Daniel Demmler*, Peter Rindal, Mike Rosulek, and Ni Trieu

PIR-PSI: Scaling Private Contact Discovery

Abstract: An important initialization step in many social-networking applications is contact discovery, which allows a user of the service to identify which of its existing social contacts also use the service. Naïve approaches to contact discovery reveal a user's entire set of social/professional contacts to the service, presenting a significant tension between functionality and privacy.
In this work, we present a system for *private* contact discovery, in which the client learns *only* the intersection of its own contact list and a server's user database, and the server learns only the (approximate) size of the client's list. The protocol is specifically tailored to the case of a small client set and large user database. Our protocol has provable security guarantees and combines new ideas with state-of-the-art techniques from private information retrieval and private set intersection.
We report on a highly optimized prototype implementation of our system, which is practical on real-world set sizes. For example, contact discovery between a client with 1024 contacts and a server with 67 million user entries takes 1.36 sec (when using server multi-threading) and uses only 4.28 MiB of communication.

DOI 10.1515/popets-2018-0037
Received 2018-02-28; revised 2018-06-15; accepted 2018-06-16.

1 Introduction

With the widespread use of smartphones in the last decade, social networks and their connected instant messaging services are on the rise. Services like Facebook Messenger or WhatsApp connect more than a billion users worldwide.[1]

***Corresponding Author: Daniel Demmler:** TU Darmstadt , E-mail: daniel.demmler@crisp-da.de
Peter Rindal: Oregon State University, E-mail: rosulekm@engr.orst.edu
Mike Rosulek: Oregon State University, E-mail: rindalp@oregonstate.edu
Ni Trieu: Oregon State University, E-mail: trieun@oregonstate.edu

1 https://blog.whatsapp.com/10000631/Connecting-One-Billion-Users-Every-Day

Contact discovery happens when a client initially joins a social network and intends to find out which of its existing contacts are also on this network. Even after this initial client join, it is also run periodically (*e.g.*, daily) in order to capture a client's contacts that join the network later on. A trivial approach for contact discovery is to send the client's entire address book to the service provider, who replies with the intersection of the client's contacts and the provider's customers. This obviously leaks sensitive client data to the service provider. In fact, a German court has recently ruled that such trivial contact discovery in the WhatsApp messaging service violates that country's privacy regulations [28]. Specifically, a user cannot send her contact list to the WhatsApp servers for contact discovery, without written consent of all of her contacts.

A slightly better approach (often called "naïve hashing") has the client *hash* its contact list before sending it to the server. However, this solution is insecure, since it is prone to offline brute force attacks if the input domain is small (e.g., telephone numbers). Nonetheless, naïve hashing is used internally by Facebook and was previously used for contact discovery by the Signal messaging app. The significance of a truly private contact discovery was highlighted by the creators of Signal [35].

1.1 State of the Art & Challenges

Contact discovery is fundamentally about identifying the intersection of two sets. There is a vast amount of literature on the problem of **private set intersection (PSI)**, in which parties compute the intersection of their sets without revealing anything else about the sets (except possibly their size). A complete survey is beyond the scope of this work, but we refer the reader to Pinkas et al. [39], who give a comprehensive comparison among the major protocol paradigms for PSI.

In contact discovery, the two parties have **sets of vastly different sizes**. The server may have 10s or 100s of millions of users in its input set, while a typical client has less than 1000.[2] However, most research on PSI is optimized for the case where two parties have sets of similar size. As a result, many PSI protocols have com-

2 A 2014 survey by Pew Research found that the average number of Facebook friends is 338 [45].

munication and computation costs that scale with the size of the larger set. For contact discovery, it is imperative that the client's effort (especially communication cost) scales *sublinearly* with the server's set size. Concretely, in a setting where the client is a mobile device, we aim for communication of at most a few megabytes. A small handful of works [8, 29, 41] focus on PSI for asymmetric set sizes. We give a comprehensive comparison of these works to ours in Sect. 1.3 and Sect. 7.

Even after solving the problems related to the client's effort, the computational cost to the server can also be prohibitive. For example, the server might have to perform expensive exponentiations for each item in its set. Unfortunately no known techniques allow the server to have computational cost sublinear in the size of its (large) input set. The best we can reasonably hope for (which we achieve) is for the server's computation to consist almost entirely of fast symmetric-key operations which have hardware support in modern processors.

If contact discovery were a one-time step only for new users of a service, then the difference between a few seconds in performance would not be a significant concern. Yet, *existing users* must also perform contact discovery to maintain an up-to-date view. Consider a service with 100 million users, each of which performs maintenance contact discovery once a week. This is only possible if the marginal cost of a contact discovery instance costs less than 6 milliseconds for the service provider (one week is roughly 600 million milliseconds)! To be truly realistic and practical, private contact discovery should be as fast as possible.

1.2 Overview of Results & Contributions

We propose a new approach for private contact discovery that is practical for realistic set sizes. We refer to our paradigm as **PIR-PSI**, as it combines ideas from private information retrieval (PIR) and standard 2-party PSI.

Techniques: Importantly, we split the service provider's role into two to four non-colluding servers. With 2 servers, each one holds a copy of the user database. When a third server is used, 2 of the servers can hold secret shares of the user database rather than hold it in the clear. With 4 servers, all servers can hold secret shares. By using a computational PIR scheme, a single-server solution is possible. Most of our presentation focuses on our main contribution, the simpler 2-server version, but in Sect. 8 we discuss the other variants in detail. Note that multiple non-colluding servers is the traditional setting for PIR, and is what allows our approach to have sublinear cost (in the large server set size) for the client while still hiding its input set.

Roughly speaking, we combine highly efficient state-of-the-art techniques from 2-server PIR and 2-party private set intersection. The servers store their sets in a Cuckoo table so that a client with n items needs to probe only $O(n)$ positions of the server's database to compute the intersection. Using the state-of-the-art PIR scheme of Boyle, Gilboa & Ishai [4, 5], each such query requires $O(\kappa \log N)$ bits of communication, where N is the size of the server's data. In standard PIR, the client learns the positions of the server's data in the clear. To protect the servers' privacy, we modify the PIR scheme so that *one of the servers* learns the PIR output, but blinded by masks known to the client. This server and the client can then perform a standard 2-party PSI on masked values. For this we use the efficient PSI scheme of [30].

Within this general paradigm, we identify several protocol-level and systems-level optimizations that improve performance over a naïve implementation by several orders of magnitude. For example, the fact that the client probes *randomly distributed* positions in the server's database (a consequence of Cuckoo hashing with random hash functions) leads to an optimization that reduces cost by approximately 500×.

As a contribution of independent interest, we performed an extensive series of experiments (almost a trillion hashing instances) to develop a predictive formula for computing ideal parameters for Cuckoo hashing. This allows our protocol to use very tight hashing parameters, which can also yield similar improvements in all other cuckoo hashing based PSI protocols. A more detailed description can be found in Section 3.2.

Performance: Let n be the size of the client's set, let N the size of the server's set ($n \ll N$), and let κ be the computational security parameter. The total communication for contact discovery is $O\big(\kappa n \log(N \log n/\kappa n)\big)$. The computational cost for the client is $O\big(n \log(N \log n/\kappa n)\big)$ AES evaluations, $O(n)$ hash evaluations, and κ exponentiations. The exponentiations can be done once-and-for all in an initialization phase and re-used for subsequent contact discovery events between the same parties.

Each server performs $O\big((N \log n)/\kappa\big)$ AES evaluations, $O(n)$ hash evaluations, and κ (initialization-time) exponentiations. While this is indeed a large number of AES calls, hardware acceleration (i.e., AES-NI instructions and SIMD vectorization) can easily allow processing of a billion items per second per thread on typical server hardware. Furthermore, the server's computational effort in PIR-PSI is highly parallelizable, and we explore the effect of parallelization on our protocol.

Privacy/security for the client: If a corrupt server does not collude with the other server, then it learns nothing about the client's input set except its size. In the case where the two servers collude, even if they are malicious (*i.e.*, deviating arbitrarily from the protocol) they learn no more than the client's *hashed* input set. In other words, **the failure mode for PIR-PSI is to reveal no more than the naïve-hashing approach.** Since naïve-hashing is the status quo for contact discovery in practice, PIR-PSI is a strict privacy improvement.

Furthermore, the non-collusion guarantee is forward-secure. Compromising both servers leaks nothing about contact-discovery interactions that happened previously (when at most one server was compromised).

In PIR-PSI, malicious servers can use a different input set for each client instance (*e.g.*, pretend that Alice is in their database when performing contact discovery with Bob, but not with Carol). That is, the servers' effective data set is not anchored to some public root of trust (*e.g.*, a signature or hash of the "correct" data set).

Privacy/security for the server: A semi-honest client (*i.e.*, one that follows the protocol) learns no more than the intersection of its set with the servers' set, except its size. A malicious client can learn different information, but still no more than $O(n)$ bits of information about the servers' set (n is the purported set size of the client). We can characterize precisely what kinds of information a malicious client can learn.

Other features: PIR-PSI requires the servers to store their data-set in a fairly standard Cuckoo hashing table. Hence, the storage overhead is constant and updates take constant time.

PIR-PSI can be easily extended so that the client privately learns associated data for each item in the intersection. In the case of a secure messaging app, the server may hold a mapping of email addresses to public keys. A client may wish to obtain the public keys of any users in its own address book.

As mentioned previously, PIR-PSI can be extended to a 3-server or 4-server variant where some of the servers hold only *secret shares* of DB, with security holding if no two servers collude (cf. Sect. 8). This setting may be a better fit for practical deployments of contact discovery, since a service provider can recruit the help of other independent organizations, neither of which need to know the provider's user database. Holding the user database in secret shared form reduces the amount of data that the service provider retains about its users[3] and gives stronger defense against data exfiltration.

1.3 Related Work & Comparison

PSI with asymmetric set sizes: As discussed in the previous section, many protocols for private set intersection are not well-suited when the two parties have input sets of very different sizes. For example, [30, 39] are the fastest PSI protocols for large sets of similar size, but require communication at least $O\big(\lambda(N+n)\big)$ where λ is a statistical security parameter. This cost makes these approaches prohibitive for contact discovery, where N is very large.

Only a handful of works specifically consider the case of asymmetric set sizes. Chen, Laine & Rindal (CLR) [8] use somewhat-homomorphic encryption to reduce communication to logarithmic in the large set size N. Kiss et al. (KLSAP) [29] describe an approach that defers $O(N)$ communication to a pre-processing phase, in which the server sends a large Bloom filter containing $\mathsf{AES}(k,x)$ for each of its items x. To perform contact discovery, the parties use Yao's protocol to obliviously evaluate AES on each of the client's items, so the client can then probe the Bloom filter for membership. Resende and Aranha (RA) [41] describe a similar approach in which the server sends a large message during the pre-processing phase. In this case, the large message is a more space-efficient Cuckoo filter.

In Sect. 7 we give a detailed comparison between these works and PIR-PSI. The main qualitative difference is the security model. The protocols listed above are in a two-party setting involving a single server, whereas PIR-PSI involves several *non-colluding* servers. We discuss the consequences of this security model more thoroughly in Sect. 4. Besides this difference, the KLSAP & RA protocols require significant offline communication and persistent storage for the client.

Keyword PIR: Chor, et al. [9] defined a variant of PIR called *keyword PIR,* in which the client has an item x, the server has a set S, and the client learns whether $x \in S$. Our construction can be viewed as a kind of multi-query keyword PIR with symmetric privacy guarantee for the server (the client learns *only* whether $x \in S$). In [5, Appendix A] a similar method is proposed that could be used to implement private contact discovery.

[3] https://www.reuters.com/article/us-usa-cyber-signal/signal-messaging-app-turns-over-minimal-data-in-first-subpoena-idUSKCN1241JM

The keyword-PIR method would have computation cost of $O(Nn\rho)$ AES invocations, where the items in the parties' sets come from the set $\{0,1\}^\rho$. By contrast, our protocol has server computation cost $O(N \log n)$. We can imagine all parties first hashing their inputs down to a small ρ-bit fingerprint before performing contact discovery. But with N items in the server's set, we must have $\rho \geq s+2\log N$ to limit the probability of a collision to 2^{-s}. In practice $\rho > 60$ is typical, hence $n\rho$ is much larger than $\log n$.

Signal's private contact discovery.: Recently the Signal messaging service announced a solution for private contact discovery based on Intel SGX, which they plan to deploy soon [34]. The idea is for the client to send their input set directly into an SGX enclave on the server, where it is privately compared to the server's set. The enclave can use remote attestation to prove the authenticity of the server software.

The security model for this approach is incomparable to ours and others, as it relies on a trusted hardware assumption. Standard two-party PSI protocols rely on standard cryptographic hardness assumptions. Our PIR-PSI protocol relies on cryptographic assumptions as well as an assumption of non-collusion between two servers. It is also worth pointing out that commercial uses of Intel SGX currently require a license and that there is ongoing research that focuses on applying side-channel attacks like Spectre and Meltdown to extract confidential data from SGX enclaves [32].

2 Preliminaries

Throughout the paper we use the following notation: The large server set has N items, while the small client set has n items. The length of the item is ρ bits. In our implementation we use $\rho = 128$ bits. We write $[m] = \{1, \ldots, m\}$. The computational and statistical security parameters are denoted by κ, λ, respectively. In our implementation we use $\kappa = 128$ and $\lambda = 40$.

2.1 Secure computation

Secure two-party computation allows two distrusting parties to compute a joint function on their inputs without revealing any additional information except for the result itself. In 1982, two different approaches [23, 47] have been introduced and opened up this field of research. Over the last decade, secure computation has been widely studied and applied in a variety of applications. We consider security in the presence of an adversary that can be semi-honest (passive) or malicious (active). A semi-honest adversary follows the protocol without deviation, but tries to infer information about private content from the exchanged messages. A malicious adversary can arbitrarily deviate from the protocol and modify, re-order or suppress messages in order to break the protocol's security.

Table 1. Notation: Parameters and symbols used.

Parameter	Symbol
symmetric security paramter [bits]	$\kappa = 128$ bits
statistical security paramter [bits]	$\lambda = 40$ bits
element length [bits]	$\rho = 128$ bits
client set size [elements]	n
server set size [elements]	N
cuckoo table expansion (Sect. 3.2)	e
cuckoo table size [elements] (Sect. 3.2)	$m = e \cdot N$
DPF bins (Sect. 3.4)	β
DPF bin size [elements] (Sect. 3.4)	μ
PIR block size (Sect. 3.5)	b
scaling factor (Sect. 5.3)	c
server database	DB
server cuckoo hash table	CT

2.2 Private Information Retrieval

Private Information Retrieval (PIR) was introduced in the 1990s by Chor et al. [10]. It enables a client to query information from one or multiple servers in a privacy preserving way, such that the servers are unable to infer which information the client requested. In contrast to the query, the servers' database can be public and may not need to be protected. When first thinking about PIR, a trivial solution is to have a server send the whole database to the client, who then locally performs his query. However, this is extremely inefficient for large databases. There exists a long list of works that improve PIR communication complexity [1, 4–6, 13, 14, 16, 20, 24–27, 33, 36, 46]. Most of these protocols demand multiple non-colluding servers. In this work, we are interested in 2-server PIR schemes.

2.3 Distributed Point Functions

Gilboa and Ishai [22] proposed the notion of a distributed point function (DPF). For our purposes, a DPF with domain size N consists of the following algorithms:

DPF.Gen:a randomized algorithm that takes index $i \in [N]$ as input and outputs two (short) *keys* k_1, k_2.

DPF.Expand: takes a short key k as input and outputs a long *expanded key* $K \in \{0,1\}^N$.[4]

The correctness property of a DPF is that, if $(k_1, k_2) \leftarrow \mathsf{DPF.Gen}(i)$ then $\mathsf{DPF.Expand}(k_1) \oplus \mathsf{DPF.Expand}(k_2)$ is a string with all zeros except for a 1 in the ith bit.

A DPF's security property is that the marginal distribution of k_1 alone (resp. k_2 alone) leaks no information about i. More formally, the distribution of k_1 induced by $(k_1, k_2) \leftarrow \mathsf{DPF.Gen}(i)$ is computationally indistinguishable from that induced by $(k_1, k_2) \leftarrow \mathsf{DPF.Gen}(i')$, for all $i, i' \in [N]$.

PIR from DPF: Distributed point functions can be used for 2-party PIR in a natural way. Suppose the servers hold a database DB of N strings. The client wishes to read item $DB[i]$ without revealing i. Using a DPF with domain size N, the client can compute $(k_1, k_2) \leftarrow \mathsf{DPF.Gen}(i)$, and send one k_b to each server. Server 1 can expand k_1 as $K_1 = \mathsf{DPF.Expand}(k_1)$ and compute the inner product:

$$K_1 \cdot DB \overset{\text{def}}{=} \bigoplus_{j=1}^{N} K_1[j] \cdot DB[j]$$

Server 2 computes an analogous inner product. The client can then reconstruct as:

$$(K_1 \cdot DB) \oplus (K_2 \cdot DB) = (K_1 \oplus K_2) \cdot DB = DB[i]$$

since $K_1 \oplus K_2$ is zero everywhere except in position i.

BGI construction: Boyle, Gilboa & Ishai [4, 5] describe an efficient DPF construction in which the size of the (unexpanded) keys is roughly $\kappa(\log N - \log \kappa)$ bits, where κ is the computational security parameter.

Their construction works by considering a full binary tree with N leaves. To expand the key, the $\mathsf{DPF.Expand}$ algorithm performs a PRF evaluation for each node in this tree. The (unexpanded) keys contain a PRF block for each level of the tree.

As described, this gives unexpanded keys that contain κ bits for each level of a tree of height $\log N$. To achieve $\kappa(\log N - \log \kappa)$ bits total, BGI suggest the following "early termination optimization": Treat the expanded key as a string of N/κ characters over the alphabet $\{0,1\}^\kappa$. This leads to a tree of height $\log(N/\kappa) = \log N - \log \kappa$. An extra κ-bit value is required to deal with the longer characters at the leaves, but overall the total size of the unexpanded keys is roughly $\kappa(\log N - \log \kappa)$ bits. In practice, we use hardware-accelerated AES-NI as the underlying PRF, with $\kappa = 128$.

4 The original DPF definition also requires efficient random access to this long expanded key. Our usage of DPF does not require this feature.

PARAMETERS: Set sizes m and n; Two parties: sender $\mathcal{S}$ and receiver $\mathcal{R}$

FUNCTIONALITY:
- Wait for an input $X = \{x_1, x_2, \ldots, x_n\} \subseteq \{0,1\}^*$ from sender $\mathcal{S}$ and an input $Y = \{y_1, y_2, \ldots, y_m\} \subseteq \{0,1\}^*$ from receiver $\mathcal{R}$
- Give output $X \cap Y$ to the receiver $\mathcal{R}$.

Fig. 1. Private Set Intersection Functionality $\mathcal{F}_{\mathsf{psi}}^{m,n}$

2.4 Private Set Intersection (PSI)

Private Set Intersection (PSI) is an application of secure computation that allows parties, each holding a set of private items, to compute the intersection of their sets without revealing anything except for the intersection itself. We describe the ideal functionality for PSI in Fig. 1. Based on oblivious polynomial evaluation, the first PSI protocol was formally introduced in 2004 by [18]. However, this protocol requires a quadratic number of expensive public key operations. Over the last decade, many PSI protocols [8, 11, 12, 17, 29, 30, 37–41, 44] were proposed with linear (or even sub-linear) communication and computation complexity, which made PSI become practical for many applications. These PSI protocols can be classified into two different settings based on the size of party's input set: (1) symmetric, where the sets size have approximately the same size; (2) asymmetric, where one of the sets is severely smaller than the other.

The most efficient PSI approaches [30, 39] for symmetric sets are based on efficient Oblivious Transfer (OT) extension.

3 Our Construction: PIR-PSI

We make use of the previously described techniques to achieve a practical solution for privacy-preserving contact discovery, called **PIR-PSI**. We assume that the service provider's large user database is held on 2 separate servers. To perform private contact discovery, a client interacts with both servers simultaneously. The protocol's best security properties hold when these two servers do not collude. Variants of our construction for 3 and 4 servers are described in Sect. 8, in which some of the servers hold only *secret shares* of the user database.

We develop the protocol step-by-step in the following sections. The full protocol can be found in Fig. 2.

3.1 Warmup: PIR-PEQ

At the center of our construction is a technique for combining a *private equality test* (PEQ – a special case of PSI when the parties have one item each) [38] with a PIR query. Suppose a client holds private input i, x and wants to learn whether $DB[i] = x$, where the database DB is the private input of the servers.

First recall the PIR scheme from Sect. 2.2 based on DPFs. This PIR scheme has *linear reconstruction* in the following sense: the client's output $DB[i]$ is equal to the *XOR of the responses* from the two servers.

Suppose a PIR scheme with linear reconstruction is modified as follows: the client sends an additional mask r to server #1. Server #1 computes its PIR response v_1 and instead of sending it to the client, sends $v_1 \oplus r$ to server #2. Then server #2 computes its PIR response v_2 and can reconstruct the masked result $v_2 \oplus (v_1 \oplus r) = DB[i] \oplus r$. We refer to this modification of PIR as **designated-output PIR**, as the client designates server #2 to learn the (masked) output.

The client can now perform a standard 2-party secure computation with server #2. In particular, they can perform a PEQ with input $x \oplus r$ from the client and $DB[i] \oplus r$ from the server. As long as the PEQ is secure, and the two servers do not collude, then the servers learn nothing about the client's input. If the two servers collude, they can learn i but not x.

This warm-up problem is not yet sufficient for computing private set intersection between a set X and DB, since the client may not know which location in DB to test against. Next we will address this by structuring the database as a Cuckoo hash table.

3.2 Cuckoo hashing

Cuckoo hashing has seen extensive use in Private Set Intersection protocols [30, 37–40] and in related areas such as privacy preserving genomics [7]. This hashing technique uses an array with m entries and k hash functions $h_1, \ldots, h_k : \{0,1\}^* \to [m]$. The guarantee is that an item x will be stored in a hash table at one of the locations indexed by $h_1(x), ..., h_k(x)$. Furthermore, only a single item will be assigned to each entry. Typically, k is quite small (we use $k = 3$). When inserting x into the hash table, a random index $i \in [k]$ is selected and x is inserted at location $h_i(x)$. If an item y currently occupies that location, it is evicted and y is re-inserted using the same technique. This process is repeated until all items are inserted or some upper bound on the number of trials have been reached. In that latter case, the procedure can either abort or place the item in a special location called the stash. We choose cuckoo hashing parameters such that this happens with sufficiently low probability (see Sect. 5.2 and Appendix B), i.e., no stash is required.

In our setting the server encodes its set DB into a Cuckoo hash table CT of size $m = e \cdot N$, where $e > 1$ is an expansion factor. That way, the client (who has the much smaller set X) must probe only $k|X|$ positions of the Cuckoo table to compute the intersection. Using the PIR-PEQ technique just described makes the communication linear in $|X|$ but only logarithmic in $|DB|$.

3.3 Hiding the cuckoo locations

There is a subtle issue if one applies the PIR-PEQ idea naïvely. When the client learns that $y \in (DB \cap X)$, he/she will in fact learn whether y is placed in position $h_1(y)$ or $h_2(y)$ or $h_3(y)$ of the Cuckoo table. But this *leaks more than the intersection* $DB \cap X$, in the sense that it cannot be simulated given just the intersection! The placement of y in the Cuckoo table CT depends indirectly on all the items in DB.[5] Note that this is not a problem for other PSI protocols, since there the party who processes their input with Cuckoo hashing is the one who receives output from the PEQs. For contact discovery, we require these to be different parties.

To overcome this leakage, we design an efficient oblivious shuffling procedure that obscures the cuckoo location of an item. First, let us start with a simple case with two hash functions h_1, h_2, where the client holds a single item x. This generalizes in a natural way to $k = 3$ hash functions. Full details are provided in Appendix A.

The client will generate and send two PIR queries, for positions $h_1(x)$ and $h_2(x)$ in CT. The client also sends two masks r_1 and r_2 to server #1 which serve as masks for the designated-output PIR. Server #1 randomly chooses whether to swap these two masks. That is, it chooses a random permutation $\sigma : \{1,2\} \to \{1,2\}$ and masks the first PIR query with $r_{\sigma(1)}$ and the second with

5 For instance, say the client holds set X and (somehow) knows the server has set $DB = X \cup \{z\}$ for some unknown z. It happens that for many $x \in X$ and $i \in [k]$, $h_i(x)$ equals some location ℓ. Then with good probability some $x \in X$ will occupy location ℓ. However, after testing location ℓ the client learns no $x \in X$ occupies this location. Then the client has learned some information about z (namely, that $h_i(z) = \ell$ is likely for some $i \in [k]$), even though z is not in the intersection.

$r_{\sigma(2)}$. Server #2 then reconstructs the designated PIR output, obtaining $CT[h_1(x)] \oplus r_{\sigma(1)}, CT[h_2(x)] \oplus r_{\sigma(2)}$. The client now knows that if $x \in DB$, then server #2 must hold either $x \oplus r_1$ or $x \oplus r_2$.

Now instead of performing a private equality test, the client and server #2 can perform a standard **2-party PSI** with inputs $\{x \oplus r_1, x \oplus r_2\}$ from the client and the designated PIR values $\{CT[h_1(x)] \oplus r_{\sigma(1)}, CT[h_2(x)] \oplus r_{\sigma(2)}\}$ from server #2. This technique perfectly hides whether x was found at $h_1(x)$ or $h_2(x)$. While it is possible to perform a separate 2-item PSI for each PIR query, it is actually more efficient (when using the 2-party PSI protocol of [30]) to combine all of the PIR queries into a single PSI with $2n$ elements each.

Because of the random masks, this approach may introduce a false positive, where $CT[j] \neq x$ but $CT[j] \oplus r = x \oplus r'$ (for some masks r and r'), leading to a PSI match. In our implementation we only consider items of length 128, so the false positive probability taken over all client items is only $2^{-128+\log_2((2n)^2)}$, by a standard union bound.

3.4 Optimization: Binning queries

The client probes the servers' database in positions that are determined by the Cuckoo hash functions. Under the reasonable assumptions that (1) the client's input items are chosen independently of the Cuckoo hash functions and (2) the cuckoo hash functions are random functions, the client probes the cuckoo table CT in *uniformly distributed* positions.

Knowing that the client's queries are randomly distributed in the database, we can take advantage of the fact that the queries are "well-spread-out" in some sense. Consider dividing the server's CT ($m = N \cdot e$ entries) into β bins of equal size. The client will query the database in nk random positions, so the distribution of these queries into the β bins can be modeled as a standard balls and bins problem. We can choose a number β of bins and a maximum load μ so that Pr[there exists a bin with $\geq \mu$ balls] is below some threshold (say, 2^{-40} in practice). With such parameters, the protocol can be optimized as follows.

The parties agree to divide CT into β regions of equal size. The client computes the positions of CT that he/she wishes to query, and collects them according to their region. The client adds dummy PIR queries until there are *exactly* μ queries to each region. The dummy items are necessary because revealing the number of (non-dummy) queries to each region would leak information about the client's input to the server. For each region, the server treats the relevant m/β items as a sub-database, and the client makes exactly μ PIR queries to that sub-database.

This change leads to the client making more PIR queries than before (because of the dummy queries), but each query is made to a much smaller PIR instance. Looking at specific parameters shows that binning can give **significant performance improvements.**

It is well-known that with $\beta = O\big(nk/\log(nk)\big)$ bins, the maximum number of balls in any bin is $\mu = O\big(\log(nk)\big)$ with very high probability. The total number of PIR queries (including dummy ones) is $\beta\mu = \Theta(nk)$. That is, the binning optimization with these parameters increases the number of PIR queries by a constant factor. At the same time, the PIR database instances are all smaller by a large factor of $\beta = O\big(nk/\log(nk)\big)$. The main bottleneck in PIR-PSI is the computational cost of the server in answering PIR queries, which scales linearly with the size of the PIR database. Reducing the size of all effective PIR databases by a factor of β has a significant performance impact. In general, tuning the constant factors in β (and corresponding μ) gives a wide trade-off between communication and computation.

3.5 Optimization: Larger PIR Blocks

So far we have assumed a one-to-one correspondence between the entries in the server's cuckoo table and the server's database for purposes of PIR. That is, we invoke PIR with an v-item database corresponding to a region of the cuckoo table with v entries.

Suppose instead that we use PIR for an $v/2$-item database, where each item in the PIR database consists of a block of 2 cuckoo table entries. The client generates each PIR query for a single item, but now the PIR query returns a block of 2 cuckoo table entries. The server will feed both entries into the 2-party PSI, so that these extra neighboring items are not leaked to the client.

This change affects the various costs in the following ways: (1) It reduces the number of cryptographic operations needed for the server to answer each PIR query by half; (2) It does not affect the computational cost of the final inner product between the expanded DPF key and PIR database entries, since this is over the same amount of data; (3) It reduces the communication cost of each PIR query by a small amount (κ bits); (4) It doubles all costs of the 2-party PSI, since the server's PSI input size is doubled.

Of course, this approach can be generalized to use a PIR blocks of size b, so that a PIR database of size v/b is used for v cuckoo table entries. This presents a trade-off between communication and computation, discussed further in in Sect. 5.

3.6 Asymptotic Performance

With these optimizations the computational complexity for the client is the generation of $\beta\mu = O(n)$ PIR queries of size $O(\log(N/\kappa\beta))$. As such they perform $O(n\log(N/\kappa\beta)) = O(n\log(N\log n/\kappa n))$ calls to a PRF and send $O(\kappa n\log(N/(\kappa n\log n)))$ bits. The servers must expand each of these queries to a length of $O(N/\beta)$ bits which requires $O(N\mu/\kappa) = O(N\log n/\kappa)$ calls to a PRF.

4 Security

4.1 Semi-Honest Security

The most basic and preferred setting for PIR-PSI is when at most one of the parties is passively corrupt (a.k.a. semi-honest). This means that the corrupt party does not deviate from the protocol. Note that restricting to a single corrupt party means that we assume *non-collusion* between the two PIR servers.

Theorem 1. *The $\mathcal{F}_{\text{PIR-PSI}}$ protocol (Fig. 2) is a realization of $\mathcal{F}_{psi}^{n,N}$ secure against a semi-honest adversary that corrupts at most one party in the $\mathcal{F}_{psi}^{nk,\beta\mu}$ hybrid model.*

Proof. In the semi-honest non-colluding setting it is sufficient to show that the transcript of each party can be simulated given their input and output. That is, we consider three cases where each one of the parties is individually corrupt.

Corrupt Client: Consider a corrupt client with input X and output $Z = X \cap DB$. We show a simulator that can simulate the view of the client given just X and Z. First observe that the simulator playing the role of both servers knows the permutation $\pi = \pi_2 \circ \pi_1$ and the vector of masks r. As such, response v can be computed as follows. For $x \in Z$ the simulator randomly samples one of k masks $r_{i_1}, \dots, r_{i_k} \in r$ which the client will use to mask x and add $x \oplus r_{i_j}$ to v. Pad v with random values not contained in union u to size $\beta\mu$ and forward v to the ideal $\mathcal{F}_{\text{PSI}}^{nk,\beta\mu}$. Conditioned on no spurious collisions between v and u in the real interaction (which happen with negligible probability, following the discussion in Sect. 3.3) this ideal interaction perfectly simulates the real interaction.

One additional piece of information learned by the client is that cuckoo hashing on the set DB with hash function $h_1, ..., h_k$ succeeded. However, by the choice of cuckoo parameter, this happens with overwhelming probability and therefore the simulator can ignore the case of cuckoo hashing failure.

Corrupt server: Each server's view consists of:

– PIR queries (DPF keys) from the client; since a single DPF key leaks nothing about the client's query index, these can be simulated as dummy DPF keys.
– Messages in the oblivious masking step, which are uniformly distributed as discussed in Sect. 3.3 and Appendix A.
– In the case of server #2, masked PIR responses from server #1, which are uniformly distributed since they are masked by the $\vec{r}$ values. □

4.2 Colluding Servers

If the two servers collude, they will learn both DPF keys for every PIR query, and hence learn the locations of all client's queries into the cuckoo table. These locations indeed leak information about the client's set, although the exact utility of this leakage is hard to characterize. The servers still learn nothing from the PSI subprotocol by colluding since only one of the servers is involved.

It is worth providing some context for private contact discovery. The state-of-the-art for contact discovery is a naïve (insecure) hashing protocol, where both parties simply hash each item of their set, the client sends its hashed set to the server, who then computes the intersection. This protocol is insecure because the server can perform a dictionary attack on the client's inputs.

However, *any PSI protocol* (including ours) can be used in the following way. First, the parties hash all their items, and then use the hashed values as inputs to the PSI. As long as the hash function does not introduce collisions, pre-hashing the inputs preserves the correctness of the PSI protocol.

A side effect of pre-hashing inputs is that the parties never use their "true" inputs to the PSI protocol. Therefore, the PSI protocol cannot leak more than the hashed inputs — identical to what the status quo naïve hashing protocol leaks. Again, this observation is true for *any* PSI protocol. In the specific case of PIR-PSI, if parties pre-hash their inputs, then even if the two servers collude (even if they are malicious), the overall protocol can never leak more about the client's inputs than naïve hashing. Relative to existing solutions implemented in current applications, that use naïve hashing, there is no extra security risk for the client to use PIR-PSI.

Parameters: X is the client's set, DB is the set held by server #1 and #2, where $X, DB \subseteq \{0,1\}^\rho$. $n = |X|, N = |DB|$, k is the number of cuckoo hash functions. The protocol uses an instance of $\mathcal{F}_{\text{PSI}}$ with input length ρ. λ, κ are the statistical and computational security parameter.

1. **[Cuckoo Hashing]** Both servers agree on the same k random hash functions $h_1, ..., h_k : \{0,1\}^\rho \to [m]$ and cuckoo table size $m = |CT|$ such that inserting N items into cuckoo hash table CT succeeds with probability $\geq 1 - 2^{-\lambda}$.
   The servers compute a cuckoo hash table CT such that for all $y \in DB$, $CT[h_i(y)] = y$ for some $i \in k$.
2. **[Query]** Upon the client receiving their set X,
   (a) Send $n = |X|$ to the servers. All parties agree on the number of bins $\beta = O(n/\log n)$, and their size $\mu = O(\log n)$ (see Sect. 5.3). Define the region DB_i as all locations $j \in [m]$ of CT such that $(i-1)\frac{m}{\beta} < j \leq i\frac{m}{\beta}$.
   (b) For $x \in X, h \in \{h_1, ..., h_k\}$, let $h(x)$ index the j'th location in bin i. The client adds (x, j) to bin $B[i]$.
   (c) For bin $B[i]$,
      i. Pad $B[i]$ to size μ with the pair $(\perp, 0)$.
      ii. For $(x, j) \in B[i]$ in a random order, the client constructs the keys $k_1, k_2 = \mathsf{DPF.Gen}(j)$. Send k_s to server $\#s$.
      iii. Server $\#s$ expands their key $K_s = \mathsf{DPF.Expand}(k_s)$ and compute $v_s[\ell] = DB_i \cdot K_s$ where k_s is the ℓ'th DPF key received.
3. **[Shuffle]** Observe that, $(v_1 \oplus v_2)[\ell] = CT[j_\ell]$ where j_ℓ is the ℓ'th PIR location.
   (a) The client samples a permutation π of $\beta\mu$ items such that for the i'th $x \in X$, and $j \in [k]$ it holds that the $\pi\big((i-1)k+j\big)$'th DPF key corresponded to the query of item x at location $h_j(x)$.
   (b) The client samples $w_1, w_2 \leftarrow \{0,1\}^\kappa$ and sends w_1 to server #1, and w_2 to server #2. Define shared terms $t = \textsc{prg}(w_2)$, $(r||s||\pi_1) = \textsc{prg}(w_1)$ where r, s, t are random vectors of the same size as v_i and π_1 is a random permutation of $\beta\mu$ items. The client sends π_2 to server #2 such that $\pi_2 \circ \pi_1 = \pi$ and sends $p_0 = t \oplus \pi(s)$ to server #1.
   (c) Server #1 samples a random permutation σ of $\beta\mu$ items such that for all $i \in [\beta\mu/k]$ it holds that $\sigma(j) \in S_i$ where $j \in S_i = (i-1)k + \{1, ..., k\}$. Server #1 sends $p_1 = \pi_1\big(\sigma(r) \oplus s\big)$ and $p_2 = p_0 \oplus v_1$ to server #2.
   (d) Server #2 computes $v = v_2 \oplus \pi_2(p_1) \oplus t \oplus p_2$.
4. **[PSI]** The client then computes the masked versions of the $x_i \in X$ as $x_i' = \{x_i \oplus r_{\pi((i-1)k+1)}, ..., x_i \oplus r_{\pi(ik)}\}$ and computes u as the union of all these sets. The client and server #2 respectively send u, v to $\mathcal{F}_{\mathsf{psi}}^{nk,\beta\mu}$ such that the client receives $z = u \cap v$. The client outputs $\{x_i : x_i' \cap z \neq \emptyset\}$.

Fig. 2. Our 2-server PIR-PSI protocol $\mathcal{F}_{\text{PIR-PSI}}$.

4.3 Malicious Client

Service providers may be concerned about malicious behavior (i.e., protocol deviation) by clients during contact discovery. Since servers get no output from PIR-PSI, there is no concern over a malicious client inducing inconsistent output for the servers. The only concern is therefore what unauthorized information a malicious client can learn about DB.

Overall the only information the client receives in PIR-PSI is from the PSI subprotocol. We first observe that the PSI subprotocol we use ([30]) is naturally secure against a malicious client, when it is instantiated with an appropriate OT extension protocol. This fact has been observed in [31, 37]. Hence, in the presence of a malicious client we can treat the PSI subprotocol as an ideal PSI functionality. The malicious client can provide at most nk inputs to the PSI protocol — the functionality of PSI implies that the client therefore learns no more than nk bits of information about DB. This leakage is comparable to what an honest client would learn by having an input set of nk items.

Modifications for more precise leakage characterization: In DPF-based PIR schemes clients can make malformed PIR queries to the server, by sending k_1, k_2 so that $\mathsf{DPF.Expand}(k_1) \oplus \mathsf{DPF.Expand}(k_2)$ has more than one bit set to 1. The result of such a query will be the XOR of several DB positions.

However, Boyle et al. [5] describe a method by which the servers can ensure that the client's PIR queries (DPF shares) are well-formed. The technique increases the cost to the servers by a factor of roughly 3× (but adds no cost to the client).

The client may also send malformed values in the oblivious masking phase. But since the servers use those values independently of DB, a simulator (who sees the client's oblivious masking messages to both servers) can simulate what masks will be applied to the PIR queries. Overall, if the servers ensure validity of the client's PIR values, we know that server #1's input to PSI will consist of a collection of nk individual positions from DB, each masked with values that can be simulated.

5 Implementation

We implemented a prototype of our $\mathcal{F}_{\text{PIR-PSI}}$ protocol described in Fig. 2. Our implementation uses AES as the underlying PRF (for the distributed point function of [5]) and relies on the PSI implementation of [30] and the oblivious transfer from [42]. Our implementation is publicly available [43].

5.1 System-level Optimizations

We highlight here system-level optimizations that contribute to the high performance of our implementation. We analyze their impact on performance in Appendix C.

Optimized DPF full-domain evaluation: Recall that the DPF construction of [5] can be thought of as a large binary tree of PRF evaluations. Expanding the short DPF key corresponds to computing this entire tree in order to learn the values at the leaves. The process of computing the values of all the leaves is called "full-domain evaluation" in [5].

DPF full-domain evaluation is the major computational overhead for the servers in our protocol. To limit its impact our implementation takes full advantage of instruction vectorization (SIMD). Most modern processors are capable of performing the same instruction on multiple (*e.g.*, 8) pieces of data. However, to fully utilize this feature, special care has to be taken to ensure that the data being operated on is in cache and contiguous.

To meet these requirements, our implementation first evaluates the top 3 levels of the DPF binary tree, resulting in 8 remaining independent subtrees. We then perform SIMD vectorization to traverse all 8 subtrees simultaneously. Combining this technique with others, such as the removal of `if` statements in favor of array indexing, our final implementation is roughly 20× faster then a straight-forward (but not careless) sequential implementation and can perform full-domain DPF evaluation at a rate of 2.6 billion bits/s on a single core.

Single-pass processing: With the high raw throughput of DPF evaluation, it may not be surprising that it was no longer the main performance bottleneck. Instead, performing many passes over the dataset (once for each PIR query) became the primary bottleneck by an order of magnitude. To address this issue we further modify the workflow to evaluate all DPFs (PIR queries) for a single bin in parallel using vectorization.

That is, for all μ DPF evaluations in a given bin, we evaluate the binary trees in parallel, and traverse the leaves in parallel. The values at the leaves are used to take an inner product with the database items, and the parallel traversal ensures that a given database item only needs to be loaded from main memory (or disk) once. This improves (up to 5×) the performance of the PIR protocol on large datasets, compared to the straightforward approach of performing multiple sequential passes of the dataset.

Parallelization: Beyond the optimizations listed above, we observe that our protocol simply is very amenable to parallelization. In particular, our algorithm can be parallelized both within the DPF evaluation using different subtrees and by distributing the PIR protocols for different bins between several cores/machines. In the setting where thousands of these protocols are being executed a day on a fixed dataset, distributing bin evaluations between different machines can be extremely attractive due to the fact that several protocol instances can be batched together to gain even greater benefits of vectorization and data locality. The degree of parallelism that our protocol allows can be contrasted with more traditional PSI protocols which require several global operations, such as a single large shuffle of the server's encoded set (as in [30, 37–40]).

5.2 Cuckoo Hashing Parameters

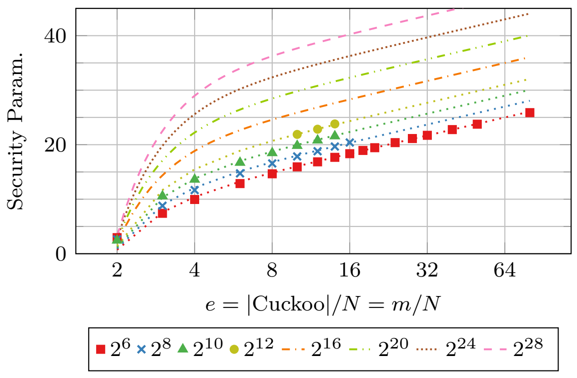

Fig. 3. Empirical (marks) and interpolated (dashed/dotted lines) cuckoo success probability for $k = 2$ hash functions and different set sizes N.

To achieve optimal performance it is crucial to minimize the size of the cuckoo table and the number of hash functions. The table size affects how much work the servers have to perform and the number of hash functions k affects the number of client queries. As such we wish to minimize both parameters while ensuring cuckoo hashing failures are unlikely as this leaks a single bit about DB. Several works [15, 19] have analyzed cuckoo hashing from an asymptotic perspective and show that the failure probability decreases exponentially with increasing table size. However, the exact relationship between the number of hash functions, stash size, table size and security parameter is unclear from such an analysis.

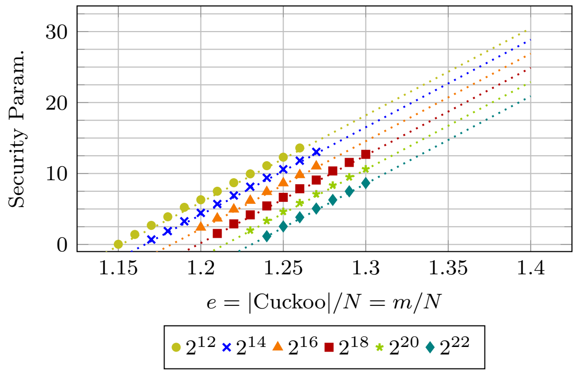

Fig. 4. Empirical (marks) and interpolated (dashed/dotted lines) cuckoo success probability for $k = 3$ hash functions and different set sizes N.

We solve this problem by providing an accurate relationship between these parameters through extensive experimental analysis of failure probabilities. That is, we ran Cuckoo-hashing instances totalling nearly 1 trillion items hashed, over two weeks for a variety of parameters. As a result our bounds are significantly more accurate and general than previous experiments [8, 39]. We analyzed the resulting distribution to derive highly predictive equations for the failure probability. We find that $k = 2$ and $k \geq 3$ behave significantly different and therefore derive separate equations for each.

Our extrapolations are graphed in Figs. 3 & 4, and the specifics of the formulas are given in Appendix B.

5.3 Parameter Selection for Cuckoo Hashing & Binning

Traditional use of cuckoo hashing instructs the parties to sample new hash functions for each protocol invocation. In our setting however it can make sense to instruct the servers, which hold a somewhat static dataset, to perform cuckoo hashing once and for all. Updates to the dataset can then be handled by periodically rebuilding the cuckoo table once it has reached capacity. This leaves the question of what the cuckoo hashing success probability should be. It is standard practice to set statistical failure events like this to happen with probability 2^{-40}. However, since the servers perform cuckoo hashing only occasionally (and since hashing failure applies only to initialization, not future queries), we choose to use more efficient parameters with a security level of $\lambda = 20$ bits, i.e., $\Pr[\text{Cuckoo failure}] = 2^{-20}$. We emphasize that once the items are successfully placed into the hash table, all future *lookups* (*e.g.*, contact discovery instances) are error-free, unlike, say, in a Bloom filter.

We also must choose the number of hash functions to use.[6] Through experimental testing we find that overall the protocol performs best with $k = 3$ hash functions. The parameters used can be computed by solving for $e \approx 1.4$ in Equation 2 given that $\lambda = 20$.

To see why this configuration was chosen we must also consider another important parameter, the number of bins β. Due to binning being performed for each protocol invocation by the client, we must ensure that it succeeds with very high probability and therefore we use the standard of $\lambda = 40$ to choose binning parameters. An asymptotic analysis shows that the best configuration is to use $\beta = O(n/\log n)$ bins, each of size $\mu = O(\log n)$. However, this hides the exact constants which give optimal performance. Upon further investigation we find that the choice of these parameters result in a communication/computation trade-off.

For the free variable c, we set the number of bins to be $\beta = cn/\log_2 n$ and solve for the required bin size μ. As can be seen in Fig. 5 of the appendix, the use of $k = 3$ and scaling factor $c = 4$ result the best running time at the expense of a relatively high communication of 11 MiB. However, at the other end of the spectrum is $k = 2$ and $c = 1/16$ results in the smallest communication of 2.4 MiB. The reason $k = 2$ achieves smaller communication for any fixed c is that the client sends $k = 2$ PIR queries per item instead of three. However, $k = 2$ requires that the cuckoo table is three times larger than for $k = 3$ and therefore the computation is slower.

Varying c affects the number of bins β. Having fewer bins reduces the communication due to the bins being larger and thereby having better real/dummy query ratio. However, larger bins also increases the overall work, since the work is proportional to the bin size μ times N. We aim to minimize both the communication and running time. We therefore decided on choosing $k = 3$ and $c = 1/4$ as our default configuration, the circled data-point in Fig. 5. However, we note that in specific settings it may be desirable to adjust c further.

The PIR block size b also results in a computation/communication trade-off. Having a large block size gives shorter PIR keys and therefore less work to expand the DPF. However, this also results the server having a larger input set to the subsequent PSI which makes that phase require more communication and computation. Due to the complicated nature of how all these

6 Although the oblivious shuffling procedure of Sect. 3.3 can be extended in a natural way to include a stash, we use a stash-free variant of Cuckoo hashing.

Table 2. Our protocol's total contact discovery communication cost and running time using T threads, and $\beta = cn/\log_2 n$ bins and PIR block size of b. LAN: 10 Gbps, 0.02 ms latency.

Param.				Comm.	Running time [seconds]		
N	n	c	b	[MiB]	$T=1$	$T=4$	$T=16$
2^{28}	2^{10}	4	16	28.3	4.07	1.60	0.81
		0.25	1	4.93	33.02	13.22	5.54
	2^{8}	3	16	7.10	3.61	1.30	0.65
		1	1	2.20	14.81	6.92	3.40
	4	1	32	0.06	1.93	–	–
	1	1	32	0.03	1.21	–	–
2^{26}	2^{10}	2	8	12.7	1.61	0.72	0.41
		0.25	1	4.28	7.22	3.65	1.36
	2^{8}	6	16	10.3	0.98	0.51	0.26
		0.25	4	1.36	4.36	1.90	0.97
	4	1	32	0.06	0.56	–	–
	1	1	32	0.03	0.48	–	–
2^{24}	2^{10}	1	8	8.61	0.67	0.36	0.22
		0.25	1	3.85	2.28	0.94	0.50
	2^{8}	4	8	4.81	0.49	0.22	0.18
		1	1	1.68	1.26	0.57	0.36
	4	1	32	0.05	0.19	–	–
	1	1	32	0.03	0.16	–	–
2^{20}	2^{10}	0.5	4	2.10	0.22	0.10	0.06
		0.25	1	2.98	0.32	0.21	0.16
	2^{8}	2	4	1.95	0.20	0.09	0.06
		0.25	4	1.13	0.24	0.18	0.15
	4	1	32	0.05	0.14	–	–
	1	1	32	0.03	0.13	–	–

parameters interact with each other, we empirically optimized the parameters to find that a PIR block size between 1 and 32 gives the best trade-off.

6 Performance

In this section we analyze the performance of PIR-PSI. We ran all experiments on a single benchmark machine which has 2x 18-core Intel Xeon E5-2699 2.30 GHz CPU and 256 GB RAM. Specifically, we ran all parties on the same machine, communicating via localhost network, and simulated a network connection using the Linux `tc` command: a LAN setting with 0.02 ms round-trip latency, 10 Gbps network bandwidth; a WAN setting with a simulated 80 ms round-trip latency, 100 Mbps network bandwidth. We process elements of size $\rho = 128$ bits.

Our protocol can be separated into two phases: the *server's init* phase when database is stored in the Cuckoo table, and the *contact discovery* phase where client and server perform the private intersection.

6.1 PIR-PSI performance Results

In our *contact discovery* phase, the client and server first perform the pre-processing of PSI between n and $3n$ items which is independent of parties' inputs. We refer this step as pre-processing phase which specifically includes base OTs, and $O(n)$ PRFs. The online phase consists of protocol steps that depend on the parties' inputs. To understand the scalability of our protocol, we evaluate it on the range of server set size $N \in \{2^{20}, 2^{24}, 2^{26}, 2^{28}\}$ and client set size $n \in \{1, 4, 2^8, 2^{10}\}$. The small values of $n \in \{1, 4\}$ simulate the performance of incremental updates to a client's set. Tab. 2 presents the communication cost and total *contact discovery* time with online time for both single- and multi-threaded execution with $T \in \{1, 4, 16\}$ threads.

As discussed in Section 5.3, there are a communication and computation trade-off on choosing the different value c and b which effects the number of bins and how many items are selected per PIR query. The interplay between these two variable is somewhat complex and offer a variety of communication computation trade-offs. For smaller $n \in \{1, 4\}$, we set $b = 32$ to drastically reduce the cost of the PIR computation at the cost of larger PSI. For larger n, we consider parameters which optimize running time and communication separately, and show both in Tab. 2.

Our experiments show that our PIR-PSI is highly scalable. For the database size $N = 2^{28}$ and client set size $n = 2^{10}$, we obtain an overall running time of 33.02 s and only 4.93 MiB bits of communications for the contact discovery phase using a single thread. Alternatively, running time can be reduced to just 4 s for the cost of 28 MiB communication. Increasing the number of threads from 1 to 16, our protocol shows a factor of 5× improvement, due to the fact that it parallelizes well. When considering the smallest server set size of $N = 2^{20}$ with 16 threads, our protocol requires only 1.1 MiB of communication and 0.24 s of contact discovery time.

We point out that the computational workload for the client is small and consists only of DPF key generation, sampling random values and the classical PSI protocol in the size of the client set. This corresponds to 10% of the overall running time (e.g., 0.3 s of the total 3.6 s for $N = 2^{28}$ and $n = 2^8$). Despite our experiments being run on a somewhat powerful server, the overwhelming bottleneck for performance is the computational cost for the server. Furthermore, after parameter agreement, the PIR step adds just a single round of communication between the client device and the servers before the PSI starts. Hence, our reported performance is also representative of a scenario in which the client is a computationally weaker device connected via a mobile network. Detailed numbers on the performance impact of the optimizations from Sect. 5.1 are provided in Appendix C.

6.2 Updating the Client and Server Sets

In addition to performing PIR-PSI on sets of size n and N, the contact discovery application requires periodically adding items to these sets. In case the client adds a single item x to their set X, only the new item x needs to be compared with the servers' set. Our protocol can naturally handle this case by simply having the client use $X' = \{x\}$ as their input to the protocol. However, a shortcoming of this approach is that we cannot use binning and the PIR query spans the whole cuckoo table.

For a database of size $N = 2^{24}$, our protocol requires only 0.16 s and 0.19 s to update 1 and 4 items, respectively. When increasing the size to $N = 2^{28}$, we need 1.9 s to update one item. Our update queries are cheap in terms of communication, roughly 30–50 kiB. We remark that update queries can be parallelized well due to the fact that DPF.Gen and DPF.Expand can each be processed in a divide and conquer manner. Also, several update queries from different users can be batched together to offer very high throughput. However, our current implementation only supports parallelization at the level of bins/regions, and not for a single DPF query.

The case when a new item is added to the servers' set can easily be handled by performing a traditional PSI between one of the servers and the client, where the server only inputs their new item. One could also consider batching several server updates together, and then performing a larger PSI or applying our protocol to the batched server set.

7 Comparison with Prior Work

In this section we give a thorough qualitative & quantitative comparison between our protocol and those of CLR [8], KLSAP [29], and RA [41]. We obtained the implementations of CLR & KLSAP from the respective authors, but the implementation of RA is not publicly available. Because of that, we performed a comparison on inputs of size $N \in \{2^{16}, 2^{20}, 2^{24}\}$ and $n \in \{5535, 11041\}$ to match the parameters used in [41, Table 1&2]. While the experiments of RA were performed on an Intel Haswell i7-4770K quadcore CPU with 3.4 GHz and 16 GB RAM, we ran the KLSAP and CLR protocols on our own hardware, described in the previous section. We remark that RA's benchmark machine has 3.4 GHz, which is 1.48× *faster than our machine.* The number of cores and RAM available on our hardware does not influence the results of the single-threaded benchmarks ($T = 1$). Results of the comparison are summarized in Tab. 3.

7.1 CLR protocol

The high level idea of the protocol of Chen, Laine & Rindal (CLR) [8] is to have the client encrypt each element in their dataset under a homomorphic encryption scheme, and send the result to the server. The server evaluates the intersection circuit on encrypted data, and sends back the result for the receiver to decrypt.

The CLR protocol has communication complexity $O(\rho n \log(N))$, where the items are ρ bits long. Ours has communication complexity $O(\kappa n \log(N/(\kappa n \log n)))$, with no dependence on ρ since the underlying PSI protocol [30] has no such dependence. For small items (*e.g.*, $\rho = 32$ as reflected in Tab. 3), CLR uses less communication than our protocol, *e.g.*, 20 MiB as opposed to 37 MiB. However, their protocol scales very poorly for string length of 128 bits as it would require significantly less efficient FHE parameters. Furthermore, CLR can not take advantage of the fact that most contact lists have significantly fewer than 5535 entries. That is, the cost for $n = 1$ and $n = 5535$ is roughly equivalent, because of the way FHE optimizations like batching are used. The main computational bottleneck in CLR is the server performing $O(n)$ homomorphic evaluations on large circuits of size $O(N/n)$. The comparable bottleneck in our protocol is performing DPF.Expand and computing the large inner products. Since these operations take advantage of hardware-accelerated AES, PIR-PSI is significantly faster than CLR, *e.g.*, 20× for $N = 2^{24}$.

The server's initialization in CLR involves hashing the N items into an appropriate data structure (as in PIR-PSI), but also involves pre-computing the many coefficients of polynomials. Hence our initialization phase is much faster than CLR, *e.g.*, 40× for $N = 2^{24}$. The CLR protocol does not provide a full analysis of security against malicious clients. Like our protocol, the leakage allowed with a malicious client is likely to be minimal.

7.2 KLSAP protocol

In the KLSAP [29] protocol, the server sends a Bloom filter of size $O(\lambda N)$ to the client in an offline phase. During later contact discovery phases, the client refers to this Bloom filter.

The size of this Bloom filter is considerable: nearly 2 GiB for $N = 2^{24}$ server items. This data, which must be stored by the client, may be prohibitively large

Table 3. Comparison of PIR-PSI to CLR, KLSAP, and RA with $T \in \{1, 4\}$ threads. LAN: 10 Gbps, 0.02 ms latency. WAN: 100 Mbps, 80 ms latency. Best results marked in bold. Online communication reported in parenthesizes. Cells with "–" denote that the setting is not supported, due to limitations in the respective implementation. Cells with "*" indicate that the numbers are scaled for a fair comparison of error probability. CLR and RA use 32 bit items, while PIR-PSI and KLSAP process 128 bit items.

Protocol	Parameters		Communication	Running time [seconds]				Client	Server Init.	
				LAN (10 Gbps)		WAN (100 Mbps)		Storage	[seconds]	
	N	n	Size [MiB]	$T=1$	$T=4$	$T=1$	$T=4$	[MiB]	$T=1$	$T=4$
CLR [8]	2^{24}	11041	**21.1**	38.6	19.7	41.0	22.1	**0**	76.8	20.6
		5535	**12.5**	34.0	16.3	36.0	18.2		71.2	18.5
	2^{20}	11041	11.5	3.7	3.2	4.9	4.4		9.1	2.5
		5535	5.6	3.5	1.9	4.1	2.5		5.1	1.4
	2^{16}	11041	4.1/4.4	1.8	1.4	2.2	1.8		1.2	0.3
		5535	2.6	0.9	0.6	1.1	0.9		0.9	0.3
KLSAP [29]	2^{24}	11041	2049 (43.3)	90.4	–	265.1	–	1941	8.32	–
		5535	1070 (21.7)	52.3	–	128.3	–	1016		
	2^{20}	11041	1968 (43.3)	82.1	–	259.9	–	1860	0.58	–
		5535	989 (21.7)	44.8	–	124.7	–	935		
	2^{16}	11041	1963 (43.3)	81.8	–	259.6	–	1855	0.04	–
		5535	984 (21.7)	44.0	–	121.4	–	930		
RA [41]	2^{24}	11041	171.67 (0.67)*	**1.08***	–	18.39*	–	171.00*	333.62	–
		5535	168.34 (0.34)*	**0.75***	–	17.61*	–	168.00*		
	2^{20}	11041	**11.36 (0.67)***	0.67*	–	**3.41***	–	10.69*	20.78	–
		5535	**10.84 (0.34)***	0.34*	–	2.89*	–	10.50*		
	2^{16}	11041	**1.34 (0.67)***	0.66*	–	**1.33***	–	0.67*	1.30	–
		5535	**1.00 (0.34)***	0.33*	–	**0.85***	–	0.66*		
Ours	2^{24}	11041	32.46	2.18	**1.65**	**5.63**	**5.13**	**0**	**2.690**	–
		5535	21.45	1.34	**1.11**	**3.72**	**2.77**			
	2^{20}	11041	22.86	**0.37**	**0.31**	3.70	**3.59**		**0.089**	–
		5535	11.67	**0.29**	**0.24**	**2.50**	**2.29**			
	2^{16}	11041	12.83	**0.28**	**0.29**	2.55	**2.55**		**0.004**	–
		5535	7.66	**0.21**	**0.20**	1.85	**1.85**			

for mobile client applications. By contrast, our protocol (and CLR) requires no long-term storage by the client.

In the contact discovery phase, KLSAP runs Yao's protocol to obliviously evaluate an AES circuit for each of the client's items. Not even counting the boom filter, this requires slightly more communication than our approach (1.5×). Additionally, it requires more computation by the (weak) client: evaluating many AES garbled circuits (thousands of AES calls per item) vs. running many instances DPF.Gen ($\log N$ AES calls per item) followed by a specialized PSI protocol (constant number of hash/AES per item). Even though the server in our protocol must perform $O(N)$ computation during contact discovery, our considerable optimizations result in a much faster discovery phase ($40\times$ for $N = 2^{24}$).

When the server makes changes to its set in KLSAP, it must either re-key its AES function (which results in re-sending the huge Bloom filter), or send incremental updates to the Bloom filter (which breaks forward secrecy, as a client can query its items in both the old and new versions of the Bloom filter).

KLSAP is easily adapted to secure against a malicious client. This stems from the fact that the contact discovery phase uses Yao's protocol with the client acting as garbled circuit evaluator. Hence it is naturally secure against a malicious client (provided that one uses malicious-secure OTs).

Subtleties about hashing errors: The way that KLSAP uses a Bloom filter also leads to qualitative differences in the error probabilities compared to PIR-PSI. In KLSAP the server publishes a Bloom filter for all clients, who later query it for membership. The false-positive rate (FPR) of the Bloom filter is the probability that a single item not in the server's set is mistakenly classified as being in the intersection. Importantly, the FPR for this global Bloom filter is *per client item.* In KLSAP this FPR is set to 2^{-30}, which means after processing a combined 1 million client items the probability of some client receiving a false positive may be as high as 2^{-10}!

By contrast, the PIR-PSI server places its items in a Cuckoo table once-and-for all (with hashing error probability 2^{-20}). As long as this *one-time event* is successful, all subsequent probes to this data structure are error-free (we store the entire $\rho = 128$ bit item in the Cuckoo table, not just a short fingerprint as in [41]). If the hashing is unsuccessful, the server simply tries again with different hash functions. All of our other failure events (*e.g.*, probability of a bad event within our 2-party PSI protocol) are calibrated for error probability 2^{-40} *per contact* *discovery instance*, not per item! To have a comparable guarantee, the Bloom filter FPR of KLSAP would have to be scaled by a factor of $\log_2(n)$.

7.3 RA protocol

The RA [41] protocol uses a similar approach to KLSAP, in that it uses a relatively large representation of the server's set, which is sent in an offline phase and stored by the client. The downsides of this architecture discussed above for KLSAP also apply to RA (client storage, more client computation, false-positive rate issues, forward secrecy).

RA's implementation uses a Cuckoo filter that stores for each item a 16-bit fingerprint. This choice leads to a relatively high false-positive rate of $2^{-13.4}$. To achieve the failure events with error probability 2^{-40} per contact discovery instance (in line with our protocol), the Cuckoo filter FPR of RA would be $2^{-(40+\log_2(n))}$. Therefore, their protocol would have to be modified to use 56-bit and 57-bit fingerprints for $n = 5535$ and $n = 11041$, respectively. This change increases the communication cost, transmission time, and offline storage requirements $3.44-3.5\times$, relative to the numbers reported in [41, Table 1]. In Tab. 3 we report the *scaled* communication costs, the *scaled* online running time, and the *scaled* client's storage, but refrain from trying to scale the server's initialization times. As can be seen our protocol running time is $1.2 - 3.2\times$ faster than RA for sufficiently large N. We also have a $100\times$ more efficient server initialization phase and achieve communication complexity of $O(n \log N)$ as compared to $O(N)$ of RA. This difference can easily be seen by how the communication of RA significantly increases for larger N.

In RA, the persistent client storage is not a Bloom filter but a more compact Cuckoo filter. This reduces the client storage, but it still remains linear in N. For $N = 2^{28}$ the storage requirement is 2.57 GiB to achieve an error probability of 2^{-40} per contact discovery instance.

The RA protocol does not provide any analysis of security against malicious clients.

8 Other Extensions

Although this work mainly focuses on the setting of pure contact discovery with two servers, our protocol can be easily modified for other settings.

PSI with associated data (PSI+AD): refers to a scenario where the client has a set A of keys and the server has a set B of key-value pairs, and the client wishes to learn $\{(k, v) \mid (k, v) \in B \text{ and } k \in A\}$. In the context of an encrypted messaging service, the keys may be phone numbers or email addresses, and the values may be the user's public key within the service.

PIR-PSI can be modified to support associated data, in a natural way. The server's Cuckoo hash table simply holds key-value pairs, and the 2-party PSI protocol is replaced by a 2-party PSI+AD protocol. The client will then learn *masked* values for each item in the intersection, which it can unmask. The PSI protocol of [30] that we use is easily modified to allow associated data.

3- and 4-Server Variant: We described PIR-PSI in the context of two non-colluding servers, who store identical copies of the service provider's user database. Since both servers hold copies of this sensitive database, they are presumably both operated by the service provider, so the promise of non-collusion may be questionable. Using a folklore observation from the PIR literature, we can allow servers to hold only *secret shares* of the user database, at the cost of adding more servers.

Consider the case of 3 servers. The service provider can recruit two independent entities to assist with private contact discovery, without entrusting them with the sensitive user database. The main idea is to let servers #2 and #3 hold additive secret shares of the database and jointly simulate the effect of a single server that holds the database in the clear.

Recall the 2-party DPF-PIR scheme of [5], that we use. The client sends DPF shares k_1, k_2 to the servers, who expand the keys to K_1, K_2 and performs an inner product with the database. The client XORs the two responses to obtain result $(K_1 \cdot DB) \oplus (K_2 \cdot DB) = (K_1 \oplus K_2) \cdot DB = DB[i]$.

In our 3-server case, we have server #1 holding DB, and servers #2 and #3 holding DB_2, DB_3 respectively, where $DB = DB_2 \oplus DB_3$. We simply let the client send DPF share k_1 to server #1, and send k_2 to *both* of the other servers. All servers expand their DPF share and perform an inner product with their database/share. The client will receive $K_1 \cdot DB$ from server #1, $K_2 \cdot DB_2$ from server #2, and $K_2 \cdot DB_3$ from server #3. The XOR of all responses is indeed

$$\begin{aligned} &(K_1 \cdot DB) \oplus (K_2 \cdot DB_2) \oplus (K_2 \cdot DB_3) \\ &= (K_1 \cdot DB) \oplus K_2 \cdot (DB_2 \oplus DB_3) \\ &= K_1 \cdot DB \oplus K_2 \cdot DB = (K_1 \oplus K_2) \cdot DB = DB[i] \end{aligned}$$

Now the entire PIR-PSI protocol can be implemented with this 3-server PIR protocol as its basis. The computational cost of each server is identical to the 2-server

PIR-PSI, and is performed in parallel by the independent servers. Hence, the total time is minimally affected. The client's total communication is unaffected since server #2 can forward K_2 to server #3. The protocol security is the same, except that the non-collusion properties hold now only if server #1 doesn't collude with any of the other servers. If servers #2 & #3 collude, then they clearly learn DB, but as far as the client's privacy is concerned, the situation simply collapses to 2-server PIR-PSI.

Similarly, server #1 can also be replaced by a pair of servers, each with secret shares (and this sharing of DB can be independent of the other sharing of DB). This results in a 4-server architecture with security for the client as long as neither of servers #1 & #2 collude with one of the servers #3 & #4, and where no single server holds DB in the clear.

2-Server with OPRF Variant: An alternative to the 3-server variant above is to leverage a pre-processing phase. Similar to [41], the idea is to have server #1 apply an oblivious PRF to their items instead of a hash function, which will ensure that the database is pseudorandom in the view of server #2, who does not know the PRF key. In particular, let server #1 sample a key k for the oblivious PRF F used in [41] and update the database as $DB'_i := F_k(DB_i)$, which is then sent to server #2. When a client wishes to compute the intersection of its set X with DB, they first perform an oblivious PRF protocol with server #1 to learn $X' = \{F_k(x) \mid x \in X\}$. Note that this protocol ensures that the client does not learn k. The client can now engage in our standard two-server PIR-PSI protocol to compute $Z' = X' \cap DB'$ and thereby infer $Z = X \cap DB$.

The advantage of this approach is that server #2 does not learn any information about the plaintext database DB since the PRF was applied to each record. Moreover, this holds even if server #2 colludes with one of the clients. The added performance cost of this variant has two components. First, server #1 must update its database by applying the PRF to it. As shown by [41], a single CPU core can process roughly 50,000 records per second, which is sufficiently fast given that this is a one-time cost. The second overhead is performing the oblivious PRF protocol with the clients. This requires three exponentiations per item in X, which represents an acceptable overhead given that $|X|$ is small.

Single Server Variant: We also note that our PIR-PSI architecture has the potential to be extended to the single-server setting. Several PIR protocols [1–3], based on fully homomorphic encryption have been shown to offer good performance while at the same time removing the two-server requirement. With some modifications to our architecture, we observe that such PIR protocols can be used. The main challenge to overcome is how to secret share and shuffle the results of the PIR before being forwarded to the PSI protocol. First, a PIR protocol which allows the result to be secret shared is required. We observer that typical PIR protocols (e.g. [3]) can support such a functionality by adding a random share to the result ciphertext. Given this, a two party variant of step 3 of Fig. 2 can be implemented using standard two-party shuffling protocols. We leave the optimization and exact specification of such a single-server PIR-PSI protocol to future work, but note its feasibility.

9 Deployment

We now turn our attention to practical questions surrounding the real-world deployment of our multi-server PIR-PSI protocol. As briefly discussed in the previous section, the requirement that a single organization has two non-colluding servers may be hard to realize. However, we argue that the 3-server or 2-server with an OPRF variants make deployment significantly simpler. Effectively, these variants reduce the problem to finding one or two external semi-honest parties that will not collude with the service provider (server #1). A natural solution to this problem is to leverage existing cloud providers such as Microsoft Azure. Given that these companies have a significant interest to maintain their reputation, they would have a large incentive to not collude. Indeed, Microsoft has informally proposed such a setting [21] where secure computation services are provided under a non-collusion assumption. Alternatively, privacy-conscious organization such as the Electronic Frontier Foundation (EFF) could serve as the second server.

Acknowledgments

We thank the anonymous reviewers of PETS 2018 for their helpful comments on our paper. This work has been co-funded by the German Federal Ministry of Education and Research (BMBF) and by the Hessen State Ministry for Higher Education, Research and the Arts (HMWK) within CRISP, by the DFG as part of projects S5 and E4 within the CRC 1119 CROSSING. Authors from Oregon State University have been partially supported by a Google faculty research award.

References

[1] C. Aguilar-Melchor, J. Barrier, L. Fousse, and M.-O. Killijian. XPIR : Private Information Retrieval for Everyone. In: *PoPETs* 2016.2 (2016), pp. 155 –174.

[2] S. Angel and S. Setty. Unobservable Communication over Fully Untrusted Infrastructure. In: *OSDI*. 2016, pp. 551–569.

[3] S. Angel, H. Chen, K. Laine, and S. Setty. PIR with compressed queries and amortized query processing. In: *Proceedings of the 2018 IEEE Symposium on Security and Privacy*. SP '18. IEEE Computer Society, 2018.

[4] E. Boyle, N. Gilboa, and Y. Ishai. Function Secret Sharing. In: *EUROCRYPT 2015, Part II*. Vol. 9057. LNCS. Springer, Heidelberg, 2015, pp. 337–367.

[5] E. Boyle, N. Gilboa, and Y. Ishai. Function Secret Sharing: Improvements and Extensions. In: *ACM CCS 16*. ACM Press, 2016, pp. 1292–1303.

[6] C. Cachin, S. Micali, and M. Stadler. Computationally Private Information Retrieval with Polylogarithmic Communication. In: *EUROCRYPT'99*. Vol. 1592. LNCS. Springer, Heidelberg, 1999, pp. 402–414.

[7] G. S. Cetin et al. Private Queries on Encrypted Genomic Data. In: *IACR Cryptology ePrint Archive* 2017 (2017), p. 207. URL: http://eprint.iacr.org/2017/207.

[8] H. Chen, K. Laine, and P. Rindal. Fast Private Set Intersection from Homomorphic Encryption. In: *ACM CCS 17*. ACM Press, 2017, pp. 1243–1255.

[9] B. Chor, N. Gilboa, and M. Naor. *Private Information Retrieval by Keywords*. Cryptology ePrint Archive, Report 1998/003. http://eprint.iacr.org/1998/003. 1998.

[10] B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan. Private Information Retrieval. In: *J. ACM* 45.6 (1998), pp. 965–981.

[11] E. De Cristofaro, J. Kim, and G. Tsudik. Linear-Complexity Private Set Intersection Protocols Secure in Malicious Model. In: *ASIACRYPT 2010*. Vol. 6477. LNCS. Springer, Heidelberg, 2010, pp. 213–231.

[12] E. De Cristofaro and G. Tsudik. Practical Private Set Intersection Protocols with Linear Complexity. In: *FC 2010*. Vol. 6052. LNCS. Springer, Heidelberg, 2010, pp. 143–159.

[13] D. Demmler, A. Herzberg, and T. Schneider. RAID-PIR: Practical Multi-Server PIR. In: *ACM Workshop on Cloud Computing Security*. CCSW '14. ACM, 2014, pp. 45–56.

[14] C. Devet, I. Goldberg, and N. Heninger. Optimally Robust Private Information Retrieval. In: *Proceedings of the 21st USENIX Conference on Security Symposium*. Security'12. USENIX Association, 2012, pp. 13–13. URL: http://dl.acm.org/citation.cfm?id=2362793.2362806.

[15] M. Dietzfelbinger et al. Tight thresholds for cuckoo hashing via XORSAT. In: *International Colloquium on Automata, Languages, and Programming*. Springer. 2010, pp. 213–225.

[16] C. Dong and L. Chen. A Fast Single Server Private Information Retrieval Protocol with Low Communication Cost. In: *ESORICS 2014, Part I*. Vol. 8712. LNCS. Springer, Heidelberg, 2014, pp. 380–399.

[17] C. Dong, L. Chen, and Z. Wen. When private set intersection meets big data: an efficient and scalable protocol. In: *ACM CCS 13*. ACM Press, 2013, pp. 789–800.

[18] M. J. Freedman, K. Nissim, and B. Pinkas. Efficient Private Matching and Set Intersection. In: *EUROCRYPT 2004*. Vol. 3027. LNCS. Springer, Heidelberg, 2004, pp. 1–19.

[19] A. Frieze, P. Melsted, and M. Mitzenmacher. An analysis of random-walk cuckoo hashing. In: *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*. Springer, 2009, pp. 490–503.

[20] C. Gentry and Z. Ramzan. Single-Database Private Information Retrieval with Constant Communication Rate. In: *ICALP 2005*. Vol. 3580. LNCS. Springer, Heidelberg, 2005, pp. 803–815.

[21] R. Gilad-Bachrach et al. Secure Data Exchange: A Marketplace in the Cloud. In: (2016).

[22] N. Gilboa and Y. Ishai. Distributed Point Functions and Their Applications. In: *EUROCRYPT 2014*. Vol. 8441. LNCS. Springer, Heidelberg, 2014, pp. 640–658.

[23] O. Goldreich, S. Micali, and A. Wigderson. How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In: *19th ACM STOC*. ACM Press, 1987, pp. 218–229.

[24] J. Groth, A. Kiayias, and H. Lipmaa. Multi-query Computationally-Private Information Retrieval with Constant Communication Rate. In: *PKC 2010*. Vol. 6056. LNCS. Springer, Heidelberg, 2010, pp. 107–123.

[25] R. Henry. Polynomial Batch Codes for Efficient IT-PIR. In: *PoPETs* 2016.4 (2016), pp. 202–218. URL: https://doi.org/10.1515/popets-2016-0036.

[26] R. Henry, Y. Huang, and I. Goldberg. One (Block) Size Fits All: PIR and SPIR with Variable-Length Records via Multi-Block Queries. In: *NDSS 2013*. The Internet Society, 2013.

[27] R. Henry, F. G. Olumofin, and I. Goldberg. Practical PIR for electronic commerce. In: *ACM CCS 11*. ACM Press, 2011, pp. 677–690.

[28] Huffington Post. *Nach Gerichtsurteil: WhatsApp-Nutzern können Abmahnkosten drohen (German)*. http://www.huffingtonpost.de/2017/06/27/whatsapp-abmahnung-anwalt-medien-gericht-nutzer_n_17302734.html. 2017.

[29] Á. Kiss et al. Private Set Intersection for Unequal Set Sizes with Mobile Applications. In: *PoPETs* 2017.4 (2017), pp. 177–197.

[30] V. Kolesnikov, R. Kumaresan, M. Rosulek, and N. Trieu. Efficient Batched Oblivious PRF with Applications to Private Set Intersection. In: *ACM CCS 16*. ACM Press, 2016, pp. 818–829.

[31] M. Lambæk. Breaking and Fixing Private Set Intersection Protocols. https://eprint.iacr.org/2016/665. MA thesis. Aarhus University, 2016.

[32] Large-Scale Data & Systems (LSDS) Group, Imperial College, London. *spectre-attack-sgx*. Github Repository. https://github.com/lsds/spectre-attack-sgx. 2017.

[33] W. Lueks and I. Goldberg. Sublinear Scaling for Multi-Client Private Information Retrieval. In: *FC 2015*. Vol. 8975. LNCS. Springer, Heidelberg, 2015, pp. 168–186.

[34] M. Marlinspike. *Technology preview: Private contact discovery for Signal*. Signal blog post. https://signal.org/blog/private-contact-discovery/. 2017.

[35] M. Marlinspike. *The Difficulty Of Private Contact Discovery*. Whisper Systems blog post. https://whispersystems.org/blog/contact-discovery/. 2014.

[36] T. Mayberry, E.-O. Blass, and A. H. Chan. PIRMAP: Efficient Private Information Retrieval for MapReduce. In: *FC 2013*. Vol. 7859. LNCS. Springer, Heidelberg, 2013, pp. 371–385.

[37] M. Orrù, E. Orsini, and P. Scholl. Actively Secure 1-out-of-N OT Extension with Application to Private Set Intersection. In: *Topics in Cryptology – CT-RSA 2017: The Cryptographers' Track at the RSA Conference 2017, San Francisco, CA, USA, February 14–17, 2017, Proceedings*. Springer International Publishing, 2017, pp. 381–396.

[38] B. Pinkas, T. Schneider, and M. Zohner. Faster Private Set Intersection Based on OT Extension. In: *USENIX Security 14*. SEC'14. USENIX Association, 2014, pp. 797–812.

[39] B. Pinkas, T. Schneider, and M. Zohner. *Scalable Private Set Intersection Based on OT Extension*. Cryptology ePrint Archive, Report 2016/930. http://eprint.iacr.org/2016/930. 2016.

[40] B. Pinkas, T. Schneider, G. Segev, and M. Zohner. Phasing: Private Set Intersection Using Permutation-based Hashing. In: *USENIX Security 15*. USENIX Association, 2015, pp. 515–530.

[41] A. C. D. Resende and D. F. Aranha. Faster Unbalanced Private Set Intersection. In: *FC 2018*. LNCS. Springer, Heidelberg, 2018.

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

[43] P. Rindal. *libPSI: A repository for private set intersection.* https://github.com/osu-crypto/libPSI.

[44] P. Rindal and M. Rosulek. Improved Private Set Intersection Against Malicious Adversaries. In: *EUROCRYPT 2017, Part I*. Vol. 10210. LNCS. Springer, Heidelberg, 2017, pp. 235–259.

[45] A. Smith. *6 new facts about Facebook*. Pew Research Center Fact Tank. http://www.pewresearch.org/fact-tank/2014/02/03/6-new-facts-about-facebook/. 2014.

[46] J. T. Trostle and A. Parrish. Efficient Computationally Private Information Retrieval from Anonymity or Trapdoor Groups. In: *ISC 2010*. Vol. 6531. LNCS. Springer, Heidelberg, 2011, pp. 114–128.

[47] A. C.-C. Yao. Protocols for Secure Computations (Extended Abstract). In: *23rd FOCS*. IEEE Computer Society Press, 1982, pp. 160–164.

A Hiding Cuckoo Locations

We previously described a simple approach to hide the Cuckoo location for a single item of the client. At first glance, it seems trivial to generalize this approach to many items for the client – simply repeat the procedure above once for each client item. However, it requires server #2 to know that two PIR queries correspond to the same logical client item (*e.g.*, two queries correspond to $h_1(x)$ and $h_2(x)$ for the same x, so their masks can be randomly swapped). This turns out to be incompatible with another of our optimizations (see Sect. 3.4) that lets the servers learn some information about the location of the PIR queries. It is safe to leak this information about the *collective* set of client queries (*e.g.*, a certain number of the client's queries are made to this region in CT) but not about *specific* queries (*e.g.*, client has an x where $h_1(x)$ is in this region and $h_2(x)$ is in that region).

We therefore generalize this oblivious masking technique as the following functionality:

- The client holds a permutation π that maps its logical inputs to the indices of those PIR queries. That is, for each item x_j of the client, the $\pi(2j+1)$'th PIR query is for $CT[h_1(x_j)]$ and the $\pi(2j+2)$'th PIR query is for $CT[h_2(x_j)]$.
- The client also holds a vector $\vec{r}$ of masks
- Server #1 chooses a random permutation σ with the property that $\{\sigma(2j+1), \sigma(2j+2)\} = \{2j+1, 2j+2\}$ for all j. That is, σ consists of swaps of adjacent items only.
- Servers #1 & #2 have vectors of PIR responses $\vec{v}_1, \vec{v}_2$, respectively.
- The goal is for server #2 to learn:

$$\vec{v}_1 \oplus \vec{v}_2 \oplus \pi(\sigma(\vec{r})) \overset{\text{def}}{=} \vec{v}_1 \oplus \vec{v}_2 \oplus (r_{\pi(\sigma(1))}, \ldots, r_{\pi(\sigma(2n))})$$

We claim that this results in masks r_{2j+1}, r_{2j+2} being "routed" to the two PIR queries corresponding to logical item x_j. Indeed, since $\vec{v}_1 \oplus \vec{v}_2$ comprise the *unmasked* PIR outputs, the definition of π implies that

$$\vec{v}_1 \oplus \vec{v}_2 = \pi\big(CT[h_1(x_1)], CT[h_2(x_1)], \ldots, CT[h_1(x_n)], CT[h_2(x_n)]\big)$$

Hence, server #2's output is the following vector permuted by π:

$$\big(CT[h_1(x_1)], CT[h_2(x_1)], \ldots\big) \oplus (r_{\sigma(1)}, r_{\sigma(2)}, \ldots)$$

By the construction of σ, we see that masks r_{2j+1} and r_{2j+2} are indeed paired up with $CT[h_1(x_j)]$ and $CT[h_2(x_j)]$, as desired.

Hence, the client can compute the $2n$ values of the form $x_j \oplus r_{2j+1}, x_j \oplus r_{2j+2}$, and use these as input to a conventional 2-party PSI protocol. Server #2 can use its output from this oblivious masking as its input to the PSI. From the output of this PSI subprotocol, the client can deduce the intersection.

To actually achieve this oblivious masking functionality, we do the following: The client picks three random mask vectors $\vec{r}, \vec{s}, \vec{t}$ of length m and generates a 2-out-of-2 secret sharing of π as $\pi = \pi_2 \circ \pi_1$. The client sends $\vec{t}$ and π_2 to server #2 and $\vec{r}$, $\vec{s}$, π_1 and $\vec{t} \oplus \pi(\vec{s})$ to server #1. Server #1 sends $\pi_1\big(\sigma(\vec{r})\oplus\vec{s}\big)$ and $[\vec{t}\oplus\pi(\vec{s})\oplus\vec{v}_1]$ to server #2, who can then compute $\vec{v}$:

$$\begin{aligned}\vec{v} &= \vec{v}_2 \oplus \pi_2\Big(\pi_1\big(\sigma(\vec{r})\oplus\vec{s}\big)\Big) \oplus \vec{t} \oplus [\vec{t}\oplus\pi(\vec{s})\oplus\vec{v}_1]\\ &= \vec{v}_2 \oplus \pi\big(\sigma(\vec{r})\big) \oplus \pi(\vec{s}) \quad \oplus \vec{t} \oplus \quad \vec{t}\oplus\pi(\vec{s})\oplus\vec{v}_1\\ &= \vec{v}_1 \oplus \vec{v}_2 \oplus \pi\big(\sigma(\vec{r})\big)\end{aligned}$$

In order to be compatible with further optimizations, we must show that the servers learn nothing about the client's permutation π, which captures which PIR queries correspond to the same logical client input.

Server #1 receives $\vec{r}, \vec{s}, \pi_1$, and $\vec{t}\oplus\pi(\vec{s})$. These values are randomly selected by the client, so server #1 learns nothing about π from this oblivious masking process.

Server #2 receives $\vec{t}$ and π_2 from the client, and $\pi_1(\sigma(\vec{r})\oplus\vec{s})$ as well as $\vec{t}\oplus\pi(\vec{s})\oplus\vec{v}_1$ from server #1. Since the values $\vec{t}, \pi_2, \vec{r}, \vec{s}$ are each uniformly distributed, the entire view of server #2 is random. Hence, server #2 likewise learns nothing about π from the oblivious masking process.

Saving bandwidth: We can save bandwidth in the oblivious masking procedure by observing that many of the client's messages are random, and can instead be chosen pseudorandomly.

Recall that $\vec{r}, \vec{s}, \pi_1, \vec{t}\oplus\pi(\vec{s})$ are sent to server #1 and $\vec{t}, \pi_2$ to server #2. The client can send a small seed w to server #1 and use this seed to pseudorandomly choose $(\vec{r}, \vec{s}, \pi_1) = \text{PRG}(w)$. Similarly, the client can send a seed to server #2 and use it pseudorandomly define $\vec{t}$.

Now that π_1 is fixed, the client can solve for appropriate π_2 such that $\pi_2 \circ \pi_1 = \pi$. The client must send other values explicitly: $\vec{t}\oplus\pi(\vec{s})$ to server #1 and π_2 to server #2.

B Cuckoo Hashing Failure Probability Formula

Let $e > 1$ be the expansion factor denoting that N items are inserted into a cuckoo table of size $m = eN$. Fig. 3 shows the security parameter (i.e., λ, such that the probability of hashing failure is $2^{-\lambda}$) of Cuckoo hashing with $k = 2$ hash functions. As N becomes larger, λ scales linearly with $\log_2 N$ and with the stash size s, which matches the results of [15]. For $e \geq 8$ and $k = 2$, we interpolate the relationship as the linear equation

$$\lambda = \big(1 + 0.65s\big)\big(3.3\log_2(e) + \log_2(N) - 0.8\big) \qquad (1)$$

For smaller values of e, we observe that λ quickly converges to 1 at $e = 2$. We approximate this behavior by subtracting $\big(5\log_2(N) + 14\big)e^{-2.5}$ from Equation 1. We note that these exact interpolated parameters are specific to our implementation which uses a specific eviction policy (linear walk) and re-insert bounds (100). However, we observed similar bounds for other parameters and evictions strategies (e.g. random walks or 200 re-insert bound).

We also consider the case $k = 3$, shown in Fig. 4 and find that it scales significantly better that $k = 2$. For instance, at $e = 2$ we find $\lambda \approx 100$ for interesting set sizes while the same value of e applied to $k = 2$ results in $\lambda \approx 1$. As before we find that λ grows linearly with the expansion factor e. Unlike in the case of $k = 2$, we observe that increasing N has a slight negative effect on λ. Namely, doubling N roughly decreases λ by 2. However, the slope at which λ increases for $k = 3$ is much larger than $k = 2$ and therefore this dependence on $\log N$ has little impact on λ. We summarize these findings for $k = 3$ as the linear equation

$$\lambda = a_N e + b_N \qquad (2)$$

where $a_N \approx 123.5$ and $b_N \approx -130 - \log_2 N$. Here we use an approximation to hide an effect that happens for small $N \leq 512$. In this regime we find that the security level quickly falls. In particular, the slope a_N and intercept b_N go to zero roughly following the normal distribution CDF. By individually interpolating these variable we obtain accurate predictions of λ for $N \geq 4$. Our interpolations show that $a_N = 123.5 \cdot \text{CDF}_{\text{NORMAL}}(x = N, \mu = 6.3, \sigma = 2.3)$ and $b_N = -130 \cdot \text{CDF}_{\text{NORMAL}}(x = N, \mu = 6.45, \sigma = 2.18) - \log_2 N$.

For $k = 3$ we do not consider a stash due to our experiments showing it having a much smaller impact as compared to $k = 2$. Additionally, we do not compute exact parameters for $k > 3$ due to the diminishing returns. In particular, $k = 4$ follows the same regime as $k = 3$ but only marginally improves the failure probability.

C Effect of the Optimizations

In this section, we discuss the effect of our optimizations on the performance. By far the most important optimization employed is the use of *binning*. Observe in Tab. 4 that the running time with all optimizations enabled is 1.0 s while the removal of binning results in a running time of 1906 s. This can be explained by the overall reduction of asymptotic complexity to $O(N \log n)$ with binning as opposed to $O(Nn)$ without binning.

Table 4. Online running time in seconds of the protocol with all optimizations enabled compared with the various optimizations of Sect. 5.1 individually disabled.

N	n	All Opt. Enabled	No Batching	No Blocking	No Vectorization	No Binning
2^{24}	2^{12}	1.0	2.1	3.7	40.1	1906

Another important optimization is the use of PIR blocks which consist of more than cuckoo table item. This *blocking* technique allows for a better balance between the cost of the PIR compared to the cost of the subsequent PSI. Increasing the block size logarithmically decreases the cost of the PIR while linearly increasing the cost of the PSI. Since the PIR computation is so much larger than the PSI (assuming $n \ll N$) setting the block size to be greater than 1 gave significant performance improvements. In practice we found that setting b to be within 1 and 32 gave the best results. Tab. 4 shows that setting b to optimize running time gives a 3.7× improvement.

We also consider the effect that our highly optimized DPF implementation has on the overall running time. *Vectorization* refers to an implementation of the DPF with the full-domain optimization implemented similar as described by the [5, Figure 4]. We then improve on their basic construction to take full advantage of CPU vectorization and fixed-key AES. The result is a 40× difference in overall running-time.

The final optimization is to improve memory locality of our implementation by carefully accessing the cuckoo table. Instead of computing each PIR query individually, which would require loading the large cuckoo table from memory many times, our *batching* optimization runs all DPF evaluations for a given database location at the same time. This significantly reduces the amount of data that has to be fetched from main memory. For a dataset of size $N = 2^{24}$ we observe that this optimization yields 2.1× improvement, and an even bigger 5× improvement when applied to a larger dataset of $N = 2^{28}$ along with using $T = 16$ threads.

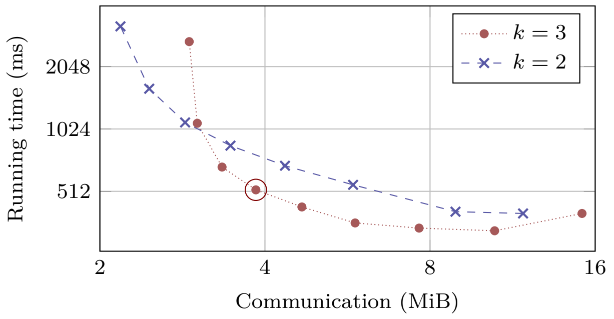

Fig. 5. Communication and computation trade-off for $n = 2^{10}, N = 2^{24}, T = 16$ threads, k cuckoo hash function, no stash, with the use of $\beta = cn/\log_2 n$ bins where $c \in \{2^{-5}, 2^{-4}, \ldots, 2^3\}$ and are listed left to right as seen above. The configuration $(k = 2, c = 2^{-5})$ did not fit on the plot. The highlighted point $(k = 3, c = 1/4)$ is the default parameter choice that is used.