Simple, Fast Malicious Multiparty Private Set Intersection

Ofri Nevo  
The Open University  
ofrine@gmail.com

Ni Trieu  
Arizona State University  
nitrieu@asu.edu

Avishay Yanai  
VMware Research  
ay.yanay@gmail.com

ABSTRACT

We address the problem of multiparty private set intersection against a malicious adversary. First, we show that when one can assume no collusion amongst corrupted parties then there exists an extremely efficient protocol given only symmetric-key primitives. Second, we present a protocol secure against an adversary corrupting any strict subset of the parties. Our protocol is based on the recently introduced primitives: oblivious programmable PRF (OPPRF) and oblivious key-value store (OKVS).

Our protocols follow the client-server model where each party is either a client or a server. However, in contrast to previous works where the client has to engage in an expensive interactive cryptographic protocol, our clients need only send a single key to each server and a single message to a pivot party (where message size is in the order of the set size). Our experiments show that the client’s load improves by up to 10× (compared to both semi-honest and malicious settings) and that factor increases with the number of parties.

We implemented our protocol and conducted an extensive experiment over both LAN and WAN and up to 32 parties with up to \(2^{20}\) items each. We provide a comparison of the performance of our protocol and the state-of-the-art for both the semi-honest setting (by Chandran et al.) and the malicious setting (by Ben Efraim et al. and Garamella et al.).

1 INTRODUCTION

Private set intersection (PSI) allows several parties, each holding a set of items, to learn the intersection of these sets and nothing else. Over the last several years, two-party PSI has become truly practical with extremely fast cryptographically secure implementations [3, 29, 32]. These protocols can process millions of items in seconds and are only a small factor slower than the naive and insecure method of exchanging hashed values. PSI (both two-party and multiparty) has many privacy-preserving applications such as private contact discovery [6, 16], measuring the effectiveness of online advertising [19] and password checkup [15]. Recently, private contact tracing applications related to COVID-19 [1, 7, 9, 35] found PSI as the ultimate cryptographic tool, allowing multiple parties (diagnosed users and healthcare providers) to privately match contact information and notify users who may have been infected. There are numerous applications that are better suited to the multiparty case, for example, several calendar users wish to find a commonly available time slot for a meeting; several companies wish to combine their data to find a target audience for an ad campaign [19]; a set of enterprises with private audit logs of connections to their corporate networks wish to identify similar activities in all networks. Recently, a variant of multiparty PSI [28] has been used for cache sharing in edge computing, which allows multiple network operators to store a set of common data items with the highest access frequencies in their capacity-limited shared cache while maintaining the privacy of their datasets. We can fairly say that today, PSI is one of the most motivated questions within the field of secure computation, which is well reflected in the progress made in the recent several years.

In this work, we consider the problem of multiparty PSI and devise protocols that are secure in the presence of a malicious adversary who may statically corrupt any strict subset of the parties.

1.1 State of the Art for Multiparty PSI

The complexity of various concretely efficient multiparty PSI protocols is presented in Table 1. Below we consider the works most relevant to ours.

1.1.1 Kolesnikov et al. The first concretely efficient multiparty PSI protocol was presented by Kolesnikov et al. in CCS’17 [23] which is implemented using fast oblivious transfer (OT) extension and is secure in the random oracle model. This protocol has two versions, one against a semi-honest adversary and the other against an augmented semi-honest adversary (who may change the corrupted parties’ inputs prior to the execution), such that in both versions the adversary may corrupt an arbitrary strict subset of the parties. That is, if the total number of parties is \(n\), the adversary may corrupt any \(t < n\) parties. While the performance of their semi-honest version improves as \(t\) decreases, their augmented semi-honest version performs evenly, no matter what \(t\) is (e.g. a case where the parties are relatively reliable, in which we can assume \(t < n/2\) or \(t = 1\) would not improve the protocol’s performance). The main contribution of [23] is the introduction of a two-party functionality called oblivious programmable PRF (OPPRF) which is run between a sender and a receiver. The sender has a set of points \(P = \{(x_i, y_i)\}\) that it wants to ‘program’ (with distinct \(x_i\)’s and pseudorandom \(y_i\)’s) and the receiver has a set of queries \(\{q_i\}\). For each query \(q_i\) the functionality outputs a PRF evaluation on \(q_i\) to the receiver, under the following condition. If \(q_i = x_j\) for some \(j\) then the functionality outputs \(y_j\) and otherwise it outputs \(F_k(q_i)\) (where \(k\) is a random key chosen by the functionality). The functionality guarantees that the receiver cannot tell whether the obtained result is ‘programmed’ or not and that the sender could not tell what are the receiver’s queries.

The first phase of the protocols in [23] requires the parties to obtain many shares of zero. The main difference between the two versions is that in the semi-honest setting an expensive conditional zero-sharing protocol is required, which incurs an OPPRF invocation between each pair of parties; whereas for the augmented semi-honest a cheap unconditional zero-sharing protocol is sufficient, which requires each pair of parties to exchange only a symmetric key.

When the receiver has only a single query, a protocol for OPPRF can be instantiated very efficiently using only oblivious transfers (OT). [23] demonstrated an efficient extension in order to allow the receiver to have multiple queries as follows. The receiver maps its
queries \(q_1, \ldots, q_m\) to \(m'\) bins, \(B_1, \ldots, B_{m'}\), using cuckoo hashing with \(k\) hash functions \(h_1, \ldots, h_k\), such that each bin has at most one query in it. The sender, however, maps its points into \(m'\) bins with simple hashing using all \(h_1, \ldots, h_k\), so each point \((x_i, y_i)\) is inserted to all bins \(B_{h_j(x_i)}\) for \(j \in [k]\). By this, except with negligible probability, each sender’s bin contains at most \(O(\log m)\) points. Now, the sender and receiver can run \(m'\) instantiations of the single-query OPPRF, such that in the \(i\)-th instantiation the sender inputs all points that were mapped to its \(B_i\) and the receiver inputs the query that was mapped to its \(B_i\) (or some dummy query if that bin is empty).

That approach, however, is not secure against a malicious sender. The sender may map the point \((x, y)\) only to a subset of the required bins \(B_{h_j(x)}(\ldots, B_{h_k(x)}\) instead of all of them. Suppose that the adversary learns whether the receiver obtained \(y\) or not (this information may be leaked in real-world scenarios). Such leakage is not isolated, i.e. if the sender put \((x, y)\) only in one bin \(B_{h_j(x)}\) and the receiver indeed obtained \(y\), necessarily means that the receiver put its query \(q = x\) in bin \(B_{h_j(x)}\), which leaks information related to other queries that could have been put in that bin.

Recently, Pinkas et al. [29] proposed a two-party PSI secure against a malicious adversary. Their protocol relies on cuckoo hashing, and yet, protects from the malicious sender’s attack described above. At the core of their construction is a hidden malicious version of OPPRF supporting multiple queries\(^1\). We use that maliciously secure OPPRF in our protocols and present the details in Appendix B for completeness. Gargiella et al. [13] used that version of OPPRF to replace the OPPRF in [23] in order to obtain a protocol that is secure against a malicious adversary. Their protocol, as we discuss in Section 1.2, is secure against \(t < n - 1\) parties, however, when \(t < n - 1\) their protocol’s performance remains the same (i.e., as if it has to protect against a coalition of \(t = n - 1\) corrupted parties).

1.1.2 Chandran et al. A concurrent and independent work by Chandran et al. [2] improves the above protocol, against a semi-honest adversary, as well as extends it to circuit-based PSI (where any post-processing function may be privately operated on the intersection) and to Quorum PSI (allowing the protocol to output values that are intersected by only a subset of the parties, instead of all of them). [2] however, considers a weaker adversary, who may corrupt at most \(t < n/2\) of the parties (i.e. honest majority). That relaxation of the adversarial power allows removing the expensive conditional zero-sharing that is the bottleneck in [23] and use an \((n, t)\)-secret sharing scheme (e.g. Shamir’s) instead. This ensures that any subset of at most \(t\) parties could not reveal intermediate results during the execution of the protocol. In contrast to [2], the protocol we present in this work is maliciously secure even in the dishonest majority setting (i.e. \(n/2 \leq t < n\)). Furthermore, even in the honest majority setting, our protocol offers slightly better security as we can pick the \(t + 1\) parties with the highest reputation to process the intersection (i.e. to play as servers). This means that only if this particular set of \(t + 1\) parties collude they can reveal information, whereas in [2] any \(t + 1\) parties may do so.

1.1.3 Ben Efraim et al. Another concurrent and independent work by Ben Efraim et al. [10] presents the first concretely efficient maliciously secure multiparty PSI, cleverly combining results from semi-honest multiparty PSI [18] and malicious two-party PSI [31], which are based on garbled bloom filter (GBF). A bloom filter (BF) is a data structure mainly used for recording the membership of items in a set. A set of items \(A = \{a_1, \ldots, a_m\}\) is encoded to a codeword \(B = (b_1, \ldots, b_m)\) (where \(m = O(m\lambda)\) and \(b_1 \in \{0, 1\}\)) using a set of \(k\) hash functions \(h_1, \ldots, h_k\). For every \(a \in A\) and for every \(j \in [k]\) it holds that \(b_{h_j(a)} = 1\) and all other positions in \(B\) equal 0. Thus, to check whether an item \(x\) belongs to \(A\), check whether \(\land_{j \in [k]} b_{h_j(a)} = 1\). For every \(a \not\in A\) it holds that \(\land_{j \in [k]} b_{h_j(a)} = 0\) only with negligible probability (which accounts to ‘false positive’).

A garbled bloom filter, introduced by Dong et al. [8] allows encoding \(A = \{a_1, \ldots, a_m\}\) to a codeword \(B = \{b_1, \ldots, b_m\}\) (with \(b_i \in \{0, 1\}\)) such that for every \(a \in A\) it holds that \(\lor_{i \in [k]} b_{h_j(a)} = 0\) whereas for \(a \not\in A\) it holds that \(\lor_{i \in [k]} b_{h_j(a)} = 1\) equals a random value except with negligible probability. The false positive rate for GBF is negligible, just like in a plain BF. A combination of GBF and oblivious transfer (OT) leads to a very simple two-party PSI (against a semi-honest adversary).

We note that a newer version of their PaXoS construction was introduced in [32] and solves a minor security issue. We stress that future construction should consider using the fixed version in [32].
That simple protocol is insecure when the sender or receiver is malicious. A malicious sender may encode more than \( m \) items into the GBF (e.g. by setting \( \gamma_i = 0 \) for every \( i \in \{m'\} \)) and the receiver may input 1 in every OT instance, by which it obtains the entire GBF of the sender, and may perform a brute force attack to extract the sender’s input set \( X \). Protecting against a malicious sender is easy, by using a random OT instead. In random OT (ROT) both messages \( m_0 \) and \( m_1 \) are chosen uniformly at random by the functionality and are given to the sender as an output. The new protocol is exactly as above, except that the sender does not encode \( X \) to a GBF. Instead, the parties run \( m' \) ROT instances, by which the receiver obtains \( R \) as before.

Let \( m'_0, m'_1 \) be the random messages used in the \( i \)-th ROT instance, then for each \( x_i \in X \) the sender computes \( x'_i = \sum_{j \in [k]} m^i_j(x_i) \) and send \( x'_i \) to the receiver. The receiver in turn computes \( y'_i = \bigoplus_{j \in [k]} r^i_j(y_i) \) for every \( y_i \in Y \). The receiver concludes that \( y_i \) is in the intersection iff \( y'_i \in \{x'_1, \ldots, x'_m\} \). This way the sender may input only \( m \) items to the PSI protocol since it has to explicitly compute their random representation and send them to the receiver.

Preventing against a malicious receiver is more involved. This was first addressed by Rindal and Rosulek [31] using the cut-and-choose technique, which allows the receiver to prove that it indeed encoded only \( m \) items in its bloom filter \( Y' \). The result protocol, which is secure against malicious adversaries, is quadratic in \( \lambda \) (the statistical security parameter) whereas the protocols we present in this paper are linear in \( \lambda \). That means that the storage, computation, and communication (i.e. number of ROTs) of [10] are much larger than the set size \( m \). We can observe from [10, Table 9] that the final bloom filter size and the number of ROTs performed in the protocol are almost 200x and 300x larger than the plain set size \( m \) whereas in our protocols the concrete complexity is larger only by a small factor (2-3).

We believe that, just like in the malicious two-party setting, a transition from GBF-based [31] to GCT-based [29] protocols will take place in the malicious multiparty setting as well and that the GCT approach will prevail.

1.1.4 Other Multiparty PSI Protocols. The first multiparty PSI was proposed by Freedman, Nissim and Pinkas [12], relying on oblivious polynomial evaluation (OPE), which in turn is based on homomorphic encryption (e.g. Paillier). In the two-party version, Alice interpolates a polynomial \( p(x) = \sum_{i=0}^m a_i x^i \) whose roots are her items \( x_1, \ldots, x_m \) and sends the encrypted coefficients \( Enc_{ek}(a_i) \) to Bob (where \( (ek, dk) \) is the encryption-decryption key pair and \( dk \) is known only to Alice). For every item \( y_i \) of Bob, he then homomorphically computes the ciphertext \( y_i' = Enc_{ek}(r_i \cdot p(y_i) + y_i) \), for a uniformly random \( r_i \), and sends it back to Alice. Alice then decrypts \( y_i' = Dec_{dk}(y_i') \) and concludes that \( y_i' \) is in the intersection iff \( y_i' \in X \). It is easy to see that this protocol is correct and secure against a semi-honest adversary. That approach is followed by other works, like [4, 5, 17, 21, 33, 34].

The recent work by Ghosh and Nilges [14] replaces the expensive homomorphic encryption with an efficient protocol for oblivious polynomial evaluation (OLE). Their asymptotic communication complexity is near-optimal, however, their protocol requires the parties to perform polynomial interpolations over a large number of points (i.e. the polynomial degree is the set size \( O(m) \)), which renders their protocol impractical for large sets (e.g. more than few tens of thousands). As a result, it was not implemented.

Other protocols follow the bloom filter approach described above. Miyaji et al. [25, 26] combine bloom filters with additively homomorphic encryption to obtain a non-colluding server-aided solution, and Zhang et al. [36] achieve maliciously secure multiparty PSI, but in a model in which the two ‘servers’ \( P_0 \) and \( P_1 \) do not collude (in fact, their collusion would make the protocol insecure even against a semi-honest adversary).

1.2 Overview of Our Results & Techniques

Our aim is at constructing a scalable maliciously secure multiparty PSI protocol. We make use of two main building blocks: oblivious programmable PRF (OPPRF) and oblivious key-value store (OKVS). The former is the basis for the fastest multiparty PSI protocols in the semi-honest setting [2, 23]. Pinkas et al. [29] strengthened the original cuckoo hashing based OPPRF construction of [23] to the malicious setting (see details in Appendix B).

An OKVS [13] is a data structure in which a sender has a set of key-value mapping \( \{(x_i, y_i)\} \) (pseudo)random \( y_i \)’s, and she wishes to hand that mapping over to a receiver (or receivers), allowing the receiver to evaluate the mapping on any input but without revealing the keys \( x_i \). Correctness of the data structure must ensure that if the other party evaluates the OKVS on some \( q = x_j \) then the result is \( y_j \). Obliviousness here is similar to that of the OPPRF: given the OKVS, the receiver cannot tell what keys \( x_i \)’s are encoded. The most compact OKVS that one can think of is a polynomial. That is, the OKVS \( S = (a_0, \ldots, a_m) \) is the set of coefficients of an \((m-1)\)-degree polynomial \( p(x) = \sum_{i=0}^{m-1} a_i x^i \) where \( m \) is the number of points and \( p \) is interpolated over those points \( \{(x_i, y_i)\} \). Given the coefficients \( S \), the receiver can evaluate the polynomial \( p \) on every query. This OKVS is size-optimal: it encodes \( m \) points using exactly \( m \) entries (coefficients). Correctness is obvious; obliviousness follows from the fact that if the \( y_i \)’s are (pseudo)random then so is the polynomial, and \( p \) is independent of the \( x_i \). When \( m \) is large, however, that OKVS construction is not practical as it requires interpolation and multi-point evaluation, which are super linear in the degree. The PaXoS data structure [29, 32], which is based on cuckoo hashing, is proven to be a much more practical OKVS [13], which compromises a bit on compactness (i.e. its size is \( 1.5 - 2.5 \)x larger than the number of points \( m \)), but it is very fast to encode and decode (in analogy to interpolation and evaluation). While our protocols can be instantiated with any OKVS, we rely on that specific construction in our implementation.

The main difference between the two primitives is that OPPRF actively enforces the receiver to evaluate the function \( F \) on a limited number of queries, whereas OKVS is simply a data structure that is sent in the clear to the receiver, thus, no limit on the number of evaluation is set. This difference has a significant impact on their performance. Specifically, an OT-based OPPRF [22] incurs about 4.2 – 4.5x more communication and is about 2x slower. In addition, an OKVS is merely a single message sent from the sender to the receiver while an OPPRF requires a 2-round protocol.

We present PSI protocols for two different settings. In the first one we assume no collusion among the parties (i.e. \( n \) parties and


\[ t \leq 1 \) and in the second we assume an arbitrary collusion (i.e. \( n \) parties and \( t < n \)). We give a high-level idea of our techniques:

- **No collusion.** In this case we do not even require an OPPRF. Specifically, we reduce the problem of multiparty PSI to the problem of two-party PSI. As an example, consider three parties \( P_1, P_2, P_3 \) with sets \( A^1, A^2, A^3 \) respectively. Party \( P_1 \) picks a random key \( k \) and send it to \( P_2 \). In addition, \( P_1 \) generates an OKVS using the points \( (a^1_i, F_k(a^1_i)) \) for every \( a^1_i \in A^1 \) and sends it to \( P_3 \). Then \( P_2 \) computes \( b^2_j = F_k(a^2_j) \) for every \( a^2_j \) and \( P_3 \) evaluates \( b^3_i = S(a^3_i) \) for every \( a^3_i \in A^3 \). Note that at this point if \( x \) is in the intersection then both \( P_2 \) and \( P_3 \) have \( b^2_j = b^3_j = F_k(x) \) for some \( j \) and \( j' \). Otherwise (if \( x \) is not in the intersection) then either \( P_2 \) or \( P_3 \) (or both) does not have \( F_k(x) \). Thus, \( P_2 \) and \( P_3 \) can run a two-party PSI protocol over the inputs \( \{b^2_i \}_{i \in [n]} \) and \( \{b^3_i \}_{i \in [n]} \) and obtain the intersection \( A^1 \cap A^2 \cap A^3 \). Furthermore, we observe that instead of running the usual two-party PSI (e.g. \([29, 31]\)), they can run a server-aided PSI with \( P_1 \) being the server. Since a malicious server-aided PSI is much faster than a plain malicious PSI (~0.8 seconds vs. ~5 seconds for sets size of \( m = 2^{20} \)) the overall running time for the three party PSI decreases from ~9 seconds (with plain PSI) to ~4.8 seconds for sets of size \( m = 2^{20} \), almost 2x improvement. We extend this simple idea to an arbitrary number of parties, resulting with an extremely fast protocol. For instance, \( n = 32 \) parties with set size of \( m = 2^{20} \) complete the protocol in 10 seconds.

Since a server-aided two-party PSI does not require OT (e.g. public-key base OT), our multiparty PSI protocol relies only on symmetric-key primitives. To the best of our knowledge, this is the only construction with such a property.

- **Arbitrary collusion.** This is the challenging setting, in which the adversary may corrupt any strict subset of the parties. We present a simple protocol that can be described in a modular fashion using only high level primitives OPPRF and OKVS (sealing lower-level complex primitives like OT). Our protocol can be calibrated as a function of \( t \) such that the smaller \( t \) is the faster the protocol. For example, with \( n = 15 \) and \( m = 2^{20} \) the runtime of our protocol are \( \{7.2, 22.8, 32.5, 58.23\} \) seconds for \( t = \{1, 4, 7, 14\} \) respectively. In the worst case, when \( t = n - 1 \), our protocol converges with the protocol of Garimella et al. \([13]\) (which is the same as the augmented semi-honest version of \([23]\), except the OKVS instantiation); both have the same performance. Calibration of the protocol according to the upper bound on the number of corrupted parties is not trivial. That is, the augmented semi-honest protocol by \([23]\) and the malicious protocols by \([10, 13]\) protect from a collusion of \( n - 1 \) parties even though \( t \) may be smaller. In addition, the semi-honest honest majority protocol by \([2]\) protects against a collusion of \( n/2 - 1 \) parties even though \( t \) may be smaller. It is not known how to improve the performance of these protocols in accordance to smaller \( t \).

To withstand a collusion of up to \( t \) parties, our protocol (very informally) reduces the problem of \( n \)-party PSI to the problem of \( (t + 1) \)-party PSI. Specifically, \( n - t - 1 \) parties play as clients, with a very lightweight computational and network load. In addition, \( t \) parties play as servers, and the last party plays as a pivot. The challenge is to share the clients’ sets to the possession of the pivot and the \( t \) servers in a way that does not reveal anything about the intersection of the honest clients’ sets. To this end, we utilize a technique similar to that in the non-collusion setting: Each client picks a random PRF key for each server and sends it to that server. Then, the client generates an OKVS where the keys are its items and the values are a combination of PRF evaluation using all these keys, and send that OKVS to the pivot party. At this point, for each item in the intersection (of all parties’ sets) the servers and the pivot (in total \( t + 1 \) parties) have a sharing of zero. In contrast, for items not in the intersection, their sharing is for a random value. The servers and pivot find these items for which the shares sum up to zero by running a dedicated ZeroXOR protocol.

We compare our protocols to recent (implemented) multiparty PSI protocols \([2, 10, 13]\).

## 2 PRELIMINARIES

Denote the set \( \{1, \ldots, n\} \) by \([n]\). Definitions for [oblivious, programmable] PRF (i.e. OPPRF, PPRF, and OPPPRF) are given below, taken almost verbatim from \([23]\). Denote by \( k \) and \( \lambda \) the computational and statistical security parameters, respectively. PPT is short for probabilistic polynomial time. We denote the concatenation of two bit strings \( x \) and \( y \) by \( x || y \). In our PSI protocols, we denote the set of party \( i \) of size \( m \) by \( A^i = \{a^i_1, \ldots, a^i_m\} \).

### 2.0.1 Private Set Intersection.

The functionality for \( n \)-party private set intersection is given in Functionality 2.1. Note that in the semi-honest setting the functionality may give the intersection output to all parties (rather than to \( P_n \) only) and the adversary always sets \( abort = 0 \). Also note that even though the functionality allows an unbounded bit-length for the items, in practice (and in our protocols in particular) it is sufficient to consider items of length \( k \), so it is possible to input the item \( H(x) \) instead of the original item \( x \in \{0, 1\}^k \) where \( H : \{0, 1\}^* \rightarrow \{0, 1\}^k \) is a collision-resistant hash function.

### 2.0.2 Oblivious PRF.

An oblivious PRF (OPRF) \([11]\) is a 2-party protocol in which the sender learns a PRF key \( k \) and the receiver learns \( F(k, q_1), \ldots, F(k, q_m) \), where \( F \) is a PRF and \( \{q_1, \ldots, q_m\} \) are inputs chosen by the receiver. Note that we consider a variant of OPRF where the receiver obtains outputs of multiple statically chosen queries. The OPRF ideal functionality is given in Functionality 2.2.

### 2.0.3 Programmable PRF (PPRF).

A programmable PRF consists of the following algorithms:
FUNCTIONALITY 2.2. (Oblivious PRF - \(F^{P,m}_{\text{oprf}}\))

Parameters: A PRF \(F\), and a bound \(m\) on the number of queries.

Behavior: Wait for input \((q_1, \ldots, q_m)\) from the receiver \(R\) where \(q_i \in \{0,1\}^k\). Sample a random PRF key \(k\) and give it to the sender \(S\). Give \(\{F(k, q_1), \ldots, F(k, q_m)\}\) to \(R\).

- KeyGen(\(\kappa, \mathcal{P}\)) \(\rightarrow\) (\(k, \text{hint}\)): Given a security parameter \(\kappa\) and set of points \(\mathcal{P} = \{(a_1, t_1), \ldots, (a_n, t_m)\}\) with distinct \(a_i\)-values, where \(a_i, t_i \in \{0,1\}^k\), generate a PRF key \(k\) and (public) auxiliary information hint. We denote the set \(\{a_i\}\) by keys(\(\mathcal{P}\)) and the set \(\{t_i\}\) by vals(\(\mathcal{P}\)).
- \(F(k, \text{hint}, x) \rightarrow y\): Evaluates the PRF on input \(x\), giving output \(y \in \{0,1\}^k\).

A programmable PRF satisfies correctness if for all \((x, y) \in \mathcal{P}\), and \((k, \text{hint}) \leftarrow \text{KeyGen}(\kappa, \mathcal{P})\) it holds that \(F(k, \text{hint}, x) = y\). For security consider Experiment 2.3.

EXPERIMENT 2.3. \(\text{Exp}^A(\mathcal{P}, Q, \kappa)\)

1. For each \(a_i \in \mathcal{P}\) choose random \(t_i' \in \{0,1\}^k\)
2. \((k, \text{hint}) \leftarrow \text{KeyGen}(\kappa, \{(a_i, t_i') | a_i \in \text{keys}(\mathcal{P})\})\).
3. return \(\mathcal{A}(\text{hint}, \{F(k, \text{hint}, q) | q \in Q\})\)

We say that a programmable PRF is \((m_1, m_2)\)-secure if for all \(\mathcal{P}_1, \mathcal{P}_2, Q\) where \(|\mathcal{P}_1| = |\mathcal{P}_2| = m_1, |Q| = m_2\), and all PPT adversary \(\mathcal{A}\):

\[
\Pr[\text{Exp}^A(\mathcal{P}_1, Q, \kappa)] - \Pr[\text{Exp}^A(\mathcal{P}_2, Q, \kappa)] \leq \text{negl}(\kappa)
\]

Intuitively, security means that it is hard to tell which set of points is programmed, given hint and \(m_2\) outputs of the PRF, if the points were programmed to random outputs. Note that this definition implies that unprogrammed PRF outputs (i.e., those not set by the input to KeyGen) are pseudorandom. The ‘hint’ is part of the syntax since all constructions of PPRF leak some object to the receiver in addition to the PRF outputs. This object is called a hint and security is guaranteed even though the hint is known to the receiver.

Oblivious Programmable PRF (OPPRF). The formal definition of an oblivious programmable PRF functionality is given in Functionality 2.4. It is similar to the plain OPRF functionality except that (1) it allows the sender to initially provide a set of points \(\mathcal{P}\) which will be programmed into the PRF; (2) it additionally gives the “hint” value to the receiver. OPPRF construction for both the semi-honest and malicious setting were proposed by Kolesnikov et al. [23] and by Pinkas et al. [29, 30] ([29] proposes the malicious construction).

FUNCTIONALITY 2.4. \((F^{P,m_1,m_2}_{\text{opprf}})\)

Parameters: A programmable PRF \(F\), an upper bound \(m_1\) on the number of points to be programmed, and a bound \(m_2\) on the number of queries.

Behavior: Wait for input \(\mathcal{P} = \{(a_1, t_1), \ldots, (a_{m_1}, t_{m_1})\}\) from the sender \(S\) and input \((q_1, \ldots, q_{m_2})\) from the receiver \(R\). Run \((k, \text{hint}) \leftarrow \text{KeyGen}(\kappa, \mathcal{P})\). Give \((k, \text{hint})\) to \(S\) and \((\text{hint}, F(k, \text{hint}, q_1), \ldots, F(k, \text{hint}, q_{m_2}))\) to \(R\).

3 PSI WITH NO COLLUSION

This section serves as a warm-up and presents simple protocols for \(n\)-party PSI. Even though the general protocols in this section are not the most efficient ones, the purpose of presenting them is twofold: (1) demonstrating the simplicity of basing the PSI protocol on the higher-level abstraction of OKVS; and (2) this presentation yields the most efficient three-party PSI protocol to date, for both the semi-honest and malicious settings.
We reduce the problem of $n$-party PSI in the following way. The parties repeat this process recursively until party $P_i$ for all malicious 
the result. 

Note that the above simple recursive protocol has a caveat: a party $P_i$ could encode $(a'_i, F(k, a'_i))$ into an OKVS where $a'_i \not\in A_i^1$. Note that when $P_i$ has a key $k$ and an OKVS $S$, both objects do not imply any limit on the number of queries to them (i.e. $P_i$ can compute $F(k, \cdot)$ and Decode($S$, $\cdot$) arbitrarily many times). In order to weaken the threshold assumption (i.e. to make the protocol secure even against collusion), one may use an $F_{\text{opprf}}^{n,1,m}$ in place of the OKVS. That is, in Step 2 of Protocol 3.1, $P_1$ runs an OPPRF protocol with each of $P_2, \ldots, P_n$. Now, by the definition of OPPRF, $P_1$ can make only a limited number of queries.

Although that modification seems to strengthen the protocol security, it would not satisfy the security requirement defined by functionality $F_{\text{psi}}^{n,1,m}$. Recall that the functionality outputs to $P_n$ only the items that are in the intersection of all sets. However, in the modified protocol the adversary, who corrupts parties $P_1, P_2$ (or $i, j$), may learn the intersection of the sets of parties $P_1, P_2, P_3$ by having $P_1, P_2$ agree on the same input set $A^1 = A^2_j$ and compare their OPPRF results. An equal OPPRF results on a query $x$ means that $x \in A^1 \cap A^2_j$ whereas an unequal results on $x$ means that $x \not\in A^1$. Such an intersection of three parties is not permitted by functionality $F_{\text{psi}}^{n,1,m}$.

3.1.3 Three-party and dishonest majority. Note that when $n = 3$, the above adversarial behavior is not considered as an attack, since the intersection of the sets of three parties is actually the intersection of all sets, which is allowed to be revealed. Thus, we find such a modification to Protocol 3.1 useful for implementing $F_{\text{psi}}^{3,2,m}$. That is, to securely compute the intersection of three sets even when two of the parties are corrupted and colluding. 

3.1.4 Complexity and Security. The protocol recursively invokes itself with decreasing number of parties, where our base case is a two-party PSI. That means that each of $P_1, \ldots, P_{n-2}$ encrypts a single OKVS and decodes $i = 1$ instances of OKVS. Furthermore, that means that the protocol has $O(n)$ rounds of communication, which may be the bottleneck when the number of parties is large.
Theorem 3.2. Protocol 3.1 \( (\pi^{n,1,m}) \) securely computes functionality \( F_{n,1,m}^{\pi} \) in the \( F_{n-1,1,m} \)-hybrid and random oracle model in the presence of a malicious adversary.

Proof: Correctness is clear from the definitions of OKVS, PRF and PSI. We turn to show security by presenting a simulator to each of the following four cases, for each case we describe simulation in both the semi-honest and malicious settings.

Corrupted \( P_1 \). In the semi-honest setting, the simulator is given the \( P_1 \)'s input \( A^1 = \{a^1_1, \ldots, a^1_m\} \), he inputs it to the ideal-world functionality and obtains an empty output. In the real execution \( P_1 \) receives no further messages, thus, which trivially concludes the simulation. In the malicious setting, the simulator has to extract \( P_1 \)'s actual input. To do so, the simulator internally runs \( P_1 \) and for each call \( H(x) \) to the random oracle, the simulator enters \( x \) to a list \( L \). Then, playing the role of \( P_2 \), who receives \( k \) and \( P_1 \) (\( i = 3, \ldots, n \)) who receives \( S \), the simulator concludes with the actual input set \( \hat{A}^1 = \{x \in L \mid F(k, H(x)) = \text{Decode}(S, H(x))\} \).

The simulator inputs \( \hat{A}^1 \) to the ideal world functionality. Note that except with negligible probability, for every \( x' \) that was not queried the random oracle, it follows that \( F(k, H(x')) \neq \text{Decode}(S, H(x')) \), and thus \( x' \) does not appear in the real execution result intersection. This conclusion simulates the case because \( P_1 \) does not receive any further message in both worlds.

Corrupted \( P_2 \). The simulator is given \( A^2 \), outputs it to the ideal-world functionality and receives nothing back. In the real execution \( P_2 \) receives a random key from \( P_1 \), so the simulator generates a random key and sends it to \( P_2 \), which concludes the simulation in the semi-honest case.

In the malicious setting, the simulator runs \( P_2 \) internally and gives a random key \( k \). The simulator observes \( P_2 \)'s calls to the random oracle and records them in the set \( L \). Then, the simulator observes the set of values, \( L' \), input by \( P_2 \) to the \( F_{n-1,1,m} \) functionality and concludes with the set \( \hat{A}^2 = \{x \in L' \mid \exists y \in L : F^{-1}(k, y) = H(x)\} \). The simulator inputs \( \hat{A}^2 \) to the ideal world functionality. Note that for each value \( y \in L' \) for which \( F^{-1}(k, y) \) is not a random oracle output on some value from \( L \), the probability that \( F^{-1}(k, y) \) is a random oracle output for some value in \( A^2 \) for \( i \neq 2 \) is negligible, since there are at most \( (n-1)m \) random oracle outputs in the range \( \{0, 1\}^t \), the probability that \( F^{-1}(k, y) \) is one of them is negligible. Therefore, with high probability \( y \) would not impact the result intersection in the real execution.

Corrupted \( P_1 (3 \leq i < n) \). The simulator is given \( A^i \), sends it to the ideal-world functionality and receives no output. In the real execution \( P_1 \) receives an OKVS from \( P_1 \), so the simulator computes \( S \leftarrow \text{Encode}((k_i, v_i)) \) with \( m \) random pairs \( (k_i, v_i) \) and sends \( S \) to \( P_1 \). By the obliviousness property of \( S \), it is not possible to distinguish between \( S \) output by the simulator and the OKVS that has \( A^i \) as keys in the real execution.

In the malicious setting the simulator extracts \( P_1 \)'s input set as follows: it runs \( P_1 \) internally with the random OKVS \( S \) as its first message. It observes the set of \( P_1 \)'s random oracle queries and records them in the list \( L \). Then, it receives \( P_1 \)'s input set \( L' \) to the \( F_{n-1,1,m} \) functionality and concludes with the set \( \hat{A}^i = \{x \mid x \in L \land \text{Decode}(S, H(x)) \in L'\} \). As before, for a value \( y \) in \( L' \) that is not in the range of \( \text{Decode}(S, \cdot) \) or is \( \text{Decode}(S, r) \) where \( r \) not being a random oracle respond to any value in \( L \), with high probability \( y \) has no impact on the result intersection in the real execution. Therefore we may ignore it in the ideal world simulation.

Corrupted \( P_n \). The simulation here works exactly as in the previous case with \( S \) being the OKVS sent to \( P_n \). The simulator inputs the concluded set \( \hat{A}^i \) to the ideal world functionality and indeed receives an output \( X \) - the intersection of all parties’ sets. The simulator hands \( \{x \mid \text{Decode}(S, x)\} \subseteq X \) to \( P_n \) (in the internal execution) and outputs whatever \( P_n \) outputs. As argued in the previous case, with high probability both worlds use the same input set of \( P_n \), therefore the result intersection is the same.

3.2 Reducing to \( O(1) \) Rounds

Protocol 3.1 has a constant number of rounds. The idea is to ‘push’ the computation workload to a small number of designated parties, specifically, to parties \( P_{n-1} \) and \( P_n \). \( P_1 \) generates the PRF keys \( k_i \) for all \( i \in [2, n-2] \), and hands \( k_i \) to \( P_i \), and uses the XOR of all \( F(k_i, a^j_i) \) as \( \bigoplus_{i=2}^{n-2} F(k_i, a^j_i) \) to encode an OKVS \( S_n \), which it then sends to \( P_n \). \( P_n \) learns an OKVS \( S_n \), so decide it on every \( a^{n-1} \in A^{n-1} \), which equals \( \bigoplus_{i=2}^{n-2} F(k_i, a^j_i) \) if \( a^{n-1} \) was encoded in \( S_n \). Similarly, party \( P_{n-1} \) receives the OKVS \( S_i \) (encoded using key \( k_i \) received from \( P_1 \) from party \( P_1 \in \{P_2, \ldots, P_{n-2}\} \), so she can decide it on every \( a^{n-1} \in A^{n-1} \). Again, if \( a^{n-1} \) was encoded then the result is \( \bigoplus_{i=2}^{n-2} F(k_i, a^{n-1}) \). So for a value \( x \) that is in the intersection, both \( P_{n-1} \) and \( P_n \) compute the same value, which looks pseudo-random to them (Because both parties learn only the pseudo-random values encoded in the OKVS’s without knowing the keys).

Note that, similar to Protocol 3.1, \( P_{n-1} \) and \( P_n \) augment their input to \( F_{2,1,m}^{\pi} \) to be the concatenation of the plain item and its PRF evaluation. This is required in order to mitigate a similar attack to the one described above: a malicious \( P_1 \) might encode \( (x, \bigoplus_{i=2}^{n-2} F(k_i, x')) \) in \( S_n \) (rememeber, \( P_1 \) chooses all keys), where \( x' \in A^l \) for all \( i \in [2, n-1] \) and \( x \in A^l \), but neither \( x \) nor \( x' \) are in the intersection. Now, when \( P_n \) computes \( \text{Decode}(S_n, x) \) she obtains \( \bigoplus_{i=2}^{n-2} F(k_i, x') \). Therefore, \( P_{n-1}, P_n \) invoke \( F_{2,1,m}^{\pi} \) with \( \bigoplus_{i=2}^{n-2} F(k_i, x') \) as one of the values in their sets, leading \( P_n \) to falsely output the value \( x \).

Let remark that in the case of \( n = 3 \), party \( P_2 \) acts as if she is party \( P_{n-1} \). Namely, \( P_2 \) preforms steps 5 and 6, while she does not preform step 3. As a consequence, in step 4, \( P_2 \) computes \( A^{n-1} = \{a^{n-1}_j \mid F(k_{n-1}, a^{j}_{n-1})\} \) \( d_{n-1}^{-1} \in A^{n-1} \).

3.2.1 Discussion. Note that even a slight modification to Protocol 3.3 may turn it insecure. For example, suppose \( P_1 \), for \( i \in [2, n-2] \), sends the OKVS \( S_i \) directly to \( P_n \); then \( P_n \) could compute \( v' \leftarrow \bigoplus_{i=2}^{n-2} \text{Decode}(S_i, v) \) and \( v'' \leftarrow \text{Decode}(S_n, v) \), compare the two values \( v', v'' \) and deduce if \( v \in A^1, v \in A^i \) and \( v \in A^1 \) or neither. Since OKVS
does not imply any limit on the number of queries to it, $P_n$ can preform this test with any $\sigma$, thus learning more information than what the functionality allows.

In addition, a collusion of even two parties would break the security of Protocol 3.5: If $P_1$ colludes with $P_{n-1}$, $P_1$ may send the PRF keys $k_1$ to $P_{n-1}$. Now, $P_{n-1}$ may run $\text{Decode}(S_1, x)$ on unlimited number of values $x$, by that, it receives either $F(k_1, x)$ or some pseudorandom value, depending on whether $x \in A^i$ or not. If inputs are drawn from a rather small domain then $P_{n-1}$ may completely reveal $P_1$’s input set. We remark that the protocol remains secure against a collusion of any subset in $\mathcal{P}([P_2, \ldots, P_{n-2}]) \times \{P_1, P_{n-1}, P_n}\}$ (where $\mathcal{P}$ denotes the power set), as each party $P_i \in \{P_2, \ldots, P_{n-2}\}$ holds only its own key $k_i$ and set $S_i$, which do not leak information regarding any other parties’ input.

### 3.2.2 Complexity and Security

We begin by the analysis of the computational complexity. Party $P_1$ computes a single OKVS, but performs $O(nm)$ calls to $F$ in order to do so. Party $P_1$ $(i \in \{2, \ldots, n-2\})$ computes a single OKVS with work linear in $m$. $P_{n-1}$ decodes $O(n)$ instances of OKVS, each on $O(m)$ values, which incurs computation of $O(nm)$. Finally, $P_n$ decodes a single OKVS on $O(m)$ values.

We continue with the round complexity. Party $P_1$ sends $k_1$ to $P_1$ $(i \in \{2, \ldots, n-2\})$ in the first round. She also sends $S_1$ to $P_n$ in the same round. $P_1$ $(i \in \{2, \ldots, n-2\})$ sends $S_i$ to $P_{n-1}$ at the second round. Parties $P_{n-1}$ and $P_n$ invoke $\mathcal{F}_{\text{psi}}^{\pi}$ in the third and last round. Overall, the round complexity is 4 rounds more than the protocol for two parties. We instantiate $\mathcal{F}_{\text{psi}}^{\pi}$ using the server-aided PSI by Kamara et al [20] which is 2 rounds. Therefore, our protocol has an overall of 4 rounds.

Finally, consider communication complexity. Each party $P_i$ $(i \in \{1, \ldots, n-2\})$ sends an OKVS encoded with $O(m)$ values. Party $P_1$ also sends $O(n)$ $\kappa$-length keys. $P_{n-1}$ and $P_n$ communication complexity is determined by the exact protocol used to compute the functionality $\mathcal{F}_{\text{psi}}^{\pi}$, which is $O(m)$ as it can be instantiated with a server-aided version.

**Theorem 3.4.** Protocol 3.3 ($\pi_{\text{psi}}^{\pi}$) securely computes functionality $\mathcal{F}_{\text{psi}}^{\pi}$ in the $\mathcal{F}_{\text{psi}}^{\pi}$-hybrid model in the presence of a malicious adversary.

**Proof:**

To show correctness, we separate the proof to the case where $x$ is in the intersection and the cases where $x$ does not belong to $A^i$, for each $i \in [n]$. Case 1: $x \in A^i$. $P_1$ sends $S_1$ to $P_n$ without encoding $x$ as a key in $S_1$. Then, $y_n = \text{Decode}(S_n, x)$ is a pseudorandom value that with overwhelming probability not equal to $y_{n-1} = \bigoplus_{i=2}^{n-2} \text{Decode}(S_i, x)$. Thus, even if $x \not\in A^i$ for all $i \in \{2, \ldots, n\}$, the values $x \parallel y_{n-1}$ and $x \parallel y_n$ input to $\mathcal{F}_{\text{psi}}^{\pi}$ would not match, therefore $x$ is not output as part of the intersection.

Case 2: $x \not\in A^i$ for some $i \in \{2, \ldots, n-2\}$. Party $P_i$ sends $S_i$ to $P_{n-1}$ without encoding $x$ as a key. Thus, $\text{Decode}(S_i, x)$ is a pseudorandom value that with overwhelming probability not equal to $F(k_i, x)$. Therefore, if $P_n$ has $x$, it inputs $x \parallel \bigoplus_{i=2}^{n-1} \text{Decode}(S_i, x)$ which is a pseudorandom value not equal to $x = \bigoplus_{i=2}^{n-1} \text{F}(k_i, x)$ whereas if $P_n$ has $x$ it inputs $x \parallel \hat{x}$, meaning that $x$ is not output as part of the intersection.

Case 3: $x \not\in A^i \lor x \not\in A^n$. Parties $P_{n-1}$ and $P_n$ concatenate their plain-text values in the beginning of each value of their sets $\hat{A}^{n-1}$ and $\hat{A}^n$ respectively. Thus, they do not obtain a value corresponding to $x$ from $\mathcal{F}_{\text{psi}}^{\pi}$, from the correctness of this functionality.

**Simulation.** We turn to show security by presenting a simulator to each of the following four cases, for each case, we describe simulation in both the semi-honest and malicious settings.

**Corrupted $P_1$.** In the semi-honest setting the simulator is given $P_1$’s input $A^i = \{a^i_1, \ldots, a^i_m\}$, it inputs it to the ideal-world functionality and obtains an empty output. In the real execution, $P_1$ receives no further messages, which trivially concludes the simulation.

In the malicious setting, the simulator has to extract $P_1$’s actual input. To do so, the simulator internally runs $P_1$ and for each call $H(x)$ to the random oracle, the simulator enters $x$ to a list $L$. Then, playing the role of $P_n$ who receives $S_n$ and $P_i$ $(i = 2, \ldots, n-1)$ who receives $k_i$, the simulator concludes with the actual input set $A^i = \{x \in L \mid \bigoplus_{i=2}^{n-1} \text{F}(k_i, x) = \text{Decode}(S_n, H(x))\}$. The simulator inputs $A^i$ to the ideal-world functionality. Note that except with negligible probability, for every $x'$ that was not queried to the random oracle, it follows that $\bigoplus_{i=2}^{n-1} \text{F}(k_i, x') \neq \text{Decode}(S_n, H(x'))$. 
and thus $x'$ does not appear in the real execution result intersection. This conclude the simulation because $P_1$ does not receive any further message in both worlds.

**Corrupted $P_i$ ($2 \leq i \leq n - 2$).** In the semi-honest setting, the simulator is given $P_i$’s input $A_i$, so it inputs that set to the ideal-world functionality. In addition, $P_i$ receives a random key from $P_i$ in the real execution, so the simulator generates a random key $k_i$ and set it as $P_i$’s view, which concludes the simulation since $P_i$ receives no further messages.

In order to extract $P_i$’s actual input in the malicious setting, the simulator internally runs $P_i$ and for each call $H(x)$ to the random oracle, the simulator enters $x$ to a list $L$. Then, playing the role of $P_{n-1}$ who receives $S_i$, the simulator concludes with the actual input set $\hat{A}^i = \{x \in L \mid \text{Decode}(S_i, H(x)) = F(k_i, H(x))\}$ where $k_i$ is the key that the simulator gives $P_i$ in the internal execution. The simulator inputs $\hat{A}^i$ to the ideal world functionality. Note that, except with negligible probability, for every $x'$ that was not queried the random oracle, it follows that $\text{Decode}(S_i, H(x')) \neq F(k_i, H(x'))$, and thus $x'$ does not appear in the real execution result intersection. This concludes the simulation since $P_i$ receives no further messages in the real execution.

**Corrupted $P_{n-1}$.** In the semi-honest case, the simulator has $A^{n-1}$, so it inputs that set to the ideal world functionality. In the real execution $P_{n-1}$ receives an OKVS from $P_i$ ($2 \leq i \leq n - 2$), so the simulator computes $S_i \leftarrow \text{Encode}((\{k_i, u_i\})_i \mid \text{with } m \text{ random pairs (}k_j, u_j)\text{ and sends } S_j \text{ to } P_{n-1}, \text{ for each } i \in (2, \ldots, n-2).$ By the obliviousness property of $S_i$, it is not possible to distinguish between $S_i$ output by the simulator and an OKVS that encodes $A^i$ as keys in the real execution. This concludes the simulation since $P_{n-1}$ receives no further messages in the real execution.

In the malicious setting, the simulator extracts $P_{n-1}$’s input set as follows: it runs $P_{n-1}$ internally with the $n-2$ random $S_i$ as its first messages, as described above. It observes the set of $P_{n-1}$’s random oracle queries and records them in a list $L$. Then, it receives $P_{n-1}$’s input set $L'$ to the $\mathcal{F}_{\text{psi}}^{2, m}$ functionality and concludes with the set $\hat{A}^{n-1} = \{x \in L \mid x \mid \hat{x} \notin L'\}$ where $\hat{x} = \bigoplus_{i=2}^{n-3} \text{Decode}(S_i, H(x))$. The simulator inputs $\hat{A}^{n-1}$ to the ideal world functionality. As before, values $x$ that are not in $L$ or not in $L'$ would not be found in the intersection in the real execution (except with negligible probability) and therefore can be ignored in the ideal world execution. This concludes the simulation as $P_{n-1}$ receives no further messages in the real execution.

**Corrupted $P_n$.** In the semi-honest case, the simulator has $A^n$, inputs it to the ideal world functionality, and obtains the intersection $X$ back. The simulator sends a random OKVS $S_n$ and the set $\hat{X} = \{x \mid \text{Decode}(S_n, x) \mid \hat{x} \notin X\}$ to $P_n$ and outputs whatever it outputs. By the obliviousness property of the OKVS, $S_n$ and $\hat{X}$ in both worlds are computationally indistinguishable and expose the same correlation, i.e. for each $x \mid \hat{x} \in \hat{X}$ it follows that $\text{Decode}(S_n, x) = \hat{x}$.

The extraction of $P_n$’s actual input in the malicious setting follows. The simulator runs $P_n$ internally with the random OKVS, $S_n$, as its first message. It observes the set of $P_n$’s random oracle queries and records them in the list $L$. Then, it receives $P_n$’s input set $L'$ to the $\mathcal{F}_{\text{psi}}^{2, m}$ functionality and concludes with the set $\hat{A}^n = \{x \in L \mid x \mid \hat{x} \notin L'\}$ where $\hat{x} = \text{Decode}(S_n, H(x))$. The simulator inputs $\hat{A}^n$ to the ideal world functionality and receives $X$ back. It sends to $P_n$ the set $\hat{X} = \{x \mid \text{Decode}(S_n, H(x))\} \chi \notin X$ and outputs whatever it outputs.

\[\square\]

## 4 PSI WITH ARBITRARY COLLUSION

Recall the insecurity of Protocol 3.3 against a collusion of two parties. Specifically, when $P_1$ colludes with $P_{n-1}$, they have both the keys $k_j$ and the OKVSes $S_i$ for all $i \in [2, n-2]$, which means they can reveal $P_1$’s input if the domain is small enough. Furthermore, when $P_{n-1}$ and $P_n$ collude, they can reveal the intersection of all parties $P_1, \ldots, P_{n-2}$, which is not allowed by the functionality.

We can mitigate the above attacks as follows: First, $P_1$ picks key $k_{j}$ for $P_i$ for $i \in [2, n-3]$ and computes $S_n$ based on these keys. Now, each $P_i$ for $i \in [2, n-3]$ picks an additional key $k'_i$ and computes its $S_i$ by $S_i \leftarrow \text{Encode}((\{a'_i, F(a'_i)\})_i \mid \text{with } F(a'_i) = F(k'_i, a'_i) \oplus F(k_i, a_i)$, and sends it to $P_{n-1}$. In addition, $P_i$ sends $k'_i$ to $P_{n-2}$, which computes $\{a_i^{n-2}\} \bigoplus_{i=2}^{n-3} F(k'_i, a_i) \oplus F(k_{n-1}, a_{n-1}^{n-2}), m \}$. At this point, the ‘important information’ of the parties is spread amongst three parties $P_{n-2}, P_{n-1}$ and $P_n$. Specifically, for an item $a$ in the intersection, party $P_{n-2}$ holds $a_{n-2} = \bigoplus_{i=2}^{n-3} F(k'_i, a_i)$, party $P_{n-1}$ holds $a_{n-1} = \bigoplus_{i=2}^{n-3} F(k'_i, a_i)$ and party $P_n$ holds $a_n = \bigoplus_{i=2}^{n-3} F(k_i, a_i)$. Notice that $\bigoplus_{i=2}^{n-3} F(k'_i, a_i) \oplus \bigoplus_{i=2}^{n-3} F(k_i, a_i) = 0$. For other values $a$ which are not in the intersection, the result of $\bigoplus_{i=2}^{n-3} F(k'_i, a_i)$ is pseudorandom. To find out the items for which the sum $\bigoplus_{i=2}^{n-3} F(k'_i, a_i) \oplus \bigoplus_{i=2}^{n-3} F(k_i, a_i) = 0$ the three parties $P_{n-2}, P_{n-1}$ and $P_n$ run a sub-protocol called ZeroXOR, which outputs exactly those items. This solves the aforementioned issues since now there are no two parties that have sufficient information to reveal the intersection of the honest parties.

In Section 4.1 we introduce the ZeroXOR functionality and protocol and in Section 4.2 we present our protocol that uses it in order to resist an arbitrary corruption of $t < n$ parties.

### 4.1 ZeroXOR

Let us introduce the ZeroXOR functionality. Intuitively, it allows $n$ parties, where $P_{i} \in [n]$ holds a set of key-value pairs $X_i = \{(x_i, y_i)\}_i \in \{m\}$, to determine all keys that satisfy the two conditions: (1) the key is in the intersection set of all parties’ keys; (2) the XOR of the values associated with these common key from each party is zero. We formally present the ZeroXOR functionality and its construction in Figure 4.1 and Figure 4.2, respectively.

**FUNCTIONALITY 4.1. ($\mathcal{F}_{\text{ZeroXOR}}^{n, m}$)**

**Parameters:** $n$ parties.

**Behavior:** Wait for input $X_i = \{(x_i', y_i')\}_{i \in \{m\}$, from $P_{i} \in [n]$ where $(x_i, y_i) \in (0, 1)^k, (0, 1)^l$.

Give $P_n$ the set $\{x \mid \forall i \in [n] : (x, y'_i) \in X_i \text{ and } \bigoplus_{i \in [n]} y'_i = 0\}$. 

---

9
The zero-sharing functionality and protocol can be extended to any set of parties $\mathcal{P}$.

Thus, we omit the proof of the following theorem.

PROTOCOL 4.2. ($\pi_{\text{zeroXOR}}$)

PARAMETERS: A PRF $F$, an OPPRF functionality, $n$ parties where $P_i$ has the set $X_i = \{(x^i_j, y^i_j)\}_{j \in [m]}$.

Protocol:

1. $P_{\{i\}[n]}$ invokes $\text{ZeroShare}$ on $x^i_j$ and obtains its share $s^i_j = S(K_i, x^i_j)$ for every $j \in [m]$.
2. $P_{\{1,\ldots,n\}-1}$ and $P_n$ jointly invoke $\mathcal{F}_{\text{opprf}}^{F,m,m}$,
   - $P_i$ acts as a sender, programming $\mathcal{P} \rightarrow (x^i_j, y^i_j)_{j \in [m]}$.
   - $P_n$ acts as a receiver with queries $(x^n_j)_{j \in [m]}$.
   - $P_n$ obtains $((x^i_j, y^i_j))_{j \in [m]}$ where $z^i_j = y^i_j$ if $(x^i_j, y^i_j) \in X_i$ and a pseudorandom value otherwise.
3. Party $P_n$ outputs $\{s^m_j | s^m_j + y^n_j = \bigoplus_{i \in [n-1]} s^i_j\}$.

One could use an OPPRF to implement our ZeroXOR as follows. Each party $P_i \in [n-1]$ with a set of key-value pairs $X_i = \{(x^i_j, y^i_j)\}_{j \in [m]}$ allows $P_n$ to submit $(x^j, y^j)_{j \in [m]}$ as queries, and to obtain the associated responses $z^j$ from $P_i$. Now, $z^j$ is equal to $y^j$ if $x^j = x^j$, otherwise, $z^j$ is pseudorandom. Consequently, if all parties have the key $x^j$, the XOR of all responses $z^j_i$, $i \in [n-1]$, are equal to $P_n$'s value $y^j$, by which $P_n$ concludes that $x^j$ is in the intersection.

While the above correctly implements ZeroXOR functionality and may be adequate in some scenarios, it is not secure in general. Concretely, $P_n$ learns the actual values of the common items of other parties $P_i$ even if their keys are not in the intersection. To address this security issue, we rely on the zero-sharing idea of [23], which serves as a one-time pad over the values associated with the parties’ keys. The zero-sharing functionality and protocol are given in Section 2. Note that the zero-sharing construction of [23] is ‘unconditional’, i.e., it produces an unlimited number of pseudorandom zero-shares derived from short seeds that can be exchanged in a one-time initialization step.

The security of ZeroXOR follows in a straightforward way from the security of its building blocks (e.g. OPPRF and zero-sharing). Thus, we omit the proof of the following theorem.

THEOREM 4.3. Protocol $\pi_{\text{zeroXOR}}^{F,m}$ (Figure 4.2) securely implements $\mathcal{F}_{\text{zeroXOR}}^{F,m}$ (Figure 4.1) in the presence of a malicious adversary corrupting $t < n$ parties, in the $\mathcal{F}_{\text{opprf}}^{F,m,m}$ hybrid model.

4.2 The Protocol

The construction of our $\pi_{\text{opprf}}^{n,t,m}$ is formally presented in Protocol 4.4 and the intuition follows. The idea described above specifically for $t = 2$ can be extended to any $t < n$ as follows. Let $v = n - t$, the parties $P_1, \ldots, P_{v-1}, P_0, P_{v+1}, \ldots, P_n$ are separated to three parts. The first part $P_1, \ldots, P_{v-1}$ take a role of a client; the third part $P_{v+1}, \ldots, P_n$ take a role of a server; and the final $P_0$ is a pivot. Each client $P_i$ generates and sends a key $k^i_j$ to every server $P_j$. In addition, the client $P_i$ generates an OKVS $S_i$ such that each item $a^j_i \in A^j$ is associated with the XOR of the PRF results using all keys, namely, $\bigoplus_{j \in [v+1, n]} F_j(a^j_i)$. Each client $P_i$ sends $S_i$ to the pivot party, which decodes and XOR them according to its own set. That is, for every item $a^j_i \in A^j$, compute $\bigoplus_{j \in [v+1, n]} F_j(a^j_i)$. A server $P_j$ has all keys $k^j_i$ for $i \in [v-1]$. It uses those keys to obtain $\bigoplus_{j \in [v-1]} F_j(a^j_i)$ for every $a^j_i \in A^j$. If $x$ is in the intersection then the values obtained by the pivot and the $t$ servers are XORed to zero, and otherwise, they are XORRed to a pseudorandom value. To find which ones are XORRed to zero the pivot and servers invoke the ZeroXOR functionality. It holds that the $t + 1$ parties $P_0, \ldots, P_n$ (i.e. the pivot and the $t$ servers) hold the information in order to determine which items are in the intersection. In addition, any subset of $t$ or fewer parties could not determine the intersection.

4.2.1 Complexity and security. In the following, we analyze the performance of our protocol, considering the only dependency in $m$ and $t$. All complexities also depend on the computational security parameter $\kappa$, which we omit.

The computational complexity for clients is proportional to the set size $m$ and the number of corrupted parties $t$, since each client $P_i$ for $i \in [v-1]$ generates an OKVS based on all $t$ keys $k^i_j$ for $j \in [v+1, n]$. A single OKVS is sent from each client to the pivot party, and thus the communication complexity of a client depends only on $m$.

The computational complexity for the pivot party depends on $n - t$ since it decodes the OKVS given from each client. The computation and communication complexities of the ZeroXOR protocol depend on the cost of the OPPRF, which is linear in $m$. Therefore, the overall (communication and computation) cost for the pivot party is $O(m(n - t))$. 

PROTOCOL 4.4. ($\text{PSI with collusion - } \pi_{\text{opprf}}^{n,t,m}$)

PARAMETERS: There are $n$ parties $P_1, \ldots, P_n$ and an adversary $\mathcal{A}$. Party $P_i$ has the set $A^i = \{a^j_i\}_{j \in [m]}$. The protocol uses an OKVS scheme (Encode, Decode) and a PRF $F$ modeled as a random oracle in the malicious setting.

Protocol 4:

1. Let $v = n - t$. That is, the parties are $P_1, \ldots, P_{v-1}, P_0, \ldots, P_n$ s.t. $\{|P_{v+1}, \ldots, P_n|\} = t$.
2. Party $P_i$ for $i \in [1, v-1]$ chooses keys $k^i_j$ for $j \in [v+1, n]$ and sends $k^i_j$ to $P_j$.
3. Party $P_i$ for $i \in [1, v-1]$ sends $S_i$ to $P_0$ where $S_i \leftarrow \text{Encode}((u^j, a^j_i)_{j \in [v+1, n]})_{q_n|m}(
ad)$
4. Party $P_0$ received $S_i$ for $i \in [1, v-1]$. It computes the key-values set $X^0 = \left\{ (a^0_j, \bigoplus_{k=1}^v \text{Decod}(S_i, a^j_i))_{q_n|m} \right\}$
5. Party $P_i$ for $i \in [v+1, n]$ received keys $k^i_j$ for $j \in [v-1]$. It computes the key-values set $X^1 = \left\{ (a^j_i, \bigoplus_{k=1}^v F_k(a^j_i))_{q_n|m} \right\}$
6. Parties $P_0, \ldots, P_n$ invoke functionality $\mathcal{F}_{\text{zeroXOR}}^{F,m}$ with their corresponding sets $X_0, \ldots, X_m$, by which $P_n$ obtains the intersection.

\*In case that $\mathcal{A}$ is malicious, party $P_i$ uses $H(a^j_i)$ instead of $a^j_i$ in steps (3)-(5) above, where $H$ is a random oracle.
Servers receive only keys from the clients which do not depend on the set size. Their computation is a PRF computation per key per item. In addition, they are engaged in the ZeroXOR protocol, which incurs a linear overhead in $m$ for all parties, except for $P_n$, who is involved in $t$ OPPRF invocations (with each of $P_{i}$ for $i \in \{0, n - 1\}$). Thus, for server $P_i$ $(i \in \{0+1, n-1\})$ the computation and communication complexities are $O(m(n-t))$ and $O(m)$ respectively and for the server $P_n$ they are $O(m(n-t))$ and $O(mt)$.

Consider the round complexity of the protocol. Steps (2)-(3) are run in parallel, steps (4)-(5) are computation only, and step 6 is the ZeroXOR protocol which incurs one round for the ZeroShare protocol in addition to the round complexity of the OPPRF. Overall, there are 4 rounds.

**Theorem 4.5.** Protocol 4.4 securely computes functionality $\mathcal{F}^{n,t,m}_{\psi}$ in the $\mathcal{F}^{F,n,m}_{\text{zeroXOR}}$-hybrid and random oracle model in the presence of a malicious adversary corrupting $t < n$ parties.

**Proof Sketch.** As explained in the introduction, each client $P_i$ ($i \in \{0, 1\}$) essentially produces a conditional zero sharing for each item $x \in A^t$. That is, it provide the pivot with an OKVS $S$ and the servers with keys $k^i$ $(j \in \{0 + 1, n\})$ such that if they query these object on $x \in A^t$ they obtain the shares $s_0, s_{n+1}, ..., s_n$ such that $\oplus_{j=0}^{n} s_j = 0$. Otherwise, if even one of $P_0, ..., P_n$, say $P_k$, does not query about $x$, then the probability that it holds $s_k$ such that $\oplus_{j=0}^{n} s_j = 0$ is negligible. Now, to obtain only those items for which their shares sum up to zero, the pivot and the servers use the ZeroXOR functionality.

As a corollary, combining the conditional zero-sharing produced by all clients leads to that the pivot and the servers have a shares of zero only for items that are in the intersection of all parties.

Extracting any party’s input is done by the simulator internally running the party $P_i$ and for each call $H(x)$ to the random oracle, the simulator enters the input $x$ to a list $L$. After the party sent her derived OKVS $S_i$ to $P_0$ (for $P_i$, $i \in \{0, n - 1\}$) or key-value set $X_i$ to $\mathcal{F}^{\text{zeroXOR}}$ (for $P_i$, $i \in \{0, n\}$), the simulator can conclude with the actual input set $A^t$, similarly to the proof of Theorem 3.4. We denote the set or parties corrupted by $A$ as $C$. The case where $C \subseteq \{P_0, ..., P_{n-1}\}$ is trivial, as none of these parties receive any information which depends on any input set $A^t$. For the case where $P_0 \in C$, we note that any input $a_q^i$ received after $S_i$ is encrypted using $P_i$’s $t$ generated PRF keys. Thus, simulating $S_i$ is easy as $S_i$ appears random to $P_0$, even if she receives any $t-1$ PRF keys. Next, assume Party $P_i \in C$, $i \in \{0 + 1, n\}$, $P_i$ receives only PRF keys from $P_j$, $j \in \{1, v - 1\}$, and not any item which depends on other party’s input. Thus, all the simulator’s protocol to do is generate random PRF keys and hand it to $A$. $P_n$ also receives the outputs from $\mathcal{F}^{F,t+1,m}_{\text{zeroXOR}}$ so the simulator outputs whatever $P_n$ outputs.

### 5 PERFORMANCE EVALUATION

We implemented our protocols 3.3 and 4.4 for the cases of no collusion and arbitrary collusion, respectively, and compared them with the state-of-the-art multiparty PSI protocols by Kolesnikov et al. [23] (in both the semi-honest and augmented semi-honest settings), Chandran et al [2] (for semi-honest honest majority) and Ben-Efraim et al. [10] (for malicious dishonest majority). Note that the comparison with the augmented semi-honest version of [23] covers also a comparison with the malicious version of [13], since they only diverge in the OPPRF instantiation and the former is faster. In our reports, for $t = 1$ we used our protocol 3.3 and for $t > 1$ we used protocol 4.4. When $t = n - 1$, $\pi_{\psi}$ protocol 4.4 in fact requires only performing ZeroXOR with n parties, each holding $X_i = (a^i_q, \psi^i_q)_{q \in \mathbb{F}_n}$. That is, when $t = n - 1$ we have no clients, the pivot party is $P_1$ and $P_2, ..., P_n$ are servers.

Similar to [2, 23], we used a single machine 2x 36-core Intel Xeon 2.30GHz CPU and 256GB of RAM and simulated network using the Linux tc command. Our LAN setting has 0.02ms round-trip latency and 10 Gbps network bandwidth. Our WAN setting has 96ms round-trip latency and 200 Mbps network bandwidth. Similar to [2, 23], in order to ensure parallelism as promised in our protocols, each party uses a separated thread to communicate with each other party.

<table>
<thead>
<tr>
<th>$m$</th>
<th>$2^{12}$</th>
<th>$2^{16}$</th>
<th>$2^{20}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encode</td>
<td>0.052</td>
<td>0.103</td>
<td>2.838</td>
</tr>
<tr>
<td>Decode</td>
<td>0.003</td>
<td>0.005</td>
<td>0.99</td>
</tr>
</tbody>
</table>

Table 2: OKVS performance: Run time in seconds of the PaXoS [29] algorithms Encode and Decode.

Our implementation uses the table-based OPPRF2 code from [23], OPPRF code from [22]. We use Encode and Decode based on PaXoS data structure [29] and give a detailed running time for it in Table 2. For the PaXoS cuckoo table we use the expansion parameter of 2.5, i.e. the number of bins in the cuckoo table is 2.5m. We instantiate the PRF F using AES-NI. All evaluations were performed with item input length of 128 bits, statistical security parameter $\lambda = 40$ and computational security parameter $\kappa = 128$. When $t = n - 1$, our $\pi_{\psi}$ protocol can be optimized by only performing ZeroXOR with n parties, each holding $X_i = (a^i_q, \psi^i_q)_{q \in \mathbb{F}_n}$. Our complete implementation is available on GitHub: https://github.com/asu-crypto/mPSI

### 5.1 Comparison with Prior Work

For the most direct comparison, we consider the following values of $(n, t) \in \{(4, 1, 3), (10, 1, 4, 9), (15, 1, 4, 7, 14)\}$. Note that $(4, 1), (10, 4), (15, 7)$ were reported explicitly in the experimental analysis of [2]. The settings with $t > \frac{n}{3}$ is not supported by the protocol [2]. Due to lack of time, we are unable to run the implementation of ENOC [10] (which was on Github ~3 months ago) on our benchmark machine (due to errors in the build process which we could not handle at that time frame). Therefore, ENOC runtime is taken from [10, Table 5]. Table 5 [10] does not report on $n \in \{10, 15\}$, therefore we use $n \in \{8, 12\}$ for them instead (on which they do report), respectively, which is more favour to them. The communication cost of ENOC [10] is calculated from their protocol description. The exact calculation we performed is detailed in Appendix D. In their evaluation reports it can be seen that their protocol does not scale beyond 16 parties with sets larger than $2^{16}$ and beyond 32 parties even for sets larger than $2^{14}$. In contrast, our protocols perform well with 32 parties, even with sets of $2^{20}$ items.

Recall that our $\pi_{\psi}$ protocol consists of three types of parties: client $P_i \in \{0, v - 1\}$, pivot $P_0$, and server $P_{i,v} \in \{0, n\}$. Since clients...
When comparing the protocols, we find that the client’s running time of our protocol is significantly less than that of the prior works, requiring only 3.4 seconds to perform a PSI with \( m = 2^{20} \) in the LAN setting. This is a 23x improvement and 1.6~28x faster than the concurrent work CDGOSS [2] and KMPRT [23], respectively. When \( t = n - 1 \), our protocol essentially consists only of the \( n \)-party ZeroXOR protocol, which has the same communication and computation cost as the augmented semi-honest version of KMPRT.

Table 3 shows the communication overhead of the protocols. Our protocol requires 7~15x, 2.5~90x, and 18~270x less communication cost than CDGOSS [2], KMPRT [23], and ENOC [10] on the client’s side, respectively. Note that the bandwidth requirement of our client is almost constant in \( t < (n - 1) \) and \( n \) since the client’s major communication cost falls in sending encoding set to a pivot party \( P_c \). Therefore, the client’s performance of our protocol is extremely favorable when \( t \) and \( n \) are large. For the total communication cost, our protocol also shows a 3~4x, 2.6~20x, and 16~330x improvement compared to previous work [2, 10, 23], respectively.

**5.2 Extended Evaluation of Our Protocols**

To understand the scalability of our protocols, we evaluate them on the range of the number parties \( n \in \{3, 4, 5, 8, 16, 32\} \), corruption threshold \( t \in \{1, 3, \frac{n}{2}\} \) on the set size \( m \in \{2^{12}, 2^{16}, 2^{20}\} \).

We report their detailed computational performance results in Table 5, showing total running time in both LAN and WAN settings. We find that our protocols scale well in the experiments. Indeed, the performance of our protocol is mostly constant in the number of parties \( n \) when \( t \) is fixed, because the ZeroXOR protocol dominates the run time. For instance, when fixing \( t = 3 \), the total running times of our protocol for \( n = 5 \) and \( n = 32 \) are 24.11 and 25.52 seconds, respectively, for \( m = 2^{20} \).

**Acknowledgements**

The second author is partially supported by NSF awards #2031799, #2101052, and #2115075. We are grateful to Tamir Tassa as well as the CCS 2021 anonymous reviewers for their feedback and editorial suggestions.
REFERENCES


[15] Christoph Hagen, Christian Weinert, Christoph Sendner, Alexander Dimitrienko, and Thomas Schneider. All the numbers are US: large-scale abuse of contact discovery in mobile messengers. NDSS, 2021.


A SERVER-AIDED TWO-PARTY PSI

We present below the protocol by Kamara et al. [20] that computes functionality \( F_{2,1,m} \) (another protocol was recently proposed by Le et al. [24], which is better suitable to the circuit-based PSI functionality).

The protocol utilizes a third-party non-colluding server, which may be malicious. The semi-honest version of the protocol is presented in Protocol A.1.

### A.1 Kamara et al. [20]

The parties have to withstand a corrupted server, who tries to omit items from the intersection. This is done as follows. Each party augments its set \( A^i \) with \( \lambda \) copies of each element. Specifically, party \( P_i \) generates the set \( A^i = \{ a^i_1, \ldots, a^i_{\lambda} \} e_{\{m\}} \) (each item is replicated \( \lambda \) times, each it is concatenated with the next index from \( 1, \ldots, \lambda \)). Then, the parties run the semi-honest protocol above on the sets \( A^1 \) and \( A^2 \). Now, to omit a single item \( x \) from the intersection, the server has to omit \( \lambda \) pseudorandom items.
from $X$, namely, the items $F(k, x[1]), \ldots, F(k, x[|X|])$. Since all values seen by the server are pseudorandom, it is difficult to tell which pseudorandom items encode the same value and thus it is unlikely that the server omits exactly those $\lambda$ items.

Note that it is still possible for the server to omit all values from the intersection. This is easily fixed by having the parties add an agreed upon item to both sets $A^1$ and $A^2$, by which, it is guaranteed that the intersection is not empty. So if the server returns $X = \emptyset$, it is caught cheating.

Finally, note that it is still possible for the server to return $X = \hat{A}^1$ to $P_1$ (and similarly $X = \hat{A}^2$ to $P_2$) by which the parties conclude that the intersection contains all items. This is again easily fixed by agreeing on one dummy item $d_1$ which is added only to $\hat{A}^1$ and another dummy item $d_2$ which is added only to $\hat{A}^2$. This ensures that the intersection does not contain the entire set, hence, returning $X = \hat{A}^1$ is immediately treated as cheating. This is presented formally in Protocol A.2.

B. MALICIOUS OPPRF

There are two parties, sender $S$ and receiver $R$. The sender $S$ has a set of points $\mathcal{P} = \{(a_1, t_1), \ldots, (a_{m_1}, t_{m_1})\}$ and the receiver $R$ has queries $(q_1, \ldots, q_{m_2})$. The template of an OPPRF construction follows: The parties run an OPPRF which outputs a key $k$ to the sender and the PRF results $F(k, q_1), \ldots, F(k, q_{m_2})$ to the receiver. The sender computes the hint as follows. For each $a_1$ compute $i_1 = F_k(a_1) \oplus t_1$. Then, the sender generates an OKVS by $S \leftarrow$ Encode($\{(i_1, a_1)\}_{a_1 \in \mathcal{P}}$) and sends it to the receiver. For each of the receiver’s query $q_j$ and OPPRF results $F(k, q_j)$, the receiver computes the OPPRF result $y_i = F(k, q_j) \oplus \text{Decode}(S, q_j)$.

Suppose that the receiver queries the OPPRF on some $q = a_j$, then the OPPRF result is $y_i = F(k, a_j) \oplus \text{Decode}(S, a_j) = F(k, a_j) \oplus F(k, a_j) \oplus t_j = t_j$ as required. On the other hand, for a query $q \neq a_j$ for all $j$, the result $y_i = F(k, q) \oplus \text{Decode}(S, q)$ is pseudorandom because $F(k, q)$ is a pseudorandom value that has never been used before in the construction of $S$.

The above template builds on an OPPRF that supports multiple queries by the receiver (specifically $m_2$ queries) whereas concretely efficient OPPRFs directly support a single query only.

PROTOCOL A.2. (Server-Aided 2-Party PSI [20] - $\pi^{2,1,m}$)

**Parameters:** There are 2 parties $P_1$, $P_2$ and a third-party server $S$. $P_1$ and $P_2$ have sets $A^1$ and $A^2$ as input, respectively. $S$ does not have inputs. Let $F$ be a PRF.

**Protocol:**

1. $P_1$ chooses sets $D_0, D_1, D_2$ and a key $k$ such that $|D_0| = |D_1| = |D_2| = d$, sends them to $P_2$ and set $A^1 = A \cup D_0 \cup D_1$.
2. $P_2$ sets $A^2 = A^2 \cup D_0 \cup D_2$.
3. Party $P_1 (i \in (1, 2))$ sends a shuffled version of $\hat{A}^i = \{F(k, x) \mid x \in A^i\}$ to $S$.
4. $S$ computes $X = \hat{A}^1 \cap \hat{A}^2$ and sends $X$ to $P_1, P_2$.
5. $P_1$ aborts if:
   a. $\exists D_0 \notin F^{-1}(k, X)$ or $D_1 \cap F^{-1}(k, X) \neq \emptyset$
   b. $\exists x \in \hat{A}^1$ and $\exists \beta \in \{\lambda\}$ such that $x|\beta \in F^{-1}(k, X)$ and $x|\beta \notin F^{-1}(k, X)$
6. The parties output distinct items in $\{F^{-1}(k, x)\}_{x \in X} \setminus D_0$.

PROTOCOL C.1. (Zero-Sharing - $\pi^{n, m\text{\_zerosh}}$)

**Parameters:** There are $n$ parties $P_1, \ldots, P_n$ and an adversary $A$. There is a PRF $F : \{0, 1\}^k \rightarrow \{0, 1\}^k$.

**Protocol:**

1. Each party $P_i$ picks a random seed $r_{i,j}$ for $j \in [i + 1, n]$ and sends $r_{i,j}$ to $P_j$. The key $K_i$ of party $P_i$ is $(k_{i,1}, \ldots, k_{i,i-1}, k_{i,i+1}, \ldots, k_{i,n})$.
2. To obtain its share for value $x$, party $P_i$ computes $S(K_i, x) = \left( \bigoplus_{j<i} F_{k_{i,j}}(x) \right) \oplus \left( \bigoplus_{j>i} F_{k_{i,j}}(x) \right)$.

To overcome this, two approaches have been proposed. The first one is developed by [23, 30], in which the receiver uses cuckoo hashing and the sender uses a simple hashing. This way, for each bin the receiver has at most one item and the sender has $O(\log m)$ items, so they can invoke the single-query OPPRF per bin. This however is secure in the semi-honest setting only because a malicious sender, who knows (via auxiliary information) that the receiver has item $x$, may put $x$ only in one of the possible bins instead of in all of them. This way, by the PSI result it may learn in which bin the receiver put its item $x$ and by this leaking information on other items that the receiver has. We refer the reader to [23, 30] for more details.

Alternatively, [29] proposed a different approach via a data structure called PaXoS (Probe and XOR of Strings) along with a 1-out-of-N random OT that has an homomorphic properties. This approach withstands a malicious adversary. The receiver encodes its queries in a data structure $D = (d_1, \ldots, d_{m'})$ of size $m'$ (which is greater than $m_2$). Suppose that the PaXoS is parameterized with $k$ hash functions $h_1, \ldots, h_k$, then for every receiver’s query $q$ it follows that $q = \text{Decode}(D, q) = d_{h_1(q)} \oplus d_{h_2(q)} \oplus \ldots \oplus d_{h_k(q)}$.

Then, the sender and receiver run a 1-out-of-N ROT for $m'$ times, where in the $i$-th ROT the receiver obtains the value $r_i = a_i + s \land C(d_i)$ and the sender obtains $a_i$, where $s$ is a random string that is used in all ROT instances (i.e. for all $i$) and $C$ is a linear code. After running all instances of ROT, the receiver treats the results $R = (r_1, \ldots, r_{m'})$ as a PaXoS data structure. Thus, to obtain the result associated with a query $q$ it computes $y = \text{Decode}(R, q) = r_{h_1(q)} \oplus r_{h_2(q)} \oplus \ldots \oplus r_{h_k(q)} = (a_{h_1(q)} \oplus \ldots \oplus a_{h_k(q)}) \oplus s \land (C(d_{h_1(q)}) \oplus \ldots \oplus C(d_{h_k(q)}))$

From the homomorphic property of the ROT scheme, it follows that the sender may obtain the same value $y$, since it knows all ROT results $a_1, \ldots, a_{m'}$ and $s$. If the sender wants to program the point $(q, t)$ in the OPPRF (for a random $t$), it first computes $t' = t \oplus y$ (it can compute $y$ on its own) and encode the point $(q, t')$ in the OKVS sent to the receiver. Upon receiving the OKVS $S$, the receiver compute the OPPRF result $y' = \text{Decode}(R, q) \oplus \text{Decode}(S, q) = y \oplus t' = t$ as required, where $R$ is the PaXoS structure interpretation of the ROT results and $S$ is the OKVS sent by the sender.

C. ZERO SHARING PROTOCOL

See Protocol C.1.
D ESTIMATING COMMUNICATION FOR BEN Efraim ET AL.

We calculate the concrete communication complexity of [10] based on the formulae and optimal parameter instantiations they report in Table 4 and Appendix E of their paper.

The parameters are \( N_{OT} \), \( N_{CC} \), \( N_{BF} \) provided to the protocol, where \( N_{OT} \) represents the number of random OTs to perform, \( N_{CC} \) represents the number of bits to choose for the cut-and-choose check and \( N_{BF} \) represents the size of the Bloom filter.

We calculate a client’s \((P_i, i > 0)\) communication by:
\[
2N_{OT} + N_{CC} \log_2 N_{OT} + N_{CC} \log_2 N_{OT} + \kappa + N_{BF} \log_2 N_{OT} + N_{BF} \kappa
\]

And server’s \((P_0)\) communication by:
\[
2nN_{OT} + nN_{CC} \log_2 N_{OT} + nN_{CC} \log_2 N_{OT} + \kappa + nN_{BF} \log_2 N_{OT}
\]