observe that the protocol is correct. We prove the following two Lemmas:

Lemma 3.2. *The Protocol Π_{oprf} realizes the $\mathcal{F}_{oprf}$ functionality against a Malicious Sender $\mathcal{A}$ in the random oracle, $\mathcal{F}_{vole}$-hybrid model.*

Proof. The simulator $\mathcal{S}$ interacts with the Sender as follows:

- $\mathcal{S}$ plays the role of $\mathcal{F}_{\mathsf{vole}}$. When $\mathcal{A}$ sends $(\mathsf{sender}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{vole}}$, $\mathcal{S}$ waits for $\mathcal{A}$ to send $\Delta, \vec{B}$.
- On behalf of the Receiver, $\mathcal{S}$ sends uniform $r, \vec{A}$ to $\mathcal{A}$.
- Whenever $\mathcal{A}$ queries $\mathsf{H}(q, y)$, if $q = \langle \vec{K}, \mathsf{row}(y, r) \rangle - \Delta \mathsf{H}^{\mathbb{F}}(y) + w$ and $\mathsf{H}(q, y)$ has not previously been queried, $\mathcal{S}$ sends $(\mathsf{sender}, \mathsf{sid}, y)$ to $\mathcal{F}_{\mathsf{oprf}}$ and programs $\mathsf{H}(q, y)$ to the response. Otherwise H responds normally.

To prove that this simulation is indistinguishable consider the following hybrids:

- Hybrid 0: The same as the real protocol except $\mathcal{S}$ in this hybrid plays the role of $\mathcal{F}_{\mathsf{vole}}$.
- Hybrid 1: $\mathcal{S}$ in this hybrid samples $\vec{A}$ uniformly as opposed to $\vec{A} := \vec{P} + \vec{A'}$. Since $\vec{A'}$ is distributed uniformly in the view of the $\mathcal{A}$, this hybrid has an identical distribution.
- Hybrid 2: When $\mathcal{S}$ in this hybrid samples r, it aborts if any of the $\mathsf{row}(\cdot, r)$ queries have previously been made. Since r is sampled uniformly the probability of this is $O(2^{-\kappa})$ and therefore this hybrid is indistinguishable from the previous.

- Hybrid 3: $\mathcal{S}$ in this hybrid does not call Encode, and so does not abort if Encode fails. Since none of $\mathsf{row}(\cdot, r)$ queries have previously been made, the PaXoS cuckoo-graph is uniformly sampled from all (n, m)-cuckoo graphs and therefore the probability of abort is bounded by $2^{-\lambda}$ [PRTY20]. Therefore this hybrid is statistically indistinguishable from the previous. Observe that this hybrid no longer uses the Receiver's input.

- Hybrid 4: Whenever $\mathcal{A}$ queries $\mathsf{H}(q, y)$ after receiving $\vec{A}$, if $q = \langle \vec{K}, \mathsf{row}(y, r) \rangle + w$ and $\mathsf{H}(q, y)$ has previously been queried, this hybrid aborts. Otherwise it sends $(\mathsf{sender}, \mathsf{sid}, y)$ to $\mathcal{F}_{\mathsf{oprf}}$ and programs $\mathsf{H}(q, y)$ to the response.

  Observe that r is uniformly distributed prior to it being sent. Therefore, any given $q = \langle \vec{K}, \mathsf{row}(y, r) \rangle - \mathsf{H}^{\mathbb{F}}(y)\Delta + w$ is similarly distributed and $\mathcal{A}$ has a negligible probability of previously querying $\mathsf{H}(q, y)$. We conclude that this hybrid is indistinguishable from the simulation.

Lemma 3.3. *The Protocol Π_{oprf} realizes the $\mathcal{F}_{\mathsf{oprf}}$ functionality against a Malicious Receiver $\mathcal{A}$ in the random oracle, $\mathcal{F}_{\mathsf{vole}}$-hybrid model.*

Proof. The simulator $\mathcal{S}$ interacts with the Receiver as follows:

- $\mathcal{S}$ plays the role of $\mathcal{F}_{\mathsf{vole}}$ and receives $\vec{A}', \vec{C}$ from $\mathcal{A}$.

- When $\mathcal{A}$ sends $r, \vec{A}$, $\mathcal{S}$ computes $\vec{P} := \vec{A} - \vec{A}'$. For each of the previous $\mathsf{H}^{\mathbb{F}}(x)$ queries made by $\mathcal{A}$, $\mathcal{S}$ checks if $\mathsf{Decode}(\vec{P}, x, r) = \mathsf{H}^{\mathbb{F}}(x)$ and if so adds x to set X. $\mathcal{S}$ sends $(\mathsf{Receiver}, \mathsf{sid}, X)$ to $\mathcal{F}_{\mathsf{oprf}}$ and receives $\{F(x) \mid x \in X\}$ in response.

- $\mathcal{S}$ samples $w \leftarrow \{0,1\}^{\kappa}$. For each $x \in X$, $\mathcal{S}$ programs $\mathsf{H}(\mathsf{Decode}(\vec{C}, x, r) + w, x) := F(x)$. $\mathcal{S}$ sends w to $\mathcal{A}$.

To prove that this simulation is indistinguishable consider the following hybrids:

- Hybrid 0: The same as the real protocol except the $\mathcal{S}$ plays the role of $\mathcal{F}_{\mathsf{vole}}$. When $\mathcal{A}$ sends $(\mathsf{receiver}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{vole}}$, $\mathcal{S}$ waits to receive $\vec{A}', \vec{C}$.

- Hybrid 1: When $\mathcal{A}$ sends $r, \vec{A}$, $\mathcal{S}$ in this hybrid computes $\vec{P} := \vec{A} - \vec{A}'$. For each of the previous $\mathsf{H}^{\mathbb{F}}(x)$ queries made by $\mathcal{A}$, this hybrid checks if $\mathsf{Decode}(\vec{P}, x, r) = \mathsf{H}^{\mathbb{F}}(x)$ and if so adds x to set X. This hybrid sends $(\mathsf{Receiver}, \mathsf{sid}, X)$ to $\mathcal{F}_{\mathsf{oprf}}$ and receives $\{F(x) \mid x \in X\}$ in response.

- Hybrid 2: $\mathcal{S}$ in this hybrid does not sample w^{s} at the beginning of the protocol and sends a random value for c^{s} instead $H(w^{\mathsf{s}})$. Right before w^{s} is should be sent, $\mathcal{S}$ samples w^{s} and programs $H(w^{\mathsf{s}}) := c^{\mathsf{s}}$. Conditioned on $H(w^{\mathsf{s}})$ not previously being queried, this hybrid is identically distributed and therefore indistinguishable in general since w^{s} is uniform.

- Hybrid 3: When $w^{\mathsf{s}} \leftarrow \{0,1\}^{\kappa}$ is sampled, $\mathcal{S}$ in this hybrid aborts if any $\mathsf{H}(\mathsf{Decode}(\vec{C}, x, r) + w, x)$ has been made by $\mathcal{A}$. Since w^{s} was just sampled, each $\mathsf{Decode}(...) + ... + w^{\mathsf{s}}$ is uniform and therefore the probability of abort is at most $O(2^{-\kappa})$.

  $\mathcal{S}$ in this hybrid programs $\mathsf{H}(\mathsf{Decode}(\vec{C}, x, r) + w, x) := F(x)$ for all $x \in X$ and sends w^{s} to $\mathcal{A}$. Since the $F(x)$ are uniform, programming H does not change the distribution.

- Hybrid 4: $\mathcal{S}$ in this hybrid aborts if $\mathcal{A}$ ever makes an $\mathsf{H}(v, x)$ query such that $(v, x) \in \{(\mathsf{Decode}(\vec{K}, x, r) - \Delta\mathsf{H}^{\mathbb{F}}(x) + w, x) \mid x \in \mathbb{F} \setminus X\}$. Observe that

$$\begin{aligned}
\mathsf{Decode}(\vec{K}, x, r) - \Delta\mathsf{H}^{\mathbb{F}}(x) &= \langle \vec{K}, \mathsf{row}(x, r)\rangle - \Delta\mathsf{H}^{\mathbb{F}}(x) \\
&= \langle \vec{B} + \vec{P}\Delta + \vec{A}'\Delta, \mathsf{row}(x, r)\rangle - \Delta\mathsf{H}^{\mathbb{F}}(x) \\
&= \langle \vec{C} + \vec{P}\Delta, \mathsf{row}(x, r)\rangle - \Delta\mathsf{H}^{\mathbb{F}}(x) \\
&= \Delta(\langle \vec{P}, \mathsf{row}(x, r)\rangle - \mathsf{H}^{\mathbb{F}}(x)) + \langle \vec{C}, \mathsf{row}(x, r)\rangle
\end{aligned}$$

  and recall that Δ is uniformly distributed in the view of $\mathcal{A}$. So for all x s.t. $\langle \vec{P}, \mathsf{row}(x, r)\rangle \neq \mathsf{H}^{\mathbb{F}}(x)$, the distribution of $\Delta(\langle \vec{P}, \mathsf{row}(x, r)\rangle - \mathsf{H}^{\mathbb{F}}(x))$ is uniform in the view of $\mathcal{A}$. Now consider the case that $\langle \vec{P}, \mathsf{row}(x, r)\rangle = \mathsf{H}^{\mathbb{F}}(x)$. W.l.o.g., let us assume that all $\mathsf{H}^{\mathbb{F}}(x)$ queries are made prior to sending $\vec{A}$. Recall from Section 2 that $\vec{P} \in \mathbb{F}^m$ where $m \approx 2.4n$ for PaXoS and that by Equation 1 the effective input $X' = \{x \mid \mathsf{Decode}(x, \vec{P}) = H(x)\}$ is of size at most $n' := |X'| = O(n)$ with overwhelming probability. Moreover, by Conjecture 2.1, it holds that $n' = m$.

4 Private Set Intersection

Using our OPRF protocol from the previous section, we now obtain a PSI protocol via the well known transformation shown in Figure 6. The ideal functionality for PSI is given in Figure 5. Given a malicious or semi-honest

OPRF, this transformation achieves malicious or semi-honest security, respectively. While the general transformation is known and implicitly or explicitly used by used by [CKT10, DCW13, RR17a, PRTY19, PRTY20, CM20], we provide a tight analysis in the malicious setting which reduces our communication by 20% to 50% compared to [CKT10, PRTY20].

The OPRF to PSI transformation works as follows. The PSI receiver sends their set X to the OPRF functionality $\mathcal{F}_{\mathsf{oprf}}$ and receives back $F(x)$ for all $x \in X$. The sender queries $\mathcal{F}_{\mathsf{oprf}}$ to learn $F(y)$ for their y. The sender sends $Y' := \{F(y) \mid y \in Y\}$ to the receiver who can compute $X \cap Y := \{x \mid x \in X \wedge F(x) \in Y'\}$.

To ensure the correctness of this protocol it is crucial that there are not any spurious collisions between the $F(x)$ and $F(y)$ values. In particular, since F is a random function it is possible that $x \neq y \wedge F(x) = F(y)$. In the semi-honest setting, the standard approach is to define the output domain of F to be $\{0,1\}^{\mathsf{out}}$ where $\mathsf{out} := \lambda + \log_2(n_{\mathsf{x}} n_{\mathsf{y}})$. Since the X, Y are fixed prior to randomly sampling F, the probability for any $x \notin Y$ to result in $F(x) \in Y'$ is purely a statistical problem[3]. In particular,

$$\Pr_{x,Y,F}[F(x) \in \{F(y) \mid y \in Y\} \wedge x \notin Y] = 2^{-\mathsf{out}} n_{\mathsf{y}} = 2^{-\mathsf{out}+\log_2(n_{\mathsf{y}})}.$$

If we take the union bound over $x \in X$, the overall probability of a collision is $n_{\mathsf{x}} 2^{-\mathsf{out}+\log_2(n_{\mathsf{y}})} = 2^{-\mathsf{out}+\log_2(n_{\mathsf{y}} n_{\mathsf{x}})} = 2^{-\lambda}$.

In the malicious setting the situation is complicated by the fact that the simulator must extract the sender's set Y by observing the sender's $\mathcal{F}_{\mathsf{oprf}}$ queries and the value of Y'. The folklore approach is to extract $Y := \{y \mid y \in Y^* \wedge F(y) \in Y'\}$ where Y^* is the set of inputs the sender queried the $\mathcal{F}_{\mathsf{oprf}}$ at. However, in the event that there exists distinct $y, y' \in Y^*$ s.t. $F(y) = F(y')$, then more than one y is extracted for each $y^* \in Y^*$.

The probability that there exists distinct $y, y' \in Y^*$ s.t. $F(y) = F(y')$ is at most $2^{-\mathsf{out}+2\log_2(n_{\mathsf{y}}{}^*)}$ where $n_{\mathsf{y}}{}^* := |Y^*|$. Therefore, it is expected to occur when $n_{\mathsf{y}}{}^* \geq 2^{\mathsf{out}/2}$. As such, in the folklore analysis and that of [CKT10, PRTY20], it is required that $\mathsf{out} := 2\kappa$ in order for the security argument to hold.

We now present a new extraction procedure which allows $\mathsf{out} = \kappa$. In our protocol this effectively reduces the sender's communication by half, therefore reducing the overall communication by half when $|X| \ll |Y|$.

Our extraction procedure is to only extract $y \in Y^*$ if it is distinct. Intuitively, the reason security still holds is that collisions within Y^* are

[3] In the $\mathcal{F}_{\mathsf{oprf}}$ hybrid where F is truly random.

unlikely to collide with the receiver's set X. In particular, the receiver's set X is first fixed and then the function F is sampled. Thus, the probability that there exists a $y \in Y^*$ and $y \notin X$, yet $F(x) = F(y)$ is at most

$$2^{-\mathsf{out}+\log_2(n_{\mathsf{x}}n_{\mathsf{y}}{}^*)} = O(2^{-\mathsf{out}+\log_2(\kappa)+\log_2(n_{\mathsf{y}}{}^*)})$$

and therefore if $\mathsf{out} := \kappa$ the probability is $O(2^{-\kappa+\log(\kappa)+\log(n_{\mathsf{y}}{}^*)})$. Concretely, if $\kappa = 128, n_{\mathsf{x}} = 2^{30}$ then the sender would have to make an expected $n_{\mathsf{y}}{}^* = 2^{98}$ $\mathcal{F}_{\mathsf{oprf}}$ queries in order to expect to distinguish as opposed to 2^{49} queries via the folklore analysis.

Theorem 4.1. *The Protocol Π_{psi} realizes the $\mathcal{F}_{psi}$ functionality against a Malicious adversary in the $\mathcal{F}_{oprf}$-hybrid model.*

Proof. Consider a malicious sender. The simulator interacts with the sender as:

- The simulator plays the role of $\mathcal{F}_{\mathsf{oprf}}$. The simulator observes all the $(\mathsf{sender}, \mathsf{sid}, y)$ messages. Let Y^* be the set of all such y.
- When the sender sends Y', the simulator computes $\hat{Y} := \{y \mid y \in Y^* \wedge \nexists y' \in Y^* \; s.t. \; y \neq y' \wedge F(y) = F(y')\}$ and extracts $Y := \{y \mid y \in \hat{Y} \wedge F(y) \in Y'\}$ and sends Y to $\mathcal{F}_{\mathsf{psi}}$.

First, conditioned on there not being any $F(y) = F(y')$ collisions, it is easy to verify that the simulation above is correct and indistinguishable.

Now consider some collision $F(y) = F(y')$. Observe that the simulator only needs to extract y, y' if there is a noticeable probability of one of them being in X. W.l.o.g., let us assume $y \in X$. Therefore, consider the probability of $F(y') = F(x)$ for some $x \in X$. Since $|X| = n_{\mathsf{x}} = O(\kappa)$, the probability of the sender finding such a (target preimage) collision is $O(2^{-\kappa})$.

Consider a malicious receiver. The simulator is as follows:

- The simulator plays the role of $\mathcal{F}_{\mathsf{oprf}}$.
- When the receiver sends $(\mathsf{receiver}, \mathsf{sid}, X)$ to $\mathcal{F}_{\mathsf{oprf}}$, the simulator observes X and sends X' back as the $\mathcal{F}_{\mathsf{oprf}}$ would.
- The simulator forwards X to $\mathcal{F}_{\mathsf{psi}}$ and receives $Z = X \cap Y$ in response.
- The simulator computes Y' as containing all $\{F(z) \mid z \in Z\}$ along with $n_{\mathsf{y}} - |Z|$ uniform values from $\{0,1\}^{\mathsf{out}} \setminus X'$. The simulator sends Y'.

This simulation is identical to the real protocol except for the dummy items being sampled from $\{0,1\}^{\mathsf{out}} \setminus X'$ instead of $\{0,1\}^{\mathsf{out}}$. However, since $2^{\mathsf{out}} - |X| = O(2^{\kappa})$ this change is indistinguishable.

5 Oblivious Programmable PRF

We now turn our attention to constructing our circuit PSI protocol. To achieve this, we first construct a type of protocol known as an oblivious programmable PRF (OPPRF). The functionality is shown in Figure 7. The sender has a set of input pairs $(y_1, z_1), ..., (y_n, z_n)$. The functionality samples a key k such that $F_k(y_i) = z_i$ and at all other input points it outputs a random value. The receiver on input points $x_1, ..., x_n$ then obtains $F_k(x_i)$ for all i.

We instantiate this functionality using an OPRF protocol, and the $\mathsf{XoPaXoS}$ solver. The parties call the OPRF functionality $\mathcal{F}_{\mathsf{oprf}}$ with X being the receiver's input. The sender obtains k while the receiver obtains $X' = \{F_k(x_1), ..., F_k(x_n)\}$. The sender constructs a solver for $\vec{P}$ such that $\mathsf{Decode}(\vec{P}, y_i) = z_i - F_k(y_i)$ using $\mathsf{XoPaXoS}$ and sends $\vec{P}$ to the receiver who then outputs $x_i^* := x_i' + \mathsf{Decode}(\vec{P}, x_i)$ for all i. When $x_i = y_j$, then

$$x^* := F_k(x_i) + \mathsf{Decode}(\vec{P}, x_i) = F_k(x_i) + z_j - F_k(x_i) = z_j.$$

The sender outputs the key $k^* := (k, \vec{P})$ where the OPPRF function is defined as $F^*_{k^*}(x) := F_k(x) + \mathsf{Decode}(P, x)$.

With respect to security, first observe that the v_i values outside the intersection are information theoretically hidden in the $\mathcal{F}_{\mathsf{oprf}}$ hybrid. What remains to be shown is that the distribution of $\vec{P}$ does not depend on $Y \setminus X$. Recall from Section 2 that this is the exact issue $\mathsf{XoPaXoS}$ addresses compared to PaXoS. Intuitively, $\mathsf{XoPaXoS}$ ensures that each position of $\vec{P}$ is either assigned a uniformly random value or is the sum of previous positions and some $z_i - F_k(y_i)$. We prove security of this protocol in Theorem 5.1.

Theorem 5.1. *The Protocol Π_{opprf} realizes the $\mathcal{F}_{\mathsf{opprf}}$ functionality against a semi-honest adversary in the $\mathcal{F}_{\mathsf{oprf}}$-hybrid model.*

Proof. Consider a semi-honest sender. Observe that the protocol is correct. Since the receiver does not send any messages the simulation is trivial.

Consider a malicious receiver. The simulator generates the receiver's transcript as follows:

- The simulator samples uniform values $F_k(x)$ for $x \in X$.
- The simulator sends X to $\mathcal{F}_{\mathsf{opprf}}$ functionality and receives back $x'_1, ..., x'_n$.
- Samples $\vec{P}$ uniformly from all vectors such that $\mathsf{Decode}(\vec{P}, x_i) = x'_i - F_k(x_i)$.

- The simulator outputs $(\{F_k(x) \mid x \in X\}, \vec{P})$ as the transcript.

Clearly the $\mathbb{F}_k(x)$ values are identically distributed. What remains to be shown is that $\vec{P}$ has the same distribution as it would in the real protocol. Recall from Section 2 that XoPaXoS assign values to $\vec{P}$ in four ways

- During Step 5, $P_i \leftarrow \mathbb{G}$ for $i \in C'$. Recall that Step 4 identifies $\tilde{d}$ of the last $d + \lambda$ columns which form an invertible matrix for the 2-core. These columns are indexed by C. Then C' is defined as $C' = \{j \mid i \in R, M'_{i,j} = 1\} \cup ([d + \lambda] \setminus C + m')$ indexes all positions of $\vec{P}$ which interact with the 2-core along with all of the last $d + \lambda$ columns which are not used to invert.

- Next, in Step 6, the remaining $\tilde{d}$ positions of $\vec{P}$ corresponding are assigned a value such that $\mathsf{Decode}(\vec{P}, y_i) = v'_i - F_k(y_i)$ for the i in the 2-core which is equivalent to solving

$$\tilde{M}^*(P_{C_1+m'}, ..., P_{C_{\tilde{d}}+m'})^\intercal = (y'_{R_1}, ..., y'_{R_{\tilde{d}}})^\intercal.$$

  Since this is a fully determined system, there is exactly one solution.

- In Step 7 a single node i from each tree in G is assigned a uniform value.

- Lastly, observe that the rest of the system is fully determined. That is, each the the remaining P_i position are assigned a value with the form

$$P_i := v'_k - F_k(y_k) - \sum_{j \in \{...\}} P_j.$$

The analysis above can be reordered such that Step 5, 7 are performed first. Then there is exactly one solution to the correctness constraint.

6 Circuit PSI

We now construct a circuit PSI protocol from our OPPRF. Our construction (Figure 10) builds on the approach of [PSTY19], using our novel XoPaXoS and VOLE-based OPPRF from the previous section. As we will see in the experiments (Section 7), this translates into a significant speedup compared to [PSTY19]. The ideal functionality for circuit PSI is given in Figure 9. It allows both sender and receiver to input a set of associated values, which

will be secret-shared alongside the elements in the intersection. The associated values corresponding to elements in the intersection can then be used in any subsequent MPC phase, and could for example be used to compute sums [IKN+20] or inner products [SGRP19] of the intersection. Since our protocol is effectively the same as [PSTY19] with the substitution of our OPPRF and $\mathcal{F}_{\mathsf{2pc}}$ implementation, we defer the proof of security to [PSTY19].

Cuckoo Hashing. We make use of a data structure known as a cuckoo hash table. Given a set X, one can create a hash table T of size $m = \epsilon|X|$. This table is parameterized by k hash functions $h_1, \ldots, h_k : \{0,1\}^* \to \{1, 2, ..., m\}$. There is a procedure [PSZ18, DRRT18] s.t. with overwhelming probability for all $x \in X$, x can be storied in T at $T[h_j(x)]$ for a $j \in [k]$, and only one item will be stored at any position of T. We discuss concrete parameter choices for ϵ and k in Section 7.2.

We will also refer to a procedure known as *simple hashing* of a set Y where we store $y \in Y$ at *all* locations $T[h_j(y)]$. For simple hashing, each position of T may hold more than one value. It can be shown that if the table has $m = O(|Y|)$ positions, then any given location of the table will hold at most $O(\log |Y|)$ items.

Protocol. The full circuit PSI protocol is constructed using the OPPRF and cuckoo hashing. The receiver will construct a cuckoo hash table T_{x} of their set X. The sender will construct a simple has table T_{y} of their set Y.

For each $i \in [m]$ the sender will sample a random value $r_i \leftarrow \{0,1\}^\ell$ where $\ell := \lambda + \log_2 m$. For all i and $y \in T_{\mathsf{y}}[i]$, the sender will construct a list $L = \{(y', r_i)\}$ where $y' = H(y, j)$ and j is defined such that $i = h_j(y)$. That is, j is the hash function index that mapped y to this bin. The receiver constructs set X' which is defined as the collection of all $H(x, j)$ such that x is stored at $T_{\mathsf{x}}[h_j(x)]$. The sender then provides L as their input to $\mathcal{F}_{\mathsf{opprf}}$ while the receiver inputs X'. In response the receiver obtains the set X^*.

As an explanation of this, let us focus on some bin index i such that x was mapped to bin $T_{\mathsf{x}}[i]$ due to hash function h_j, i.e., $T_{\mathsf{x}}[i] = x$ and $h_j(x) = i$. Furthermore, let us assume that there is some $y \in Y$ s.t. $x = y$. Since the sender did simple hashing, they too mapped y to bin $T_{\mathsf{y}}[i]$ since $h_j(y) = i$. For this y, they programmed the OPPRF with the pair $(H(y, j), r_i)$. When the receiver inputs $H(x, j)$ to the OPPRF they receive the value r_i in response. If $x \notin Y$, then the receiver will receive a random value. Therefore, for each i, the receiver now has a value r'_i which is equal to r_i (held by the sender) if $T_{\mathsf{x}}[i] \in Y$ and otherwise r'_i is random per the

OPPRF security definition.

The final step of the protocol is to use a generic MPC protocol to compare each r'_i with r_i to check if they are equal. The output of this generic MPC will be secret shared which will be the output of the protocol.

In the event that the sender has "associated values", they will program the OPPRF with $L = \{(y', r_i||\tilde{y} \oplus w_i)\}$ where $\tilde{y}$ is the associated value for y', and w_i is an random value that the sender samples for each bin $i \in \{1, 2, ..., m\}$ in the same way as r_i. The receiver will then obtain $r'_i||w'_i$ from the OPPRF protocol for each i. The generic MPC will then take as input $\{(r'_i, w'_i)\}$ from the receiver and $\{(r_i, w_i, \tilde{x})\}$ from the sender. For each i the MPC computation will compute $q_i := (r'_i = r_i)$ and $z_i := q_i \cdot \big((w'_i \oplus w_i)||\tilde{x}\big)$ and then output secret shares of q_i and z_i.

7 Performance Evaluation

7.1 Theoretical Comparison.

All protocols compared here are largely based on efficient symmetric key primitives – with the exception of the DH-PSI protocol – and can be instantiated with $O(n_{\mathsf{x}} + n_{\mathsf{y}})$ running time. Since these protocols are asymptotically similar, it becomes difficult to compare them. As we do below, one metric is to implement the protocol and compare their running times. However, the quality of the implementation has a large impact on running time. Arguably a more objective metric is the total communication which is independent of the implementation.

Table 1 shows a theoretical comparison of the communication required by various PSI protocols. We present the communication overhead in three ways. The general case in terms of $n_{\mathsf{x}}, n_{\mathsf{y}}, \kappa, \lambda$; when we fix $\kappa = 128, \lambda = 40$; and when we fix all the parameters. Many protocols contain addition parameters that allow a for some type of tradeoff. For these we chose representative values.

Our semi-honest protocol requires sending $\rho\kappa n_{\mathsf{x}} + (\lambda + \log(n_{\mathsf{y}} n_{\mathsf{y}}))n_{\mathsf{y}}$ bits plus the overhead of performing a VOLE of size ρn_{x}. Here, ρ is the rate of the linear system solver which is being employed by the protocol. We consider two values of ρ. The first is $\rho = 2.4$ which corresponds to the PaXoS solver while the second is $\rho = 1$ when Vandermonde/interpolation solver is used.

To estimate the overhead of the VOLE protocol we experimentally determined that our implementation requires a total of $2^{17}\kappa \sqrt[20]{n_{\mathsf{x}}}$ bits. We note that this is the approximate overhead of our implementation and may not

Table 1: Comparison of theoretical communication cost of various PSI protocols. Several protocols have additional parameters which have been *approximated* in terms of κ, λ. In particular, the coefficients shown below often vary (non-linearly) as a function of n, κ, λ. In these cases we chose representative values. The third column contains the overhead for fixed $\lambda = 40, \kappa = 128$ while the last three columns also fix the set sizes.

Protocol	**Communication**		$n = n_\mathsf{y} = n_\mathsf{x}$		
			2^{16}	2^{20}	2^{24}
Semi-Honest					
DH-PSI	$4\kappa n_\mathsf{x} + (\lambda + \log(n_\mathsf{x} n_\mathsf{y}))n_\mathsf{y}$	$512n_\mathsf{x} + 40n_\mathsf{y} + \log(n_\mathsf{x} n_\mathsf{y})n_\mathsf{y}$	$584n$	$592n$	$600n$
[KKRT16]	$6\kappa n_\mathsf{x} + 3(\lambda + \log(n_\mathsf{x} n_\mathsf{y}))n_\mathsf{y}$	$768n_\mathsf{x} + 120n_\mathsf{y} + 3\log(n_\mathsf{x} n_\mathsf{y})n_\mathsf{y}$	$984n$	$1008n$	$1032n$
[PRTY19] Low-Comm	$3.5\kappa n_\mathsf{x} + 1.02(2 + \lambda + \log(n_\mathsf{x}))n_\mathsf{y}$	$450n_\mathsf{x} + 43n_\mathsf{y} + 1.02\log(n_\mathsf{x})n_\mathsf{y}$	$\mathbf{509n}$	$513n$	$517n$
[PRTY19] Fast	$3.5(1 + 1/\lambda)\kappa n_\mathsf{x} + 2(\lambda + \log(n_\mathsf{x} n_\mathsf{y}))n_\mathsf{y}$	$461n_\mathsf{x} + 80n_\mathsf{y} + 2\log(n_\mathsf{x} n_\mathsf{y})n_\mathsf{y}$	$603n$	$619n$	$635n$
[PRTY20]	$9.3\kappa n_\mathsf{x} + (\lambda + \log(n_\mathsf{x} n_\mathsf{y}))n_\mathsf{y}$	$461n_\mathsf{x} + 40n_\mathsf{y} + \log(n_\mathsf{x} n_\mathsf{y})n_\mathsf{y}$	$1208n$	$1268n$	$1302n$
[CM20]	$4.8\kappa n_\mathsf{x} + (\lambda + \log(n_\mathsf{x} n_\mathsf{y}))n_\mathsf{y}$	$620n_\mathsf{x} + 40n_\mathsf{y} + \log(n_\mathsf{x} n_\mathsf{y})n_\mathsf{y}$	$678n$	$694n$	$702n$
Ours total (PaXoS)	$2.4\kappa n_\mathsf{x} + (\lambda + \log(n_\mathsf{x} n_\mathsf{y}))n_\mathsf{y} + 2^{17}\kappa {n_\mathsf{x}}^{0.05}$	$2^{24}{n_\mathsf{x}}^{0.05} + 307n_\mathsf{x} + 40n_\mathsf{y} + \log(n_\mathsf{x} n_\mathsf{y})n_\mathsf{y}$	$914n$	$426n$	$398n$
Ours total (interpolation)	$\kappa n_\mathsf{x} + (\lambda + \log(n_\mathsf{x} n_\mathsf{y}))n_\mathsf{y} + 2^{17}\kappa {n_\mathsf{x}}^{0.05}$	$2^{24}{n_\mathsf{x}}^{0.05} + 128n_\mathsf{x} + 40n_\mathsf{y} + \log(n_\mathsf{x} n_\mathsf{y})n_\mathsf{y}$	$702n$	$\mathbf{245n}$	$\mathbf{219n}$
Ours online (PaXoS)	$2.4\kappa n_\mathsf{x} + (\lambda + \log(n_\mathsf{x} n_\mathsf{y}))n_\mathsf{y} + 2^{13}\kappa {n_\mathsf{x}}^{0.13}$	$2^{20}{n_\mathsf{x}}^{0.13} + 307n_\mathsf{x} + 40n_\mathsf{y} + \log(n_\mathsf{x} n_\mathsf{y})n_\mathsf{y}$	$502n$	$398n$	$396n$
Ours online (interpolation)	$\kappa n_\mathsf{x} + (\lambda + \log(n_\mathsf{x} n_\mathsf{y}))n_\mathsf{y} + 2^{13}\kappa {n_\mathsf{x}}^{0.13}$	$2^{20}{n_\mathsf{x}}^{0.13} + 128n_\mathsf{x} + 40n_\mathsf{y} + \log(n_\mathsf{x} n_\mathsf{y})n_\mathsf{y}$	$310n$	$218n$	$217n$
Malicious					
[PRTY20]	$11.8\kappa n_\mathsf{x} + 2\kappa n_\mathsf{y}$	$1512n_\mathsf{x} + 256n_\mathsf{y}$	$1766n$	$1766n$	$1766n$
Ours total (PaXoS)	$2.4\kappa n_\mathsf{x} + (\lambda + \log(n_\mathsf{x} n_\mathsf{y}))n_\mathsf{y} + 2^{17}\kappa {n_\mathsf{x}}^{0.05}$	$2^{24}{n_\mathsf{x}}^{0.05} + 307n_\mathsf{x} + 128n_\mathsf{y}$	$960n$	$474n$	$438n$
Ours total (interpolation)	$\kappa n_\mathsf{x} + (\lambda + \log(n_\mathsf{x} n_\mathsf{y}))n_\mathsf{y} + 2^{17}\kappa {n_\mathsf{x}}^{0.05}$	$2^{24}{n_\mathsf{x}}^{0.05} + 128n_\mathsf{x} + 128n_\mathsf{y}$	$\mathbf{754n}$	$\mathbf{293n}$	$\mathbf{259n}$
Ours online (PaXoS)	$2.4\kappa n_\mathsf{x} + \kappa n_\mathsf{y} + 2^{13}\kappa {n_\mathsf{x}}^{0.13}$	$2^{20}{n_\mathsf{x}}^{0.13} + 307n_\mathsf{x} + 128n_\mathsf{y}$	$558n$	$446n$	$436n$
Ours online (interpolation)	$\kappa n_\mathsf{x} + \kappa n_\mathsf{y} + 2^{13}\kappa {n_\mathsf{x}}^{0.13}$	$2^{20}{n_\mathsf{x}}^{0.13} + 128n_\mathsf{x} + 128n_\mathsf{y}$	$366n$	$266n$	$257n$

be asymptotically correct for $n_\mathsf{x} \gg 2^{24}$. Since the cost of the VOLE is highly sublinear, the overhead it contributes quickly diminishes as n_x increase. For example, the VOLE requires 27,800 bits per element for $n_\mathsf{x} = 2^{10}$ while only requiring 38 bits per element for $n_\mathsf{x} = 2^{20}$. From this we can conclude that our protocol works best for large sets, e.g. $n_\mathsf{x} \geq 2^{16}$.

We also consider a setting where we perform a one time VOLE preprocessing phase. In this case the bulk of the VOLE computation can be performed before the X, Y sets or their sizes $n_\mathsf{x}, n_\mathsf{y}$ are known. This is akin to performing base OTs ahead of time as is done by all the protocols compared below (except DH-PSI). With preprocessing the online overhead of the VOLE decreases to approximately $2^{13}\kappa\sqrt[8]{n_\mathsf{x}}$ bits, an improvement of $16\times$. In addition, sublinear VOLE constructions are relatively new and there are likely more optimizations opportunities, like the recent work of [YWL+20] which we utilize.

As the table shows, our protocol outperforms prior work, especially for large inputs. The three protocols of [PRTY19, PRTY20] mostly differ in their linear system encoding rates. [PRTY19] considers two different types of Vandermonde / interpolation solvers which achieve rate $\rho \approx 1$ while [PRTY20] achieves rate $\rho = 2.4$ via their PaXoS solver and a significantly improved running time. Both of these works use an OT-extension type pro-

tocol which results in sending approximately 3.5κ bits per element in their encoding. We on the other hand depart from this OT-extension based technique and utilize sublinear VOLE. This has the advantage that we send only κ bits per item in the encoding. For the final PSI-from-OPRF construction, the sender will additionally send their set encoded under the OPRF which requires $\lambda + \log(n_{\mathsf{x}} n_{\mathsf{y}})$ bits per item in Y.

[KKRT16] does not encode their input into a linear system and instead uses cuckoo hashing which has a rate of $\rho \approx 1.7$. This work is also a OT-extension type protocol which requires sending 3.5κ bits per hash table element which results in an overhead of $6\kappa n_{\mathsf{x}}$. However, the cuckoo hashing approach results in the sender needing to send 3 OPRF values per item in Y. The core advantage of [KKRT16] is that cuckoo hashing is extremely efficient compared to solving a linear system and as such obtains very small running times.

In the case of malicious security, the overhead of our protocol is effectively identical except that the sender now must send larger OPRF values, i.e. κ bits per element in Y as opposed to $\lambda + \log(n_{\mathsf{x}} n_{\mathsf{y}})$ bits. On the other hand, the protocol of [PRTY20] requires increasing the number of bits per item in the linear encoding from 3.5κ to 5κ. This has the effect that they must send an overall encoding size of $11.8\kappa n_{\mathsf{x}}$. Our protocol more naturally achieves malicious security and only requires sending κ per encoding position. In addition, the [PRTY20] analysis states the sender must send OPRF values of size 2κ. However, we demonstrate that our protocol remains secure when only κ bits are sent.

7.2 Experimental Evaluation

Implementation. We implement all our protocols in C++. We use an extended version of the VOLE implementation of [SGRR19], supporting iterative bootstrapping [YWL+20] and a consistency check for malicious security [WYKW20], and assuming LPN with regular noise [BCGI18, WYKW20]. For computing the 2-core of the cuckoo graph in our PaXoS implementation, we use igraph [igr], and we rely on libOTe [Rin] for oblivious transfers and the GMW implementation used in our circuit PSI protocol.

To compare our protocols to previous work [KKRT16, CM20, PRTY20, PSTY19], we perform experiments in different network settings. To that end, we use two Amazon EC2 M5.2xlarge VMs, each featuring 8 cores at 2.5 GHz and 32 GiB of RAM. For comparability, we limit each protocol to a single core. In the LAN, without any artificial constraints, we measured a bandwidth of 5 Gbps between our machines. For settings with lower

bandwidth, we use Wondershaper [HGS] to limit incoming and outgoing traffic.

PSI. Here, we compare our semi-honest and malicious PSI implementations against the works of [KKRT16], [CM20], and [PRTY20]. The protocol of [KKRT16] is particularly fast, but comes with a comparatively large communication overhead. The semi-honest protocol of [CM20] on the other hand comes with a lower communication overhead, but more expensive computation. Finally, the PaXoS protocol of [PRTY20] features fast computation, but increased communication compared to [CM20]. We do not compare against the SpOT-light protocol [PRTY19], since [CM20] outperforms it in high-bandwidth settings[4], and our protocol has even lower communication than SpOT-low.

The results of our evaluation in the semi-honest setting are shown in Table 2. As expected, [KKRT16] outperforms all other protocols in the LAN setting, but is less effective with reduced bandwidth. For medium input sizes and bandwidths, [CM20] and [PRTY20] sometimes outperform our protocols and [KKRT16]. Our protocols particularly shine in medium to low bandwidth settings, and with large input sizes, which is to be expected given its low communication cost.

In the malicious setting, the state of the art is presented by [PRTY20]. Again, we compare communication and running time in different bandwidth settings, and present our results in Table 3. While in the LAN, [PRTY20] sometimes outperforms our implementation, we are consistently faster as bandwidth decreases.

Since the vector OLE implementation underlying our protocols uses the iterative bootstrapping approach of [YWL+20], our protocols have the distinctive feature that a part of the computation can be performed in a one-time, data-independent setup phase. Our implementation of this setup phase could be improved by tuning the LPN parameters (and thus the bootstrapping iteration sizes) to the input set sizes. Currently we use the parameters from [BCGI18, YWL+20] without any additional tuning. In our tables, we highlight the best protocols when setup is amortized in gray. It can be seen that in that case, our protocol more consistently outperforms previous work.

Circuit PSI. We also compare our circuit-PSI implementation to the state of the art protocol [PSTY19]. We use the same cuckoo hashing parameters

[4] In low communication settings (10 Mbps and 1 Mbps), [CM20] takes 15% longer than [PRTY19], but at the same time up to 75% longer than our protocol.

Table 2: Comparison of our PSI protocols to previous works in the semi-honest setting. We compare the amount of data sent by both parties, as well as the total running time with different bandwidths. A dash (–) indicates experiments that either crashed or did not finish, or where only the total communication is reported. The best protocol within a setting is marked in blue if setup is included, and in gray if setup is excluded.

n	Protocol	Communication (MB)			Total running time (s)			
		P_1	P_2	Total	LAN	100 Mbps	10 Mbps	1 Mbps
2^{16}	[KKRT16]	–	–	7.730	0.1160	0.7250	6.884	68.82
	[CM20]	0.5790	4.764	5.343	0.5853	0.6437	4.870	47.49
	[PRTY20]	12.62	0.5898	13.21	0.6460	1.682	11.86	112.8
	Ours	0.9965	2.702	3.699	0.1720	0.4510	3.277	31.18
	Ours (w/setup)	1.171	3.062	4.232	0.5030	1.067	6.742	63.33
2^{18}	[KKRT16]	–	–	31.88	0.5850	2.968	28.46	283.6
	[CM20]	2.520	19.23	21.75	2.017	2.194	19.50	193.8
	[PRTY20]	51.94	2.621	54.56	1.517	5.976	47.66	464.2
	Ours	3.066	10.30	13.37	1.227	2.192	12.26	114.1
	Ours (w/setup)	3.622	10.68	14.31	1.985	3.279	16.65	151.5
2^{20}	[KKRT16]	–	–	128.5	2.441	11.93	114.8	1143
	[CM20]	10.03	77.63	87.66	8.148	9.071	78.38	780.0
	[PRTY20]	214.0	10.49	224.5	5.885	24.09	195.6	1910
	Ours	12.06	40.55	52.61	4.398	8.496	48.69	449.7
	Ours (w/setup)	12.62	40.93	53.55	5.396	9.850	53.35	487.7
2^{22}	[KKRT16]	–	–	530.1	10.19	49.30	473.6	4718
	[CM20]	44.08	313.5	357.6	34.70	41.54	319.4	3182
	[PRTY20]	815.7	46.14	861.9	22.94	93.67	751.3	–
	Ours	47.28	161.7	208.9	23.93	40.67	199.0	1794
	Ours (w/setup)	47.84	162.0	209.9	25.88	42.97	204.7	1834
2^{24}	[KKRT16]	–	–	2137	43.90	199.1	1910	–
	[CM20]	176.3	1266	1442	189.6	198.1	1289	12 860
	[PRTY20]	3364	184.5	3548	101.7	392.0	–	–
	Ours	204.2	645.7	849.9	90.74	156.4	814.2	7296
	Ours (w/setup)	204.7	646.1	850.9	92.81	158.7	819.9	7335

Table 3: Comparison of our PSI protocols to [PRTY20] in the malicious setting.

n	Protocol	Communication (MB)			Total running time (s)			
		P_1	P_2	Total	LAN	100 Mbps	10 Mbps	1 Mbps
2^{16}	[PRTY20]	12.62	2.097	14.71	0.6510	1.808	13.13	125.5
	Ours	1.390	2.702	4.092	0.2250	0.5260	3.627	34.77
	Ours (w/setup)	1.564	3.062	4.626	0.5560	1.147	7.109	66.72
2^{18}	[PRTY20]	51.94	8.389	60.33	1.556	6.469	52.57	513.1
	Ours	4.639	10.30	14.94	1.279	2.464	13.96	127.6
	Ours (w/setup)	5.195	10.68	15.88	2.046	3.558	18.37	165.0
2^{20}	[PRTY20]	214.0	33.55	247.6	6.119	26.12	215.2	2410
	Ours	17.31	40.55	57.86	5.150	9.599	54.09	495.0
	Ours (w/setup)	17.86	40.93	58.79	6.157	10.94	58.76	532.6
2^{22}	[PRTY20]	815.7	134.2	950.0	23.37	101.2	826.1	–
	Ours	68.25	161.7	229.9	26.50	45.19	222.5	1975
	Ours (w/setup)	68.81	162.0	230.9	28.46	47.50	228.3	2015
2^{24}	[PRTY20]	3364	536.9	3901	102.8	422.1	–	–
	Ours	271.3	645.7	917.0	104.0	174.5	881.0	7876
	Ours (w/setup)	271.9	646.1	918.0	106.0	176.8	886.7	7914

Table 4: Comparison of our Circuit-PSI protocol to [PSTY19]. Values marked with an asterisk (*) were not measured, but computed from the theoretical communication costs [PSTY19, Section 7.3].

n	Protocol	Total comm. (MB)	Total running time (s)	
			5 Gbps	100 Mbps
2^{12}	[PSTY19]	9.00*	0.965	1.34
	Ours (IKNP)	13.4	0.495	1.19
	Ours (SilentOT)	4.79	0.737	1.07
2^{16}	[PSTY19]	149*	5.01	11.3
	Ours (IKNP)	171	1.52	9.03
	Ours (SilentOT)	21.1	4.05	5.34
2^{20}	[PSTY19]	2540*	72.0	172
	Ours (IKNP)	2830	23.3	149
	Ours (SilentOT)	277	103	120

as [PSTY19], $\epsilon = 1.27$ and $k = 3$ hash functions, following the analysis of [PSZ18]. We note, however, that there is some disagreement in the literature regarding the correct cuckoo hashing parameters for a given statistical security level λ. For example, for $k = 3, n = 2^{20}$, and $\lambda = 40$, [PSZ18] and [DRRT18] report quite different expansion factors (1.27 vs. 1.54). In our own experiments, we found the security level to be approximated by $\lambda = 240\epsilon - 256 - \log_2 n$, which requires $\epsilon = 1.32$ for $n = 2^{20}$ and $\lambda = 40$. Still, we stick to the parameters used by [PSTY19] for comparability.

Like [PSTY19], our construction uses a generic two-party computation phase in the end (Step 5 in Figure 10). We implement two variants of this step: one using the standard IKNP OT extension [IKNP03] to implement the GMW offline phase, and one using the more recent SilentOT [BCG+19].

Our results in Table 4 show that our protocols outperform [PSTY19] in both high and low-bandwidth settings. Since the main communication bottleneck is the GMW phase, the SilentOT variant works particularly well in the low-communication setting. In the LAN, our IKNP variant still outperforms [PSTY19] (who also used IKNP) in terms of running time, which showcases the efficiency of our novel OPPRF construction.

8 Conclusion

In this paper, we have shown how to combine two cryptographic primitives, namely Vector-OLE and linear system solvers like (Xo)PaXoS, into highly efficient O(P)PRF and PSI protocols. Our final protocols outperform previous work in terms of communication, and as a consequence, in terms of running time in bandwidth-constrained environments. From a theoretical perspective, we provide a more efficient reduction from OPRF to PSI.

As discussed in Section 2, there are many ways to implement the linear system solvers we require for VOLE-PSI. One approach, based on polynomial interpolation, promises to result in the lowest communication complexity, but as previous work has shown, this comes at the cost of expensive computation. The approach presented in this paper, using PaXoS, allows for fast computation, but incurs a higher communication blowup of asymptotically $2.4\kappa n$. It remains an open question whether there are more efficient (i.e., smaller) data structures that also allow for linear encoding and decoding. Should these become available, they will directly improve the communication complexity of our protocols.

References

[ADI+17] Benny Applebaum, Ivan Damgård, Yuval Ishai, Michael Nielsen, and Lior Zichron. Secure arithmetic computation with constant computational overhead. In *CRYPTO (1)*, volume 10401 of *Lecture Notes in Computer Science*, pages 223–254. Springer, 2017.

[BCG+19] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, Peter Rindal, and Peter Scholl. Efficient two-round OT extension and silent non-interactive secure computation. In *ACM Conference on Computer and Communications Security*, pages 291–308. ACM, 2019.

[BCGI18] Elette Boyle, Geoffroy Couteau, Niv Gilboa, and Yuval Ishai. Compressing vector OLE. In *ACM Conference on Computer and Communications Security*, pages 896–912. ACM, 2018.

[BKM+20] Prasad Buddhavarapu, Andrew Knox, Payman Mohassel, Shubho Sengupta, Erik Taubeneck, and Vlad Vlaskin. Private matching for compute. *IACR Cryptol. ePrint Arch.*, 2020:599, 2020.

[BLL+21] Fabrice Benhamouda, Tancrède Lepoint, Julian Loss, Michele Orrù, and Mariana Raykova. On the (in)security of ROS. In Anne Canteaut and François-Xavier Standaert, editors, *Advances in Cryptology - EUROCRYPT 2021 - 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17-21, 2021, Proceedings, Part I*, volume 12696 of *Lecture Notes in Computer Science*, pages 33–53. Springer, 2021.

[BM74] Allan Borodin and R. Moenck. Fast modular transforms. *J. Comput. Syst. Sci.*, 8(3):366–386, 1974.

[CKT10] Emiliano De Cristofaro, Jihye Kim, and Gene Tsudik. Linear-complexity private set intersection protocols secure in malicious model. In *ASIACRYPT*, volume 6477 of *Lecture Notes in Computer Science*, pages 213–231. Springer, 2010.

[CM20] Melissa Chase and Peihan Miao. Private set intersection in the internet setting from lightweight oblivious PRF. In *CRYPTO*

(3), volume 12172 of *Lecture Notes in Computer Science*, pages 34–63. Springer, 2020.

[CO18] Michele Ciampi and Claudio Orlandi. Combining private set-intersection with secure two-party computation. In *SCN*, volume 11035 of *Lecture Notes in Computer Science*, pages 464–482. Springer, 2018.

[CT10] Emiliano De Cristofaro and Gene Tsudik. Practical private set intersection protocols with linear complexity. In *Financial Cryptography*, volume 6052 of *Lecture Notes in Computer Science*, pages 143–159. Springer, 2010.

[DCW13] Changyu Dong, Liqun Chen, and Zikai Wen. When private set intersection meets big data: an efficient and scalable protocol. In *ACM Conference on Computer and Communications Security*, pages 789–800. ACM, 2013.

[DRRT18] Daniel Demmler, Peter Rindal, Mike Rosulek, and Ni Trieu. PIR-PSI: scaling private contact discovery. *Proc. Priv. Enhancing Technol.*, 2018(4):159–178, 2018.

[Gil99] Niv Gilboa. Two party RSA key generation. In *CRYPTO*, volume 1666 of *Lecture Notes in Computer Science*, pages 116–129. Springer, 1999.

[GPR+21] Gayathri Garimella, Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. Oblivious key-value stores and amplification for private set intersection. In Tal Malkin and Chris Peikert, editors, *Advances in Cryptology – CRYPTO 2021*, pages 395–425, Cham, 2021. Springer International Publishing.

[HEK12] Yan Huang, David Evans, and Jonathan Katz. Private set intersection: Are garbled circuits better than custom protocols? In *NDSS*. The Internet Society, 2012.

[HGS] Bert Hubert, Jacco Geul, and Simon Séhier. wondershaper: Command-line utility for limiting an adapter's bandwidth.

[igr] igraph: Library for the analysis of networks.

[IKN+20] Mihaela Ion, Ben Kreuter, Ahmet Erhan Nergiz, Sarvar Patel, Shobhit Saxena, Karn Seth, Mariana Raykova, David Shanahan, and Moti Yung. On deploying secure computing: Private

intersection-sum-with-cardinality. In *EuroS&P*, pages 370–389. IEEE, 2020.

[IKNP03] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In *CRYPTO*, volume 2729 of *Lecture Notes in Computer Science*, pages 145–161. Springer, 2003.

[KKRT16] Vladimir Kolesnikov, Ranjit Kumaresan, Mike Rosulek, and Ni Trieu. Efficient batched oblivious PRF with applications to private set intersection. In *ACM Conference on Computer and Communications Security*, pages 818–829. ACM, 2016.

[KLS+17] Ágnes Kiss, Jian Liu, Thomas Schneider, N. Asokan, and Benny Pinkas. Private set intersection for unequal set sizes with mobile applications. *Proc. Priv. Enhancing Technol.*, 2017(4):177–197, 2017.

[KRS+19] Daniel Kales, Christian Rechberger, Thomas Schneider, Matthias Senker, and Christian Weinert. Mobile private contact discovery at scale. In *USENIX Security Symposium*, pages 1447–1464. USENIX Association, 2019.

[KS12] Kazuki Kobayashi and Tomoharu Shibuya. Generalization of lu's linear time encoding algorithm for LDPC codes. In *ISITA*, pages 16–20. IEEE, 2012.

[LM10] Jin Lu and José M. F. Moura. Linear time encoding of LDPC codes. *IEEE Trans. Inf. Theory*, 56(1):233–249, 2010.

[Mea86] Catherine A. Meadows. A more efficient cryptographic matchmaking protocol for use in the absence of a continuously available third party. In *IEEE Symposium on Security and Privacy*, pages 134–137. IEEE Computer Society, 1986.

[OOS17] Michele Orrù, Emmanuela Orsini, and Peter Scholl. Actively secure 1-out-of-n OT extension with application to private set intersection. In *CT-RSA*, volume 10159 of *Lecture Notes in Computer Science*, pages 381–396. Springer, 2017.

[PRTY19] Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. Spot-light: Lightweight private set intersection from sparse OT extension. In *CRYPTO (3)*, volume 11694 of *Lecture Notes in Computer Science*, pages 401–431. Springer, 2019.

[PRTY20] Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. PSI from PaXoS: Fast, malicious private set intersection. In *EUROCRYPT (2)*, volume 12106 of *Lecture Notes in Computer Science*, pages 739–767. Springer, 2020.

[PSSZ15] Benny Pinkas, Thomas Schneider, Gil Segev, and Michael Zohner. Phasing: Private set intersection using permutation-based hashing. In *USENIX Security Symposium*, pages 515–530. USENIX Association, 2015.

[PSTY19] Benny Pinkas, Thomas Schneider, Oleksandr Tkachenko, and Avishay Yanai. Efficient circuit-based PSI with linear communication. In *EUROCRYPT (3)*, volume 11478 of *Lecture Notes in Computer Science*, pages 122–153. Springer, 2019.

[PSWW18] Benny Pinkas, Thomas Schneider, Christian Weinert, and Udi Wieder. Efficient circuit-based PSI via cuckoo hashing. In *EUROCRYPT (3)*, volume 10822 of *Lecture Notes in Computer Science*, pages 125–157. Springer, 2018.

[PSZ14] Benny Pinkas, Thomas Schneider, and Michael Zohner. Faster private set intersection based on OT extension. In *USENIX Security Symposium*, pages 797–812. USENIX Association, 2014.

[PSZ18] Benny Pinkas, Thomas Schneider, and Michael Zohner. Scalable private set intersection based on OT extension. *ACM Trans. Priv. Secur.*, 21(2):7:1–7:35, 2018.

[Rin] Peter Rindal. libote: A fast, portable, and easy to use oblivious transfer library.

[RR17a] Peter Rindal and Mike Rosulek. Improved private set intersection against malicious adversaries. In *EUROCRYPT (1)*, volume 10210 of *Lecture Notes in Computer Science*, pages 235–259, 2017.

[RR17b] Peter Rindal and Mike Rosulek. Malicious-secure private set intersection via dual execution. In *ACM Conference on Computer and Communications Security*, pages 1229–1242. ACM, 2017.

[SGRP19] Phillipp Schoppmann, Adrià Gascón, Mariana Raykova, and Benny Pinkas. Make some ROOM for the zeros: Data sparsity

in secure distributed machine learning. In *ACM Conference on Computer and Communications Security*, pages 1335–1350. ACM, 2019.

[SGRR19] Phillipp Schoppmann, Adrià Gascón, Leonie Reichert, and Mariana Raykova. Distributed vector-OLE: Improved constructions and implementation. In *ACM Conference on Computer and Communications Security*, pages 1055–1072. ACM, 2019.

[WYKW20] Chenkai Weng, Kang Yang, Jonathan Katz, and Xiao Wang. Wolverine: Fast, scalable, and communication-efficient zero-knowledge proofs for boolean and arithmetic circuits. *IACR Cryptol. ePrint Arch.*, 2020:925, 2020.

[YWL+20] Kang Yang, Chenkai Weng, Xiao Lan, Jiang Zhang, and Xiao Wang. Ferret: Fast extension for correlated OT with small communication. In *CCS*, pages 1607–1626. ACM, 2020.

Parameters: There are two parties, a Sender and a Receiver with a set $X \subseteq \mathbb{F}$ where $|X| = n$.

Protocol: Upon input $(\mathsf{sender}, \mathsf{sid})$ from the Sender and $(\mathsf{receiver}, \mathsf{sid}, X)$ from the Receiver, the protocol specifies the following:

1. The Sender samples $w^{\mathsf{s}} \leftarrow \mathbb{F}$ and sends $c^{\mathsf{s}} := \mathsf{H}^{\mathbb{F}}(w^{\mathsf{s}})$ to the Receiver.

2. The Receiver samples $r \leftarrow \{0,1\}^{\kappa}, w^{\mathsf{r}} \leftarrow \mathbb{F}$ and solves the systems

$$\begin{bmatrix} \mathsf{row}(x_1, r) \\ ... \\ \mathsf{row}(x_n, r) \end{bmatrix} \vec{P} = (\mathsf{H}^{\mathbb{F}}(x_1), ..., \mathsf{H}^{\mathbb{F}}(x_n))$$

   for $\vec{P}$ as a function of their set $\{x_1, ..., x_n\} = X \subset \mathbb{F}$.

3. The Sender sends $(\mathsf{sender}, \mathsf{sid})$ and the Receiver sends $(\mathsf{receiver}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{vole}}$ with dimension m and $|\mathbb{F}| \approx 2^{\kappa}$. The parties respectively receive $\Delta, \vec{B}$ and $\vec{C} := \vec{A}'\Delta + \vec{B}, \vec{A}'$.

4. The Receiver sends $r, w^{\mathsf{r}}, \vec{A} := \vec{P} + \vec{A}'$ to the Sender who defines $\vec{K} := \vec{B} + \vec{A}\Delta$.

5. The Sender sends w^{s} to the Receiver who aborts if $c^{\mathsf{s}} \neq \mathsf{H}^{\mathbb{F}}(w^{\mathsf{s}})$. Both parties define $w := w^{\mathsf{r}} + w^{\mathsf{s}}$.

6. The Receiver outputs $X' := \{\mathsf{H}(\mathsf{Decode}(\vec{C}, x) + w, x) \mid x \in X\}$.

Subsequently, upon each input $(\mathsf{sender}, \mathsf{sid}, y)$ from the Sender, the protocol specifies that the Sender outputs $F(y) = \mathsf{H}(\mathsf{Decode}(\vec{K}, y, r) - \Delta\mathsf{H}^{\mathbb{F}}(y) + w, y)$.

Figure 4: Protocol Π_{oprf} which realizes the Oblivious PRF functionality $\mathcal{F}_{\mathsf{oprf}}$.

Parameters: There are two parties, a sender with set $Y \subset \mathbb{F}$ and a receiver with a set of key $X \subseteq \mathbb{F}$. Let $n_{\mathsf{y}}, n_{\mathsf{x}}, n_{\mathsf{x}}' \in \mathbb{Z}$ be public parameters where $n_{\mathsf{x}} \leq n_{\mathsf{x}}'$.

Functionality: Upon receiving $(\mathsf{sender}, \mathsf{sid}, Y)$ from the sender and $(\mathsf{receiver}, \mathsf{sid}, X)$ from the receiver. If $|Y| > n_{\mathsf{y}}$, abort. If the receiver is malicious and $|X| > n_{\mathsf{x}}'$, then abort. If the receiver is honest and $|X| > n_{\mathsf{x}}$, then abort.
The functionality outputs $X \cap Y$ to the receiver.

Figure 5: Ideal functionality $\mathcal{F}_{\mathsf{psi}}$ of Private Set Intersection.

Parameters: There are two parties, a sender with set $Y \subset \mathbb{F}$ and a receiver with a set of key $X \subseteq \mathbb{F}$.
In the Semi-honest setting, let $\mathsf{out} := \lambda + \log_2(n_{\mathsf{x}}) + \log_2(n_{\mathsf{y}})$. In the malicious setting let $\mathsf{out} := \kappa$. Let $\mathcal{F}_{\mathsf{oprf}}$ be the OPRF functionality with $n = n_{\mathsf{x}}$ and $n_{\mathsf{x}}' := n'$ and the output length out.

Protocol:

1. The sender sends $(\mathsf{sender}, \mathsf{sid})$ and receiver sends $(\mathsf{receiver}, \mathsf{sid}, X)$ to $\mathcal{F}_{\mathsf{oprf}}$. The receiver receives $X' = \{F(x) \mid x \in X\}$.

2. For $y \in Y$, the sender sends $(\mathsf{sender}, \mathsf{sid}, y)$ to $\mathcal{F}_{\mathsf{oprf}}$ and receives back $F(y)$.

3. The sender sends $Y' := \{F(y) \mid y \in Y\}$ to the receiver in a random order.

4. The receiver outputs $\{x \mid F(x) \in Y', x \in X\}$.

Figure 6: Protocol Π_{psi} which realizes the PSI functionality $\mathcal{F}_{\mathsf{psi}}$.

Parameters: There are two parties, a sender with input $L = \{(y_1, z_1), ..., (y_{n_{\mathsf{y}}}, z_{n_{\mathsf{y}}})\}$ where $y_i \in \mathbb{F}, z_i \in \{0,1\}^{\mathsf{out}}$ and a receiver with a set $X \subseteq \mathbb{F}$ where $|X| = n_{\mathsf{x}}$.

Functionality: Upon input $(\mathsf{sender}, \mathsf{sid}, L)$ from the sender and $(\mathsf{receiver}, \mathsf{sid}, X)$ from the receiver, the functionality samples a random function $F : \mathbb{F} \to \{0,1\}^{\mathsf{out}}$ such that $F_k(y) = z$ for each $(y, z) \in L$ and sends $X' := \{F_k(x) \mid x \in X\}$ to the receiver.
Subsequently, upon input $(\mathsf{sender}, \mathsf{sid}, y)$ from the sender, the functionality returns $F(y)$ to the sender.

Figure 7: Ideal functionality $\mathcal{F}_{\mathsf{opprf}}$ of Oblivious Programmable PRF.

Parameters: There are two parties, a sender with $L = \{(y_1, z_1), ..., (y_\ell, z_\ell)\}$ and a receiver with a set $X \subseteq \mathbb{F}$ where $|X| = n$.

Protocol: Upon input $(\mathsf{sender}, \mathsf{sid}, L)$ from the sender and $(\mathsf{receiver}, \mathsf{sid}, X)$ from the receiver, the parties do the following:

1. The sender sends $(\mathsf{sender}, \mathsf{sid}, L)$ and the receiver sends $(\mathsf{receiver}, \mathsf{sid}, X)$ to $\mathcal{F}_{\mathsf{oprf}}$ with $|\hat{\mathbb{F}}| \approx 2^\kappa$. The parties respectively receive k and $X' = \{F_k(x) \mid x \in X\}$.
2. The sender uses the $\mathsf{XoPaXoS}$ solver to compute $\vec{P} \in \mathbb{F}^m$ over the field $\mathbb{F}$ such that $\vec{P} \leftarrow \mathsf{Encode}((y_1, z_1 - F_k(y_1)), ..., (y_\ell, z_\ell - F_k(y_\ell)))$ and sends it to the receiver.
3. The receiver outputs $\{x_1^*, ..., x_n^*\}$ such that $x_i^* := x_i' + \mathsf{Decode}(P, x_i)$.

Subsequently, upon input $(\mathsf{sender}, \mathsf{sid}, y)$ from the sender, output: $F_k(x) + \mathsf{Decode}(P, x)$.

Figure 8: Protocol Π_{opprf} which realizes the Oblivious Programmable PRF functionality $\mathcal{F}_{\mathsf{opprf}}$.

Parameters: There are two parties, a sender with set $Y \subset \mathbb{F}$, associated values $\tilde{Y} \subset \{0,1\}^{\sigma_\mathsf{y}}$ and a receiver with a set of keys $X \subseteq \mathbb{F}$, associated values $\tilde{X} \subset \{0,1\}^{\sigma_\mathsf{x}}$ where $|Y| = |\tilde{Y}| = n_\mathsf{y}, |X| = |\tilde{X}| = n_\mathsf{x}$. The functionality is parameterized by $\mathsf{Reorder} : \mathbb{F}^n \to (\pi : [n] \to [m])$ which on input X outputs a injective function π.

Functionality: Upon receiving $(\mathsf{sender}, \mathsf{sid}, Y, \tilde{Y})$ from the sender and $(\mathsf{receiver}, \mathsf{sid}, X, \tilde{X})$ the functionality computes $\pi \leftarrow \mathsf{Reorder}(X)$ and uniformly samples $Q^0, Q^1 \in \{0,1\}^m, Z^0, Z^1 \in \{0,1\}^{(\sigma_\mathsf{x}+\sigma_\mathsf{y})\times m}$ such that

$$\begin{aligned} &q_{i'}^0 \oplus q_{i'}^1 = 1, \;\; z_{i'}^0 \oplus z_{i'}^1 = (\tilde{x}_{i'} || \tilde{y}_i) && \text{if } \exists x_i \in X, y_j \in Y \text{ s.t. } x_i = y_j, \\ &q_{i'}^0 \oplus q_{i'}^1 = 0, \;\; z_{i'}^0 \oplus z_{i'}^1 = 0 && \text{otherwise} \end{aligned}$$

where $i' = \pi(i)$. Output Q^0, Z^0, π to the receiver and Q^1, Z^1 to the sender.

Figure 9: Ideal functionality $\mathcal{F}_{\mathsf{cpsi}}$ of Circuit Private Set Intersection.

Parameters: There are two parties, a sender with set $Y \subset \mathbb{F}$, associated values $\tilde{Y} \subset \{0,1\}^{\sigma_\mathsf{y}}$ and a receiver with a set of key $X \subseteq \mathbb{F}$, associated values $\tilde{X} \subset \{0,1\}^{\sigma_\mathsf{x}}$ where $|Y| = |\tilde{Y}| = n_\mathsf{y}, |X| = |\tilde{X}| = n_\mathsf{x}$. The protocol is parameterized by an expansion factor ϵ, cuckoo hash table size $m = \epsilon n_\mathsf{x}$, and k hash functions $h_j : \{0,1\}^* \to m$.

Protocol:

1. The receiver constructs a cuckoo hash table T_x of X such that $x \in X$, there exists a $j \in [k]$ such that $H(x||j) = T_\mathsf{x}[h_j(x)]$.

2. The sender constructs a simple hash table T_y of Y such that $y \in Y$, for all $j \in [k]$ it holds that $H(y||j) \in T_\mathsf{y}[h_j(y)]$.

3. For all i, the sender samples random $r_i \in \{0,1\}^\ell, w_i \in \{0,1\}^{\sigma_\mathsf{y}}$ and for all $y' \in T_\mathsf{y}[i]$, the receiver defines $L := \{(y', r_i||\tilde{y} \oplus w_i)\} \in (\mathbb{F} \times \{0,1\}^{\ell+\sigma_\mathsf{y}})^m$ where $\tilde{y}$ is associated value for y s.t. $y' = H(y,j)$.

4. The sender sends $(\mathsf{sender}, \mathsf{sid}, L)$ and the receiver sends $(\mathsf{receiver}, \mathsf{sid}, T_\mathsf{x}')$ to $\mathcal{F}_\mathsf{opprf}$ where $T_\mathsf{x}' := (H(1, T_\mathsf{x}[1]), ..., H(m, T_\mathsf{x}[m]))$. The receiver receives $X^* = \{(r_i'||w_i') \mid i \in [m]\}$.

5. For each i, the sender sends $(\mathsf{receiver}, \mathsf{sid}, r_i'||w_i')$ and the receiver sends $(\mathsf{sender}, \mathsf{sid}, r_i||w_i||\tilde{x})$ to $\mathcal{F}_\mathsf{2pc}$ where $\tilde{x}$ is the associated value with $x = T_\mathsf{x}[i]$ (or zero if $T_\mathsf{x}[i]$ is empty). $\mathcal{F}_\mathsf{2pc}$ computes a circuit $\mathcal{C}$ that for each $i \in [m]$:

   (a) Sets $q_i := 1$ if $r_i' = r_i$ and $q_i := 0$ otherwise,

   (b) Outputs secret shares q_i^0, q_i^1 of q_i and z_i^0, z_i^1 of $z_i := q_i \cdot \big((w_i' \oplus w_i)||\tilde{x}\big)$.

Figure 10: Protocol Π_cpsi which realizes the circuit PSI functionality $\mathcal{F}_\mathsf{cpsi}$.

Parameters: There are two parties, a sender and a receiver. The functionality is parameterized by a circuit $\mathcal{C} : \{0,1\}^{\mathsf{in}_1+\mathsf{in}_2} \to \{0,1\}^{\mathsf{out}_1+\mathsf{out}_2}$.

Functionality: Upon receiving $(\mathsf{sender}, \mathsf{sid}, X)$ from the sender and $(\mathsf{receiver}, \mathsf{sid}, Y)$ where $X \in \{0,1\}^{\mathsf{in}_1}$ and $Y \in \{0,1\}^{\mathsf{in}_2}$, the functionality computes $(Z_1, Z_2) := \mathcal{C}(X, Y)$ and returns $Z_1 \in \{0,1\}^{\mathsf{in}_1}$ to the receiver and $Z_2 \in \{0,1\}^{\mathsf{in}_2}$ to the sender.

Figure 11: Ideal functionality $\mathcal{F}_\mathsf{2pc}$ of generic two party computation.