Simple, Fast Malicious Multiparty Private Set Intersection

Ofri Nevo
The Open University, Israel

Ni Trieu
Arizona State University

Avishay Yanai
VMware Research

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 Garimella et al.).

CCS CONCEPTS

• **Security and privacy** → **Cryptography**; *Privacy protections*; • **Theory of computation** → *Cryptographic protocols.*

KEYWORDS

private set intersection

ACM Reference Format:

Ofri Nevo, Ni Trieu, and Avishay Yanai. 2021. Simple, Fast Malicious Multiparty Private Set Intersection. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21), November 15–19, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3460120.3484772

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 naïve 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

CCS '21, November 15–19, 2021, Virtual Event, Republic of Korea

ACM ISBN 978-1-4503-8454-4/21/11…$15.00
https://doi.org/10.1145/3460120.3484772

Protocol	Communication		Computation		Corruption Threshold	Rounds	Security	Concretely Efficient
	Leader	Client	Leader	Client				
HV17 [17]	$O(nm\lambda)$	$O(m\lambda)$	$O(nm\log m\kappa)$	$O(m\kappa)$	$t < n$	4	semi-honest	No
	$O((n^2 + nm\log m)\kappa)$	$O((n + m\log m)\kappa)$	$O(m^2)$			7	malicious	
GN19 [14]	$O((n^2 + nm)\kappa)$		$O(nm\log m)$	$O(m\log^2 m)$	$t < n$	12	malicious	No
KMPRT17 [23]	$O(nm(\lambda + \kappa))$	$O(m(\lambda + \kappa))$	$O(n\kappa)$	$O(m\kappa)$	$t < n$	3	augmented semi-honest	Yes
		$O(mt(\lambda + \kappa))$		$O(mt\kappa)$		4	semi-honest	
CDGOSS21 [2]	$O(nm(\lambda + \kappa + \log m))$	$O(m(\lambda + \kappa + \log m))$	$O(nm\kappa)$	$O(m\kappa)$	$t < \lfloor (n+1)/2 \rfloor$	8	semi-honest	Yes
ENOC21 [10]	$O(nm\kappa^2 + nm\kappa\log(m\kappa))$	$O(m\kappa^2 + m\kappa\log(m\kappa))$	$O(nm\kappa)$		$t < n$	8	malicious	Yes
Ours- 3.3	$O((m+n)\kappa)$	$O(m\kappa)$	$O(nm\kappa)$	$O(m\kappa)$	t=1	5	malicious	Yes
Ours- 4.4	$O(m\kappa \cdot \max\{t, n-t\})$	$O(m\kappa)$	$O(m\kappa(n-t))$	$O(mt\kappa)$	$t < n$	4	malicious	Yes

Table 1: Analytic comparison of related work with our protocols. Notation: n parties; at most t are corrupted and colluding; each party holds a set of size m. λ and κ are statistical and computational security parameters, respectively.

to 'program' (with distinct x's and pseudorandom y'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 $B_{h_1(x)}, \ldots, B_{h_k(x)}$ 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 $B_{h_1(x)}$ and the receiver indeed obtained y, that necessarily means that the receiver put its query $q = x_i$ in bin $B_{h_1(x)}$, 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. Garimella 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. $n/2 \le t < n$). 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 $A = (a_1, \ldots, a_m)$ (with $a_i \in \{0, 1\}^*$) is encoded to a codeword $B = (b_1, \ldots, b_{m'})$ (where $m' = O(m\lambda)$ and $b_i \in \{0, 1\}$) 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

[1]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].

positions in B equal 0. Thus, to check whether an item x belongs to A, check whether $\bigwedge_{j\in[k]} b_{h_j(a)} = 1$. For every $\hat{a} \notin A$ it holds that $\bigwedge_{j\in[k]} b_{h_j(\hat{a})} = 1$ 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 $b_i \in \{0,1\}^\lambda$) such that for every $a \in A$ it holds that $\bigoplus_{i\in[k]} b_{h_i(a)} = 0^\lambda$ whereas for $\hat{a} \in A$ it holds that $\bigoplus_{i\in[k]} b_{h_i(\hat{a})}$ 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). Specifically, let a sender $\mathcal{S}$ and a receiver $\mathcal{R}$ have the sets $X = (x_1, \ldots, x_m)$ and $Y = (y_1, \ldots, y_m)$ respectively. The receiver encodes $Y' = BF(Y) = (y'_1, \ldots, y'_{m'})$ and the sender encodes $X' = GBF(X) = (x'_1, \ldots, x'_{m'})$ (note that $x'_i \in \{0,1\}^\lambda$ and $y'_i \in \{0,1\}$). Then, for $i \in [m']$ the parties invoke an OT where the sender inputs two strings (m_0, m_1) and the receiver inputs bit b and obtains m_b, where $m_0 \xleftarrow{\$} \{0,1\}^\lambda$, $m_1 = x'_i$ and $b = y'_i$. Let $R = (r_1, \ldots, r_{m'})$ be the vector of OT results. The receiver concludes that $y \in Y$ is in the intersection iff $\bigoplus_{i\in[k]} r_{h_i(y)} = 0^\lambda$.

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 $y'_i = 0^\lambda$ 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^i, m_1^i be the random messages used in the i-th ROT instance, then for each $x_i \in X$ the sender computes $x'_i = \bigoplus_{j\in[k]} m_1^{h_j(x_i)}$ and send x'_i to the receiver. The receiver in turn computes $y'_i = \bigoplus_{j\in[k]} r_{h_j(y_i)}$ 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 λ (the statistical security parameter) whereas the protocols we present in this paper are linear in λ. 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 200× and 300× 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 $p(x) = \sum_{i=0}^{m} \alpha_i x^i$ whose roots are her items $x_1, \ldots, x_m$ and sends the encrypted coefficients $Enc_{ek}(\alpha_i)$ 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 $y^* = Dec_{dk}(y'_i)$ and concludes that y^* is in the intersection iff $y^* \in X$. 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 $(\{x_i, y_i\})$ with (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 $S = (\alpha_0, \ldots, \alpha_m)$ is the set of coefficients of an $(m-1)$-degree polynomial $p(x) = \sum_{i=0}^{m-1} \alpha_i x^i$ 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's are (pseudo)random then so is the polynomial, and p is independent of the x's. 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 $1.5-2.5\times$ 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 $4.2-4.5\times$ more communication and is about $2\times$ 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 S using the points $(a_i^1, F_k(a_i^1))$ for every $a_i^1 \in A^1$ and sends it to P_3. Then P_2 computes $b_i^2 = F_k(a_i^2)$ for every a_i^2 and P_3 evaluates $b_i^3 = S(a_i^3)$ for every $a_i^3 \in A^3$. Note that at this point if x is in the intersection then both P_2 and P_3 have $b_j^2 = b_{j'}^3 = F_k(x)$ 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 $\{b_i^2\}_{i\in[n]}$ and $\{b_i^3\}_{i\in[n]}$ 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* $2\times$ *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 runtimes 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) multi-party PSI protocols [2, 10, 13].

2 PRELIMINARIES

Denote the set $\{1, \ldots, n\}$ by $[n]$. Definitions for [oblivious, programmable] PRF (i.e. OPRF, PPRF, and OPPRF) are given below, taken almost verbatim from [23]. Denote by κ and λ 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 $A^i = \{a_1^i, \ldots, a_m^i\}$.

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 κ, so it is possible to input the item $H(x)$ instead of the original item

$x \in \{0,1\}^*$ where $H : \{0,1\}^* \to \{0,1\}^\kappa$ is a collision-resistant hash function.

> *FUNCTIONALITY 2.1.* (Multiparty PSI - $\mathcal{F}_{\text{psi}}^{n,t,m}$)
> PARAMETERS: The number of parties n, the number of corrupted parties $t < n$ and the size of each input set m.
>
> BEHAVIOR: Wait for input $A^i = \{a_1^i, \dots, a_m^i\} \subset \{0,1\}^*$ from party P_i and abort $\in \{0,1\}$ from the adversary. If abort = 0, give output $\bigcap_{i \in [n]} A^i$ to P_n. Otherwise give $\perp$ to P_n.

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 $F(k, q_1), \dots, F(k, q_m)$, where F is a PRF and $(q_1, \dots, q_m)$ 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.

> *FUNCTIONALITY 2.2.* (Oblivious PRF - $\mathcal{F}_{\text{oprf}}^{F,m}$)
> PARAMETERS: A PRF F, and a bound m on the number of queries.
>
> BEHAVIOR: Wait for input $(q_1, \dots, q_m)$ from the receiver $\mathcal{R}$ where $q_i \in \{0,1\}^\kappa$. Sample a random PRF key k and give it to the sender $\mathcal{S}$. Give $\{F(k, q_1), \dots, F(k, q_m)\}$ to the receiver.

2.0.3 Programmable PRF (PPRF). A programmable PRF consists of the following algorithms:

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

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

> *EXPERIMENT 2.3.* ($\mathsf{Exp}^{\mathcal{A}}(\mathcal{P}, Q, \kappa)$)
> (1) For each $a_i \in \mathcal{P}$ choose random $t_i' \leftarrow \{0,1\}^\kappa$
> (2) $(k, \mathsf{hint}) \leftarrow \mathsf{KeyGen}(\kappa, \{(a_i, t_i') \mid a_i \in \mathsf{keys}(\mathcal{P})\})$.
> (3) return $\mathcal{A}(\mathsf{hint}, \{F(k, \mathsf{hint}, q) \mid q \in Q\})$

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 $\mathcal{A}$:

$$\left|\Pr[\mathsf{Exp}^{\mathcal{A}}(\mathcal{P}_1, Q, \kappa)] - \Pr[\mathsf{Exp}^{\mathcal{A}}(\mathcal{P}_2, Q, \kappa)]\right| \le \mathsf{negl}(\kappa)$$

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.* ($\mathcal{F}_{\text{opprf}}^{F,m_1,m_2}$)
> 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 $\mathcal{P} = \{(a_1, t_1), \dots, (a_{m_1}, t_{m_1})\}$ from the sender $\mathcal{S}$ and input $(q_1, \dots, q_{m_2})$ from the receiver $\mathcal{R}$. Run $(k, \mathsf{hint}) \leftarrow \mathsf{KeyGen}(\kappa, \mathcal{P})$. Give (k, hint) to $\mathcal{S}$ and $(\mathsf{hint}, F(k, \mathsf{hint}, q_1), \dots, F(k, \mathsf{hint}, q_{m_2}))$ to $\mathcal{R}$.

2.0.4 Key-Value Store (KVS). A Key Value Store consists of two algorithms:

- Encode takes as input a set of (k_i, v_i) key-value pairs from the key-value domain, $\mathcal{K} \times \mathcal{V}$, and outputs an object S (or, with negligible probability, an error indicator $\perp$).
- Decode takes as input an object S, a key x and outputs a value y.

A KVS is **correct** if, for all $A \subseteq \mathcal{K} \times \mathcal{V}$ with distinct keys:

- $Pr[\mathsf{Encode}(A) = \perp]$ is negligible.
- if $\mathsf{Encode}(A) = S \neq \perp$ and $(k, v) \in A$ then $\mathsf{Decode}(S, k) = v$.

Oblivious Key-Value Store (OKVS)[13]. Consider Experiment 2.5.

> *EXPERIMENT 2.5.* ($\mathsf{Exp}^{\mathcal{A}}(\mathcal{K} = (k_1, \dots, k_m))$)
> (1) for $i \in [m]$: choose uniform $v_i \leftarrow \mathcal{V}$
> (2) return $\mathcal{A}(\mathsf{Encode}(\{(k_1, v_1), \dots (k_m, v_m)\}))$

We say that a KVS is oblivious if for all $\mathcal{K}_1, \mathcal{K}_2$ of size m and all PPT adversaries $\mathcal{A}$:

$$\left|\Pr[\mathsf{Exp}^{\mathcal{A}}(\mathcal{K}_1)] - \Pr[\mathsf{Exp}^{\mathcal{A}}(\mathcal{K}_2)]\right| \le \mathsf{negl}(\kappa)$$

In other words, if the values v_i are chosen uniformly then the output of Encode hides the choice of the keys k_i.

The key difference between OPPRF and OKVS is that an OPPRF limits the number of queries the receiver can make, whereas in OKVS the receiver is limited by its computational power only. We show that, despite that relaxation, it is possible to replace some invocations of OPPRF within a PSI protocol with invocations of OKVS, which improves performance.

It is proven in [13] that the PaXoS data structure [29] satisfies the correctness and obliviousness OKVS's requirements described above and we use it in our implementation.

2.0.5 Unconditional Zero Sharing [23]. As the name suggests, the unconditional zero sharing provides the parties with a sharing function $S : \{0,1\}^\kappa \times \{0,1\}^\ell \to \{0,1\}^\kappa$ and a key K_i for party P_i, such that for every x, we have that $s_i = S(K_i, x)$ is P_i's random share, and $\bigoplus_{i=1}^n s_i = 0$. The functionality from [23] is given below for completeness of the presentation. Its construction $\pi^{F,n}_{\text{zeroShare}}$ is presented in Protocol C.1.

> *FUNCTIONALITY 2.6.* (Zero-Sharing - $\mathcal{F}^{F,n}_{\text{zeroShare}}$)
> PARAMETERS: n parties. The dictionary store is initialized to $\emptyset$.
>
> BEHAVIOR: Upon an input x from P_i, if store$[x]$ does not exist, generate random values $s_1, \ldots, s_n$ s.t. $\bigoplus_{i=1}^{n} s_i = 0$ and store store$[x][i] = s_i$ for $i \in [n]$. Output store$[x][i]$ to P_i.

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.

In Section 3.1, we present a recursive multiparty PSI protocol for the case of no collusion, that is, the adversary corrupts at most one party. In particular, this covers an important setting of 3 parties and an honest majority (which was extensively explored in the MPC literature, e.g. [27]). Obviously, if a multiparty protocol incurs $O(1)$ rounds, then the recursive protocol incurs $O(n)$ rounds. In Section 3.2 we present an optimization of the first protocol, which has only $O(1)$ rounds.

3.1 A Recursive Construction with $O(n)$ rounds

We reduce the problem of n-party PSI with no collusion (i.e. $t = 1$) to the problem of $n-1$-party PSI with no collusion. The idea is that party P_1 chooses a random PRF key k, which she sends to P_2. She then encodes her input A^1 into an OKVS S as $S \leftarrow \text{Encode}(\{(a^1_j, F(k, a^1_j))\}_{a^1_j \in A^1})$, which she sends to $P_3, \ldots, P_n$. P_2, in turn, computes $\tilde{A}^2 = \{F(k, a^2_j) \mid a^2_j \in A^2\}$. $P_i \in \{P_3, \ldots, P_n\}$ decodes the given OKVS on its values A^i and obtains $\tilde{A}^i = \{\text{Decode}(S, a^i_j) \mid a^i_j \in A^i\}$. Now, parties $P_2, \ldots, P_n$ run $\mathcal{F}^{n-1}_{\text{psi}}$ with their new sets $\tilde{A}^i$. The parties repeat this process recursively until party P_n obtains the result.

Note that the above simple recursive protocol has a caveat: a malicious P_1 could encode $(a', F(k, a''))$ in the OKVS where $a' \in A^i$ for all $i = 3, \ldots, n$ and $a'' \in A^2$ but neither a' nor a'' are in the intersection (suppose that P_1 has that auxiliary information). This way, P_n incorrectly obtains a' in the output, since now all parties P_i ($i \in \{1, \ldots, n\}$) input $F(k, a'')$ to $\mathcal{F}^{n-1}_{\text{psi}}$.

We can easily mitigate that attack. Our protocol (Protocol 3.1) instructs $P_2, \ldots, P_n$ to augment the items they input to $\mathcal{F}^{n-1}_{\text{psi}}$: instead of only $\tilde{a}^i_j = \text{Decode}(S, a^i_j)$ party P_i inputs both a^i_j and $\tilde{a}^i_j$ (a concatenation of them). This ensures that $\mathcal{F}^{n-1}_{\text{psi}}$ outputs only the correct intersection.

3.1.1 Recursion Base Case: Server-Aided Two-Party PSI. The template above shows a reduction from n-party PSI to $(n-1)$-party PSI. Our base case would be a protocol for two parties. We observe that, since there is at most one corrupted party, this base case can be instantiated by a server-aided two-party PSI, where one of $P_1, \ldots, P_{n-2}$ takes the role of the server. Specifically, we can use the server-aided PSI in Kamara et al. [20] or the one by Le et al. [24]. Both protocols allow two parties to obtain the intersection of their sets using an untrusted third party where it is assumed that the third party does not collude with neither of the parties. Since these protocols with a non-colluding server are much more efficient our overall construction becomes more efficient as well. For completeness, a description of those protocols is given in Appendix A.

> *PROTOCOL 3.1.* (Recursive PSI - $\pi^{n,1,m}_{\text{psi}}$)
> PARAMETERS: There are $n > 2$ parties $P_1, \ldots, P_n$ and an adversary $\mathcal{A}$. The protocol uses the functionality $\mathcal{F}^{n-1,1,m}_{\text{psi}}$ and an OKVS scheme (Encode, Decode).
>
> PROTOCOL:
>
> (1) Party P_1 chooses a PRF key k and sends it to P_2.
> (2) P_1 runs[a] $S \leftarrow \text{Encode}(\{(a^1_j, F(k, a^1_j))\}_{a^1_j \in A^1})$, and sends S to $P_3, \ldots, P_n$.
> (3) Party P_2 computes $\tilde{A}^2 = \{a^2_j || F(k, a^2_j)\}_{a^2_j \in A^2}$.
> (4) Party $P_i \in \{P_3, \ldots, P_n\}$ computes $\tilde{A}^i = \{a^i_j || \text{Decode}(S, a^i_j)\}_{a^i_j \in A^i}$.
> (5) Parties $P_2, \ldots, P_n$ invoke $\mathcal{F}^{n-1,1,m}_{\text{psi}}$ where $\tilde{A}^i$ is P_i's input set.
> (6) Party P_n obtains the intersection $X = \{x || \tilde{x}\}_{x || \tilde{x} \in \cap_i \tilde{A}^i}$ from $\mathcal{F}^{n-1,1,m}_{\text{psi}}$ and outputs $\{x \mid x || \tilde{x} \in X\}$.
>
> [a] In case that $\mathcal{A}$ is malicious, party P_i uses $H(a^i_j)$ instead of a^i_j in steps (2)-(4) above, where H is a random oracle.

3.1.2 Discussion: Insecurity in the Face of Collusion. We demonstrate the reason the above protocol is insecure when the adversary corrupts two or more parties. If P_2 colludes with P_i, P_2 could send the PRF key k to P_i. Now, P_i can call $\text{Decode}(S, x)$ on any x and receive either $F(k, x)$ or some random value, depending on whether $x \in A^1$ or not. If the inputs are known to be from a relatively small domain (e.g phone numbers), P_i can perform a check on every input in the domain and expose all P_1's input items.

Note that the attack above is possible since 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 $\text{Decode}(S, \cdot)$ arbitrarily many times). In order to weaken the threshold assumption (i.e. to make the protocol secure even against collusion), one may use an $\mathcal{F}^{F,m_1,m_2}_{\text{opprf}}$ in place of the OKVS. That is, in Step 2 of Protocol 3.1, P_1 runs an OPPRF protocol with each of $P_3, \ldots, P_n$. Now, by the definition of OPPRF, P_i 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 $\mathcal{F}^{n,t,m}_{\text{psi}}$. 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_i, P_j ($2 < i, j$) may learn the intersection of the sets of parties P_1, P_i, P_j by having P_i, P_j agree on the same input set $A^i = A^j$ and compare their OPPRF results. An equal OPPRF results on a query x means that $x \in A^1 \cap A^i \cap A^j$ whereas an unequal results on x means that

$x \notin A^1$. Such an intersection of three parties is not permitted by functionality $\mathcal{F}_{\text{psi}}^{n,t,m}$.

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 $\mathcal{F}_{\text{psi}}^{3,2,m}$. 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 ($\pi_{\text{psi}}^{n,1,m}$) securely computes functionality $\mathcal{F}_{\text{psi}}^{n,1,m}$ in the $\mathcal{F}_{\text{psi}}^{n-1,1,m}$-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 $A^1 = \{a_1^1, \ldots, a_m^1\}$, 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_i ($i = 3, \ldots, n$) who receives S, the simulator concludes with the actual input set $\hat{A}^1 = \{x \in L \mid F(k, H(x)) = \mathsf{Decode}(S, H(x))\}$. 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 $F(k, H(x')) \neq \mathsf{Decode}(S, H(x'))$, and thus x' does not appear in the real execution result intersection. This concludes the simulation because P_1 does not receive any further message in both worlds.

Corrupted P_2. The simulator is given A^2, inputs 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 k 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 $\mathcal{F}_{\text{psi}}^{n-1,1,m}$ 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^i (for $i \neq 2$) is negligible, since there are at most $(n-1)m$ random oracle outputs in the range $\{0,1\}^\kappa$, 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_i ($3 \leq i < n$). The simulator is given A^i, sends it to the ideal world functionality and receives no output. In the real execution P_i receives an OKVS from P_1, so the simulator computes $S \leftarrow \mathsf{Encode}(\{(k_i, v_i)\})$ with m random pairs (k_i, v_i) and sends S to P_i. By the obliviousness property of S, it is not possible to distinguish between S output by the simulator and the OKVS that has A^1 as keys in the real execution.

In the malicious setting the simulator extracts P_i's input set as follows: it runs P_i internally with the random OKVS S as its first message. It observes the set of P_i's random oracle queries and records them in the list L. Then, it receives P_i's input set L' to the $\mathcal{F}_{\text{psi}}^{n-1,1,m}$ functionality and concludes with the set $\hat{A}^i = \{x \mid x \in L \wedge \mathsf{Decode}(S, H(x)) \in L'\}$. As before, for a value y in L' that is not in the range of $\mathsf{Decode}(S, \cdot)$ or is $\mathsf{Decode}(S, r)$ 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 $\{x || \mathsf{Decode}(S, x)\}_{x \in X}$ 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 3.3 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. Party 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 $F(k_i, a_j^1)$ as $\bigoplus_{i=2}^{n-2} F(k_i, a_j^1)_{j \in [m]}$ to encode an OKVS S_n, which she then sends to P_n. Party P_n learns an OKVS S_n, so she decodes it on every $a^n \in A^n$, which equals $\bigoplus_{i=2}^{n-2} F(k_i, a^n)$ if a^n 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 $P_i \in \{P_2, \ldots, P_{n-2}\}$, so she can decode 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 $\mathcal{F}_{\text{psi}}^{2,1,m}$ to be the concatenation of the plain item and its PRF evaluation. This is required in order to mitigate a similar

PROTOCOL 3.3. (PSI - $\pi^{n,1,m}_{\text{psi-opt}}$)
PARAMETERS: There are n parties $P_1, \ldots, P_n$ and an adversary $\mathcal{A}$. The protocol uses the functionality $\mathcal{F}^{2,1,m}_{\text{psi}}$, and an OKVS scheme (Encode, Decode).
PROTOCOL[a]:

(1) P_1 chooses $k_i \in \{0,1\}^\kappa$ uniformly and sends k_i to P_i, for $i = 2, \ldots, n-2$.
(2) P_1 computes $S_n \leftarrow \text{Encode}(\{(a^1_j, \bigoplus_{i=2}^{n-2} F(k_i, a^1_j))\}_{j\in[m]})$ and sends S_n to P_n.
(3) P_i ($i \in \{2, \ldots, n-2\}$) computes $S_i \leftarrow \text{Encode}(\{(a^i_j, F(k_i, a^i_j))\}_{j\in[m]})$ and sends S_i to P_{n-1}.
(4) P_{n-1} computes
$$\tilde{A}^{n-1} = \{a^{n-1}_j \mid\mid \bigoplus_{i=2}^{n-2} \text{Decode}(S_i, a^{n-1}_j)\}_{j\in[n]}$$
(5) P_n computes
$$\tilde{A}^n = \{a^n_j \mid\mid \text{Decode}(S_n, a^n_j)\}_{j\in[n]}$$
(6) Parties P_{n-1}, P_n invoke $\mathcal{F}^{2,1,m}_{\text{psi}}$ with inputs $\tilde{A}^{n-1}$ and $\tilde{A}^n$, respectively. P_n obtains $X = \{x||\tilde{x}\}_{x||\tilde{x}\in\tilde{A}^{n-1}\cap\tilde{A}^n}$ and outputs the intersection $\{x \mid x||\tilde{x} \in X\}$.

[a] In case that $\mathcal{A}$ is malicious, party P_i uses $H(a^i_j)$ instead of a^i_j in steps (2)-(5) above, where H is a random oracle.

attack to the one described above: a malicious P_1 might encode $(x, \bigoplus_{i=2}^{n-1} F(k_i, x'))$ in S_n (rememeber, P_1 chooses all keys), where $x' \in A^i$ for all $i \in \{2, \ldots, n-1\}$ and $x \in A^n$, but neither x nor x' are in the intersection. Now, when P_n computes $(\text{Decode}(S_n, x))$ she obtains $\bigoplus_{i=2}^{n-1} F(k_i, x')$. Therefore, P_{n-1}, P_n invoke $\mathcal{F}^{2,1,m}_{\text{psi}}$ with $\bigoplus_{i=2}^{n-1} F(k_i, x')$ as one of the values in their sets, leading P_n to falsely output the value x.

Let us 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 4 and 6, while she does not preform step 3. As a consequence, in step 4, P_2 computes $\tilde{A}^{n-1} = \{a^{n-1}_j \mid\mid F(k_{n-1}, a^{n-1}_j)\}_{a^{n-1}_j\in A^{n-1}}$.

3.2.1 *Discussion.*

Note that even a slight modification to Protocol 3.3 may turn it insecure. For example, suppose P_i, 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 $v \in A^1$, $v \in \cap^i A^i$ or neither. Since OKVS does not imply any limit on the number of queries to it, P_n can preform this test with any v, thus learning more information than what the functionality allows.

In addition, a collusion of even two parties would break the security of Protocol 3.3: If P_1 colludes with P_{n-1}, P_1 may send the PRF keys k_i to P_{n-1}. Now, P_{n-1} may run $\text{Decode}(S_i, x)$ on unlimited number of values x, by that, it receives either $F(k_i, x)$ 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_i's input set. We remark that the protocol remains secure against a collusion of any subset in $\mathbb{P}(\{P_2, \ldots, P_{n-2}\}) \times \{P_1, P_{n-1}, P_n\}$ (where $\mathbb{P}$ 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_i ($i \in \{2, \ldots, n-2\}$) 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_i to P_i ($i \in \{2, \ldots, n-2\}$) in the first round. She also sends S_n to P_n in the same round. P_i ($i \in \{2, \ldots, n-2\}$) sends S_i to P_{n-1} at the second round. Parties P_{n-1} and P_n invoke $\mathcal{F}^{2,1,m}_{\text{psi}}$ in the third and last round. Overall, the round complexity is 2 rounds more than the protocol for two parties. We instantiate $\mathcal{F}^{2,1,m}_{\text{psi}}$ 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 $O(n)$ κ-length keys. P_{n-1} and P_n communication complexity is determined by the exact protocol used to compute the functionality $\mathcal{F}^{2,1,m}_{\text{psi}}$, which is $O(m)$ as it can be instantiated with a server-aided version.

THEOREM 3.4. *Protocol 3.3 ($\pi^{n,1,m}_{\text{psi-opt}}$) securely computes functionality $\mathcal{F}^{n,1,m}_{\text{psi}}$ in the $\mathcal{F}^{2,1,m}_{\text{psi}}$-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 the intersection. P_1 encodes the point $(x, \bigoplus_{i=2}^{n-2} F(k_i, x))$ into S_n, which is sent to P_n. Party P_i for $i \in \{2, \ldots, n-2\}$ encodes $(x, F(k_i, x))$ into S_i, which is sent to P_{n-1}. Party P_{n-1} decodes each S_i with key x, obtains $F(k_i, (x)$ for all $i \in \{2, \ldots, n-2\}$, and sums them up, resulting with $\bigoplus_{i=2}^{n-2} F(k_i, (x)$. This is exactly the value obtained by P_n when decoding S_n on x. Thus, both P_{n-1} and P_n adds that value to their sets $\tilde{A}^{n-1}$ and $\tilde{A}^n$, respectively, so P_n outputs x as part of the intersection.

Case 2: $x \notin A^1$. P_1 sends S_n to P_n without encoding x as a key in S_n. Thus, $y_n = \text{Decode}(S_n, x)$ 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 $x \in A^i$ for all $i \in \{2, \ldots, n\}$, the values $x||y_{n-1}$ and $x||y_n$ input to $\mathcal{F}^{2,1,m}_{\text{psi}}$ would not match, therefore x is not output as part of the intersection.

Case 3: $x \notin A^i$ 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-1} has x, it inputs to $\mathcal{F}^{2,1,m}_{\text{psi}}$ $x||\hat{x}$ where $\hat{x}$ is a pseudorandom value not equal to $\tilde{x} = \bigoplus_{i=2}^{n-2} F(k_i, (x)$ whereas if P_n has x it inputs $x||\tilde{x}$, meaning that x is not part of the intersection.

Case 4: $x \notin A^{n-1}$ or $x \notin A^n$. Parties P_{n-1} and P_n concatenate their plain-text values in the beginning of each value of their sets $\tilde{A}^{n-1}$ and $\tilde{A}^n$ respectively. Thus, they do not obtain a value corresponding to x from $\mathcal{F}^{2,1,m}_{\text{psi}}$, 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 $A^1 = \{a^1_1, \dots, a^1_m\}$, 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 $(i = 2, \dots, n-1)$ who receives k_i, the simulator concludes with the actual input set $\hat{A}^1 = \{x \in L \mid \bigoplus_{i=2}^{n-2} F(k_i, x) = \text{Decode}(S_n, H(x))\}$. The simulator inputs $\hat{A}^1$ to the ideal-world functionality. Note that except with negligible probability, for every x' that was not queried to the random oracle, it follows that $\bigoplus_{i=2}^{n-2} F(k_i, x) \neq \text{Decode}(S_n, H(x'))$, 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 $(2 \le i \le n-2)$. 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_1 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 as 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 $(2 \le i \le n-2)$, so the simulator computes $S_i \leftarrow \mathsf{Encode}(\{(k_i, v_i)\})$ with m random pairs (k_j, v_j) and sends S_i to P_{n-1}, for each $i \in (2, \dots, n-2)$. 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 $\mathcal{F}^{2,1,m}_{\text{psi}}$ functionality and concludes with the set $\hat{A}^{n-1} = \{x \in L \mid x||\tilde{x} \in L'\}$ where $\tilde{x} = \bigoplus_{i=2}^{n-2} \text{Decode}(S_i, H(x))$. 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 $\tilde{X} = \{x||\text{Decode}(S_n, x)\}_{x \in X}$ to P_n and outputs whatever it outputs. By the obliviousness property of the OKVS, S_n and $\tilde{X}$ in both worlds are computationally indistinguishable and expose the same correlation, i.e. for each $x||\tilde{x} \in \tilde{X}$ it follows that $\text{Decode}(S_n, x) = \tilde{x}$.
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 $\mathcal{F}^{2,1,m}_{\text{psi}}$ functionality and concludes with the set $\hat{A}^n = \{x \in L \mid x||\tilde{x} \in L'\}$ where $\tilde{x} = \text{Decode}(S_n, H(x))$. The simulator inputs $\hat{A}^n$ to the ideal world functionality and receives X back. It sends to P_n the set $\tilde{X} = \{x||\text{Decode}(S_n, H(x))\}_{x \in X}$ and outputs whatever it outputs.

□

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_i and the OKVSes S_i for all $i \in [2, n-2]$, which means they can reveal P_i'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 $P_1, \dots, P_{n-2}$, which is not allowed by the functionlity.

We can mitigate the above attacks as follows: First, P_1 picks key k_i 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 $S_i \leftarrow \mathsf{Encode}(\{(a^i_j, \hat{F}(a^i_j))\}_{j \in [n]})$ where $\hat{F}(a^i_j) = F(k'_i, a^i_j) \oplus F(k_i, a^i_j)$, and sends it to P_{n-1}. In addition, P_i sends k'_i to P_{n-2}, who computes $\{a^{n-2}_j || \bigoplus_{i=2}^{n-3} F_{k'_i}(a^{n-2}_j) \oplus F_{k_{n-2}}(a^{n-2}_j)\}_{j \in [m]}$. 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 $a_{n-2} = \bigoplus_{i=2}^{n-3} F_{k'_i}(a)$, party P_{n-1} holds $a_{n-1} = \bigoplus_{i=2}^{n-3} F_{k_i}(a) \oplus F_{k'_i}(a)$ and party P_n holds $a_n = \bigoplus_{i=2}^{n-3} F_{k_i}(a)$. Notice that $a_{n-2} \oplus a_{n-1} \oplus a_n = 0$. For other values a which are not in the intersection, the result of $a_{n-2} \oplus a_{n-1} \oplus a_n$ is pseudorandom. To find out the items for which the sum $a_{n-2} \oplus a_{n-1} \oplus a_n = 0$ 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 $X_i = \{(x^i_j, y^i_j)\}_{j\in[m]}$, 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. ($\mathcal{F}^{F,n,m}_{\text{zeroXOR}}$)
Parameters: n parties.
Behavior: Wait for input $X_i = \{(x^i_j, y^i_j)\}_{j\in[m]}$, from $P_{i\in[n]}$ where $(x^i_j, y^i_j) \subset (\{0,1\}^\kappa, \{0,1\}^\ell)$.
Give P_n the set $\left\{x \mid \forall_{i\in[n]} : (x, y^i) \in X_i \text{ and } \bigoplus_{i\in[n]} y^i = 0\right\}$.

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 $\{x^n_j\}_{j\in[m]}$ as queries, and to obtain the associated responses z^i_j from P_i. Now, z^i_j is equal to $y^i_{j'}$ if $x^n_j = x^i_{j'}$, otherwise, z^i_j is pseudorandom. Consequently, if all parties have the key x^n_j, the XOR of all responses $z^i_j, \forall i \in [n-1]$, are equal to P_n's value y^n_j, by which P_n concludes that x^n_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 associated 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-sharings 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 $\pi^{F,n,m}_{\text{zeroXOR}}$ (Figure 4.2) securely implements $\mathcal{F}^{F,n,m}_{\text{zeroXOR}}$ (Figure 4.1) in the presence of a malicious adversary corrupting $t < n$ parties, in the $\mathcal{F}_{\text{zeroShare}}, \mathcal{F}^{F,m_1,m_2}_{\text{opprf}}$-hybrid model.*

4.2 The Protocol

The construction of our $\pi^{n,t,m}_{\text{psi}}$ 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 $P_1, \ldots, P_{v-1}, P_v, P_{v+1}, \ldots, P_n$ 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_v is a pivot. Each client P_i generates and sends a key k^j_i to every server P_j. In addition, the client P_i generates an OKVS S_i such that each item $a^i_q \in A^i$ is associated with the XOR of the PRF results using all keys, namely, $\bigoplus_{j\in[v+1,n]} F_{k^j_i}(a^i_q)$. Each client P_i sends S_i to the pivot party, who decodes and XOR them according to its own set. That is, for every item $a^v_q \in A^v$, compute $\bigoplus_{i\in[v-1]} \text{Decode}(S, a^v_q)$. A server P_j has all keys k^j_i for $i \in [v-1]$. It uses those keys to obtain $\bigoplus_{i\in[v-1]} F_{k^j_i}(a^j_q)$ for every $a^j_q \in A^j$. 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 XORed to a pseudorandom value. To find which ones are XORed to zero the pivot and servers invoke the ZeroXOR functionality. It holds that the $t+1$ parties $P_v, \ldots, P_n$ (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.

PROTOCOL 4.2. ($\pi^{F,n,m}_{\text{zeroXOR}}$)
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) $P_{i\in[n]}$ invokes $\mathcal{F}_{\text{zeroShare}}$ on x^i_j and obtains its share $s^i_j = S(K_i, x^i_j)$ for every $j \in [m]$.
(2) $P_{i\in\{1,\ldots,n-1\}}$ and P_n jointly invoke $\mathcal{F}^{F,m,m}_{\text{opprf}}$:
   - P_i acts as a sender, programming $\mathcal{P} = \{(x^i_j, s^i_j \oplus y^i_j)\}_{j\in[m]}$
   - P_n acts as a receiver with queries $\{x^n_j\}_{j\in[m]}$.
   - P_n obtains $\{(x^n_j, z^i_j)\}_{j\in[m]}$ 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 $\left\{x^n_j \mid s^n_j + y^n_j = \bigoplus_{i\in[n-1]} z^i_j\right\}$

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 κ, 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^j_i 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 $O(m(n-t))$.

Servers receive only keys from the clients which does 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 $i \in [v, n-1]$). Thus, for server P_i ($i \in [v+1, n-1]$) 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 4.4. (PSI with collusion - $\pi_{\text{psi}}^{n,t,m}$)

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 [a]:

(1) Let $v = n - t$. That is, the parties are $P_1, \ldots, P_v, \ldots, P_n$ 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_v where

$$S_i \leftarrow \text{Encode}(\{(a_q^i, \bigoplus_{j=v+1}^{n} F_{k_i^j}(a_q^i))\}_{q\in[m]})$$

(4) Party P_v received S_i for $i \in [1, v-1]$. It computes the key-values set

$$X^v = \left\{(a_q^v, \bigoplus_{i=1}^{v-1} \text{Decode}(S_i, a_q^v)\right\}_{q\in[m]}$$

(5) Party P_i for $i \in [v+1, n]$ received keys $\{k_j^i\}$ for $j \in [1, v-1]$. It computes the key-values set

$$X^i = \left\{(a_q^i, \bigoplus_{j=1}^{v-1} F_{k_j^i}(a_q^i)\right\}_{q\in[m]}$$

(6) Parties $P_v, \ldots, P_n$ invoke functionality $\mathcal{F}_{\text{zeroXOR}}^{F,t+1,m}$ with their corresponding sets $X_v, \ldots, X_n$, by which P_n obtains the intersection.

[a] 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.

protocol in addition to the round complexity of the OPPRF. Overall, there are 4 rounds.

THEOREM 4.5. *Protocol 4.4 securely computes functionality $\mathcal{F}_{\text{psi}}^{n,t,m}$ in the $\mathcal{F}_{\text{zeroXOR}}^{F,n,m}$-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 ($i \in [v-1]$) essentially produces a *conditional zero sharing* for each item $x \in A^i$. That is, it provide the pivot with an OKVS S and the servers with keys k_i^j ($j \in [v+1, n]$) such that if they query these object on $x \in A^i$ they obtain the shares $s_v, s_{v+1}, \ldots, s_n$ such that $\oplus_{j=v}^n s_j = 0$. Otherwise, if even one of $P_v, \ldots, P_n$, say P_k, does not query about x, then the probability that it holds s_k such that $\oplus_{j=v}^n s_j = 0$ 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_v (for $P_i, i \in [1, v-1]$) or key-value set X_i to $\mathcal{F}_{\text{zeroXOR}}^{F,t+1,m}$ (for $P_i, i \in [v, n]$), the simulator can conclude with the actual input set A^i, similarly to the proof of Theorem 3.4. We denote the set or parties corrupted by $\mathcal{A}$ as C. The case where $C \subseteq \{P_1, \ldots, P_{v-1}\}$ is trivial, as none of these parties receive any information which depends on any input set A^j. For the case where $P_v \in C$, we note that any input a_q^i received through 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_v, even if she receives any $t-1$ PRF keys. Next, assume Party $P_i \in C, i \in [v+1, n]$. 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 has to do is generate random PRF keys and hand it to $\mathcal{A}$. P_n also receives the outputs from $\mathcal{F}_{\text{zeroXOR}}^{F,t+1,m}$, 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 diverse 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 $t = n - 1$, $\pi_{\text{psi}}^{n,t,m}$ protocol 4.4 in fact requires only performing ZeroXOR with n parties, each holding $X^i = \{(a_q^i, 0^\kappa)\}_{q\in m}$. That is, when $t = n - 1$ we have no clients, the pivot party is P_1 and $P_2, \ldots, P_n$ 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.

m	2^{12}	2^{16}	2^{20}
Encode	0.052	0.103	2.838
Decode	0.003	0.005	0.99

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

Our implementation uses the table-based OPPRF[2] code from [23], OPRF 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 $\pi_{\text{psi}}^{n,t,m}$ protocol can be optimized by only performing ZeroXOR with n parties, each holding $X^i = \{(a_q^i, 0^\kappa)\}_{q\in m}$. 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

[2] Note that the table-based OPPRF which is secure against a semi-honest adversary is about 3× slower than the state-of-the-art malicious PaXoS-based OPPRF.

Sett.	Protocols	(n,t)	(4,1)			(4,3)			(10,1)			(10,4)			(10,9)			(15, 1)			(15, 4)			(15, 7)			(15, 14)		
		m	2^{12}	2^{16}	2^{20}	2^{12}	2^{16}	2^{20}	2^{12}	2^{16}	2^{20}	2^{12}	2^{16}	2^{20}	2^{12}	2^{16}	2^{20}	2^{12}	2^{16}	2^{20}	2^{12}	2^{16}	2^{20}	2^{12}	2^{16}	2^{20}	2^{12}	2^{16}	2^{20}
LAN	KMPRT[23](aug)	Client	0.21	1.05	15.57	**0.21**	**1.05**	**15.57**	0.277	1.36	18.77	0.277	1.36	18.77	**0.277**	**1.36**	**18.77**	0.34	1.76	25.18	0.34	1.76	25.18	0.34	1.76	25.18	**0.34**	**1.76**	**25.18**
		Total	0.24	1.29	19.19	**0.24**	**1.29**	19.19	**0.365**	2.59	38.04	0.365	2.59	38.04	**0.365**	**2.59**	**38.04**	0.47	3.78	58.23	0.47	3.78	58.23	0.47	3.78	58.23	**0.47**	**3.78**	**58.23**
	KMPRT[23]		0.24	1.29	19.19	0.34	3.16	52.25	0.365	2.97	46.08	0.67	6.77	98.04	1.01	2.97	46.08	0.46	4.28	64.28	0.81	8.01	154.2	1.37	13.47	201.12	1.85	20.61	304.36
	ENOC[10]		1.31	14.24	56.87	1.31	14.24	56.87	2.41	23.24	-	2.41	23.24	-	2.41	23.24	-	3.57	32.82	-	3.57	32.82	-	3.57	32.82	-	3.57	32.82	-
	CDGOSS [2]		0.23	1.6	23.8		-	-	-	-	-	0.31	2.48	31.45	-	-	-	-	-	-	-	-	-	0.44	3.27	39.45	-	-	-
	Ours	Client	**0.06**	**0.15**	**3.34**	**0.21**	**1.05**	**15.57**	**0.06**	**0.16**	**3.35**	**0.06**	**0.17**	**3.39**	**0.28**	**1.36**	**18.77**	**0.06**	**0.13**	**3.95**	**0.06**	**0.11**	**3.52**	**0.06**	**0.16**	**3.48**	**0.34**	**1.76**	**25.18**
		Total	**0.07**	**0.25**	**5.16**	**0.24**	**1.29**	**19.19**	**0.08**	**0.35**	**6.07**	**0.24**	**1.42**	**22**	**0.37**	**2.59**	**38.04**	**0.1**	**0.35**	**7.25**	**0.26**	**1.46**	**22.8**	**0.31**	**2.12**	**32.56**	**0.47**	**3.78**	**58.23**
WAN	KMPRT[23](aug)	Client	1.29	2.67	21.59	**1.29**	**2.67**	**21.59**	2.46	5.08	41.07	2.46	5.08	41.07	**2.46**	**5.08**	**41.07**	3.33	6.95	61.37	3.33	6.95	61.37	3.33	6.95	61.37	**3.33**	**6.95**	**61.37**
		Total	1.75	7.31	95.19	**1.75**	**7.31**	**95.19**	3.04	13.88	219.81	3.04	13.88	219.81	**3.04**	**13.88**	**219.81**	3.35	20.76	336.84	3.35	20.76	336.84	3.35	20.76	336.84	**3.35**	**20.76**	**336.84**
	KMPRT[23]		1.75	7.31	95.19	3.18	17.47	233.1	3.3	26.42	400.9	4.2	37.6	615.4	7.81	112.8	1915	3.63	39.11	664.08	6.24	110.52	1824.3	9.87	150.85	2641	16.42	263.2	-
	ENOC[10]		-	-	-		-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	CDGOSS [2]		1.9	7	69.6		-	-	-	-	-	3	10.4	153.9	-	-	-	-	-	-	-	-	-	3.3	15.4	244.8	-	-	-
	Ours	Client	**0.06**	**0.83**	**8.52**	**1.29**	**2.67**	**21.59**	**0.05**	**0.75**	**7.87**	**0.06**	**0.59**	**7.89**	**2.46**	**5.08**	**41.07**	**0.05**	**0.92**	**8.39**	**0.06**	**0.58**	**14.57**	**0.06**	**0.59**	**15.4**	**3.33**	**6.95**	**61.37**
		Total	**0.36**	**1.83**	**18.48**	**1.75**	**7.31**	**95.19**	**0.37**	**3.08**	**20.28**	**1.78**	**8.42**	**124.26**	**3.04**	**13.88**	**219.81**	**0.38**	**5.7**	**24.67**	**1.79**	**8.8**	**116.22**	**1.94**	**11.31**	**180.56**	**3.35**	**20.76**	**336.84**

Table 3: Running time in second of multiparty PSI protocols for n parties with corruption threshold t on sets of size m. KMPRT[23] and CDGOSS[2] in semi-honest setting, our protocol and ENOC[10] are in malicious setting. Cells with – denote trials that are not supported or not reported by the protocol.

Protocols	(n,t)	(4,1)			(4,3)			(10,1)			(10,4)			(10,9)			(15, 1)			(15, 4)			(15, 7)			(15, 14)		
	m	2^{12}	2^{16}	2^{20}	2^{12}	2^{16}	2^{20}	2^{12}	2^{16}	2^{20}	2^{12}	2^{16}	2^{20}	2^{12}	2^{16}	2^{20}	2^{12}	2^{16}	2^{20}	2^{12}	2^{16}	2^{20}	2^{12}	2^{16}	2^{20}	2^{12}	2^{16}	2^{20}
KMPRT[23](aug)	Client	1.84	31.69	556.39	1.84	31.69	556.39	1.84	31.69	556.39	1.84	31.69	556.39	1.84	31.69	556.39	1.85	31.69	556.39	1.85	31.69	556.39	1.85	31.69	556.39	1.85	31.69	556.39
	Total	7.36	126.76	2225.56	7.36	126.76	2225.56	18.4	316.9	5563.9	18.4	316.9	5563.9	18.4	316.9	5563.9	27.75	475.35	8345.85	27.75	475.35	8345.85	27.75	475.35	8345.85	27.75	475.35	8345.85
KMPRT[23]	Client	1.84	31.69	556.39	4.92	77.8	1402	1.84	31.69	556.39	8.3	131.7	2373.5	14.76	233.41	4208	1.85	31.69	556.39	8.3	131.7	2373.5	13.1	207.5	3741	22.96	363.09	6547
	Total	7.36	126.76	2225.56	19.68	311.2	5608	18.4	316.9	5563.9	44.7	706.2	12730.4	147.6	2334.1	42080	27.75	475.35	8345.85	62.25	987.75	17801.25	103.4	1635.4	29487.9	344.4	5446.35	98205
ENOC[10]	Client	44.76	655.88	10204	44.76	655.88	10204	44.76	655.88	10204	44.76	655.88	10204	44.76	655.88	10204	44.76	655.88	10204	44.76	655.88	10204	44.76	655.88	10204	44.76	655.88	10204
	Total	258.91	2279.16	31080.96	258.91	2279.16	31080.96	714.44	6370.22	93654.3	714.44	6370.22	93654.3	714.44	6370.22	93654.3	1094.04	10995.6	147396.98	1094.04	10995.6	147396.98	1094.04	10995.6	147396.98	1094.04	10995.6	147396.98
CDGOSS [2]	Client	1.3	19.9	318		-	-	-	-	-	2	30.8	492.1	-	-	-	-	-	-	-	-	-	2.4	38.8	620.1	-	-	-
	Total	3.2	49.4	790.2		-	-	-	-	-	12.3	192.4	3077.2	-	-	-	-	-	-	-	-	-	22.5	353.4	5652.9	-	-	-
Ours	Client	**0.16**	**2.62**	**41.94**	**1.84**	**31.69**	**556.39**	**0.16**	**2.62**	**41.94**	**0.16**	**2.62**	**41.94**	**1.84**	**31.69**	**556.39**	**0.16**	**2.62**	**41.94**	**0.16**	**2.62**	**41.94**	**0.16**	**2.62**	**41.94**	**1.85**	**31.69**	**556.39**
	Total	**0.79**	**12.58**	**201.33**	**7.36**	**126.76**	**2225.56**	**1.77**	**28.31**	**452.98**	**3.76**	**62.67**	**1054.6**	**18.4**	**316.9**	**5563.9**	**2.59**	**41.42**	**662.7**	**4.54**	**75.17**	**1254.62**	**5.23**	**85.37**	**1416.9**	**27.75**	**475.35**	**8345.85**

Table 4: Communication (in MB) of multiparty PSI protocols for n parties with corruption threshold t on set of size m. KMPRT[23] and CDGOSS[2] in semi-honest setting, our protocol and ENOC[10] are in malicious setting. Cells with – denote trials that are not supported or not reported by the paper.

Sett.	n	3	4		5			8			16			32		
	t/m	1	1	2	1	2	3	1	3	4	1	3	4	1	3	16
LAN	2^{12}	0.06	0.07	0.21	0.06	0.26	0.28	0.07	0.29	0.29	0.08	0.3	0.37	0.1	0.31	0.53
	2^{16}	0.2	0.25	1.25	0.23	1.57	1.45	0.28	1.46	1.56	0.43	1.47	2.51	0.74	1.48	4.57
	2^{20}	4.67	5.16	19.05	5.26	25.78	24.11	5.5	25.04	26.7	7.57	25.4	39.28	10.71	25.52	76.89
WAN	2^{12}	0.25	0.06	1.67	0.36	1.87	1.8	0.36	1.8	1.83	0.37	1.81	2.04	0.4	1.8	2.61
	2^{16}	1.02	1.83	6.27	1.74	6.44	7.16	2.66	7.2	8.66	4.73	7.77	12.92	5.44	9.83	25.12
	2^{20}	10.57	18.48	70	18.78	70.18	93.98	19.36	97.54	119.02	25.21	109.38	210.04	34.05	131.82	422.46

Table 5: Running time in seconds of party P_n (the one with most workload) of our protocols.

(4, 1), (10, 4), (15, 7) were reported explicitly in the experimental analysis of [2]. The settings with $t > \frac{n}{2}$ 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] ∼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 runtimes are 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 $\pi_{\text{psi}}^{n,t,m}$ protocol consists of three types of parties: client $P_{i\in[1,v-1]}$, pivot P_v, and server $P_{i\in[v+1,n]}$. Since clients $P_{i\in[1,v-1]}$ do not involve into the entire PSI computation process, we report their running time separately. In contract, all parties in protocols [2, 10, 23] require to participate in the mostly full computation process. In terms of communication cost, all protocols are asymmetric with respect to the server(s) and other parties (e.g. clients). Thus, similar to [2], we separate the client and the total communication costs, where by 'total' we refer to the communication of sent/received data of all parties.

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 $(n, t) = (15, 7)$ for set size $m = 2^{20}$ in the LAN setting. This is a $10 - 23\times$ and $1.6 - 82\times$ improvement in running time compared to [2, 23] in the semi-honest setting and [10] in malicious setting. For the total running time, our protocol shows 1.2–6.5× and 1.2–8.5× 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 $7 - 15\times$, $2.5 - 90\times$, and $18 - 270\times$ 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_v. 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 $3 - 4\times$, $2.6 - 20\times$, and $16 - 330\times$ improvement compared to previous work [2, 10, 23], respectively.

[3]https://github.com/ArielCyber/Malicious-MPSI

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 $t \in \{1, 3, \lceil \frac{n}{2} \rceil\}$ 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.

REFERENCES

[1] Alex Berke, Michiel Bakker, Praneeth Vepakomma, Kent Larson, and Alex 'Sandy' Pentland. Assessing disease exposure risk with location data: A proposal for cryptographic preservation of privacy, 2020.
[2] Nishanth Chandran, Nishka Dasgupta, Divya Gupta, Sai Lakshmi Bhavana Obbattu, Sruthi Sekar, and Akash Shah. Efficient linear multiparty psi and extensions to circuit/quorum psi. Cryptology ePrint Archive, Report 2021/172, 2021. https://eprint.iacr.org/2021/172.
[3] Melissa Chase and Peihan Miao. Private set intersection in the internet setting from lightweight oblivious PRF. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part III*, volume 12172 of *LNCS*, pages 34–63. Springer, Heidelberg, August 2020.
[4] Jung Hee Cheon, Stanislaw Jarecki, and Jae Hong Seo. Multi-party privacy-preserving set intersection with quasi-linear complexity. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, 95-A(8):1366–1378, 2012.
[5] Dana Dachman-Soled, Tal Malkin, Mariana Raykova, and Moti Yung. Secure efficient multiparty computing of multivariate polynomials and applications. In *ACNS*, pages 130–146, 2011.
[6] Daniel Demmler, Peter Rindal, Mike Rosulek, and Ni Trieu. PIR-PSI: scaling private contact discovery. *Proc. Priv. Enhancing Technol.*, 2018(4):159–178, 2018.
[7] Samuel Dittmer, Yuval Ishai, Steve Lu, Rafail Ostrovsky, Mohamed Elsabagh, Nikolaos Kiourtis, Brian Schulte, and Angelos Stavrou. Function secret sharing for psi-ca: With applications to private contact tracing. Cryptology ePrint Archive, Report 2020/1599, 2020. https://eprint.iacr.org/2020/1599.
[8] Changyu Dong, Liqun Chen, and Zikai Wen. When private set intersection meets big data: an efficient and scalable protocol. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *CCS*, pages 789–800. ACM, 2013.
[9] Thai Duong, Duong Hieu Phan, and Ni Trieu. Catalic: Delegated PSI cardinality with applications to contact tracing. In Shiho Moriai and Huaxiong Wang, editors, *ASIACRYPT 2020, Part III*, volume 12493 of *LNCS*, pages 870–899. Springer, Heidelberg, December 2020.
[10] Aner Ben Efraim, Olga Nissenbaum, Eran Omri, and Anat Paskin-Cherniavsky. Psimple: Practical multiparty maliciously-secure private set intersection. Cryptology ePrint Archive, Report 2021/122, 2021. https://eprint.iacr.org/2021/122.
[11] Michael J. Freedman, Yuval Ishai, Benny Pinkas, and Omer Reingold. Keyword search and oblivious pseudorandom functions. In Joe Kilian, editor, *TCC*, 2005.
[12] Michael J. Freedman, Kobbi Nissim, and Benny Pinkas. Efficient private matching and set intersection. In *EUROCRYPT*, pages 1–19, 2004.
[13] Gayathri Garimella, Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. Oblivious key-value stores and amplification for private set intersection. Cryptology ePrint Archive, Report 2021/883, 2021. https://ia.cr/2021/883.
[14] Satrajit Ghosh and Tobias Nilges. An algebraic approach to maliciously secure private set intersection. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT*, volume 11478, pages 154–185. Springer, 2019.
[15] Google. https://blog.google/technology/safety-security/password-checkup. 2019.
[16] Christoph Hagen, Christian Weinert, Christoph Sendner, Alexandra Dmitrienko, and Thomas Schneider. All the numbers are US: large-scale abuse of contact discovery in mobile messengers. *NDSS*, 2021.
[17] Carmit Hazay and Muthuramakrishnan Venkitasubramaniam. Scalable multi-party private set-intersection. In *Public-Key Cryptography - PKC 2017 - 20th IACR International Conference on Practice and Theory in Public-Key Cryptography, Amsterdam, The Netherlands, March 28-31, 2017, Proceedings, Part I*, pages 175–203, 2017.
[18] Roi Inbar, Eran Omri, and Benny Pinkas. Efficient scalable multiparty private set-intersection via garbled bloom filters. In Dario Catalano and Roberto De Prisco, editors, *SCN*, volume 11035, pages 235–252. Springer, 2018.
[19] 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 *IEEE European Symposium on Security and Privacy, EuroS&P 2020, Genoa, Italy, September 7-11, 2020*, pages 370–389. IEEE, 2020.
[20] Seny Kamara, Payman Mohassel, Mariana Raykova, and Seyed Saeed Sadeghian. Scaling private set intersection to billion-element sets. In *Financial Cryptography and Data Security - 18th International Conference, FC 2014, Christ Church, Barbados, March 3-7, 2014, Revised Selected Papers*, pages 195–215, 2014.
[21] Lea Kissner and Dawn Xiaodong Song. Privacy-preserving set operations. In *Advances in Cryptology - CRYPTO 2005: 25th Annual International Cryptology Conference, Santa Barbara, California, USA, August 14-18, 2005, Proceedings*, pages 241–257, 2005.
[22] Vladimir Kolesnikov, Ranjit Kumaresan, Mike Rosulek, and Ni Trieu. Efficient batched oblivious PRF with applications to private set intersection. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 818–829. ACM Press, October 2016.
[23] Vladimir Kolesnikov, Naor Matania, Benny Pinkas, Mike Rosulek, and Ni Trieu. Practical multi-party private set intersection from symmetric-key techniques. In *CCS*, 2017.
[24] Phi Hung Le, Samuel Ranellucci, and S. Dov Gordon. Two-party private set intersection with an untrusted third party. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 2403–2420, 2019.
[25] Atsuko Miyaji, Kazuhisa Nakasho, and Shohei Nishida. Privacy-preserving integration of medical data - A practical multiparty private set intersection. *J. Medical Syst.*, 41(3):37:1–37:10, 2017.
[26] Atsuko Miyaji and Shohei Nishida. A scalable multiparty private set intersection. In *NSS*, pages 376–385, 2015.
[27] Payman Mohassel, Mike Rosulek, and Ye Zhang. Fast and secure three-party computation: The garbled circuit approach. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *CCS*, pages 591–602, 2015.
[28] Duong Tung Nguyen and Ni Trieu. Mpccache: Privacy-preserving multi-party cooperative cache sharing at the edge. Cryptology ePrint Archive, Report 2021/317, 2021. https://eprint.iacr.org/2021/317.
[29] Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. PSI from PaXoS: Fast, malicious private set intersection. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part II*, volume 12106 of *LNCS*, pages 739–767. Springer, Heidelberg, May 2020.
[30] Benny Pinkas, Thomas Schneider, Oleksandr Tkachenko, and Avishay Yanai. Efficient circuit-based PSI with linear communication. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT*, volume 11478, pages 122–153. Springer, 2019.
[31] Peter Rindal and Mike Rosulek. Improved private set intersection against malicious adversaries. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT*, volume 10210, pages 235–259, 2017.
[32] Peter Rindal and Phillipp Schoppmann. Vole-psi: Fast oprf and circuit-psi from vector-ole. Eurocrypt, 2021. https://eprint.iacr.org/2021/266.
[33] Yingpeng Sang and Hong Shen. Privacy preserving set intersection protocol secure against malicious behaviors. In *PDCAT*, pages 461–468, 2007.
[34] Yingpeng Sang and Hong Shen. Privacy preserving set intersection based on bilinear groups. In *ACSC*, pages 47–54, 2008.
[35] Ni Trieu, Kareem Shehata, Prateek Saxena, Reza Shokri, and Dawn Song. Epione: Lightweight contact tracing with strong privacy. *CoRR*, abs/2004.13293, 2020.
[36] En Zhang, Feng-Hao Liu, Qiqi Lai, Ganggang Jin, and Yu Li. Efficient multi-party private set intersection against malicious adversaries. In *Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop, CCSW@CCS 2019, London, UK, November 11, 2019*, pages 93–104, 2019.

A SERVER-AIDED TWO-PARTY PSI

We present below the protocol by Kamara et al. [20] that computes functionality $\mathcal{F}_{\mathrm{psi}}^{2,1,m}$ (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

PROTOCOL A.1. ($\pi_{\text{psi}}^{2,1,m}$)

PARAMETERS: There are 2 parties P_1, P_2 and a server S. P_1 and P_2 have sets A^1 and A^2 as input, respectively. Let F be a PRF.

PROTOCOL:

(1) P_1 samples a random PRF key k, and sends it to P_2.
(2) Party $P_i (i \in (1, 2))$ sends $\tilde{A}^i = \{F(k, a^i_j)\}_{j\in[m]}$ to S after shuffling the set.
(3) S computes $X = \tilde{A}^1 \cap \tilde{A}^2$ and returns the result to P_1, P_2.
(4) The parties output $\{F^{-1}(k, x)\}_{x\in X}$.

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

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 $A^1 \leftarrow A \cup D_0 \cup D_1$.
(2) P_2 sets $A^2 \leftarrow A^2 \cup D_0 \cup D_2$.
(3) Party $P_i (i \in (1, 2))$ sends a shuffled version of $\tilde{A}^i = \{F(k, x)\}_{x\in A^i}$ to S.
(4) S computes $X = \tilde{A}^1 \cap \tilde{A}^2$ and sends X to P_1, P_2.
(5) P_i aborts if:
  (a) Either $D_0 \not\subset F^{-1}(k, X)$ or $D_i \cap F^{-1}(k, X) \neq \emptyset$
  (b) There exists $x \in A^i$ and $\alpha, \beta \in [\lambda]$ such that $x||\alpha \in F^{-1}(k, X)$ and $x||\beta \notin F^{-1}(k, X)$
(6) The parties output distinct items in $\{F^{-1}(k, x)\}_{x\in X} \setminus D_0$.

augments its set A^i with λ copies of each element. Specifically, party P_i generates the set $\tilde{A}^i = \{a^i_j||1, \ldots, a^i_j||\lambda\}_{j\in[m]}$ (each item is replicated λ times, each time it is concatenated with the next index from $1, \ldots, \lambda$). Then, the parties run the semi-honest protocol above on the sets $\tilde{A}^1$ and $\tilde{A}^2$. Now, to omit a single item x from the intersection, the server has to omit λ pseudorandom items from X, namely, the items $F(k, x||1), \ldots, F(k, x||\lambda)$. 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 λ 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 $X = \tilde{A}^1$ to P_1 (and similarly $X = \tilde{A}^2$ to P_2) by which the parties conclude that the intersection includes all items. This is again easily fixed by agreeing on one dummy item d_1 which is added only to A^1 and another dummy item d_2 which is added only to A^2. This ensures that the intersection does not contain the entire set, hence, returning $X = \tilde{A}^1$ is immediately treated a cheating. This is presented formally in Protocol A.2.

B MALICIOUS OPPRF

There are two parties, sender $\mathcal{S}$ and receiver $\mathcal{R}$. The sender $\mathcal{S}$ has a set of points $\mathcal{P} = \{(a_1, t_1), \ldots, (a_{m_1}, t_{m_1})\}$ and the receiver $\mathcal{R}$ has queries $(q_1, \ldots, q_{m_2})$. The template of an OPPRF construction follows: The parties run an OPRF which outputs a key k to the sender and the PRF results $F(k, q_1), \ldots, F(k, q_m)$ to the receiver. The sender computes the hint as follows. For each a_i compute $\hat{t}_i = F_k(a_i) \oplus t_i$. Then, the sender generates an OKVS by $S \leftarrow \mathsf{Encode}(\{(a_i, \hat{t}_i)\}_{i\in[m]})$ and sends it to the receiver. For each of the receiver's query q_i and OPRF results $F(k, q_1)$, the receiver computes the OPPRF result $y_i = F(k, q_1) \oplus \mathsf{Decode}(S, q_i)$.

Suppose that the receiver queries the OPRF on some $q = a_j$, then the OPPRF result is $y_i = F(k, a_j) \oplus \mathsf{Decode}(S, a_j) = F(k, a_j) \oplus F(k, a_j) \oplus t_j = t_j$ as required. On the other hand, for a query $q \neq a_j$ for all j, the result $y_i = F(k, q) \oplus \mathsf{Decode}(S, q)$ 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 receives (specifically m_2 queries) whereas concretely efficient OPPRFs directly support a single query only.

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 $O(\log m_1)$ 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 $q = \mathsf{Decode}(D, q) = d_{h_1(q)} \oplus d_{h_2(q)} \oplus \ldots \oplus d_{h_k(q)}$.

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 $r_i = a_i + s \wedge C(d_i)$ 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

$$\begin{aligned} y &= \mathsf{Decode}(R, q) = r_{h_1(q)} \oplus r_{h_2(q)} \oplus \ldots \oplus r_{h_k(q)} \\ &= (a_{h_1(q)} \oplus \ldots \oplus a_{h_k(q)}) \oplus s \wedge (C(d_{h_1}(q)) \oplus \ldots \oplus C(d_{h_k}(q))) \\ &= (a_{h_1(q)} \oplus \ldots \oplus a_{h_k(q)}) \oplus s \wedge C(d_{h_1}(q) \oplus \ldots \oplus d_{h_k}(q)) \\ &= (a_{h_1(q)} \oplus \ldots \oplus a_{h_k(q)}) \oplus s \wedge C(q) \end{aligned}$$

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 S sent to the receiver. Upon receiving the OKVS S, the receiver compute the OPPRF result $y' = \mathsf{Decode}(R, q) \oplus \mathsf{Decode}(S, q) = y \oplus t' = t$ as required, where R is the PaXoS structure interpretation of the ROT results and S is the OKVS sent by the sender.

PROTOCOL C.1. (Zero-Sharing - $\pi^{F,n}_{\text{zeroShare}}$)
PARAMETERS: There are n parties $P_1, \ldots, P_n$ and an adversary $\mathcal{A}$. There is a PRF $F : \{0,1\}^\kappa \times \{0,1\}^\ell \to \{0,1\}^\kappa$.

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 $(k_{1,i}, \ldots, k_{i-1,i}, k_{i,i+1}, \ldots, k_{i,n})$.
(2) To obtain its share for value x, party P_i computes

$$S(K_i, x) = \left(\bigoplus_{j<i} F_{k_{j,i}}(x)\right) \oplus \left(\bigoplus_{j>i} F_{k_{i,j}}(x)\right)$$

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:

$$2N_{OT}\kappa + N_{CC}\log_2 N_{OT} + N_{CC}\log_2 N_{OT} + \kappa + N_{BF}\log_2 N_{OT} + N_{BF}\kappa$$

And server's (P_0) communication by:

$$2nN_{OT}\kappa + nN_{CC}\log_2 N_{OT} + nN_{CC}\log_2 N_{OT} + \kappa + nN_{BF}\log_2 N_{OT}$$