Practical Multi-party Private Set Intersection from Symmetric-Key Techniques

Vladimir Kolesnikov
Bell Labs
Murray Hill, New Jersey, USA
vladimir.kolesnikov@nokia-bell-labs.com

Naor Matania
Bar-Ilan University
Tel Aviv, Israel
naorm1991@gmail.com

Benny Pinkas
Bar-Ilan University
Tel Aviv, Israel
benny@pinkas.net

Mike Rosulek
Oregon State University
Corvallis, Oregon, USA
rosulekm@oregonstate.edu

Ni Trieu
Oregon State University
Corvallis, Oregon, USA
trieun@oregonstate.edu

ABSTRACT

We present a new paradigm for multi-party private set intersection (PSI) that allows n parties to compute the intersection of their datasets without revealing any additional information. We explore a variety of instantiations of this paradigm. Our protocols avoid computationally expensive public-key operations and are secure in the presence of any number of semi-honest participants (i.e., without an honest majority).

We demonstrate the practicality of our protocols with an implementation. To the best of our knowledge, this is the first implementation of a multi-party PSI protocol. For 5 parties with data-sets of 2^{20} items each, our protocol requires only 72 seconds. In an optimization achieving a slightly weaker variant of security (augmented semi-honest model), the same task requires only 22 seconds.

The technical core of our protocol is oblivious evaluation of a *programmable* pseudorandom function (OPPRF), which we instantiate in three different ways. We believe our new OPPRF abstraction and constructions may be of independent interest.

CCS CONCEPTS

• **Theory of computation → Cryptographic protocols**; • **Security and privacy** → *Privacy protections*; *Cryptography*; Symmetric cryptography and hash functions;

KEYWORDS

Private Set Intersection; Oblivious PRF; Secure Multiparty Computation

1 INTRODUCTION

In the problem of private set intersection (PSI), several parties each hold a set of items and wish to learn the intersection of these sets and nothing else. Today, two-party PSI is a truly practical primitive, with extremely fast cryptographically secure implementations [26, 38, 40]. Incredibly, these implementations are only a small factor slower than the naïve and insecure method of exchanging hashed values. Among the specific functions of interest in secure multiparty computation (MPC), PSI is probably one of the most strongly motivated by practice. Indeed, already today companies such as Facebook routinely use PSI to share and mine shared information [34, 52]. In 2012, (at least some of) this sharing was performed with insecure naïve hashing, where players send and compare hashes of their set elements. Today, companies are able and willing to tolerate a reasonable performance penalty, with the goal of achieving stronger security [52]. We believe that the ubiquity and the scale of private data sharing, and PSI in particular, will continue to grow as big data becomes bigger and privacy becomes a more recognized issue. We refer reader to [38–40] for additional discussion and motivation of PSI.

In our work, we consider multi-party PSI in the semi-honest model. By *"multi-party"* we refer to cases where more than two parties wish to compute the intersection of their private data sets. This is a natural generalization of the practically very useful two-party PSI, creating opportunities for much richer data sharing than what was possible with two-party PSI. Consider, for example, a scenario where several organizations, e.g., Facebook, an advertiser, and a third-party data provider, wish to combine their data to find a target audience for an ad campaign. As another application, consider a set of enterprises which have private audit logs of connections to their corporate networks, and wish to identify similar activities in all networks.

We note that the multi-party setting in secure computation is notoriously difficult to tackle. Existing protocols in generic MPC, such as garbled circuits, are significantly more complex and costly in the multi-party case compared to the two-party case. Quite surprisingly, each player in our protocols expend effort *similar* to that in the two-party case.

1.1 State of the Art for Two-Party PSI

We focus on the discussion of the state-of-the-art of semi-honest PSI protocols. We note that the earliest PSI protocols, based on Diffie-Hellman assumptions, can be traced back to the 1980s [19, 30, 47], and refer the reader to [39] for an overview of the many different

CCS '17, October 30-November 3, 2017, Dallas, TX, USA
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-4946-8/17/10…$15.00
https://doi.org/10.1145/3133956.3134065

protocol paradigms for PSI. Protocols based on oblivious transfer extension have proven to be the fastest in practice. We note that the OT-based protocols do not have the lowest *communication* cost. In settings where computation is not a factor, but communication is at a premium, the best protocols are in [5, 23, 41]. In the semi-honest version of these protocols, each party sends only $2n$ group elements, where n is the number of items in each set. However, these protocols require a number of exponentiations proportional to the number of items, making their performance slow in practice. Concretely, [38] found Diffie-Hellman-based protocols to be over 200× slower than the OT-based ones.

Current state-of-the-art semi-honest PSI protocols in the two-party setting are [26, 40]. They both use bucketing to reduce the number of comparisons, and rely on oblivious PRF evaluation. Until our work, these ideas were not used in PSI protocols for the multi-party case.

Most work on concretely efficient PSI is in the random oracle model, and with security against semi-honest, rather than malicious, adversaries. Some notable exceptions are [12, 16, 20] in the standard model, and [7, 8, 10, 12, 37, 44, 45] with security against malicious adversaries.

Lastly, we note that there are efficient constructions for generic MPC [2, 25, 27, 29, 32, 43, 49–51], which can be used for implementing any functionality. In particular, these protocols can be used for securely implementing PSI, in either the two-party or multi-party settings. However, circuits for computing PSI are relatively large. A natural circuit for two-party PSI performs $\mathcal{O}(n^2)$ comparisons, whereas more efficient circuits are of size $\mathcal{O}(n \log n)$ [18, 40]. However, as demonstrated in [40], secure evaluation of these circuits is about two orders of magnitude slower than the most efficient PSI protocols.

1.2 State of the Art for Multi-party PSI

A multi-party PSI protocol was first proposed by Freedman, Nissim, and Pinkas [12]. The protocol of [12] is based on oblivious polynomial evaluation (OPE) which is implemented using additively homomorphic encryption, such as Paillier encryption scheme. The basic idea is to represent a dataset as a polynomial whose roots are its elements, and send homomorphic encryptions of the coefficients of this protocol to obliviously evaluate it on the other party's inputs. Relying on the OPE technique, Kissner and Song [24] proposed a multi-party PSI protocol with quadratic computation and communication complexity in both the size of dataset and the number of parties. The computation overhead is reduced to be linear in number of participants in [46], which was based on bilinear groups. Furthermore, an efficient solution with quasi-linear complexity in the size of dataset is proposed in [6]. In both [6, 46], the maximum number of the corrupted parties are assumed to be $n/2$. Very recent work [17] describes new protocols which run over a star network topology, and are secure in the standard model against either semi-honest or malicious adversaries. The basic idea is to designate one party to run a version of the protocol of [12] with all other parties. The main building block in [17] is an additively homomorphic public-key encryption scheme, with threshold decryption, whose key is mutually generated by the parties. The protocol requires computing a linear number of encryptions and decryptions (namely, exponentiations) in the input sets. In contrast, our main building block is based on Oblivious Transfer extensions where the number of exponentiations does not depend on the size of the dataset. [17] does not include implementation, but we expect that our protocols are much faster due to building from symmetric primitives. We describe the performance of representative multi-party PSI protocols in the semi-honest settings in Table 1.

We mention that multi-party PSI was also investigated in the server-aided model, based on the existence of a server which does not collude with clients [1, 31]. Information-theoretic PSI protocols, possible in the multi-party setting, are considered in [3, 28, 36].

1.3 Our Contributions

We design a modular approach for multi-party PSI that is secure against an arbitrary number of colluding semi-honest parties. Our approach can be instantiated in a number of ways providing tradeoffs for security guarantees and computation and communication costs.

We implemented several instantiations of our PSI approach. To our knowledge, this is the first implementation of multi-party PSI. We find that multi-party PSI is practical, for sets with a million items held by around 15 parties, and even for larger instances. The main reason for our protocol's high performance is its reliance on fast symmetric-key primitives. This is in contrast with prior multi-party PSI protocols, which require expensive public-key operations for each item. Our implementation will be made available on GitHub.

Our PSI Approach. The main building block of our protocol, which we believe to be of independent interest, is *oblivious, programmable PRF (OPPRF).* Recall, oblivious PRF (OPRF) is a 2-party protocol in which the sender learns a PRF key k and the receiver learns $F(k, r)$, where F is a PRF and r is the receiver's input. In an OPPRF, the PRF F further allows the sender to "program" the output of F on a limited number of inputs. The receiver learns the PRF output as before, but, importantly, does not learn whether his input was one on which the PRF was programmed by the sender. We propose three OPPRF constructions, with different tradeoffs in communication, computation, and the number of points that can be programmed.

Basic idea. Our PSI protocol consists of two major phases. First, in the **conditional zero-sharing phase**, the parties collectively and securely generate additive sharings of zero, as follows. Each party P_i obtains, for each of its items x_j, a share of zero, denoted s_j^i. It holds that $\sum_{i=1}^{n} s_j^i = 0$. Namely, if all parties have x_j in their sets then the sum of their obtained shares is zero (else, w.h.p., the sum is non-zero). In the second phase, parties perform **conditional reconstruction** of their shares. The idea is for each P_i to program an instance of OPPRF to output its share s_j^i when evaluated on input x_j. Intuitively, if all parties evaluate the corresponding OPPRFs on the same value x_j, then the sum of the OPPRF outputs is zero. This signals that x_j is in the intersection. Otherwise, the shares sum to a random value.

This brief overview ignores many important concerns — in particular, how the parties coordinate shares and items without revealing the identity of the items. We propose several ways to realize each

	Communication		Computation		Corruption	Security
Protocol	Leader	Client	Leader	Client	Threshold	Model
[24]	$\mathcal{O}(tnm\log(\lvert X\rvert))\lambda$		$\mathcal{O}(ntm^2)$		$n-1$	semi-honest
[6]	$\mathcal{O}((n^2m+nm)\lambda)$		$\mathcal{O}(nm+m)$		$\lfloor n/2\rfloor$	semi-honest
[17]	$\mathcal{O}(nm\lambda)$	$\mathcal{O}(m\lambda)$	$\mathcal{O}(mn\log_2(m))$	$\mathcal{O}(m)$	$n-1$	semi-honest
Ours	$\mathcal{O}(nm\lambda)$	$\mathcal{O}(m\lambda)$	$\mathcal{O}(n\kappa)$	$\mathcal{O}(\kappa)$	$n-1$	augmented semi-honest
		$\mathcal{O}(mt\lambda)$		$\mathcal{O}(t\kappa)$		semi-honest

Table 1: Communication (bits) and computation (number of exponentiations) complexities of multi-party PSI protocols in the semi-honest setting, where n is number of parties, t dishonestly colluding, each with set size m; X is the domain of the element; and λ and κ are the statistical and computational security parameters, respectively. In our protocols, the computational complexities are in an offline preprocessing phase.

of the two PSI phases, resulting in a suite of many possible instantiations. We then discuss the strengths and weaknesses of different instantiations.

A more detailed overview of the approach and the two phases is presented in Section 5, prior to the presentation of the full protocol.

2 PRELIMINARIES

2.1 Secure Computation

The security of a secure multi-party protocol is formally defined by comparing the distribution of the outputs of all parties in the execution of the protocol π to an *ideal* model where a trusted third party is given the inputs from the parties, computes f and returns the outputs. The idea is that if it is possible to simulate the view of the adversary in the real execution of the protocol, given only its view in the ideal model (when it only sees its input and output), then the adversary cannot do in the real execution anything that is impossible in the ideal model, and hence the protocol is said to be secure.

We work in the multi-party setting where the corrupt parties collude. This is modeled by considering a single monolithic adversary that obtains the views of all corrupt parties. The protocol is secure if the *joint distribution* of those views can be simulated.

Functionalities. We define a particular secure computation task by formally describing the behavior of the ideal functionality (trusted third party). The ideal functionality for multi-party PSI is given in Figure 1.

Augmented semi-honest model. We present an optimized variant of our protocols that is in a slightly weaker security model. In the **augmented semi-honest** model the adversary is allowed to change the inputs of corrupted parties (but thereafter run the protocol honestly on those inputs).

In the specific case of multi-party PSI, this additional power is relatively harmless. One can think of a multi-party PSI as computing $X_H \cap X_C$, where X_H is the intersection of all honest parties' sets and X_C is the intersection of all corrupt parties' sets. The augmented semi-honest model simply allows an adversary to choose X_C, rather than being bound to whatever X_C was chosen by the environment. Without loss of generality, an augmented semi-honest adversary can simply set all corrupt parties to have the *same* input set X_C.

We note that the augmented semi-honest model is well known [14, 16] and was used in previous work on multi-party PSI and related functionalities [12, 13]. We define and discuss this security notion at length in Appendix A.

2.2 Cuckoo Hashing

We review the basics of Cuckoo hashing [35], specifically the variant of Cuckoo hashing that involves a stash [22]. In basic Cuckoo hashing, there are m bins, a *stash*, and several random hash functions $h_1, \ldots, h_k$ (often $k = 2$), each with range $[m]$. The invariant is that any item x stored in the Cuckoo hash table is stored either in the stash or (preferably) in one of the bins $\{h_1(x), \ldots, h_k(x)\}$. Each non-stash bin holds at most one item. To insert and element x into a Cuckoo hash table, we place it in bin $h_i(x)$, if this bin is empty for any i. Otherwise, choose a random $i \in_R [k]$, place x in bin $h_i(x)$, evict the item currently in $h_i(x)$, and recursively insert the evicted item. After a fixed number of evictions, give up and place the current item in the stash.

3 PROGRAMMABLE OPRF

Our PSI approach builds heavily on the concept of oblivious PRFs (OPRF). We review the concepts here and also introduce our novel *programmable* variant of an OPRF.

3.1 Definitions

Oblivious PRF. An **oblivious PRF (OPRF)** [11] is a 2-party protocol in which the sender learns a PRF key k and the receiver learns $F(k, q_1), \ldots, F(k, q_t)$, where F is a PRF and $(q_1, \ldots, q_t)$ are inputs chosen by the receiver. Note that we are considering a variant of OPRF where the receiver can obtain several PRF outputs on statically chosen inputs. We describe the ideal functionality for an OPRF in Figure 2.

Instantiation and Security Details. While many OPRF protocols exist, we focus on the protocol of Kolesnikov et al. [26]. This protocol has the advantage of being based on oblivious-transfer (OT) extension. As a result, it uses only inexpensive symmetric-key cryptographic operations (apart from a constant number of initial public-key operations for base OTs). The protocol efficiently generates a large number of OPRF instances, which makes it a particularly good fit for our eventual PSI application that uses many OPRF instances. Concretely, the amortized cost of each OPRF instance costs roughly 500 bits in communication and a few symmetric-key operations.

Technically speaking, the protocol of [26] achieves a slightly weaker variant of OPRF than what we have defined in Figure 2. In particular, (1) PRF instances are are generated with *related keys*, and (2) the protocol reveals slightly more than just the PRF output $F(k, q)$. We stress that in the resulting PRF of [26] the construction

PARAMETERS: The number of parties n, and the size of the parties' sets m.

FUNCTIONALITY:

- Wait for an input $X_i = \{x_i^1, \ldots, x_i^m\} \subseteq \{0,1\}^*$ from each party P_i.
- Give output $\bigcap_{i=1}^{n} X_i$ to all parties.

Figure 1: PSI ideal functionality.

PARAMETERS: A PRF F and bound t.

BEHAVIOR: Wait for input $(q_1, \ldots, q_t)$ from the receiver $\mathcal{R}$. Sample a random PRF seed k and give it to the sender $\mathcal{S}$. Give $(F(k, q_1), \ldots, F(k, q_t))$ to the receiver.

Figure 2: The OPRF ideal functionality $\mathcal{F}_{\text{oprf}}^{F,t}$

PARAMETERS: A programmable PRF F, and upper bound n on the number of points to be programmed, and bound t on the number of queries.

BEHAVIOR: Wait for input $\mathcal{P}$ from the sender $\mathcal{S}$ and input $(q_1, \ldots, q_t)$ from the receiver $\mathcal{R}$, where $\mathcal{P} = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ is a set of points. Run $(k, \mathsf{hint}) \leftarrow \mathsf{KeyGen}(\mathcal{P})$ and give (k, hint) to the sender. Give $(\mathsf{hint}, F(k, \mathsf{hint}, q_1), \ldots, F(k, \mathsf{hint}, q_t))$ to the receiver.

Figure 3: The OPPRF ideal functionality $\mathcal{F}_{\text{opprf}}^{F,t,n}$

remains secure even under these restrictions. More formally, let $leak(k, q)$ denote the extra information that the protocol leaks to the receiver. [26] gives a security definition for PRF that captures the fact that outputs of F, under related keys $k_1, \ldots, k_n$, are pseudorandom even given $leak(k_i, q_i)$. Our OPPRF constructions are built on this OPRF, and as a result our constructions would inherit analogous properties as well.

For ease of presentation and reasoning, we work with the cleaner security definitions that capture the main spirit of programmable OPRF. We emphasize that, although cumbersome, it is possible to incorporate all of the [26] relaxations into the definitions. We stress that our eventual application of PSI is secure *in the standard sense* when built from such relaxed OP[P]RF building blocks.

Programmable PRF. We introduce a new notion of a programmable oblivious PRF. Intuitively, the functionality is similar to OPRF, with the additional feature that it allows the sender to program the output of the PRF on a set of points chosen by the sender. Before presenting the definition of this functionality, we discuss a PRF that supports being programmed in this way.

A **programmable PRF** consists of the following algorithms:

- $\mathsf{KeyGen}(1^\kappa, \mathcal{P}) \to (k, \mathsf{hint})$: Given a security parameter and set of points $\mathcal{P} = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ with distinct x_i-values, generates a PRF key k and (public) auxiliary information hint. We often omit the security parameter argument when it is clear from context.
- $F(k, \mathsf{hint}, x) \to y$: Evaluates the PRF on input x, giving output y. We let r denote the length of y.

A programmable PRF satisfies **correctness** if $(x, y) \in \mathcal{P}$, and $(k, \mathsf{hint}) \leftarrow \mathsf{KeyGen}(\mathcal{P})$, then $F(k, \mathsf{hint}, x) = y$. For the security guarantee, we consider the following experiment/game:

```
$\underline{\mathsf{Exp}^{\mathcal{A}}(X, Q, \kappa)}$:
    for each $x_i \in X$, chose random $y_i \leftarrow \{0,1\}^r$
    $(k, \mathsf{hint}) \leftarrow \mathsf{KeyGen}(1^\kappa, \{(x_i, y_i) \mid x_i \in X\})$
    return $\mathcal{A}\Big(\mathsf{hint}, \{F(k, \mathsf{hint}, q) \mid q \in Q\}\Big)$
```

We say that a programmable PRF is (n, t)-**secure** if for all $|X_1| = |X_2| = n$, all $|Q| = t$, and all polynomial-time $\mathcal{A}$:

$$\Big|\Pr[\mathsf{Exp}^{\mathcal{A}}(X_1, Q, \kappa) \Rightarrow 1] - \Pr[\mathsf{Exp}^{\mathcal{A}}(X_2, Q, \kappa) \Rightarrow 1]\Big| \quad \text{is negligible in } \kappa$$

Intuitively, it is hard to tell what the set of programmed points was, given the hint and t 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 reason for including a separate "hint" as part of the syntax is that our protocol constructions will naturally leak this hint to the receiver (in addition to the receiver's PRF output). We propose a definition that explicitly models this leakage and ensures that it is safe.

Oblivious Programmable PRF (OPPRF). The formal definition of an oblivious programmable PRF (OPPRF) functionality is given in Figure 3. 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.

We now give several constructions of an OPPRF, with different tradeoffs in parameters.

3.2 A Construction Based on Polynomials

Our polynomial-based construction is presented in Figure 4. We first describe the underlying programmable PRF. Let F be a PRF and define our new programmable PRF $\widehat{F}$ as follows:

- $\mathsf{KeyGen}(\mathcal{P} = \{(x_1, y_1), \ldots, (x_n, y_n)\})$: Choose a random key k for F. Interpolate a degree $n-1$ polynomial p over the points $\{(x_1, y_1 \oplus F(k, x_1)), \ldots, (x_n, y_n \oplus F(k, x_n))\}$. Let hint be the coefficients of p.
- $\widehat{F}(k, \mathsf{hint}, q) = F(k, q) \oplus p(q)$.

It is not hard to see that $\widehat{F}$ satisfies correctness since for $x_i \in \mathcal{P}$ it holds that $\widehat{F}(k, \mathsf{hint}, x_i) = F(k, x_i) \oplus p(x_i) = F(k, x_i) \oplus y_i \oplus F(k, x_i)$. Security follows from the fact that when the y_i values are distributed uniformly, so is the hint p. This is true regardless of the number of queries the receiver makes.

Finally, the OPPRF protocol for $\widehat{F}$ is straightforward if there is an OPRF protocol for F: the parties simply invoke $\mathcal{F}_{\text{oprf}}^{F,t}$ on their inputs. The sender obtains k and uses it to generate the hint as above, and sends it to the receiver. The receiver, obtaining $F(k, q_i)$ from $\mathcal{F}_{\text{oprf}}^{F,t}$, can compute its output $\widehat{F}(k, \mathsf{hint}, q_i) = F(k, q_i) \oplus p(q_i)$. The description of the OPPRF protocol is given in Figure 4. Simulation is trivial, as the parties' views in the protocol are exactly the OPPRF output.

Input of $\mathcal{S}$: n points $\mathcal{P} = \{(x_1, y_1), \ldots, (x_n, y_n)\}$, where $x_i \in \{0,1\}^*$, $x_i \neq x_j$; and $y_i \in \{0,1\}^r$

Input of $\mathcal{R}$: $Q = (q_1, \ldots, q_t) \in (\{0,1\}^*)^t$.

Protocol:

(1) $\mathcal{R}$ sends Q to $\mathcal{F}_{\text{oprf}}^{F,t}$. The sender receives k and receiver receives $F(k,q)$ for $q \in Q$.
(2) $\mathcal{S}$ interpolates the unique polynomial p of degree $n-1$ over the points $\{(x_1, y_1 \oplus F(k,x_1)), \ldots, (x_n, y_n \oplus F(k,x_n))\}$.
(3) $\mathcal{S}$ sends the coefficients of p to $\mathcal{R}$.
(4) $\mathcal{R}$ outputs $(p, F(k,q_1) \oplus p(q_1), \ldots, F(k,q_t) \oplus p(q_t))$.

Figure 4: Polynomial-based OPPRF protocol

Costs. The main advantage of this construction is that the only message that needs to be sent in addition to the $\mathcal{F}_{\text{oprf}}$ protocol is the polynomial p whose length is exactly that of n values. This seems the minimal communication overhead that is needed to achieve OPPRF from OPRF. On the other hand, the interpolation of the polynomial takes time $\mathcal{O}(n^2)$ which can be expensive for large n.

3.3 A Construction Based on Bloom Filters

Garbled Bloom filters (GBF) were introduced in [9] in the context of PSI protocols. A GBF is an array $GBF[1 \ldots N]$ of strings, associated with a collection of hash functions $h_1, \ldots, h_k : \{0,1\}^* \to [N]$. The GBF implements a key-value store, where the value associated with key x is:

$$\bigoplus_{j=1}^{k} GBF[h_j(x)]. \qquad (\star)$$

A GBF can be programmed to map specific keys to chosen values:

(1) Initialize array GBF with all entries equal to $\perp$
(2) For each key-value pair (x, v), let $J = \{h_j(x) \mid GBF[h_j(x)] = \perp\}$ be the relevant positions of GBF that have not yet been set. Abort if $J = \emptyset$. Otherwise, choose random values for $GBF[J]$ subject to the lookup equation ($\star$) equaling the desired value v.
(3) For any remaining $GBF[j] = \perp$, replace $GBF[j]$ with a randomly chosen value.

It is clear that, unless this procedure aborts, it produces a GBF with the desired key-value mapping. In [9] it was observed that the procedure aborts when processing item x if and only if x is a false positive for a *plain* Bloom filter containing the previous items (think of the plain Bloom filter obtained by interpreting a $\perp$ in GBF as 0 and anything else as 1). The false-positive probability for a plain Bloom filter has been well analyzed. In particular, to bound the probability by $2^{-\lambda}$, one can use a table with $N = n\lambda \log_2 e$ entries to store n items. In that case, the optimal number of hash functions is λ. If we set $\lambda = 40$, we get that the table size is about $60n$ and the number of hash functions is $k = 40$. In addition, by doing less hashing[21], each insert only requires two hash functions $h_1(x)$ and $h_2(x)$. The additional $k-2$ hash functions $h_i(x), i \in [3,k]$, is simulated by $h_i(x) = h_1(x) + i \times h_2(x)$.

Given the GBF construction, an OPPRF construction is relatively straightforward and similar to the polynomial-based construction.

Input of $\mathcal{S}$: n points $\mathcal{P} = \{(x_1, y_1), \ldots, (x_n, y_n)\}$, where $x_i \in \{0,1\}^*$, $x_i \neq x_j$ and $y_i \in \{0,1\}^r$

Input of $\mathcal{R}$: $Q = (q_1, \ldots, q_t) \in (\{0,1\}^*)^t$.

Protocol:

(1) $\mathcal{R}$ sends Q to $\mathcal{F}_{\text{oprf}}^{F,t}$. The sender receives k and receiver receives $F(k,q)$ for $q \in Q$.
(2) $\mathcal{S}$ inserts the n pairs
$$\{(x_1, y_1 \oplus F(k,x_1)), \ldots, (x_n, y_n \oplus F(k,x_n))\}$$
into a garbled Bloom filter denoted as G, with entries which are each r bits long. It fills the remaining empty entries with random values.
(3) $\mathcal{S}$ sends G to $\mathcal{R}$ as well as the k hash functions (the functions need not be sent explicitly, and can be defined by setting some context dependent prefixes to inputs of a known hash function).
(4) For $i = 1$ to t, $\mathcal{R}$ computes $z_i = F(k,q_i) \oplus \bigoplus_{j=1}^{k} G[h_j(q_i)]$. Finally $\mathcal{R}$ outputs $(G, z_1, \ldots, z_t)$.

Figure 5: Bloom-filter-based OPPRF protocol

Instead of the mappings $x_i \mapsto y_i \oplus F(k,x_i)$ being stored in a polynomial, they are stored in a GBF. The construction is defined in Figure 5. Security holds naturally, since if the y_i points are chosen randomly, all positions in the GBF are uniformly distributed.

Costs. The advantage of the Bloom filter based construction, compared to the polynomial-based construction, is that the insertion algorithm runs in time $\mathcal{O}(n)$ rather $\mathcal{O}(n^2)$, and is also very efficient in practice. The communication is still $\mathcal{O}(n)$ but the constant coefficient is high (the actual communication is $60n$ items rather than n) and therefore communication might be a bottleneck, especially on slow networks.

3.4 Table-Based Construction

The previous OPPRF constructions can be instantiated with any underlying OPRF that allows the receiver to evaluate the PRF on any number t of points. The resulting OPPRF constructions will inherit the same t. Meanwhile, our most efficient OPRF building block from [26] only supports $t = 1$. In this section we describe a construction tailored for the case of $t = 1$, and for small values of n (the number of programmed points).

The main idea behind this construction is as follows. For each pair (x_i, y_i) the sender $\mathcal{S}$ uses $F(k,x_i)$ as an encryption key to encrypt the corresponding value y_i. Let T be the collection of these encryptions; then T comprises the OPPRF hint. At a high level, the receiver can obtain $F(k,q)$ and use it as a key to decrypt the appropriate ciphertext from T.

The main challenges are: (1) $\mathcal{R}$ should not know whether he is getting random or programmed output values (i.e. whether $x = x_i$ for some i), and (2) $\mathcal{R}$ must learn which ciphertext from T to decrypt. We achieve both properties by using $F(k,q)$ to derive a *pointer* into the table T. In order to achieve property (1), $\mathcal{R}$ must *always* decrypt some ciphertext of T, even if $x \neq x_i$.

Concretely, suppose n is 20, so that $\mathcal{S}$ needs to program only 20 points. $\mathcal{S}$ will make a table T of size $2^5 = 32$ (next power of 2

INPUT OF $\mathcal{S}$: n points $\mathcal{P} = \{(x_1, y_1), \dots, (x_n, y_n)\}$, where $x_i \in \{0,1\}^*$, $x_i \neq x_j$; and $y_i \in \{0,1\}^r$

INPUT OF $\mathcal{R}$: $q \in \{0,1\}^*$.

PARAMETERS: random oracle $H : \{0,1\}^* \to \{0,1\}^m$, where $m = 2^{\lceil \log(n+1) \rceil}$.

PROTOCOL:

(1) $\mathcal{R}$ sends q to $\mathcal{F}^{F,t}_{\text{oprf}}$. The sender receives k and receiver receives $F(k, q)$.
(2) $\mathcal{S}$ samples $v \leftarrow \{0,1\}^\kappa$ until $\{H(F(k, x_i) \| v) \mid i \in [n]\}$ are all distinct.
(3) For $i \in [n]$, $\mathcal{S}$ computes $h_i = H(F(k, x_i) \| v)$, and sets $T_{h_i} = F(k, x_i) \oplus y_i$.
(4) For $j \in \{0,1\}^m \setminus \{h_i \mid i \in [n]\}$, $\mathcal{S}$ sets $T_j \leftarrow \{0,1\}^r$.
(5) $\mathcal{S}$ sends T and v to $\mathcal{R}$.
(6) $\mathcal{R}$ computes $h = H(F(k, q) \| v)$, and outputs $(T, v, T_h \oplus F(k, q))$.

Figure 6: Basic table-based OPPRF protocol.

greater than 20). $\mathcal{S}$ will choose a random nonce $v \in \{0,1\}^\kappa$ until $\{H(F(k, x_i) \| v) \mid i \leq 20\}$ are all distinct, where $H : \{0,1\}^* \mapsto \{0,1\}^5$ is a hash function modeled as a random oracle. For each $i \in [n]$, $\mathcal{S}$ computes $h_i = H(F(k, x_i) \| v)$, and sets $T_{h_i} = F(k, x_i) \oplus y_i$. The remaining entries of T ($32 - 20 = 12$ of them in this case) are chosen uniformly. $\mathcal{S}$ sends this nonce v together with the table T to the the receiver as part of the hint.

From the receiver's point of view, on input x he will use $F(k, q)$ to decrypt the ciphertext in position $H(F(k, q) \| v)$ of the table. The distinctness of the $H(F(k, x_i) \| v)$ values allows the sender to place encryptions of the y_i values at appropriate positions in T without any conflicts. The details are given in Figure 6. Note that the OPPRF protocol is restricted to the case of $t = 1$. Because of that, it suffices to use one-time pad encryption for the table entries.

Security & parameters. The underlying programmable PRF satisfies security based on two observations: The easy observation is that table T itself is uniformly distributed when the y_i values are uniformly distributed (as in the security definition for programmable PRF).

Next, we must argue that the nonce v leaks no information about the set of programmed points. Fix a candidate v and define $z_i = H(F(k, x_i) \| v)$. The sender tests this candidate v by seeing whether there is a collision among $\{z_i\}$ values. The receiver sees at most one value of the form $F(k, x_i)$. So by the PRF security of F, at least $n - 1$ of the other F outputs are distributed randomly from the receiver's perspective. Since H is a random oracle, it follows that at least $n - 1$ of the z_i values are distributed independent of the receiver's view (even when the receiver has oracle access to H). Finally, the condition of a collision among randomly chosen $\{z_i\}$ values is independent of any *single* z_i. Hence, the probability of a candidate v being chosen (and thus the overall distribution of v) is the same regardless of whether the receiver queried F on one of the sender's programming points.

It is important to discuss the parameter choice m (length of H output), as it greatly affects performance (the number of retries in step 2 of the protocol). We can calculate the probability that for a random v, the $\{H(s_i \| v) \mid i \in [n]\}$ values are distinct:

$$\Pr_{\text{unique}} = \prod_{i=1}^{n-1} \left(1 - \frac{i}{2^m}\right) \tag{1}$$

The expected number of restarts when sampling v is $1/\Pr_{\text{unique}}$.

Looking ahead to our PSI protocol, the OPPRF will be programmed with n items, where n is the number of items hashed into a particular bin. Different bins will have a different number of items. We must set m in terms of the *worst case* number of items per bin, so that no bin exceeds 2^m items with high probability. However, *on average,* a bin will have very few items.

Concretely, for PSI of 2^{20} items we choose hashing parameters so that no bin exceeds 30 items with high probability. Hence we set $m = 5$ (so T has 32 entries). Yet, the *expected* number of items in a bin is roughly 3. For the vast majority of bins, the sender programs the OPPRF on at most 7 points. In such a bin, only 2 trials are expected before finding a suitable v.

Costs. This OPPRF construction has favorable communication and computational cost. It requires communicating a single nonce v along with a table whose length is that of $\mathcal{O}(n)$ items. The constant in the big-O is at most 2 (the number of items is rounded up to the nearest power of 2). The computational cost of the protocol is to evaluate a random oracle H, $n\tau$ times, where τ is the number of restarts in choosing v. While these computational costs can be large in the worst case, the typical value of τ in our PSI protocol is a small constant when averaged over all of the instances of OPPRF. Our experiments confirm that this table-based OPPRF construction is indeed fast in practice.

4 EXTENDING OPPRF TO MANY QUERIES

The OPPRF constructions in the previous section are efficient when n (the number of programmed points) is small. When built from the efficient OPRF protocol of [26], they allow the receiver to evaluate the programmable PRF on only $t = 1$ point. We now show how to use a hashing technique to overcome both of these limitations. We show how to extend OPPRF constructions described in the previous section to support both a large n and a large t.

At the high level, the idea is that each party hashes their items into bins. Each bin contains a small number of inputs which allows the two parties to evaluate OPPRF bin-by-bin efficiently. The particular hashing approach we have in mind is as follows. Suppose the receiver has items $(q_1, \dots, q_t)$ on which he wants to evaluate an OPPRF. The sender has a set $\mathcal{P} = \{(x_1, y_1), \dots, (x_n, y_n)\}$ of points to program.

Cuckoo hashing. The receiver uses Cuckoo hashing (Section 2.2) to hash his items into bins. We will use a variant of Cuckoo hashing with k hash functions $h_1, \dots, h_k$, and m bins denoted as $B[1 \cdots m]$. Each item q is placed in exactly one of $\{B[h_1(q)], \dots, B[h_k(q)]\}$. Based on t and k, the parameter m is chosen so that every bin can contain at most one item with probability $1 - 2^{-\lambda}$ for a security parameter λ. We note that previous applications of Cuckoo hashing to PSI [38, 39] have used a variant of Cuckoo hashing that involves an additional stash (a place to put items when insertion fails). However, a stash renders our scheme much less efficient (every item in one party's stash must be compared to every item of another party). Instead, we propose a variant of Cuckoo hashing that avoids a stash by using 3 "primary" Cuckoo hash functions,

Probability	Bin scale & Max Bin Size	set size n				
		2^{12}	2^{14}	2^{16}	2^{20}	2^{24}
2^{-30}	ζ_1	1.15	1.13	1.13	1.13	1.12
	ζ_2	0.14	0.14	0.14	0.15	0.16
	β_1	28	28	29	30	31
	β_2	63	63	63	63	63
2^{-40}	ζ_1	1.17	1.15	1.14	1.13	1.12
	ζ_2	0.15	0.16	0.16	0.17	0.17
	β_1	27	28	29	30	31
	β_2	63	63	63	63	63

Table 2: Required number of bins $m_1 = n\zeta_1, m_2 = n\zeta_2$ to mapping n items using Cuckoo hashing, and required bin size β_1, β_2 to mapping n items into m_1 and m_2 bins using Simple hashing.

and then falling back to 2 "supplementary" Cuckoo hash functions when the first 3 fail. We empirically determine the parameters used in our hashing scheme to ensure that the hashing succeeds except with the probability less than $2^{-\lambda}$. The details are in Appendix B.

Simple hashing. Using the same set of hash functions, the sender then maps his points $\{x_1, \dots, x_n\}$ into bins, with each item being mapped under *all* of the Cuckoo hash functions (i.e., each of the sender's items appears k times in the hash table). Using standard balls-and-bins calculations based on n, k, and m, one can deduce an upper bound β such that no bin contains more than β items except with probability $1/2^\lambda$.

Denote by m_1, m_2 the number of bins used in 3-way "primary" Cuckoo hashing and 2-way "supplementary" Cuckoo hashing, respectively. Let β_1, β_2 denote the maximum bin size when using Simple hashing to map n items to m_1 and m_2 bins with no overflow, respectively. The parameters $m = m_1 + m_2$ and $\beta \in \{\beta_1, \beta_2\}$ presented in Table 2. The details of how we obtained these numbers are given in Appendix B.

Now within each bin, the receiver has at most one item q and the sender has at most β, call them $\{(x_1, y_1), \dots, (x_\beta, y_\beta)\}$. They can therefore run the basic OPPRF protocol on these inputs. Note that each of the sender's points (x, y) is mapped to several bins. The OPPRF in each of those bins will be programmed with the same (x, y). That way, if the receiver does have some $q_i = x$, then no matter which of the possible bins it is mapped to in Cuckoo hashing, the receiver will receive the correct output y.

The formal description of this protocol is given in Figure 7. The protocol requires m invocations of a single-query OPPRF, where $m = O(n)$ is the number of Cuckoo hash bins.

In sum, we are able to evaluate OPPRF for large number of programmed points n and large number of queries simply by having players hash their inputs into bins, and evaluate OPPRF per bin on small-size instances.

Caveats. One subtlety in analyzing our construction has to do with the security definition for a programmable PRF. Recall that in that definition (Section 3.1), the programmed output (y values) are chosen randomly. Yet in our protocol the sender programs different bins with *correlated* outputs. In particular, when an x_i is mapped to several bins, the OPPRF in each bin is programmed with the same (x_i, y_i) point. To deal with this, we must use the fact that

INPUT OF $\mathcal{S}$: n points $\mathcal{P} = \{(x_1, y_1), \dots, (x_n, y_n)\}$, where $x_i \in \{0,1\}^*$, $x_i \neq x_j$ and $y_i \in \{0,1\}^r$

INPUT OF $\mathcal{R}$: $Q = (q_1, \dots, q_t) \in (\{0,1\}^*)^t$.

PARAMETERS:

- Hash function $h_1, \dots, h_5$, number of bins $m \in \{m_1, m_2\}$, and max bin size $\beta \in \{\beta_1, \beta_2\}$, suitable for our hashing scheme (Table 2)

PROTOCOL:

(1) $\mathcal{R}$ hashes items Q into m bins using the Cuckoo hashing scheme defined in Section 4. Let $B_\mathcal{R}[b]$ denote the item in the receiver's bth bin (or a dummy item if this bin is empty).
(2) $\mathcal{S}$ hashes items $\{x_1, \dots, x_n\}$ into m_1 bins under 3 hash functions h_1, h_2, h_3, and hashes items $\{x_1, \dots, x_n\}$ into m_2 bins under 2 hash functions h_4, h_5. Let $B_\mathcal{S}[b]$ denote the set of items in the sender's bth bin.
(3) For $c \in [1,2]$, for each bin $b \in [m_c]$:
  (a) $\mathcal{S}$ computes $\mathcal{P}_b = \{(x_i, y_i) \mid (x_i, y_i) \in \mathcal{P}$ and $x_i \in B_\mathcal{S}[b]\}$, then pads $\mathcal{P}_b$ with dummy pairs to the maximum bin size β_c
  (b) Parties invoke an instance of $\mathcal{F}_{\text{opprf}}^{F,1,\beta_c}$ with inputs $\mathcal{P}_b$ for the sender and $B_\mathcal{R}[b]$ for the receiver.
  (c) $\mathcal{S}$ receives output (k_b, hint_b), and $\mathcal{R}$ receives output $(\mathsf{hint}_b, F(k_b, \mathsf{hint}_b, B_\mathcal{R}[b]))$.
(4) For each item $q_i \in Q$, let $z_i = F(k_b, \mathsf{hint}_b, q_i)$ where b is the bin to which $\mathcal{R}$ has hashed q_i. The receiver outputs $(\mathsf{hint}_1, \dots, \mathsf{hint}_m), (z_1, \dots, z_t)$

Figure 7: Hashing-based OPPRF protocol

the receiver is guaranteed to never query two bins on the same q (corresponding to the fact that his Cuckoo hashing assigns each q to a unique bin).

5 MULTI-PARTY PSI

We now present our main result, an application of OPPRF to multi-party PSI. We use the following notation in this section. We denote the n parties by $P_1, \dots, P_n$, and use subscripts i and j to refer to individual parties. Let $X_i \subseteq \{0,1\}^*$ denote the input set of party P_i. The goal is to securely compute the intersection $\bigcap_i X_i$. For sake of simplicity, we assume each set has m items and write $X_i = \{x_1^i, \dots, x_m^i\}$. We use subscript k to refer to a particular item x_k^i.

As discussed at the Introduction (cf. Section 1.3), our PSI protocol proceeds in two consecutive phases, **conditional zero-sharing** and **conditional reconstruction** of secrets. Importantly, OPPRF is efficient even when run on large input sets, thanks to our use of Cuckoo hash as discussed in Section 4.

5.1 Conditional Zero-Sharing

We will first describe the end goal of conditional zero-sharing and then discuss how we use multi-query OPPRF of Section 4 to achieve it. At the end of this phase, each party P_i will have a mapping $S_i : X_i \to \{0,1\}^*$ that associates each of its items $x_k^i \in X_i$ with an additive secret share $S_i(x_k^i)$. We require the following property:

if $x \in \bigcap_i X_i$ (i.e., x is in the intersection), then the corresponding shares $\{S_i(x) \mid i \in [n]\}$ will XOR to zero.

To achieve this, first consider the case of two parties P_1 and P_2. For each item $x_k^1 \in X_1$, party P_1 will choose a random string s_k and record the mapping $S_1(x_k^1) = s_k$. Then the parties can use an instance of multi-query OPPRF as follows. P_1 programs the OPPRF using points $\{(x_k^1, s_k) \mid k \in [m]\}$, and P_2 acts as receiver with input queries X_2. As a result, P_2 will obtain for every $x_k^2 \in X_2$ a corresponding OPPRF output, which we will denote $S_2(x_k^2)$. From the properties of an OPPRF, the mappings S_1 and S_2 have the desired property. If the parties share an item x_k^1 then both will have $S_1(x_k^1) = S_2(x_k^1) = s_k$, corresponding to an XOR-additive sharing of 0. The properties of the OPPRF ensure that P_2 does not know whether he is receiving real shares or random values for any item.

The case of n parties is similar. Each party P_i will act as dealer for each of their items $x_k^i \in X_i$, generating a random additive sharing of zero: $s_k^{i,1} \oplus \cdots \oplus s_k^{i,n} = 0$. Then each pair of parties P_i and P_j use an instance of OPPRF as follows. P_i programs the OPPRF using points $\{(x_k^i, s_k^{i,j}) \mid k \in [m]\}$, and P_j acts as receiver with input queries X_j. In other words, $s_k^{i,j}$ is the share that is conditionally sent from party P_i to P_j pertaining to item x_k^i.

Now each P_j has acted as OPPRF receiver for all other parties. For each item $x_k^j \in X_j$, the party has an OPPRF output from every sender P_i, along with their own share $s_k^{j,j}$. Denote by $S_j(x_k^j)$ the XOR of all of these values. It is easy to see that these S_j mappings satisfy the desired property. If some x is shared by all parties, then all pairs of parties will exchange shares corresponding to that item. All shares generated by a single party XOR to zero, so all of the $S_j(x)$ values XOR to zero as desired.

5.2 Conditional Reconstruction

The second phase of the protocol is a **conditional reconstruction** of secrets. In this phase party P_1 acts as a centralized "dealer." For each item $x \in X_1$ belonging to the dealer, he would like to determine whether x is in the intersection. It suffices for him to obtain all $S_i(x)$ values from all the parties. However, since some parties may not hold item x, they may not have a well-defined $S_i(x)$ value.

This problem can again be solved with an OPPRF. Each party P_i programs an OPPRF instance on points $\{(x, S_i(x)) \mid x \in X_i\}$, and P_1 acts as receiver with PRF queries X_1. Hence, for each item $x \in X_1$, dealer P_1 learns an associated value y^i from the OPPRF with party i. If x is indeed in the intersection, then we expect $\bigoplus_{i \neq 1} y^i = S_1(x)$. Otherwise the left-hand-side will be a random value.

5.3 Details and Discussion

A formal description of the protocol is in Figure 8.

Correctness. From the preceding high-level description, it is clear that the protocol is correct except in the event of a false positive – i.e., $S_1(x_k^1) = \bigoplus_i y_k^i$ for some $x_k^1 \in X_1$ not in the intersection. Let P_i be a party who did not have x_k^1 in their input set. That party will not program their OPPRF in Step 4 on the point x_k^1. As a result, the term y_k^i is pseudorandom. Hence the probability that of a false positive involving x_k^1 is $2^{-\ell}$. By setting $\ell = \lambda + \log_2(m)$, a union

PARAMETERS: n parties $P_1, \ldots, P_n$.

INPUT: Party P_i has input $X_i = \{x_1^i, \ldots, x_m^i\} \subseteq \{0,1\}^*$

PROTOCOL:

(1) For all $i \in [n]$ and all $k \in [m]$, party P_i chooses random $\{s_k^{i,j} \mid j \in [n]\}$ values subject to $\bigoplus_j s_k^{i,j} = 0$.
(2) For all $i, j \in [n]$, parties P_i and P_j invoke an instance of $\mathcal{F}_{\text{opprf}}^{F,m,m}$ where:
   - P_i is sender with input $\{(x_k^i, s_k^{i,j}) \mid k \in [m]\}$.
   - P_j is receiver with input X_j.

   For $x_k^j \in X_j$, let $\hat{s}_k^{i,j}$ denote the corresponding output of $\mathcal{F}_{\text{opprf}}$ obtained by P_j.
(3) For all $i \in [n]$ and $k \in [m]$, party P_i sets $S_i(x_k^i) = s_k^{i,i} \oplus \bigoplus_{j \neq i} \hat{s}_k^{j,i}$.
(4) For $i = 2$ to n, parties P_i and P_1 invoke an instance of $\mathcal{F}_{\text{opprf}}^{F,m,m}$ where:
   - P_i is sender with input $\{(x_k^i, S_i(x_k^i)) \mid k \in [m]\}$.
   - P_1 is receiver with input X_1.

   For $x_k^1 \in X_1$, let y_k^i denote the corresponding output for x_k^1 of $\mathcal{F}_{\text{opprf}}$ involving P_i.
(5) Party P_1 announces $\{x_k^1 \in X_1 \mid S_1(x_k^1) = \bigoplus_{i \neq 1} y_k^i\}$.

Figure 8: Multi-Party PSI Protocol

bound shows that the probability of *any* item being erroneously included in the intersection is $2^{-\lambda}$.

THEOREM 5.1. *The protocol of Figure 8 is secure in the semi-honest model, against any number of corrupt, colluding, semi-honest parties.*

PROOF. Let C and H be a coalition of corrupt and honest parties, respectively. To show how to simulate C's view in the ideal model, we consider two following cases based on whether all parties in C have item x:

- All parties in C have x and not all parties in H have x: if H contains only one honest party P_i, then P_i does not have x. From the output of set intersection, C can deduce that P_i does not have x. Thus, there is nothing to hide about whether P_i has x in this case.
  Consider the case that H has more than one honest party, say P_i and P_j. Suppose P_i has x, while party P_j does not. So, x does not appear in the intersection. We must show that the protocol must hide the identity of which honest party is missing x.
  In Step 2 of the protocol, there is an OPPRF instance with P_j as sender and P_i as receiver. P_j will not program the OPPRF at point x, so P_i will receive a pseudorandom output for x that is independent of the corrupt coalition's view. This causes $S_i(x)$ to be independent of the coalition's view.
  Later in Step 4, if the dealer is corrupt, both P_i and P_j act as OPPRF senders with the dealer. P_i programs the OPPRF at x using the pseudorandom value $S_i(x)$. P_j doesn't program the OPPRF at point x. The security of OPPRF is that programming the PRF at x with a random output is indistinguishable from not programming at x at all. In other words, parties P_i

and P_j have indistinguishable effect on the conditional reconstruction phase. If dealer is honest, the corrupt coalition's view is simulated from Step 2 based on the functionality of OPPRF.

- Not all corrupt parties in C have x: we must show that C should learn nothing about whether any of the honest parties hold x.

  Any honest party P_i who holds x generates corresponding shares $s^{i,j}$, to be conditionally distributed in Step 2. But some corrupt party does not query the OPPRF on x in step 2. This makes all the $s^{i,j}$ shares corresponding to x distributed uniformly. All honest parties P_j who hold x will therefore have $S_j(x)$ uniformly distributed of the coalition's view. In Step 4, the honest parties that hold x will program the OPPRF on $(x, S_j(x))$. The honest parties that don't hold x will not program the OPPRF on point x. As above, programming the PRF with a random output is indistinguishable from not programming at that point at all. Hence all honest parties have indistinguishable effect on the reconstruction phase.

□

Cost and Optimizations. In the conditional sharing phase, each party performs a multi-query OPPRF with every other party. In the reconstruction phase, each party performs just one multi-query OPPRF with the leader P_1. Recall that the cost of each of these is one instance of single-query OPPRF per Cuckoo-hashing bin.

The multi-query OPPRF scales well when sender and receiver have different number of elements. Therefore, our multi-party PSI protocol allows each party's set to have different size. The number of OPPRF instance depends on the number of bins for Cuckoo-hashing, and the OPPRF receiver is the one using Cuckoo hashing (sender uses plain hashing). Thus, our PSI protocol is more efficient by setting the leader P_1 as the party with the *smallest input set.*

We note that all of the OPPRF instances in the conditional sharing phase can be done in parallel, and all the OPPRF instances in the reconstruction phase can as well. This leads to a constant-round protocol.

Finally, recall that the multi-query OPPRF uses Cuckoo hashing. It is safe for *all* such instances, between all pairs of parties, to use the same Cuckoo hash functions. That way, a party only needs to hash their input set twice at the beginning of the protocol (once with Cuckoo hashing for when they are OPPRF receiver, and once with simple hashing for when they are OPPRF sender).

Generalization. Suppose we wish to secure the protocol against the possibility of at most t corrupt (colluding) parties. The default case is to consider $t = n - 1$. For smaller t, we can simplify the protocol. The idea is to modify the conditional zero-sharing protocol so that party P_i generates shares of zero only for $\{P_{i+1}, \ldots, P_{i+t+1}\}$ (where indices on parties are mod n). The security analysis applies also to this generalization, based on the fact that if P_i is honest, then at least one of $P_{i+1}, \ldots, P_{i+t+1}$ must also be honest.

6 FURTHER OPTIMIZATIONS

6.1 PSI in Augmented Semi-Honest Model

In this section we show an optimization to our PSI protocol which results in a protocol secure in the augmented semi-honest model (cf. Section 2 and Appendix A).

Initialization: Each party P_i picks random seeds $r_{i,j}$ for $j = i+1, \ldots, n$ and sends seed $r_{i,j}$ to P_j

Generate zero-sharing: Given an index ind, each P_i computes

$$S_i(ind) = \left(\bigoplus_{j=1}^{i-1} F(r_{j,i}, ind) \right) \oplus \left(\bigoplus_{j=i+1}^{n} F(r_{i,j}, ind) \right)$$

Figure 9: The zero-sharing protocol

Unconditional zero-sharing. The previous protocol starts with a *conditional* zero-sharing phase, where parties obtain shares of zero or shares of a random value, based on whether they share an input item x. In this section we propose an *unconditional* zero-sharing technique in which the parties always receive shares of zero.

We describe a method for generating an unlimited number of zero-sharings derived from short seeds that can be shared in a one-time initialization step. The protocol is described in Figure 9. The protocol is based on an initialization step where each pair of parties exchange keys for a PRF F, after which each party knows $n-1$ keys. Then, whenever zero-sharing is needed, party P_i generates a share as $S_i(ind) = \bigoplus_r F(r, ind)$, where ind is an index which identifies this protocol invocation, and r ranges over all the keys shared with other parties.

We first observe that the XOR of all $S_i(ind)$ shares is indeed 0, since each term $F(r_{i,j}, ind)$ appears exactly twice in the expression. As for security, consider a coalition of $t < n - 1$ corrupt parties, and let P_k be the honest party with smallest index. P_k sends random seeds to all other honest parties. These seeds are independent of all other seeds, and are unknown to the corrupt coalition. They result in set of $n - t - 1$ pseudorandom terms that are included in the shares of all honest parties other then P_k. Therefore the shares of the honest parties look pseudorandom to the coalition (subject to all shares XORing to zero).

Plugging into the PSI protocol. Suppose we modify our main PSI protocol (Figure 8) in the following ways:

- Instead of steps 1-3, the parties perform the unconditional zero-sharing phase of Figure 9. That is, they run the initialize phase to exchange seeds and then set their S_i mappings accordingly.
- Then they continue with Figure 8 starting at step 4.

The modification significantly reduces the cost of the zero-sharing phase (which was the most expensive part of Figure 8) with a zero-sharing phase that costs almost nothing. Our experiments confirm that this modified protocol is faster than the standard semi-honest-secure protocol, by a significant constant factor.

Correctness of the modified protocol follows from the same reasons as for the unmodified protocol. Namely, if some party P_i does not have an item x, then they will not program their OPPRF with P_1 at point x. This causes P_1 to obtain a random value in the reconstruction phase and subsequently not include x in the output.

Theorem 6.1. *The modified protocol (with unconditional zero-sharing) is secure in the augmented semi-honest model.*

Proof Sketch. Consider a coalition C of corrupt parties. We must show how to simulate C's view in the ideal model. If $P_1 \notin C$ then, assuming that the underlying OPRF protocol is secure, the view of C consists only of the output of the invocations of the OPPRF protocol (acting as *sender* in each one), and is therefore random. If the leader $P_1 \in C$ then the simulator sends to the ideal PSI functionality the set X_1 as the input of *every* corrupt party (this is the advantage given to the simulator in the augmented security model). Let Z denote the output of the functionality (the intersection of all sets). P_1's view contains OPPRF outputs from all honest parties, corresponding to every $x \in X$. For $x \in Z$, simulate a random sharing of zero as the corresponding OPPRF outputs. For $x \in X_1 \setminus Z$, simulate random values for the corresponding OPPRF outputs. □

Let us give an intuition on why this protocol achieves security only in the augmented model. In this modified protocol, the zero-sharing for each candidate x is generated non-interactively by the parties. So even though a corrupt party P_i does not have an item x, he can non-interactively imagine what his correct share $S_i(x)$ would be. When colluding with P_1, this allows the adversary to learn exactly what would have happened if P_i included x in its set (but only if $x \in X_1$ as well).

In the semi-honest protocol (Section 5), however, a corrupt party interacts with honest parties to generate a zero-sharing corresponding to x. At the time of the interaction, the corrupt party P_i "commits" to having x in its input set or not, depending on whether it queries the OPPRF on x. If during the (conditional) zero-sharing phase P_i does not have x in its input set, then there is no way to later guess what the "correct share" would have been.

6.2 Reducing OPPRF Hint Size

In this section we look inside the several layers of abstraction in our PSI protocol, and use a global view of things to find room for optimization. We focus on the multi-query OPPRF construction from Section 4. Recall that it works in the following way:

- The OPPRF receiver hashes their queries into m bins via a Cuckoo hashing method.
- The OPPRF sender hashes their programming-points into m bins using simple hashing, *for each Cuckoo hash function* (i.e., assigning a single item to many bins).
- In each bin, the parties perform a single-query OPPRF instance, where the receiver queries on their (unique) item in that bin.

Now look even further inside those single-query OPPRF instances. In each one, the parties invoke an OPRF instance and furthermore the sender gives a "hint" that contains the information to correct/program the OPRF outputs to the desired values.

There are two possible approaches for sending the hints that are required for these OPPRF computations. The straightforward approach sends a separate hint per OPPRF invocation, namely per bin. The other approach sends a single combined hint for all bins. Namely, this combined hint is a single polynomial or Bloom filter, which provides for each of the m possible inputs of P_i the correct "hint" for changing the output of the corresponding OPRF invocation.

The advantage of the "separate hints" approach is that in each OPPRF invocation each party P_i has only $S = \mathcal{O}(\log m / \log\log m)$ points and therefore computing the hint might be more efficient. This is relevant for the polynomial-based hint, since its computation time is quadratic in the size of the set of points. Therefore, the overhead of computing a single combined hint polynomial is $O(m^2)$ whereas the overhead of computing hints for all bins is only $\mathcal{O}(m \log^2 m / \log^2(\log m))$. On the other hand, when computing a hint per bin, the total number of points is $\mathcal{O}(m \log m / \log\log m)$, whereas if a combined hint is used, the total number of points is $\mathcal{O}(m)$. We expect (and validate in the experiments in Section 7), that a combined hint works better for the Bloom filter-based OPPRF, since the cost of this method is linear in the total number of points. On the other hand, the bottleneck of the polynomial-based OPPRF is the quadratic overhead of polynomial interpolation, thus when using that OPPRF it is preferable to use separate hints per bin.

Improvements: We can add the following improvements to the basic protocol:

- In polynomial-based OPPRF with "separate hints", the OPPRF sender does not need to pad with dummy items to the maximum bin size β before interpolating a polynomial over β pairs per bin. Instead of that, he interpolates a polynomial $p_1(x)$ over $k < \beta$ real pairs (x_i, y_i) and then add it with a polynomial $p_2(x)$ of degree $(\beta - 1)$. $p_2(x)$ can be efficiently implemented as $R(x) \prod_{i=1}^{k}(x - x_i)$, where $R(x)$ is a random polynomial of degree $(\beta - 1 - k)$. Using example hashing parameters from Section 5, the expected value of k is only 3, while the worst-case $\beta = 30$. This optimization reduces the cost of expensive polynomial interpolation.
- In polynomial-based OPPRF with combined hints, the OPPRF sender can send a combined hint for *each* hash function h_i. That is, for each Cuckoo hash function h_i, the sender computes a hint that reflects *all* of the bin-assignments under that specific h_i. The receiver hashes its items with Cuckoo hashing, and places each item according to exactly one hash function h_i. For each item, the receiver can therefore use the combined hint for that specific h_i.
- In Bloom filter-based OPPRF invocation, each of sender's item appears 5 times in hash table, there are 5 different OPRF values $F(k_{h_i}, x))$. Instead of inserting 5 pairs of the form $(x, y \oplus F(k_{h_i}, x))$ into the GBF, the sender can instead insert the concatenated value $(x, (y \oplus F(k_{h_1}, x))|| \ldots ||(y \oplus F(k_{h_5}, x)))$. This reduces the number of the GBF insertions.

6.3 3-party PSI in Standard Semi-Honest Model

Our idea for three-party PSI (3-PSI) is to have all 3 players perform an (encrypted) incremental computation of the intersection. Namely, P_1 and P_2 will first let P_2 obtain an encoding of partial intersection $X_{12} = X_1 \cap X_2$. Then P_2 and P_3 will allow P_3 to obtain some encoding of $X_{123} = X_{12} \cap X_3$. In the end, P_1 will decode the output $X_{123} = X_1 \cap X_2 \cap X_3$.

To do this, the leader P_1 chooses a random encoding e_k^1 for each of his inputs x_k^1. P_1 then acts as a sender in OPPRF, programming it on points $\{(x_k^1, e_k^1) \mid k \in [m]\}$. P_2 acts as a receiver in OPPRF using his input set X_2, and obliviously receives either one of these encodings (if his input was a corresponding match) or a random string. Denote by $\widehat{e}_k^2$ the value that P_2 obtains for each of his items x_k^2. The process repeats: P_2 will play the role of OPPRF sender with receiver P_3. P_2 will program the OPPRF on points $\{(x_k^2, \widehat{e}_k^2) \mid k \in [m]\}$ and P_3 will

PARAMETERS: 3 parties P_1, P_2, P_3.

INPUT: Party P_i has input $X_i = \{x^i_1, \ldots, x^i_m\} \subseteq \{0,1\}^*$

PROTOCOL:

(1) For all $k \in [m]$, party P_1 chooses random distinct $\{e^1_k \mid k \in [m]\}$ values.
(2) Party P_1 and P_2 invoke with an instance of $\mathcal{F}^{F,m,m}_{\text{opprf}}$ where:
   - P_1 is sender with input $\{(x^1_k, e^1_k) \mid k \in [m]\}$.
   - P_2 is receiver with input X_2.

   For $x^2_k \in X_2$, let $\widehat{e}^2_k$ denote the corresponding output of $\mathcal{F}_{\text{opprf}}$ obtained by P_2.
(3) In turn, each party P_i, $i \in \{2,3\}$, invokes with P_{i+1} an instance of $\mathcal{F}^{F,m,m}_{\text{opprf}}$ where:
   - P_i is sender with input $\{(x^i_k, \widehat{e}^i_k) \mid k \in [m]\}$.
   - P_{i+1} is receiver with input X_{i+1}.

   For $x^{i+1}_k \in X_{i+1}$, let $\widehat{e}^{i+1}_k$ denote the corresponding output of $\mathcal{F}_{\text{opprf}}$ obtained by P_{i+1} (indices are mod n)
(4) Party P_1 announces $\{x^1_k \in X_1 \mid e^1_k = \widehat{e}^1_k\}$.

Figure 10: Optimized Three-party PSI Protocol

query the OPPRF on his input set X_3. Denote by $\widehat{e}^3_k$ the value that P_3 obtains for each of his items x^3_k.

Finally, P_3 acts as OPPRF sender and programs the OPPRF on points $\{(x^2_k, \widehat{e}^2_k) \mid k \in [m]\}$, while P_1 acts as receiver and queries the OPPRF on points X_1. It is clear that if x^1_k is in the intersection, then P_1 will receive e^1_k (a value he initially chose) as OPPRF output; otherwise he will receiver a random value. A formal description of the protocol is in Figure 10.

Extending the above to $n > 3$ parties faces the following difficulty: If P_1 and P_j collude, they will learn the partial intersection $X_1 \cap \cdots \cap X_j$. Indeed, as an OPPRF receiver, P_j will receive the set of values which can be cross-checked with the encodings generated by P_1. More generally, colluding players P_i and P_j can compute partial intersection $X_i \cap \cdots \cap X_j$ by comparing their encodings.

We note that this is not an issue in 3-PSI, since colluding P_1 and P_2 can compute $X_1 \cap X_2$ anyway; colluding P_2 and P_3 cannot learn any information about the decrypted key e^1_i held by P_1 thus the corrupted parties compute $X_2 \cap X_3$ anyway; and colluding P_1 and P_3 can compute $X_1 \cap X_2 \cap X_3$ which is the desired PSI output.

With the above optimization, our 3-PSI protocol needs only 3 OPPRF executions, compared to the 4 OPPRF executions for the general protocol described in Section 5. The performance gain of the optimized protocol is not very strong when the network is slow since parties invoke OPPRF in turn and they have to wait for the previous OPPRF completed. We implemented both 3-PSI protocol variants and found this optimized variant to be $1.2 - 1.7\times$ faster.

7 IMPLEMENTATION AND PERFORMANCE

In order to evaluate the performance of our multi-party PSI protocols, we implement many of the variants described here. We do a number of experiments on a single server which has 2x 36-core Intel Xeon 2.30GHz CPU and 256GB of RAM. We run all parties in the same network, but simulate a network connection using the Linux *tc* command: a LAN setting with 0.02ms round-trip latency, 10 Gbps network bandwidth; a WAN setting with a simulated 96ms round-trip latency, 200 Mbps network bandwidth.

In our protocol, the offline phase is conducted to obtain an 128 base-OTs using Naor-Pinkas construction [33]. Our implementation uses OPRF code from [26, 42]. All evaluations were performed with a item input length 128 bits, a statistical security parameter $\lambda = 40$ and computational security parameter $\kappa = 128$. The running times recorded are an average over 10 trials. Our complete implementation is available on GitHub: https://github.com/osu-crypto/MultipartyPSI

7.1 Optimized PSI, Augmented Model

In this section we discuss the PSI protocol from Section 6 that is optimized for the augmented semi-honest model. We implemented and tested the following variants (see Section 6.2 for discussion on variant techniques of sending hints) on different set sizes $m \in \{2^{12}, 2^{14}, 2^{16}, 2^{20}\}$:

- BLOOM FILTER: where the OPPRF used a single combined garbled Bloom filter hint. In our hashing-to-bin scheme (Appendix B), sender uses $h = 5$ hash functions to insert m items into bins. With the optimization in Section 6.2, there are only m pairs inserted into the table which has $m\lambda \log_2 e$ entries. The table uses an array of $h(\lambda + \log_2(m))$-bit strings.
- POLYNOMIAL combined: where the OPPRF used combined polynomial hints per hash index. Polynomial interpolation was implemented using the NTL library[48]. Each polynomial is built on m points. The coefficients of the polynomial are $\lambda + \log_2(m)$-bit strings.
- POLYNOMIAL separated: where the OPPRF used a separate polynomial hint per bin. The coefficient of the polynomial has $\lambda + \log_2(m)$-bit strings. The degree of polynomial is β_1 for each bin in first $m\zeta_1$ bins, and β_2 for each bin in last $m\zeta_2$ bins. Here $\zeta_1, \zeta_2, \beta_1$ and β_2 are discussed in Table 2.
- TABLE: where the OPPRF used a separate table hint per bin. The table has $2^{\lceil \log_2(\beta_1) \rceil}$ entries for each bin in first $m\zeta_1$ bins, and $2^{\lceil \log_2(\beta_2) \rceil}$ entries for each bin in last $m\zeta_2$ bins. Each row has $\lambda + \log_2(m)$-bit strings.

The running times and communication overhead of our implement with 5 parties are shown in Table 3. The leader party uses up to 4 threads, each operates OPPRF with other parties. As expected, our table-based protocol achieves the fastest running times in comparison with the other OPPRF constructions. Our experiments show that it takes only one second to sample vector v and check uniqueness for all 2^{20} bins. Thus, the table-based PSI protocol costs only 22 seconds for the set size $m = 2^{20}$. The polynomial-based PSI protocol with separated hint is the next fastest protocol which requires a total time of 38 seconds, a $1.7\times$ slowdown. The slowest protocol is the polynomial-based protocol with combined hint per hash index, whose running time clearly grows quadratically with the set size. However, this protocol has the smallest communication overhead. For small set size $m = 2^{14}$, the polynomial-based PSI protocol with combined hint requires only 1.74MB for communication.

Protocol	**Running time (second)**				**Communication (MB)**			
	Set Size m							
	2^{12}	2^{14}	2^{16}	2^{20}	2^{12}	2^{14}	2^{16}	2^{20}
BLOOM FILTER	0.37	0.98	3.41	51.46	8.56	34.26	137.01	2496.2
POLY (combined hint)	7.36	194.96	-	-	**0.43**	**1.74**	-	-
POLY (separate hints)	0.32	0.74	2.33	37.89	1.46	5.98	**24.30**	**447.44**
TABLE	**0.29**	**0.57**	**1.48**	**21.93**	1.64	6.52	25.93	467.66

Table 3: The total runtime and communication of our Multi-Party PSI in augmented semi-honest model in LAN setting. The communication cost which ignore the fixed cost of base OTs for OT extension is on the *client's side*. Cells with – denote trials that either took longer than hour or ran out of memory.

Setting	Number Parties n	Threshold Corruption t	Set Size m			
			2^{12}	2^{16}	2^{20}	2^{24}
LAN	3	{1, 2}	0.21 (0.99)*	1.34 (1.19)*	**25.81 (25.23)***	409.90 (399.67)*
			0.30 (0.16)	**2.14 (1.97)**	**41.64 (41.10)**	702.3 (69.69)
	4	1	0.25 (0.12)	1.80 (1.60)	28.86 (28.27)	484.3 (478.2)
		{2, 3}	0.34 (0.21)	3.16 (2.92)	52.25 (51.65)	865.7(859.4)
	5	1	0.26 (0.12)	1.99 (1.79)	32.13 (31.49)	505.2 (499.2)
		2	0.32 (0.19)	3.44 (3.23)	49.17 (48.54)	-
		4	0.39 (0.26)	4.87 (4.61)	**71.28 (70.60)**	-
	10	1	0.39 (0.17)	2.97(2.71)	46.08 (45.28)	-
		5	0.83 (0.55)	8.79 (8.47)	136.48 (135.44)	-
		9	1.01 (0.72)	12.33 (11.98)	**182.8 (181.60)**	-
	15	1	0.46 (0.23)	4.28 (3.97)	64.28 (63.27)	-
		7	1.37 (0.77)	13.47 (12.79)	201.12 (199.34)	-
		14	**1.85 (1.32)**	20.61 (20.02)	**304.36 (302.17)**	-
WAN	3	{1, 2}	2.82 (2.34)*	10.48 (9.96)*	129.45 (128.64)*	-
			3.12 (2.64)	11.25 (10.73)	158.50 (157.64)	-
	4	1	2.65 (1.97)	12.40 (11.71)	151.9 (150.9)	-
		{2, 3}	3.18 (2.51)	17.47 (16.74)	233.1 (232.1)	-
	5	1	2.66 (1.99)	13.76 (13.06)	185.5 (184.5)	-
		2	3.21 (2.53)	20.29 (19.56)	290.9 (289.8)	-
		4	3.45 (2.78)	25.52 (24.79)	378.5 (377.4)	-
	10	1	3.30 (2.63)	26.42 (25.73)	400.9 (399.8)	-
		5	5.67 (4.98)	76.43 (75.78)	1,194 (1,193)	-
		9	7.81 (7.14)	112.8 (112.1)	1,915 (1,914)	-
	15	1	3.63 (3.15)	39.11 (38.60)	664.08 (662.80)	-
		7	9.87 (9.38)	150.85 (150.31)	2641 (2,640)	-
		14	16.42 (15.96)	263.20 (262.67)	-	-

Table 4: Total running time and online time (in parenthesis) in second of our semi-honest Multi-Party PSI for the number of parties n, $t < n$ dishonestly colluding, each with set size m. Number with $*$ shows the performance of the optimized 3-PSI protocol described in Section 6.3. Cells with – denote trials that either took longer than hour or ran out of memory.

7.2 Standard Semi-Honest PSI

In this section we discuss the standard semi-honest variant of our protocol, using conditional zero-sharing (Section 5). From the empirical results discussed in the previous section, the most efficient OPPRF instantiation is the TABLE-based hint. Thus, the OPPRF was instantiated using the TABLE-based protocol in this section.

To understand the scalability of this protocol, we evaluate it on the range of the number parties $n \in \{3, 4, 5, 10, 15\}$ on the set size $m \in \{2^{12}, 2^{16}, 2^{20}, 2^{24}\}$. We also wanted to understand the performance effect of the generalization discussed in Section 5.3 in which the protocol is tuned to tolerate an arbitrary number t of corrupted parties. In our experiments, we used $t \in \{1, \lfloor n/2 \rfloor, n-1\}$.

Our protocol scales well using multi-threading between n parties. In our implementation, the leader P_1 uses $n-1$ threads and other parties use $\min\{t+1, n-1\}$ threads so that each party operates OPPRF protocol with other parties at the same time. However, we use a single thread to perform the OPPRF subprotocol between two parties.

We proposed a better "hashing to bin" scheme (Appendix B) than the state-of-art two-party PSI [26]. Specifically, our hashing scheme removes the stash bins which consume nontrivial cost of the protocol [26] for sufficiently small sets. For example of 2^{12} set size, we see that our protocol requires 168 milliseconds compared to 211 milliseconds by [26], a difference of 1.2×.

Results. Table 4 presents the running time of our PSI protocol in both LAN and WAN setting. We report the running time for the total time and online phase. The offline phase consists of all

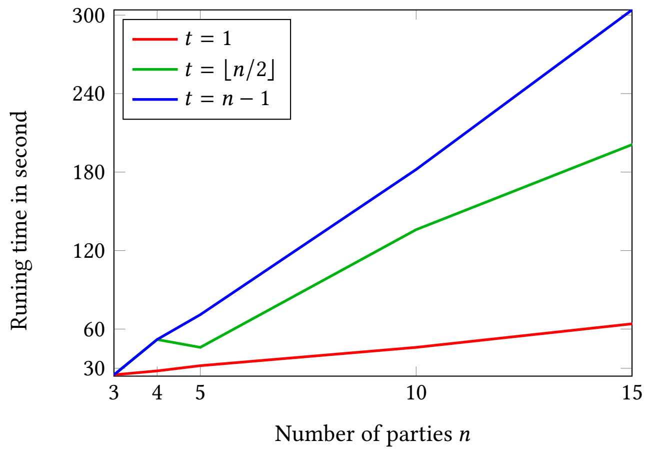

Figure 11: Total running time of our semi-honest Multi-Party PSI for the number of parties n, $t < n$ dishonestly colluding, each with set size 2^{20}, in LAN setting.

Number Parties n	Threshold Corruption t	Set Size m 2^{12}	2^{16}	2^{20}	2^{24}
3	$\{1, 2\}$	3.28	51.87	935.32	14,860
$\{4, 5\}$	1				
$\{10, 15\}$					-
4	$\{2, 3\}$	4.92	77.80	1,402	22,290
5	2	4.92	77.80	1,402	-
	4	6.56	103.74	1,870	-
10	5	9.84	155.61	2,805	-
	9	14.76	233.41	4,208	-
15	7	13.12	207.48	3,741	-
	14	22.96	363.09	6,547	-

Table 5: The numerical communication (in MB) of our Multi-Party PSI in semi-honest setting. The cost is on the *client's side* for the number of parties n, $t < n$ dishonestly colluding, each with set size m. Communication costs ignore the fixed cost of base OTs for OT extension. Cells with – denote trials that either took longer than hour or ran out of memory.

operations which do not depend on the input sets. In the three-party case, our protocol supports the full corrupted majority. For $m = 2^{20}$, our general 3-PSI protocol (Section 5) in LAN setting costs 42 seconds while the optimized protocol (Section 6.3) takes 26 seconds which is 1.6× faster. When evaluating our 3-PSI in WAN setting, we found this optimized variant to be 1.2× faster. This is primarily due to the need to wait for previous OPPRF completed.

To address the possibility of at most t parties colluding, each party performs OPPRF with $\min\{t+1, n-1\}$ other parties. Therefore the cost of the protocol is the same for $t = n - 1$ as $t = n - 2$. Hence, we report the protocol performance with the $n = 4$ and $t \in \{2, 3\}$ on the same row of the Table 4.

As we can see in the table 4, our protocol requires only 72 seconds to compute a PSI of $n = 5$ parties for $m = 2^{20}$ elements. For the same set size, when increasing the number of parties to $n = 10$, our total running time is 3 minutes and if $n = 15$ our protocol takes around 5 minutes. Figure 11 shows that our protocol's cost is linear in the size of number parties. When assuming only one corrupt party, our protocol takes only 64 seconds to compute PSI of 15 parties for $m = 2^{20}$ elements. For the small set size of $m = 2^{12}$, the PSI protocol of $n = 15$ parties takes an total time of 1.85 seconds with the online phase taking 1.32 seconds. We find that our protocol also scales to large input sets ($m = 2^{24}$) with $n \in \{3, 4, 5\}$ participants.

Table 5 reports the numerical communication costs of our implementation. The protocol is asymmetric with respect to the leader P_1 and other parties. Because the leader plays the role of receiver in most OPPRFs, the majority of his communication costs can be done in an offline phase. Hence we report the communication costs of the clients, which reflects the online cost of the protocol. For the small set size of $m = 2^{12}$, only 3.28MB communication was required in 3-PSI protocol on the client's sides. The communication complexity of our protocols is $\mathcal{O}(mt\lambda)$ bits. Thus, our protocol requires gigabytes of communication for a large set size ($m \in \{2^{20}, 2^{24}\}$). Concretely, for the large input set $m = 2^{24}$, our 3-PSI protocol uses 14.8GB of communication, roughly 0.88KB per item.

ACKNOWLEDGEMENTS

The first author was supported by Office of Naval Research (ONR) contract number N00014-14-C-0113. The second and third authors were supported by the BIU Center for Research in Applied Cryptography and Cyber Security in conjunction with the Israel National Cyber Bureau in the Prime Minister's Office. The fourth and fifth author were partially supported by NSF awards #1149647 and #1617197.

REFERENCES

[1] Aydin Abadi, Sotirios Terzis, and Changyu Dong. 2015. O-PSI: delegated private set intersection on outsourced datasets. In *ICT Systems Security and Privacy Protection*. Springer, 3–17.
[2] Arash Afshar, Payman Mohassel, Benny Pinkas, and Ben Riva. 2014. Non-Interactive Secure Computation Based on Cut-and-Choose. In *EUROCRYPT 2014 (LNCS)*, Phong Q. Nguyen and Elisabeth Oswald (Eds.), Vol. 8441. Springer, Heidelberg, Germany, Copenhagen, Denmark, 387–404. https://doi.org/10.1007/978-3-642-55220-5_22
[3] Marina Blanton and Everaldo Aguiar. 2012. Private and Oblivious Set and Multiset Operations. In *7th ACM Symposium on Information, Computer and Communications Security (ASIACCS '12)*. ACM, New York, NY, USA, 40–41. https://doi.org/10.1145/2414456.2414479
[4] Ran Canetti and Juan A. Garay (Eds.). 2013. *CRYPTO 2013, Part II*. LNCS, Vol. 8043. Springer, Heidelberg, Germany, Santa Barbara, CA, USA.
[5] Hao Chen, Kim Laine, and Peter Rindal. 2017. Fast Private Set Intersection from Homomorphic Encryption. Cryptology ePrint Archive, Report 2017/299. (2017). http://eprint.iacr.org/2017/299.
[6] Jung Hee Cheon, Stanislaw Jarecki, and Jae Hong Seo. 2012. Multi-Party Privacy-Preserving Set Intersection with Quasi-Linear Complexity. *IEICE Transactions* 95-A, 8 (2012), 1366–1378. http://search.ieice.org/bin/summary.php?id=e95-a_8_1366
[7] Dana Dachman-Soled, Tal Malkin, Mariana Raykova, and Moti Yung. 2012. Efficient Robust Private Set Intersection. *Int. J. Appl. Cryptol.* 2, 4 (July 2012), 289–303. https://doi.org/10.1504/IJACT.2012.048080
[8] Emiliano De Cristofaro, Jihye Kim, and Gene Tsudik. 2010. *Linear-Complexity Private Set Intersection Protocols Secure in Malicious Model.* Springer Berlin Heidelberg, Berlin, Heidelberg, 213–231. https://doi.org/10.1007/978-3-642-17373-8_13
[9] Changyu Dong, Liqun Chen, and Zikai Wen. 2013. When Private Set Intersection Meets Big Data: An Efficient and Scalable Protocol. In *ACM Conference on Computer &#38; Communications Security (CCS '13)*. ACM, 789–800. https://doi.org/10.1145/2508859.2516701
[10] Michael J. Freedman, Carmit Hazay, Kobbi Nissim, and Benny Pinkas. 2016. Efficient Set Intersection with Simulation-Based Security. *J. Cryptology* 29, 1 (2016), 115–155. https://doi.org/10.1007/s00145-014-9190-0
[11] Michael J. Freedman, Yuval Ishai, Benny Pinkas, and Omer Reingold. 2005. Keyword Search and Oblivious Pseudorandom Functions. In *TCC 2005 (LNCS)*, Joe Kilian (Ed.), Vol. 3378. Springer, Heidelberg, Germany, Cambridge, MA, USA, 303–324.

[12] Michael J. Freedman, Kobbi Nissim, and Benny Pinkas. 2004. Efficient Private Matching and Set Intersection. In *Advances in Cryptology - EUROCRYPT 2004 (Lecture Notes in Computer Science)*, Vol. 3027. Springer, 1–19. https://doi.org/10.1007/978-3-540-24676-3_1

[13] Ran Gelles, Rafail Ostrovsky, and Kina Winoto. 2012. Multiparty proximity testing with dishonest majority from equality testing. In *Automata, Languages, and Programming*. Springer, 537–548.

[14] Oded Goldreich. 2009. *Foundations of cryptography: volume 2, basic applications*. Cambridge university press.

[15] Shai Halevi, Yehuda Lindell, and Benny Pinkas. 2011. Secure Computation on the Web: Computing without Simultaneous Interaction. In *Advances in Cryptology - CRYPTO 2011 (Lecture Notes in Computer Science)*, Phillip Rogaway (Ed.), Vol. 6841. Springer, 132–150. https://doi.org/10.1007/978-3-642-22792-9_8

[16] Carmit Hazay and Yehuda Lindell. 2010. *Efficient secure two-party protocols: Techniques and constructions*. Springer Science & Business Media.

[17] Carmit Hazay and Muthuramakrishnan Venkitasubramaniam. 2017. Scalable Multi-Party Private Set-Intersection. Cryptology ePrint Archive, Report 2017/027. (2017). http://eprint.iacr.org/2017/027.

[18] Y. Huang, D. Evans, and J. Katz. 2012. Private Set Intersection: Are Garbled Circuits Better than Custom Protocols?. In *Network and Distributed System Security (NDSS'12)*. The Internet Society.

[19] Bernardo A. Huberman, Matt Franklin, and Tad Hogg. 1999. Enhancing Privacy and Trust in Electronic Communities. In *Proceedings of the 1st ACM Conference on Electronic Commerce (EC '99)*. ACM, 78–86.

[20] Stanislaw Jarecki and Xiaomin Liu. 2009. Efficient Oblivious Pseudorandom Function with Applications to Adaptive OT and Secure Computation of Set Intersection. In *Theory of Cryptography (Lecture Notes in Computer Science)*, Vol. 5444. Springer, 577–594. https://doi.org/10.1007/978-3-642-00457-5_34

[21] Adam Kirsch and Michael Mitzenmacher. 2008. Less Hashing, Same Performance: Building a Better Bloom Filter. *Random Struct. Algorithms* 33, 2 (Sept. 2008), 187–218. https://doi.org/10.1002/rsa.v33:2

[22] Adam Kirsch, Michael Mitzenmacher, and Udi Wieder. 2008. More Robust Hashing: Cuckoo Hashing with a Stash. In *ESA 2008 (Lecture Notes in Computer Science)*, Dan Halperin and Kurt Mehlhorn (Eds.), Vol. 5193. Springer, 611–622. https://doi.org/10.1007/978-3-540-87744-8_51

[23] Ágnes Kiss, Jian Liu, Thomas Schneider, N. Asokan, and Benny Pinkas. 2017. Private Set Intersection for Unequal Set Sizes with Mobile Applications. Cryptology ePrint Archive, Report 2017/670. (2017). http://eprint.iacr.org/2017/670.

[24] Lea Kissner and Dawn Song. 2005. Privacy-preserving Set Operations. In *Proceedings of the 25th Annual International Conference on Advances in Cryptology (CRYPTO'05)*. Springer-Verlag, Berlin, Heidelberg, 241–257. https://doi.org/10.1007/11535218_15

[25] Vladimir Kolesnikov. 2005. Gate Evaluation Secret Sharing and Secure One-Round Two-Party Computation. In *ASIACRYPT 2005 (LNCS)*, Bimal K. Roy (Ed.), Vol. 3788. Springer, Heidelberg, Germany, Chennai, India, 136–155.

[26] Vladimir Kolesnikov, Ranjit Kumaresan, Mike Rosulek, and Ni Trieu. 2016. Efficient Batched Oblivious PRF with Applications to Private Set Intersection. Cryptology ePrint Archive, Report 2016/799. http://eprint.iacr.org/2016/799.

[27] Vladimir Kolesnikov, Jesper Buus Nielsen, Mike Rosulek, Ni Trieu, and Roberto Trifiletti. 2017. DUPLO: Unifying Cut-and-Choose for Garbled Circuits. Cryptology ePrint Archive, Report 2017/344. (2017). http://eprint.iacr.org/2017/344.

[28] Ronghua Li and Chuankun Wu. 2007. *An Unconditionally Secure Protocol for Multi-Party Set Intersection*. Springer Berlin Heidelberg, Berlin, Heidelberg, 226–236. https://doi.org/10.1007/978-3-540-72738-5_15

[29] Yehuda Lindell and Ben Riva. 2015. Blazing Fast 2PC in the Offline/Online Setting with Security for Malicious Adversaries. In *ACM CCS 15*, Indrajit Ray, Ninghui Li, and Christopher Kruegel: (Eds.). ACM Press, Denver, CO, USA, 579–590.

[30] Catherine A. Meadows. 1986. A More Efficient Cryptographic Matchmaking Protocol for Use in the Absence of a Continuously Available Third Party. In *IEEE Symposium on Security and Privacy*. 134–137.

[31] Atsuko Miyaji and Shohei Nishida. 2015. A Scalable Multiparty Private Set Intersection. In *Network and System Security*. Springer, 376–385.

[32] Payman Mohassel and Ben Riva. 2013. Garbled Circuits Checking Garbled Circuits: More Efficient and Secure Two-Party Computation, See [4], 36–53. https://doi.org/10.1007/978-3-642-40084-1_3

[33] Moni Naor and Benny Pinkas. 2001. Efficient Oblivious Transfer Protocols. In *Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '01)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 448–457. http://dl.acm.org/citation.cfm?id=365411.365502

[34] Kurt Opsahl. 2013. The Disconcerting Details: How Facebook Teams Up With Data Brokers to Show You Targeted Ads. https://www.eff.org/deeplinks/2013/04/disconcerting-details-how-facebook-teams-data-brokers-show-you-targeted-ads. (2013). [Online; accessed 23-May-2016].

[35] Rasmus Pagh and Flemming Friche Rodler. 2001. Cuckoo hashing. In *European Symposium on Algorithms*. Springer, 121–133.

[36] Arpita Patra, Ashish Choudhary, and C. Pandu Rangan. 2008. Unconditionally Secure Multiparty Set Intersection Re-Visited. *IACR Cryptology ePrint Archive* 2008 (2008), 462. http://eprint.iacr.org/2008/462

[37] Arpita Patra, Pratik Sarkar, and Ajith Suresh. 2016. Fast Actively Secure OT Extension for Short Secrets. Cryptology ePrint Archive, Report 2016/940. (2016). http://eprint.iacr.org/2016/940.

[38] Benny Pinkas, Thomas Schneider, Gil Segev, and Michael Zohner. 2015. Phasing: Private Set Intersection Using Permutation-based Hashing. In *24th USENIX Security Symposium, USENIX Security 15*, Jaeyeon Jung and Thorsten Holz (Eds.). 515–530. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/pinkas

[39] Benny Pinkas, Thomas Schneider, and Michael Zohner. 2014. Faster Private Set Intersection Based on OT Extension. In *23rd USENIX Security Symposium, USENIX Security 14*, Kevin Fu and Jaeyeon Jung (Eds.). USENIX Association, 797–812. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/pinkas

[40] Benny Pinkas, Thomas Schneider, and Michael Zohner. 2016. Scalable Private Set Intersection Based on OT Extension. Cryptology ePrint Archive, Report 2016/930. (2016). http://eprint.iacr.org/2016/930.

[41] Amanda Cristina Davi Resende and Diego F. Aranha. 2017. Unbalanced Approximate Private Set Intersection. Cryptology ePrint Archive, Report 2017/677. (2017). http://eprint.iacr.org/2017/677.

[42] Peter Rindal. libOTe: an efficient, portable, and easy to use Oblivious Transfer Library. https://github.com/osu-crypto/libOTe. (????).

[43] Peter Rindal and Mike Rosulek. 2016. Faster Malicious 2-Party Secure Computation with Online/Offline Dual Execution. In *USENIX Security 2016*. USENIX Association.

[44] Peter Rindal and Mike Rosulek. 2017. Improved Private Set Intersection Against Malicious Adversaries. In *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part I (Lecture Notes in Computer Science)*, Jean-Sébastien Coron and Jesper Buus Nielsen (Eds.), Vol. 10210. 235–259. https://doi.org/10.1007/978-3-319-56620-7_9

[45] Peter Rindal and Mike Rosulek. 2017. Malicious-Secure Private Set Intersection via Dual Execution. Cryptology ePrint Archive, Report 2017/769. (2017). http://eprint.iacr.org/2017/769.

[46] Yingpeng Sang and Hong Shen. 2008. Privacy Preserving Set Intersection Based on Bilinear Groups. In *Proceedings of the Thirty-first Australasian Conference on Computer Science - Volume 74 (ACSC '08)*. Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 47–54. http://dl.acm.org/citation.cfm?id=1378279.1378290

[47] Adi Shamir. 1980. On the Power of Commutativity in Cryptography. In *Automata, Languages and Programming*. 582–595.

[48] Victor Shoup. 2003. NTL: A library for doing number theory. http://www.shoup.net/ntl. (2003).

[49] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. 2017. Authenticated Garbling and Efficient Maliciously Secure Two-Party Computation. Cryptology ePrint Archive, Report 2017/030. (2017). http://eprint.iacr.org/2017/030.

[50] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. 2017. Global-Scale Secure Multiparty Computation. Cryptology ePrint Archive, Report 2017/189. (2017). http://eprint.iacr.org/2017/189.

[51] Andrew Chi-Chih Yao. 1986. How to Generate and Exchange Secrets (Extended Abstract). In *27th FOCS*. IEEE Computer Society Press, Toronto, Ontario, Canada, 162–167.

[52] Moti Yung. 2015. From Mental Poker to Core Business: Why and How to Deploy Secure Computation Protocols? https://www.sigsac.org/ccs/CCS2015/pro_keynote.html. (2015). ACM CCS 2015 Keynote Talk.

A THE AUGMENTED SEMI HONEST MODEL

The unconditional zero-sharing protocol is secure in the augmented semi-honest model. Informally, in this model the parties controlled by the adversary are allowed to change their inputs at the beginning of the computation. (The main "power" given to the simulator in proofs in this model, is that after reading the inputs of the parties from their input tapes it can change them before sending inputs to the trusted party.)

The reason for the usage of this model, is the star-like communication infrastructure that is used by the protocols, where all parties independently interact with a single party (the "dealer"). A star structure is a very appealing communication pattern, since it does not require most parties to interact with each other or to coordinate a time in which they are all online. However, it is clear (as was demonstrated by a lower bound of [15]) that a coalition of the dealer with some corrupt parties can "save" the state of the protocol after the interaction between all honest parties and the dealer, and then continue running the protocol from that state using different options for the inputs of the corrupt parties. Note, however, that in the PSI setting the only useful input for the corrupt coalition is where the input of all its members is equal to the input that the dealer used in its interactions with the honest parties. Therefore even though they can choose other inputs and run the protocol with those inputs, they know in advance that the corresponding output will be the empty set.

This additional "power" that is given to the adversary is essential in our protocol since (in order to keep a simple communication infrastructure) not all parties interact with each other. Therefore two corrupt parties which only interact with each other may assume the power to "use" any input they would like during their execution.

We emphasize, though, that the in our protocol the corrupt parties can only set their input once, and that the only "useful" input strategy that they can use when computing the multi-party PSI functionality, is to use the same input set for all corrupt parties, since any value which is not in the intersection of the inputs of the corrupt parties will surely not be in the final PSI output.

A detailed discussion of the "power" of the augmented semi-honest model can be found in [16] Sec. 2.2.3. We present here the formal definition of this model (Def. 7.4.24 of [14]). We note that this model was implicitly used by multiple other works related to OT, such as the private equality test protocol in [13] or the multi-party PSI protocol in [12].

Definition A.1. (the augmented semi-honest model): Let Π be a two-party protocol. An augmented semi-honest behavior (with respect to Π) is a (feasible) strategy that satisfies the following conditions:

- Entering the execution *(this is the only part of the definition which differs than the standard definition of semi-honest behavior)*: Depending on its initial input, denoted u, the party may abort before taking any step in the execution of Π. Otherwise it enters the execution with any input $u' \in \{0,1\}^{|u|}$ of its choice. From this point on, u' is fixed.
- Proper selection of a random-tape: The party selects the random tape to be used in Π uniformly among all strings of the length specified by Π. That is, the selection of the random-tape is exactly as specified by Π.
- Proper message transmission or abort: In each step of Π, depending on its view of the execution so far, the party may either abort or send a message as instructed by Π. We stress that the message is computed as Π instructs based on input u', the selected random-tape and all the messages received so far.
- Output: At the end of the interaction, the party produces an output depending on its entire view of the interaction. We stress that the view consists of the initial input u, the selected random tape, and all the messages received so far.

B HASHING SCHEMES AND PARAMETER ANALYSIS

In this section we describe a new variant of Cuckoo hashing that avoids a stash. We analyze its parameters.

There are three parameters[22] that affect the Cuckoo hashing failure probability: the number of bins ζn, the number of hash functions h, and the stash size s. Let $\Pr_{n,\zeta,h}(S \geq s)$ denote the probability that when hashing n items into ζn bins (for $1 < \zeta < 2$) using h hash functions, the stash size exceeds s. [40] proved that asymptotically, $\Pr_{n,\zeta,h}(S \geq s) = O(n^{(1-h)(s+1)})$ when $h \geq 2\zeta \ln(\frac{e}{\zeta-1})$.

Our new variant works as follows to insert an item x. There are $(\zeta_1 + \zeta_2)n$ bins.

- First, use traditional Cuckoo hashing with h_1 hash functions to insert x into one of the first $\zeta_1 n$ bins.
- If the first phase fails, then use Cuckoo hashing with $h_2 = 2$ hash functions to insert the final evicted item into the last $\zeta_2 n$ bins.

The overall procedure fails if the second phase fails to find a suitable location for the final item. Note that the probability that s items will require a second phase of hashing is exactly $\Pr_{n,\zeta_1,h_1}(S \geq s)$. Hence, the failure probability of the overall procedure is:

$$\begin{aligned}\Pr_{n,\zeta_1,\zeta_2}(S \geq 0) &= \sum_{s=1}^{n} \Big(\Pr_{n,\zeta_1,h_1}(S_1 \geq s) \Pr_{s,\zeta_2,h_2=2}(S_2 \geq 0) \Big) \qquad (2)\\ &= \sum_{s=1}^{n} \big(O(n^{(1-h_1)(s+1)}) O(s^{-1}) \big)\\ &= \sum_{s=1}^{n} O(\frac{n^{(1-h_1)(s+1)}}{s})\\ &\leq \sum_{s=1}^{\infty} O(\frac{n^{(1-h_1)(s+1)}}{s})\\ &\leq O(n^{1-h_1} \log_2(\frac{n^{h_1}}{n^{h_1}-n}))\end{aligned}$$

Equation 2 allows us empirically estimate a concrete failure probability given a set of parameters $\{n, h_1, h_2, \zeta_1, \zeta_2\}$. We first fix the number of hash functions $h_1 = 3$, and determine necessary the scale of bins ζ_1, ζ_2 such that no stash is required (*i.e.* $s = 0$) except with probability $< 2^{-\lambda}$.

To obtain concrete numbers of ζ_1 when ζ_2 fixed, we run 2^{30} repetition of our Cuckoo hashing scheme, where we mapped $n \in \{2^7, 2^8, 2^9, 2^{10}, 2^{11}\}$ items to $n\zeta_1$ bins using h_1 hash functions and then mapping all failed items to $n\zeta_2$ bins using h_2 hash functions. We recorded the scale ζ_1 in Figure 12 with the solid line. To achieve the failure probability for larger n, we use linear regression by a variable $n' = n^{-2} \log_2(\frac{n^3}{n^3-n})$ to extrapolate the ζ_1. We substitute n' back to n and show the relationship between n and the predicted ζ_1 by the dash line in Figure 12. Table 2 shows the extrapolated scale ζ_1 for the Cuckoo hashing failure probability $\{2^{30}, 2^{40}\}$. We observe that for $n = 2^{20}$, our hashing scheme needs 1.3n bins with no stash size.

Simple hashing bounds. Moreover, we also need to guarantee that the maximum bin size β_1, β_2 is small when using Simple hashing to map n items to $n\zeta_1$ bins and $n\zeta_2$ bins with no overflow. [40]

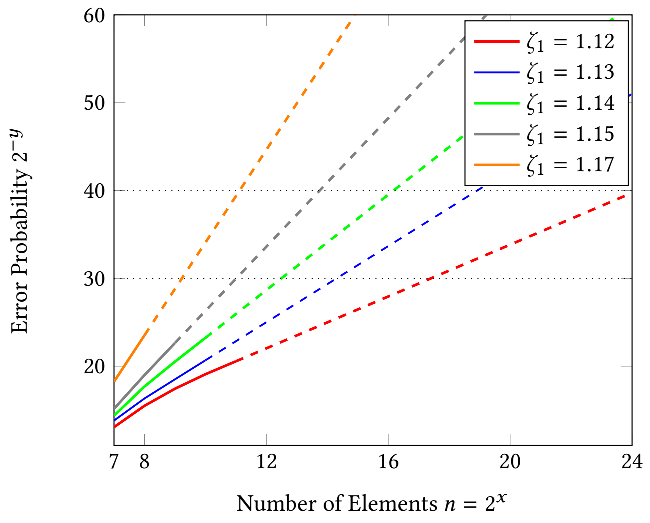

Figure 12: Required number of bins $n\zeta_1$ in first step of bucket allocation of our hashing scheme. The solid lines shows the actual measurements, the dashed lines were extrapolated using linear regression.

shows that the probability of "n balls are mapped at random to m bins, and the most occupied bin has at least k balls" is

$$\Pr(\exists \text{bin with} \geq k \text{ balls}) \leq m(\frac{en}{mk})^k \tag{3}$$

We evaluate Eq. 3 with the set sizes $n \in \{2^{12}, 2^{16}, 2^{20}, 2^{24}\}$, and depict the maximum bin size $\{\beta_i \mid i \in \{1, 2\}\}$ for the Simple hashing failure probability $\{2^{30}, 2^{40}\}$ in Table 2.