



Streaming and Unbalanced PSI from Function Secret Sharing

Samuel Dittmer¹(✉) , Yuval Ishai², Steve Lu¹ , Rafail Ostrovsky^{1,3} ,
Mohamed Elsabagh⁴, Nikolaos Kiourtis⁴, Brian Schulte⁴ ,
and Angelos Stavrou⁴

¹ Stealth Software Technologies, Inc., Los Angeles, CA 90025, USA
{samdittmer, steve, rafail}@stealthsoftwareinc.com

² Technion - Israel Institute of Technology, Haifa, Israel
yuval@cs.technion.ac.il

³ University of California, Los Angeles, Los Angeles, CA 90025, USA
rafail@cs.ucla.edu

⁴ Kryptowire, LLC, Fairfax, VA 22102, USA
{melsabagh, nkiourtis, bschulte, angelos}@kryptowire.com

Abstract. Private Set Intersection (PSI) is one of the most useful and well-studied instances of secure computation, with many variants and applications. In this work, we present new solutions to PSI and a weighted variant in which the output is the sum of the weights of keywords in the intersection. Our protocols apply to the semi-honest, two-server model and are optimized for the unbalanced case, where one of the sets is much larger than the other, and for a dynamic streaming setting, in which sets can evolve over time.

Our protocols make use of Function Secret Sharing (FSS) to aggregate numerical payloads associated with the intersection while minimizing interaction and computational overhead. They avoid the use of public-key cryptography, giving simple and concretely efficient protocols for unbalanced PSI. In the dynamic setting, we use queuing theory to eliminate leakage with minimal overhead while ensuring low wait times, giving efficient streaming unbalanced PSI.

1 Introduction

In this work, we describe a new lightweight approach to the Private Set Intersection (PSI) problem, which is geared towards the streaming and unbalanced PSI variants. We focus on the two-server model, where the bigger data set is owned by a pair of non-colluding servers. Unlike the traditional two-party setting, the two-server model allows for solutions that avoid the use of public-key cryptography. Such a two-server model is commonly used in the context of practical solutions for private information retrieval and related problems; see, e.g., [6, 15, 40, 41] and references therein.

Our technical approach makes a novel use of a Distributed Point Function (DPF) [25], an instance of Function Secret Sharing (FSS) [8], to minimize communication and computation costs and to support dynamic data sets. One key feature of FSS is that it natively supports secure keyword search on raw sets of keywords without a need for processing the keywords via a data structure for set membership. Furthermore, the FSS functionality enables adding up numerical payloads associated with multiple matches without additional interaction. These features make FSS an attractive tool for lightweight PSI solutions for a variety of applications, including private contact discovery, aggregated ads analytics, tracking breached passwords, monitoring data feeds for target keywords, and contact tracing.

The bulk of PSI research focuses on the balanced case, where the parties hold sets of roughly equal size. Recently there has been new interest paid to *unbalanced* PSI, where one set has length n and the other has length N , with $n \ll N$, and we desire total communication of $\tilde{O}(n)$.

Additionally, although PSI generally is treated as a one-shot protocol, the concept of updateable PSI was recently formalized [4] for a PSI that is executed in a dynamic *streaming* setting. In many applications it is desirable to run the PSI repeatedly over a period of time, where only a small number of elements from each set changes each epoch, and the amortized work per epoch is proportional to the size of the change. The protocols we present here offer improvements in asymptotic complexity and concrete efficiency over previous protocols in these settings, especially in the case of unbalanced streaming PSI. To further enhance their efficiency without introducing extra information leakage, we also consider a relaxation of the correctness requirement that enables answers to be slightly delayed.

1.1 Our Contribution

Focusing on the setting of a single (semi-honest) client and two non-colluding servers, we present efficient protocols for a generalized variant of PSI referred to as *PSI with weighted cardinality* (PSI-WCA). In PSI-WCA, for each keyword held by the client, there is an associated secret weight. The client learns the weight sum of the keywords in the intersection. This is targeted towards applications where the client wants to obtain a weighted score for the quality of the match between the two sets (e.g., weights can represent the importance of different keywords in a search query).¹ Determining the (unweighted) size of the intersection is a simple special case.

We present 4 protocols for PSI-WCA, two in the one-shot setting and two in the streaming setting. Our one-shot protocols include a baseline protocol for PSI-WCA and our queueing PSI-WCA, and our streaming protocols are a baseline

¹ Note that while PSI-WCA does not return to the client the actual set of keywords in the intersection, it can be used to achieve this by letting each weight include a suitable encoding of the associated keyword (e.g., consecutive powers of this keyword, cf. [15]).

streaming PSI-WCA protocol and a streaming PSI-WCA protocol with greedy scheduling. These solutions have the following features:

- **One round.** Our solutions use minimal interaction. In the one-shot case, the client sends one message to each server and receives a message from each server in return. In a streaming setting, only the servers need to send a message in each epoch when the client’s input stays the same.
- **Minimal cryptographic assumptions.** Our solutions rely only on the minimal cryptographic assumption of the existence a secure PRG, which can be instantiated with AES in practice. This gives rise to fast and simple implementations using standard hardware under conservative security assumptions. All single server solutions necessarily require public-key cryptography.
- **Weighted cardinality.** Our protocols natively realize the generalized *PSI with Weighted Cardinality* functionality that enables a fine-grained specification of a matching score.
- **Optimal server response size.** The servers only need to respond to a client query with a single element from the payload group. This is particularly useful in a streaming setting where the same client query is reused for multiple responses. In the case where the client only needs to learn the (unweighted) cardinality of the intersection, each server can send just $\lceil \log_2(n+1) \rceil$ bits to the client, where n is the client set’s size.
- **Linear client query size.** In our streaming and baseline solution, the client’s queries depend linearly on the client’s set size n and the security parameter. In a formal analysis, the security parameter is superlogarithmic in the server set size N , but the query size otherwise does not depend (even logarithmically) on N .
- **Hashing without leakage via queueing.** Hashing can be used to greatly decreases the amount of server work at the expense of leaking information about the client’s queries. We use queueing theory to delay certain responses to prevent this leakage, and show that the improvement in efficiency comes with only a small increase in expected wait time for queries to be processed.
- **Nearly linear client query size with nearly linear server computation.** We use our queueing solution to obtain a one-shot protocol with client communication $O(n \log \log n)$ and server computation $O(N \log \log n)$ (ignoring dependence on the security parameter), where n, N are the size of the set of the client and server respectively.

We compare the concrete and asymptotic efficiency to existing protocols in Tables 2 and 3 in Sect. 2.1, along with a more thorough discussion of related work. But first, we give a brief overview of the distinguishing features of each of our four protocols.

Baseline One-Shot Protocol. The baseline variant of our solution already offers several attractive efficiency features. It employs only symmetric cryptography. It involves a single round of interaction consisting of a query from the client to each server followed by a response from each server to the client. The size of the query is comparable to the size of the client’s small set Y ; concretely, in an

AES-based implementation the client sends roughly 128 bits for each bit of a keyword in Y . The answers are even shorter, and are comparable to the output size.

In terms of computation cost, our basic solution is very fast on the client side: in an AES-based implementation, the client performs roughly 4 AES calls for each bit of each keyword in Y . On the server side, with server set X , the number of AES calls scales linearly with $|X| \cdot |Y|$. While this is acceptable in settings where Y is small or when the client work is the bottleneck, e.g., when using massive parallelism on the server side (as in the recent FSS-based encrypted search system from [16]), this basic solution does not scale well when the size of Y grows.

Traditional approaches for improving server computation and making it comparable to $|X|$ employ batching techniques based on hashing or “batch codes” [1, 27, 40]. While these techniques offer a significant improvement in server computation, this comes at the cost of higher communication and setup requirements. Moreover, some of these techniques, such as those used in the related context of FSS for multi-point functions [7, 11, 37], can only efficiently apply to keywords taken from a polynomial domain size, and hence do not apply to most of our motivating applications.

Baseline Streaming Protocol. The minimal interaction pattern of our baseline protocol is particularly useful when the same query is reused for computing intersection with different sets X . We call this the *server streaming* setting. Our baseline streaming solution makes a more fine-grained use of this feature in a setting where both X and Y incrementally change with time, giving both server streaming and full streaming solutions.

This streaming solution is noteworthy for its efficiency. The total client communication and server computation cost of evaluating the intersection $X \cap Y$ over a series of t' time steps in which X and Y pick up some fixed number of new elements N' and n' , respectively, in each time step, is almost equal (in fact, asymptotic) to the cost of performing only the last step of this computation in the one-shot setting.

This efficiency is possible because, unlike other one-server PSI protocols, there is no preprocessing or setup required for the server’s set X , and so the server can use a different set X_t at each time step with no additional preprocessing cost.

Streaming PSI-WCA with Greedy Scheduling. Instead of the traditional techniques for reducing computational cost, we take the following leaner approach. Our starting point is the standard technique of partitioning the keyword domain into buckets, so that *on average* only a small number of keywords in Y fall in each bucket. This reduces the PSI task to roughly $|Y|$ instances of secure keyword search, each applying to a single bucket that contains roughly $|X|/|Y|$ elements from X . Because FSS outputs are *additively* secret-shared between the servers, the outputs for different buckets can be non-interactively summed up. However, a direct use of this approach requires the client either to reveal the number of keywords in Y that are mapped to each bucket, or alternatively to

“flatten the histogram” by using dummy queries. The former results in leaking a small amount of information about Y , whereas the latter has a significant toll on performance. To maximize performance while avoiding leakage, our solution flattens the histogram by deferring keywords from over-populated buckets to be processed with high priority in the next batch of queries, using a technique we call *greedy scheduling*. We use ideas from queueing theory to show that this approach can indeed give superior performance with no leakage, at the price of a very small expected latency in processing queries.

Queueing One-Shot Protocol. As an additional consequence of our streaming protocol with greedy scheduling, we obtain a one-shot protocol by having the client simulate the streaming protocol locally with dummy messages at every message except the first. The client runs this process until they have no deferred keywords, at which point they send the entire batch of queries as their message to the servers.

Using the queueing results of Sect. 5, we show in Sect. 4.4 that the number of iterations required for this simulated process leaks no information, and that, under an appropriate choice of parameters, the expected number of iterations is $O(\log \log n)$. Therefore the average client communication is $O(n \log \log n)$ and the server computation is $O(N \log \log n)$. This represents an improvement over the state-of-the-art unbalanced PSI protocols, which previously could achieve at best $O(N \log n)$ server computation with quasi-linear client communication.

1.2 Notation and PSI Models

The notation we use is comparable to that of existing PSI works, see e.g. [12, 40]. There are two servers, who each hold a large set of keywords X , and a client who holds a small set of keywords Y . In the basic version of the problem, the client learns the *cardinality* of the intersection of X and Y without revealing to any *single* server any information about Y (except an upper bound on its size) and without learning any additional information about X . (We assume clients to be semi-honest; lightweight protection against malicious clients can be obtained using techniques from [6, 9], and is out of scope for this work.) Following [40], we refer to this as *PSI cardinality* (PSI-CA). We also consider a generalization of PSI-CA in which the client associates to each keyword in Y an integer weight. Here the goal is for the client to obtain the sum of the weights of tokens in the intersection of X and Y . We refer to this extended variant as *PSI with weighted cardinality* (PSI-WCA).²

We then consider PSI-WCA in the *streaming* setting, where the sets change over time, presenting optimized solutions for this case. To simplify the presentation, we generally restrict our attention to the case where the client and server sets at time t are denoted by X_t and Y_t , and the parties desire to compute the

² This is similar to *private intersection-sum* [26], except that in typical use cases of the latter it is the server who holds the weights. Here both the weights and the output are owned by the client.

intersection of $X := (\cup_{i=1}^t X_i)$ and $Y := (\cup_{i=1}^t Y_i)$, or equivalently, the difference between the set $X \cap Y$ and the set computed in the previous epoch. However, we also show how a streaming variant appropriate for contact tracing, where tokens expire after T epochs, can also be efficiently implemented using our approach.

One of our variants of streaming PSI-WCA allows for *greedy scheduling* of tokens, where a stash of unmatched tokens is preserved from round to round, with guarantees that the stash size is bounded, and the distribution of which tokens are assigned to the stash from round to round reveals nothing about the tokens' values.

We also mention the simple decision version of PSI that returns a single bit representing whether or the intersection is empty. For this variant of PSI, our protocol requires only $\log_2(M+1)$ bits of communication from each server, where M is an upper bound on the intersection size. This is particularly useful in a setting where the set X changes rapidly while the set Y is unchanged, such as searching an internet feed for a set of targeted keywords.

Finally, we remark that both standard PSI, returning a bit vector, or PSI with payload, returning the payloads of items in the intersection, can be recovered from PSI-WCA by aggregating power sums, as done in [15].

1.3 Outline

In Sect. 2 we give background and related works. In Sect. 3 we provide our baseline PSI-WCA protocol, and show its extension to the server-streaming case. In Sect. 4, we give our streaming PSI-WCA with greedy scheduling protocol, and analyze the expected wait times of the underlying queues in Sect. 5. In Sect. 6 we discuss the implementation of our baseline protocol and associated benchmarks.

2 Background

2.1 Private Set Intersection

A Private Set Intersection (PSI) protocol [23] enables two parties to learn the intersection of their secret input sets X and Y , or some partial information about this intersection, without revealing additional information about the sets. Many variants of this problem have been considered in the literature. We will be interested in *unbalanced* PSI, where $|X| \gg |Y|$ and the output should be received by the party holding Y , to whom we refer as the *client*. We will further restrict the client to learn the *size* of the intersection or, more generally, a weighted sum over the intersection, while revealing no other information to the client.

PSI-Cardinality. The two functional variants of PSI we consider are PSI-Cardinality and PSI-Sum (or referred to as Weighted Cardinality in this work to distinguish the recipient). These variants are not trivially obtainable from PSI: cardinality reveals strictly less than the intersection so a PSI-CA protocol must hide the actual intersection, and PSI-Sum further requires the ability to handle a “payload” which is summed on top of hiding the intersection.

In the early work of De Cristofaro, Gasti, and Tsudik [17], the technique used to achieve PSI-CA was an extension of the DH-based PSI but with appropriate shuffling to hide which elements actually matched. The setting was meant for a balanced PSI in the sense that the communication was proportional to the sizes of both sets. The follow-up work by Ion et al. [26] was able to achieve PSI-Sum by using homomorphic encryption on the payload and expending another round to compute the sum of the matching payloads. Recent work [2, 21, 40] have focused on the unbalanced case of PSI-CA where one party has much fewer elements and therefore the communication should be proportional to the smaller set size only. Our result is the first to achieve this in 1 round in the 2-server model (with streaming) under the minimal assumption of one-way functions. We summarize our main PSI contribution in comparison to related PSI-CA works in Table 1.

Table 1. Comparison to state-of-the-art PSI-CA results. n = client set size, N = server set size, m = number of parties for multiparty solutions, DH = Diffie-Hellman style assumptions, OWF = minimal assumption of one-way functions, HE = homomorphic encryption. Security parameter factors omitted in O notation. * Bloom filter of size B . [†] Requires half round input-independent setup which is included. [‡] Delegation requires additional communication between servers which is omitted.

Result	Rounds	Comm.	Assumption	Servers	Flavor
[17]	1	$O(n + N)$	DH	1 server	PSI-CA
[26]	2	$O(n + N)$	DH+HE	1 server	PSI-CA/SUM
[18]	2	$O(mN)$	DH	m -party	PSI-CA
[2]	1.5	$O(n + B)^*$	DH	1 server	PSI-CA
[21]	$1.5^\dagger$	$O(n)^\ddagger$	OPRF	delegated	PSI-CA
[40] 1 server	2	$O(n)$	DH+HE	1 server	PSI-CA
[40] 2 server	2	$O(n)$	DH	2 server	PSI-CA
Ours	1	$O(n)$	OWF	2 server	PSI-CA/SUM

Unbalanced PSI. Most existing PSI protocols from the literature, including protocols based on linearly-homomorphic public-key encryption [26, 33], oblivious transfer [30, 35], or oblivious linear-function evaluation [24], are unsuitable for the highly unbalanced case because their communication costs scale linearly with the size of the bigger set X . This can be circumvented by PSI protocols that use simple forms of fully homomorphic encryption (FHE) [12, 13]. However, FHE-based solutions incur a high computational cost and their concrete communication overhead is large when the set Y is relatively small.

For example, when $|Y| = 128$ and $|X| = 2^{24}$, the FHE approach of [12] would require 3.1 MB of communication from the client and a response of 5.1 MB, while our approach requires a 0.26 MB message from the client in the baseline solution, and only $\log_2(M + 1)$ bits of a response from each server in the decision version

of PSI. Even the most efficient lattice based homomorphic encryption schemes require a query of at least 14 KB of communication and a response of at least 21 KB, see e.g. Table 2 of [34].

Another family of solutions, which includes [5], later extended by [36], and [29], are optimized for the online-offline model, requiring an expensive one-time setup message by the server that is linear in N , the size of the server’s message (although the constant in front of N has been improved by various compression techniques over the years, including Bloom filters and Cuckoo filters). Once the setup is complete, queries require a small amount of client communication and little to no server computation, but a larger amount of *client* computation than the other approaches discussed here. This approach is not compatible with server-side streaming because any changes to the server’s database require repeating the preprocessing to maintain security.

Recent works on contact tracing by Trieu et al. [40] and on private contact discovery by Demmler et al. [19], following a more general approach of Chor et al. [14], employ a Cuckoo hashing data structure to reduce the keyword search problem (of matching a single client token y_i with all N tokens x_j) to two invocations of PIR on a $2N$ -bit database. This work, like ours, makes use of two non-colluding servers; its main advantage over our baseline solution is that, using the efficient DPF EvalAll procedure from [9], these approaches reduce the number of AES invocations on the server side by a $O(\lambda)$ factor.

However, compared to our direct approach, this makes the solution much more complex. In particular, it requires an additional round of interaction and a bigger answer size and, perhaps most significantly, is not compatible with our streaming server mode of operation, because each time the server set changes another round of preprocessing is required. Additionally, these approaches require $O(n \log N)$ client communication, while our solutions require $O(n)$ or $O(n \log \log n)$ communication.

Table 2. Concrete efficiency features for unbalanced PSI protocols running on a server list X of size $N = 2^{24}$ and a client list Y of size $n = 128$, with client C and server S . Items marked with an asterisk are estimated from other parameter choices combined with an analysis of the theoretical dependence of those protocols on the parameter.

One-shot PSI-CA	Runtime and memory for $n = 128$, $N = 2^{24}$			
	Preprocessing (s)	Online (s)	Comm $C \rightarrow S$	Comm $S \rightarrow C$
[19]	2.69	0.49	2.4* MB	0
[12]	1368	9.1	3.1 MB	5.1 MB
[36]	333.62	0.42*	0.065 MB online	3.0 MB offline
Ours (baseline)	0	763.5*	0.26 MB	2 bytes
Ours (queueing)	0	4.5*	0.63 MB	2 bytes

Table 3. Asymptotic complexity measures for unbalanced PSI protocols with client C and server S . The parameters m and α in the second line are parameters associated with constructing a cuckoo hash table, and $m\alpha \approx n$.

One-shot PSI-CA	Asymptotic complexity of unbalanced PSI		
	Preprocessing	Online	Comm $C \rightarrow S$
[19]	λ exps	$O(N \log n / \lambda)$ AES calls	$O(\lambda n \log(N \log n / \lambda n))$
[12]	$O(N^2 / (m\alpha))$	$O(nN / (m\alpha))$	$n\lambda \log N$
[36]	$O(N)$ exps	$O(n)$ exps	$O(kn)$ online $O(kN)$ offline
Ours (baseline)	0	$O(\lambda knN)$ AES calls	λkn bits
Ours (queueing)	0	$O(\lambda kn \log \log n)$ AES calls	$O(\lambda kn \log \log n)$

Streaming PSI. Many of the common applications of PSI such as private contact discovery, password breach monitoring, and contact tracing are naturally suited to a *streaming* mode of operation, where relatively small changes to the database are made at each time step, and the client only needs to learn the changes in their intersection set. In spite of this, so far little direct attention has been paid to designing PSI protocols for the streaming setting.

The online-offline PIR-based protocols [29, 36] do have some advantages in the streaming setting when only the client values are being streamed, since the one-time setup cost no longer has to be paid in subsequent runs. However the $O(N)$ setup cost is large enough that it requires a large time window for streaming before the amortized cost is reasonable, and, critically, if the server set ever changes, the preprocessing step has to be repeated entirely.

The recently introduced notion of *incremental offline/online PIR* [32], which allows updates to the preprocessing for PIR schemes, may also be applicable to PIR-based PSI schemes, as long as the resulting protocol is still secure and the concrete metrics are good enough, but this avenue remains to be explored.

This streaming mode of operation has been formalized recently as *Updatable PSI* by Badrinarayanan et al., [4] which focuses on the case of balanced PSI, in contrast to this work. In the terminology of that paper, our streaming PSI protocol is capable of implementing both the *Addition-Only* and *Weak Deletion* variants. The first variant is where only new elements are added to the sets X , Y , and is the primary version we discuss, for ease of presentation, but we also treat the weak deletion variant, where tokens expire after some set time interval T , which has applications to contact tracing.

2.2 Function Secret Sharing

Our solution heavily builds on the tool of *function secret sharing* (FSS) [8]. A (2-party) FSS scheme for a function family $\mathcal{F}$ splits a function $f \in \mathcal{F}$ into two additive shares, where each share is a function that hides f and is described by a short key. More concretely, a function $f : \{0, 1\}^n \rightarrow \mathbb{G}$ for some finite Abelian group $\mathbb{G}$ is split into two functions f_0, f_1 , succinctly described by keys k_0, k_1 respectively, such that: (1) each key k_b hides f , and (2) for every $x \in \{0, 1\}^n$ we have $f(x) = f_0(x) + f_1(x)$.

We will use FSS for the family $\mathcal{F}$ of *point functions*, where a point function $f_{\alpha,\beta}$ evaluates to β on the special input α and to 0 on all other inputs. An FSS scheme for point functions is referred to as a *distributed point function* (DPF) [25]. We will let $\text{DPF.Gen}(1^\lambda, \alpha, \beta)$ denote the DPF key generation algorithm, which given security parameter λ and the description of a point function $f_{\alpha,\beta}$ outputs a pair of keys (k_0, k_1) (where here we assume for simplicity that the group $\mathbb{G}$ is fixed). We use DPF.Eval to denote the evaluation algorithm that on input (k_b, x) returns an output share y_b such that $y_0 + y_1 = f_{\alpha,\beta}(x)$.

We rely on the best known DPF construction from [9], which has the following performance features with an AES-based implementation: The length of each key is roughly $128n$ bits (some savings are possible when the group $\mathbb{G}$ is small). The cost of DPF.Gen is roughly $4n$ AES calls, whereas the cost of DPF.Eval is roughly n AES calls, where both can be implemented using fixed-key AES.

A direct application of DPF for secure keyword search in a 2-server setting was suggested in [8, 25]. Secure keyword search can be viewed as an extreme instance of unbalanced PSI where $|Y| = 1$. Here we generalize this in two dimensions: first, we allow a client to have multiple keywords, thus supporting a standard PSI functionality. We propose different methods for improving the cost of independently repeating the basic keyword search solution for each keyword in the client set Y . Second, we exploit the ability to use a general group $\mathbb{G}$ for implementing a *weighted* variant of PSI where each of the client's secret keywords has an associated secret weight. In fact, we use a product group for revealing multiple weighted sums.

3 Baseline Solution

3.1 The One-Shot Case

The functionality we realize is an extended “weighted” version of PSI Cardinality that attaches a weight to each client item.

Functionality PSI-WCA:

- INPUTS:
 - Each of the two servers S_0, S_1 holds the same *tokens* set $X = \{x_1, \dots, x_N\}$ of k -bit strings.
 - Client holds a set Y of pairs of the form $Y = \{(y_1, w_1), \dots, (y_n, w_n)\}$, where each y_i is a k -bit token and each w_i is an element of an Abelian group G (typically we choose to work over the integers with large enough modulus to prevent wraparound, but using an arbitrary group allows for the ability even to support product groups with multiple slots encoding different pieces of information). If the cardinality of Y is less than n , it is padded with an additional $n - |Y|$ dummy elements.
- OUTPUTS: Client outputs the sum of the weights of the tokens in the intersection; namely, the output is $w = \sum_{i: y_i \in X} w_i$ where summation is in the group G . Servers have no output.
- LEAKAGE: The size parameters leaked to the adversary are k, n, G .

The Baseline Solution. We follow the approach of Boyle et al. for secure keyword search via a direct use of distributed point functions (DPFs) [9, 25]. This departs from the approach of Chor et al. [14] and Trieu et al. [40], which uses a data structure (Cuckoo Hashing in [40]) for reducing keyword search to private information retrieval (PIR). The direct DPF-based approach requires one round of interaction and accommodates the weighted case with almost no extra overhead.

While we describe the protocol using direct interaction of the client with the two servers S_0, S_1 , in practice it may be preferable to have the client interact only with S_0 and have (encrypted) communication to and from S_1 routed via S_0 . In the following we use λ to denote a security parameter, and we consider security against a *passive* (aka semi-honest) adversary corrupting either one of the two servers or the client.

Protocol PSI-WCA:

- CLIENT-TO-SERVERS COMMUNICATION:
 1. For each client input pair (y_i, w_i) , Client generates a pair of DPF keys $(k_i^0, k_i^1) \leftarrow \text{DPF.Gen}(1^\lambda, y_i, w_i)$.
 2. Client sends the n keys k_i^b to server S_b .
- SERVERS-TO-CLIENT COMMUNICATION:
 1. Each server S_b computes $a'_b := \sum_{j=1}^N \sum_{i=1}^n \text{DPF.Eval}(k_i^b, x_j)$, where summation is in G . (Each such invocation of DPF.Eval can be implemented with roughly k invocations of fixed-key AES and does not require any communication between servers.)
 2. Letting $r \in_R G$ be a fresh secret random group element shared by the two servers, S_0 sends to Client $a_0 := a'_0 + r$ and S_1 sends $a_1 := a'_1 - r$, where addition and subtraction are in G . This can be generated using a shared pseudorandom sequence known only to the servers (e.g., a common PRF seed).
- CLIENT OUTPUT: Client outputs $w = a_0 + a_1$, where summation is in G .

The correctness of the above protocol is easy to verify. Security against a single server follows directly from the security of the DPF. Security against the Client follows from the blinding by r , which makes the pair of answers received by Client random subject to the restriction that they add to the output. We now discuss the protocol's efficiency.

Performance. Using an AES-based implementation of the DPF from [9], the above protocol has the following performance characteristics:

- ROUNDS: The protocol requires a single round of interaction, where Client sends a query to each server S_b and gets an answer in return. Client's query can be reused when the client's input Y does not change, even when the server input X changes.
- COMMUNICATION: Client sends each server $\approx 128 \cdot kn$ bits and gets back a single element of G from each server.

- **COMPUTATION:** Client performs $\approx 2kn$ (fixed-key) AES calls to generate the queries. The cost of reconstructing the answer is negligible. The computation performed by each server is dominated by $\approx knN$ AES calls. For modern processors (see Footnote 12 of [40]), each AES call requires 10 machine cycles, which enables $360 \cdot 10^6$ AES calls per second on a 3.6 GHz machine. This can be further sped up via parallelization.

3.2 Baseline Streaming Unbalanced PSI

This baseline solution can be extended directly into an incremental mode that captures a dynamic “streaming” version of the problem where the sets X and Y held by the servers and the client change in each time epoch (say, each day) by N' and n' respectively. We typically consider $N' \ll N$ and $n' \ll n$. For simplicity, we consider the case where at time step t , sets X_t and Y_t are appended to X and Y respectively, so that the set $X = \cup_{i=1}^t X_i$ is the union of all sets input by the server so far, and similarly $Y = \cup_{i=1}^t Y_i$. Then the client receives the sum of the weights of the terms in $X \cap Y$ minus the sum computed in the previous round.

More formally, we define the functionality below:

Functionality: Streaming PSI-WCA:

- **INPUTS:**
 - At each time step t , each of the two servers S_0, S_1 holds the same *tokens* $X_t = \{x_1, \dots, x_N\}$ of k -bit strings.
 - At each time step t , client holds a set Y_t of pairs of the form $Y = \{(y_1, w_1), \dots, (y_n, w_n)\}$, where each y_i is a k -bit token and each w_i is an element of an Abelian group G . If the cardinality of Y_t is less than n' , it is padded with an additional $n' - |Y_t|$ dummy elements.
- **OUTPUTS:** For each t , define $S_t := (\cup_{i=1}^t X_i) \cap (\cup_{i=1}^t Y_i)$. Then at time step t , the client outputs the sum of the weights w_i of the tokens in $S_t \setminus S_{t-1}$. Servers have no output.
- **LEAKAGE:** The size parameters leaked to the adversary are k, n, G .

The protocol to realize this functionality is a straightforward extension of the baseline functionality. The client in time step t generates and communicates new queries only for the n' tokens of Y_t introduced in that epoch. Then servers compute DPF.EVAL only to match the new n' client tokens with all tN' server tokens and the new N' server tokens with all tn' client tokens. Correctness follows from the fact that $S_t \setminus S_{t-1} = X_t \cap (\cup_{i=1}^t Y_i) \cup Y_t \cap (\cup_{i=1}^t X_i)$, and security follows immediately from the security of the PSI-WCA protocol.

This streaming solution reduces the number of AES calls per epoch on the client side from tkn' to kn' , and on the server side from $knN = t^2kn'N'$ to roughly $k \cdot (n'N + nN') \approx 2ktn'N'$. The client communication and server computation per epoch are reduced by a factor of t and $t/2$, respectively, compared to the one-shot solution. If the protocol runs for a total of t' epochs, summing from $t = 1$ to $t = t'$ we see that the total client communication is equal to $t'kn'$

and the total server computation is equal to $k(t'^2 + t')n'N'$, so that the streaming functionality can be added with no asymptotic increase in cost.

We remark that the requirement that $|X_i| = N'$ and $|Y_i| = n'$ be constant across all time steps is unnecessary to the functionality and protocol definitions given above, and only the efficiency analysis changes. In particular, we note that if $X_1 = N$ and $X_i = \emptyset$ for $i > 1$, we require zero communication from the client in subsequent epochs and $kn'N$ work per epoch. This is the *server-streaming* PSI solution discussed in Sect. 1.1.

In some applications, such as contact tracing, we instead desire tokens to expire after T epochs, and the goal is to compute the PSI-WCA functionality in the sliding window corresponding to each epoch, where the inputs consist of the $N = TN'$ and $n = Tn'$ tokens collected during the last T epochs by the servers and client, respectively. The protocol above can be modified to realize this alternate functionality by having the servers discard their client and server tokens after T epochs. The rest of the protocol is identical. This reduces the client communication and server work by a factor of T throughout the course of the protocol.

4 Unbalanced PSI-WCA with Greedy Scheduling

4.1 Overview

Similarly to the simple use of hash functions and batch codes for amortizing the server computation of multi-query PIR [1, 27], and similar techniques for standard PSI, one can use a similar approach for amortizing the server computation in PSI-WCA. The idea is to randomly partition the token domain into a small number of buckets m via a public hash function $H : \{0, 1\}^k \rightarrow [m]$ (typically $m \approx n$), and let the client match each token y_i only with the tokens in bucket $H(y_i)$. To make this possible, we need the client either to reveal the number of tokens y_i mapped to each bucket (which leaks a small amount of information about Y to the servers) or to add dummy tokens y_j^* to ensure all buckets have a fixed size except with small failure probability. Compared to more sophisticated data structures such as Cuckoo hashing, discussed in Sect. 2.1, this approach does not require additional interaction and is well-suited to the incremental mode in which new server tokens are added on the fly.

When buckets are overfilled, we can repeat the process until all tokens are processed, which gives our $O(n \log \log n)$ communication $O(N \log \log n)$ server work solution which we describe in Sect. 4.4. In our streaming solution with greedy scheduling, the protocol instead deals flexibly with overfull buckets: after hashing all tokens, if the client holds any buckets with more than b tokens, the extra tokens are stashed and added to the next day's batch.

The hashing solution increases communication to $m \cdot b$ tokens compared to n for the baseline solution, so we would like to keep $\frac{m \cdot b}{n}$ as small as possible. The incremental mode is therefore well-suited to the hashing solution. However, decreasing b increases the number of tokens that need to be stashed each day. This is the fundamental trade-off explored in Sect. 5.

To keep the buckets more evenly balanced, the protocol can instead choose $c > 1$ public hash functions. The client then inserts their n tokens into the m buckets one by one, hashing each token c times and using a greedy algorithm to determine which of the c candidate buckets to place the token into. The servers then check each of their tokens y against $H_1(y), \dots, H_c(y)$. This increases server work to $O(bcN)$, but shrinks the stash size, and hence the expected wait time.

Another consequential design choice, in both the $c = 1$ and $c > 1$ cases, is whether to fix the hash functions H_i throughout the procedure, or to choose new hash functions each day. Which choice is optimal depends on the choice of parameters, as we see in Sect. 5.3.

4.2 The Greedy Scheduling Approach

In this section, we describe our solution using *greedy scheduling* and explore optimizations and tradeoffs. The greedy scheduling protocol is a procedure for improving the efficiency of the hashing-based solution by scheduling some client tokens each epoch to be processed in a later epoch.

Other hashing-based solutions either leak some information about queries or use hashing to maintain a permanent data structure on the server side (such as a Cuckoo hash table), which requires a costly initial setup and is ill-suited for server-side streaming. Our approach uses hashing on the client side and fixes the bin size at a small constant (usually $b = 3$). The scheduling is *greedy* in that we schedule only those client tokens which can be most efficiently processed for the current epoch, and transfer the remaining tokens to a stash where they will be scheduled later. We show in Sect. 5.3 that this causes very little delay in processing in exchange for a large improvement in efficiency.

We begin with a formal definition of the streaming functionality for greedy scheduling. This definition requires a distribution ω on functions from $\{1, \dots, n+s\} \rightarrow \{0, 1\}$, where n is the size of the client's list and s is the stash size. This distribution should be thought of as a representation of the queueing process; after sampling a function $f \in \omega$, then the set $f^{-1}(1)$ corresponds to elements which are not processed in that round of the queueing process, and are instead scheduled for a future round, and transferred to the stash.

To simplify the presentation, we formally define a PSI-WCA functionality with greedy scheduling that is streaming with respect to the client, with the server's set X held constant. The baseline streaming approach described in Sect. 3.2 can be applied on top of the approach described here, and, as in Sect. 3.2, this can be done without any increase in asymptotic client communication or server computation.

Functionality: Streaming PSI-WCA with greedy scheduling:

– INPUTS:

- At each time step t , each of the two servers S_0, S_1 holds the same *tokens* $X = \{x_1, \dots, x_N\}$ of k -bit strings.

- At each time step t , client holds a set Y_t of pairs of the form $Y = \{(y_1, w_1), \dots, (y_n, w_n)\}$, where each y_i is a k -bit token and each w_i is an element of an Abelian group G . If the cardinality of Y_t is less than n , it is padded with an additional $n - |Y_t|$ dummy elements.
- As additional inputs, a stash size s and a distribution ω on functions from $\{1, \dots, n+s\} \rightarrow \{0, 1\}$ are given, with the property that if f has nonzero probability under ω , then $\sum_{i=1}^{n+s} f(i) \leq s$.
- **OUTPUTS:** For each time step t , sample a function $f_t \in \omega$. Set $\bar{Y}_1 := Y_1$, append s “dummy” elements, and assign some arbitrary bijection τ_1 from $\{1, \dots, n+s\}$ to the elements of $\bar{Y}_1$. Then recursively define the set $\bar{Y}_{t+1} = Y_{t+1} \cup \tau_t(f_t^{-1}(1))$ and τ_{t+1} to be some arbitrary bijection from $\{1, \dots, n+s\}$ to the elements of $\bar{Y}_{t+1}$. Now define $S_t := X \cap (\cup_{i=1}^t \bar{Y}_i)$, and at time step t , the client outputs the sum of the weights w_i of the tokens in $S_t \setminus S_{t-1}$. Servers have no output.
- **LEAKAGE:** The size parameters leaked to the adversary are k, n, G , along with the distribution ω and the stash size s .

We do not give the distribution ω explicitly; it is instead inferred from our description of the protocol.

4.3 Protocol Description

For our streaming protocol with greedy scheduling, we introduce the following notation. For a client with n tokens and a server with N tokens, with each token k bits, the client will hash their tokens into $m = \alpha n$ bins of size b , for some constant $\alpha < 1$. This will be done using c hash functions, choosing the bin with the smallest size if $c > 1$. Additionally, we use the bit R to distinguish between two variants of the protocol, where the hash function used is rerandomized after each epoch ($R = \text{TRUE}$) and where the hash function is kept constant ($R = \text{FALSE}$). The protocol then is close to our original PSI-WCA protocol, but with the work of DPF.Eval only done on bins where the client and server tokens could match. We give the protocol below.

Protocol Streaming PSI-WCA with greedy scheduling:

- **CLIENT-TO-SERVERS COMMUNICATION:**
 1. The client hashes their inputs, along with any elements in the stash, into m bins of size b , applying c hash functions to obtain c candidate bins and greedily assigning the element to the bin with the fewest elements. Any elements that overflow their bins are placed back into the stash. Then, for each client input pair $(y_{i,j}, w_{i,j})$, and the dummy elements, the client generates a pair of DPF keys $(k_{i,j}^0, k_{i,j}^1) \leftarrow \text{DPF.Gen}(1^\lambda, y_{i,j}, w_{i,j})$.
 2. Client sends the mb keys $k_{i,j}^d$ to server S_b .
- **SERVERS-TO-CLIENT COMMUNICATION:**
 1. Each server S_d , for each element x_k , hashes x_k by each of the c hash functions to obtain a list of indices $I_k = \{i_1, \dots, i_c\}$, and computes $a_{k,d} := \sum_{i \in I_k} \sum_{j=1}^b \text{DPF.Eval}(k_{i,j}^d, x_k)$, where summation is in G . They then compute $a'_d = \sum_k a_{k,d}$.

2. Letting $r \in_R G$ be a fresh secret random group element shared by the two servers, S_0 sends to Client $a_0 := a'_0 + r$ and S_1 sends $a_1 := a'_1 - r$, where addition and subtraction are in G . This can be generated using a shared pseudorandom sequence known only to the servers (e.g., a common PRF seed).
- CLIENT OUTPUT: Client outputs $w = a_0 + a_1$, where summation is in G .

This is a realization of the streaming PSI with greedy scheduling functionality described above, where the stash size s and the distribution ω are determined by the queueing processes described in Sect. 5. In particular, we show in those sections that $s = O(n)$, and, when $Y_i = \emptyset$ for $i > 1$, $c = 2$, and R (the rerandomization bit) is equal to FALSE, all tokens will be processed in expected time $O(\log \log n)$.

Correctness follows because each client token is guaranteed to be processed eventually, and will match a server token x_k only if the client token lies in one of the queues $H_i(x_k)$, for $i \in \{1, \dots, c\}$. Security against a single server and against a client follows from an identical argument to that in Sect. 3.1. We analyze performance when the client's set size is n per epoch, and the server's set size is a constant N .

Performance. Using an AES-based implementation of the DPF from [9], the above protocol has the following performance characteristics:

- ROUNDS: The protocol requires a single round of interaction per epoch, where Client sends a query to each server S_b and gets an answer in return.
- COMMUNICATION: Client sends each server $\approx 128 \cdot k\alpha bn = O(kn)$ bits per epoch and gets back a single element of G from each server.
- COMPUTATION: Client performs $\approx 2k\alpha bn = O(kn)$ (fixed-key) AES calls to generate the queries. The cost of reconstructing the answer is negligible. The computation performed by each server is dominated by $\approx kbcN$ AES calls.

4.4 Queueing One-Shot PSI Protocol

Our one-shot PSI protocol can be constructed as a corollary of this protocol, setting $Y_i = \emptyset$ for $i > 1$ and $|Y_1| = n$. By the proof of Proposition 4, given in the full version of this paper, we have some value $\kappa = O(\log \log n)$ such that all elements in Y_1 have wait time less than κ except with probability $\exp(O(-n))$. The client re-samples a key for the hash function until this holds, and then runs their half of the protocol for κ rounds and sends the resulting κmb keys to the server.

The server proceeds as if in the streaming with greedy scheduling case, performing $\kappa \cdot bcN$ total work, adding the outputs from each round, and sending the result to the client. We take $b = 3$ and $c = 2$. Since $\kappa = O(\log \log n)$, the client communication is $O(n \log \log n)$ and the server computation is $O(N \log \log n)$, as desired.

5 Analyzing Expected Wait Times Under Greedy Scheduling

5.1 Streaming and Bucketing

The greedy scheduling streaming protocol can be expressed in purely combinatorial terms, so that questions about average wait time and longest wait time become questions in queueing theory, allowing us to abstract away all the mechanics of PSI and FSS. We explain this in more detail in the full version of this paper [20].

The fundamental complication introduced by greedy scheduling is that some tokens will take longer than one day to be processed. Additionally, as time passes, the backlog of unprocessed tokens builds up, and the wait time increases. To understand the tradeoffs involved, we analyze the expected average and worst case wait times. When we choose parameters appropriately, the backlog in the stash reaches a steady state of reasonable size, the average wait time is small, and very large wait times are extraordinarily rare.

The results in Sect. 5.3 are a mixture of calculations using known results and extensions of existing results. We give the proofs of these results in the full version of this paper [20], and refer the interested reader to [22, 31] for some of the background in queueing theory and probability, [3] for the crucial $O(\log \log n)$ bound for queues using multiple hash functions, [10] for the derivation of the steady state of the discrete time GI-D-c queue needed for Proposition 2, and [28] for a survey of prior work and additional analytical tools.

5.2 Setting

In our analysis, we consider two metrics under four scenarios. We measure *expected wait time* and *expected worst-case wait time*, both once a steady state has been reached. Formally, the random scheduling process induces some probability distribution $W_{t,n}$ on the n -length vector of wait times of the tokens inserted at time t . If $W_{t,n}$ converges to a limiting distribution W_n as $t \rightarrow \infty$, we call W_n the steady state. We then define by an abuse of notation the expected wait time $\mathbb{E}[W] := \lim_{n \rightarrow \infty} \mathbb{E}[\frac{1}{n} \sum_{w \in W_n} w]$, and the expected worst case wait time $\mathbb{E}[\max W] := \mathbb{E}[\max_{w \in W_n} w]$.

The four scenarios we consider are (i) Fixing $c = 1$ hash function to distribute tokens, (ii) Refreshing the $c = 1$ hash function each day, (iii) fixing $c > 1$ hash functions and (iv) refreshing $c > 1$ hash functions each day.

For each scenario, we consider parameters n , the number of tokens, m the number of buckets, b the bin size, and the occupancy ratio $\alpha := n/(bm)$. Additionally we have c , the number of hash functions, and R , a single bit representing whether or not we re-randomize each day.

We performed Monte Carlo simulations of this procedure to estimate the expected wait time for fixed values of n . Using integral approximations and a computer algebra package, we can also compute estimates for the expected wait time as $n \rightarrow \infty$ for various choices of parameters. We give the results of both

kinds of experiments in Table 4, and in Table 5 we present the results given below for the expected and expected worst-case wait times.

5.3 Results

Summary of Results. The bounds on expected wait times and expected worst-case wait times we give here are primarily calculations using existing work. Proposition 3 is an extension of work by Azar et al. ([3]).

- The expected wait time decreases exponentially with b for $c = 1$ hash function, and doubly exponentially with b for $c > 1$ hash functions.
- The expected worst-case wait time is $\Theta(\log n)$ for each scenario except $c > 1$ fixed hash functions, where it is $\Theta(\log \log n)$.

Rerandomization of hash function, $c = 1$

Proposition 1. *When $c = 1$ and the rerandomization bit $R = \text{TRUE}$, and $e\alpha < 1$,*

$$\mathbb{E}[W] \leq (e\alpha)^b$$

and

$$\mathbb{E}[\max W] \leq -\frac{1 + \log n}{b \log \alpha + b} + 1.$$

Fixed hash function, $c = 1$

Proposition 2. *Fix $\alpha < 1$. When $c = 1$ and the rerandomization bit $R = \text{FALSE}$,*

$$\mathbb{E}[W] \leq \frac{C(\alpha, b)}{\sqrt{2\pi b}} \text{Li}_{\frac{1}{2}}(\alpha^b e^{(1-\alpha)b}),$$

where $\text{Li}_{\frac{1}{2}}$ is the polylogarithm of order $\frac{1}{2}$ and $C(\alpha, b) := 1 + (\alpha - 1) \frac{b}{b+1} < 1$. In particular, we have:

- (Estimate for α bounded away from 1.) For $0 < \alpha < 1 - \delta$,

$$\mathbb{E}[W] \leq \frac{C(\alpha, b)}{\sqrt{2\pi b}} (\alpha^b e^{(1-\alpha)b}) + O_{\delta, b}(\alpha^b e^{(1-\alpha)b})^2$$

for any fixed $0 < \delta < 1$.

- (Growth rate as $\alpha \rightarrow 1^-$.) For $1 - (2b)^{-\frac{1}{2}} < \alpha < 1$,

$$\mathbb{E}[W] = O\left(\frac{1}{b(1-\alpha)}\right).$$

For any positive constant $C < 1$,

$$\mathbb{E}[\max W] \leq -\frac{\log n}{C \log \alpha} + O(1),$$

where the implied constant depends on C, α , and b .

Rerandomization of hash function, $c > 1$

Proposition 3. *When $c > 1$ and the rerandomization bit $R = \text{TRUE}$, and α and b are chosen such that $0 < \alpha b < (ec^b)^{-1/(c^b-1)}$, then*

$$\mathbb{E}[W] \leq e(\alpha b)^{c^b-1}$$

and

$$\mathbb{E}[\max W] \leq -\frac{1 + \log n}{(c^b - 1) \log(\alpha b) + 1} + 1.$$

Fixed hash function, $c > 1$

Proposition 4. *When $c > 1$ and the rerandomization bit $R = \text{FALSE}$, and α and b are chosen such that $0 < \alpha b < 1 - \frac{1}{b \cdot 2^{c-2}}$, then*

$$\mathbb{E}[W] \leq \frac{(\alpha b)^{c^b}}{1 - (\alpha b)^{c^b/2}}$$

and

$$\mathbb{E}[\max W] \leq \frac{b \log \log n}{\log c} + O(1).$$

Remark 1. The upper bounds on α in Proposition 1 and on αb in Propositions 3 and 4 were chosen to emphasize reasonable parameter choices and to increase the readability of the statements of results and proofs.

The full picture is messier; for example, it is possible to show that, in the $c > 1$ and $R = \text{FALSE}$ regime, as $\alpha \rightarrow 1$, and b and c are fixed, we have $\mathbb{E}[W] \approx \alpha^{bc}$, rather than α^{c^b} . Describing the transitions in behavior of $\mathbb{E}[W]$ on this parameter space is regrettably outside the scope of this paper.

Table 4. Experimental wait times

Wait time for (α, b, n)	(0.313, 2, 25000)	$\lim_{n \rightarrow \infty} (0.313, 2, n)$	(0.417, 3, 25000)	$\lim_{n \rightarrow \infty} (0.417, 3, n)$
$c = 1, R = \text{TRUE}$	0.05319	0.05567	0.04512	0.04604
$c = 1, R = \text{FALSE}$	0.05904	0.06022	0.04961	0.05063
$c = 2, R = \text{TRUE}$	0.00073	0.00075	0.00009	0.00008
$c = 2, R = \text{FALSE}$	0.00076	0.00074	0.00007	0.00008

Table 5. Upper bounds on asymptotic wait times, as a function of (α, b, c)

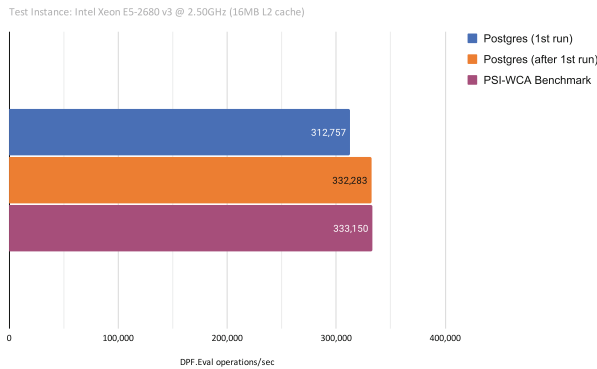
Wait time	Average	Worst-case
$c = 1, R = \text{TRUE}$	$(\alpha e)^{-b}$	$-\frac{1+\log n}{b \log \alpha} + 1$
$c = 1, R = \text{FALSE}$	$\frac{1}{\sqrt{2\pi b}} \text{Li}_{\frac{1}{2}}(\alpha^b e^{(1-\alpha)b})$	$-\frac{\log n}{C \log \alpha} + O(1)$
$c > 1, R = \text{TRUE}$	$(\alpha b)^{c^b - 1}$	$-\frac{1+\log n}{(c^b - 1) \log(\alpha b) + 1} + 1$
$c > 1, R = \text{FALSE}$	$\frac{(\alpha b)^{c^b}}{1 - (\alpha b)^{c^b/2}}$	$\frac{b \log \log n}{\log c} + O(1)$

6 Implementation and Benchmarks

In this section we will present our PSI-WCA implementation for the isolated back end servers. We are using PostgreSQL as our database engine for storing the tokens that are transmitted from the infected users and our test instances are configured as follows:

- CPU: Intel Xeon E5-2680 v3 @ 2.50 GHz with 16 MB L2 cache - 10 logical cores.
- RAM & Storage: 30 GB RAM, 60 GB SSD hard disk.
- OS: PostgreSQL 12 running on Debian 10.

Each of these test instances are virtual machines that are hosted in an OpenStack cloud platform provided by the Texas Advanced Computing Center (TACC) ([38, 39]). The operations defined by PSI-WCA are built as a custom extension that is loaded directly by the Postgres database engine. More specifically, the extension defines a custom aggregate function that accepts a key split k and the tokens database table T as input and operates directly in T in order to calculate $\sum_{u_i \in T} \text{DPF.Eval}(k, u_i) \bmod G$.

**Fig. 1.** PSI-WCA PostgreSQL extension performance

Each logical core in our test instance is capable of achieving a theoretical maximum of 333,150 DPF.Eval operations per second when running a simple

benchmark that operates on 1000 tokens and 1000 key splits that have been preloaded in RAM. For evaluating our PostgreSQL extension we generated a test data set of 100,000,000 infected tokens that has a total size of 8056 MB. In Fig. 1 it is shown that the first run of DPF.Eval on this test data set results in 312,757 operations per second which is 6.12% lower than the theoretical maximum. However, for any subsequent run our PostgreSQL extension utilizes memory caching and additional optimizations in the query execution plan which allows it to reach 332,283 DPF.Eval operations per second, thus decreasing the overhead to 0.26%.

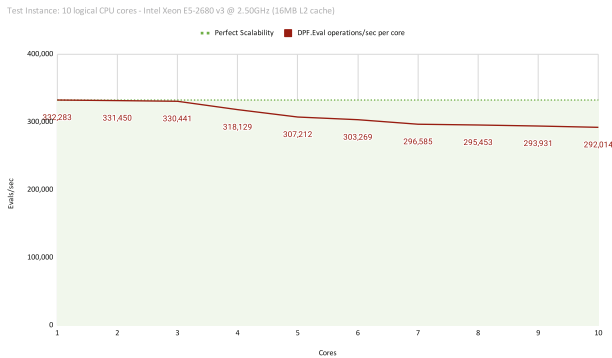


Fig. 2. PSI-WCA PostgreSQL extension DPF.Eval operations/sec per core

We have also evaluated how the DPF.Eval operations per second are affected when utilizing multiple cores in our test instance. For this benchmark we utilized the native functionality provided by PostgreSQL that spawns multiple worker processes for calculating the query results and is defined using the `max_parallel_workers_per_gather` parameter. This introduces some additional small overhead since PostgreSQL reserves a portion of CPU time in the main query process for managing the background parallel workers. As seen in Fig. 2, the DPF.Eval operations per second scale linearly when utilizing more than one core with a maximum overhead of 12% when utilizing all 10 cores in the test instance. For simplicity, the numbers in Table 2 are computed based on using a single core.

Acknowledgments. We thank Paul Bunn, Quinn Grier, and Will Yang for helpful and insightful conversations. This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA) and NIWC Pacific under contract N66001-15-C-4065 and by DARPA, AFRL/RIKD, USAF, and AFMC under contract FA8750-18-C-0054. Y. Ishai was supported in part by ERC Project NTSC (742754), BSF grant 2018393, and ISF grant 2774/20. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes not withstanding any copyright notation thereon. The views, opinions and/or findings expressed are those

of the author and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

References

1. Angel, S., Chen, H., Laine, K., Setty, S.T.V.: PIR with compressed queries and amortized query processing. In: Proceedings of the 2018 IEEE Symposium on Security and Privacy, SP 2018, 21–23 May 2018, San Francisco, California, USA, pp. 962–979. IEEE Computer Society (2018). <https://doi.org/10.1109/SP.2018.00062>
2. Angelou, N.: Asymmetric private set intersection with applications to contact tracing and private vertical federated machine learning. CoRR abs/2011.09350 (2020). <https://arxiv.org/abs/2011.09350>
3. Azar, Y., Broder, A.Z., Karlin, A.R., Upfal, E.: Balanced allocations. In: Proceedings of the Twenty-Sixth Annual ACM Symposium on Theory of Computing, pp. 593–602 (1994)
4. Badrinarayanan, S., Miao, P., Xie, T.: Updatable private set intersection. Proc. Priv. Enhanc. Technol. **2022**(2), 378–406 (2022). <https://doi.org/10.2478/popets-2022-0051>
5. Baldi, P., Baronio, R., Cristofaro, E.D., Gasti, P., Tsudik, G.: Countering GAT-TACA: efficient and secure testing of fully-sequenced human genomes. In: Chen, Y., Danezis, G., Shmatikov, V. (eds.) Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, 17–21 October 2011, pp. 691–702. ACM (2011). <https://doi.org/10.1145/2046707.2046785>
6. Boneh, D., Boyle, E., Corrigan-Gibbs, H., Gilboa, N., Ishai, Y.: Lightweight techniques for private heavy hitters. IACR Cryptol. ePrint Arch. 2021, 17 (2021). Conference version: IEEE S&P 2021
7. Boyle, E., Couteau, G., Gilboa, N., Ishai, Y.: Compressing vector OLE. In: CCS 2018, pp. 896–912 (2018)
8. Boyle, E., Gilboa, N., Ishai, Y.: Function secret sharing. In: Oswald, E., Fischlin, M. (eds.) EUROCRYPT 2015. LNCS, vol. 9057, pp. 337–367. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46803-6_12
9. Boyle, E., Gilboa, N., Ishai, Y.: Function secret sharing: improvements and extensions. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) ACM CCS 2016: 23rd Conference on Computer and Communications Security, Vienna, Austria, 24–28 October 2016, pp. 1292–1303. ACM Press (2016). <https://doi.org/10.1145/2976749.2978429>
10. Bruneel, H., Wuyts, I.: Analysis of discrete-time multiserver queueing models with constant service times. Oper. Res. Lett. **15**(5), 231–236 (1994)
11. de Castro, L., Polychroniadou, A.: Lightweight, maliciously secure verifiable function secret sharing. In: Dunkelman, O., Dziembowski, S. (eds.) EUROCRYPT 2022. LNCS, vol. 13275, pp. 150–179. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-06944-4_6
12. Chen, H., Huang, Z., Laine, K., Rindal, P.: Labeled PSI from fully homomorphic encryption with malicious security. In: ACM CCS 2018, pp. 1223–1237 (2018)
13. Chen, H., Laine, K., Rindal, P.: Fast private set intersection from homomorphic encryption. In: ACM CCS 2017, pp. 1243–1255 (2017)
14. Chor, B., Gilboa, N., Naor, M.: Private information retrieval by keywords. IACR Cryptol. ePrint Arch. **1998**, 3 (1998). <http://eprint.iacr.org/1998/003>

15. Corrigan-Gibbs, H., Boneh, D., Mazières, D.: Riposte: an anonymous messaging system handling millions of users. In: 2015 IEEE Symposium on Security and Privacy, SP 2015, pp. 321–338 (2015)
16. Dauterman, E., Feng, E., Luo, E., Popa, R.A., Stoica, I.: Dory: an encrypted search system with distributed trust. Cryptology ePrint Archive, Report 2020/1280 (2020). <https://eprint.iacr.org/2020/1280>
17. De Cristofaro, E., Gasti, P., Tsudik, G.: Fast and private computation of cardinality of set intersection and union. In: Pieprzyk, J., Sadeghi, A.-R., Manulis, M. (eds.) CANS 2012. LNCS, vol. 7712, pp. 218–231. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-35404-5_17
18. Debnath, S.K., Stanica, P., Kundu, N., Choudhury, T.: Secure and efficient multiparty private set intersection cardinality. Adv. Math. Commun. **15**(2), 365–386 (2021)
19. Demmler, D., Rindal, P., Rosulek, M., Trieu, N.: PIR-PSI: scaling private contact discovery. Proc. Priv. Enhanc. Technol. **2018**(4), 159–178 (2018). <https://doi.org/10.1515/popets-2018-0037>
20. Dittmer, S., et al.: Authenticated garbling from simple correlations. Cryptology ePrint Archive (2022)
21. Duong, T., Phan, D.H., Trieu, N.: Catalic: delegated PSI cardinality with applications to contact tracing. In: Moriai, S., Wang, H. (eds.) ASIACRYPT 2020. LNCS, vol. 12493, pp. 870–899. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-64840-4_29
22. Eisenberg, B.: On the expectation of the maximum of IID geometric random variables. Stat. Probab. Lett. **78**(2), 135–143 (2008)
23. Freedman, M.J., Nissim, K., Pinkas, B.: Efficient private matching and set intersection. In: Cachin, C., Camenisch, J.L. (eds.) EUROCRYPT 2004. LNCS, vol. 3027, pp. 1–19. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24676-3_1
24. Ghosh, S., Nilges, T.: An algebraic approach to maliciously secure private set intersection. In: Ishai, Y., Rijmen, V. (eds.) EUROCRYPT 2019. LNCS, vol. 11478, pp. 154–185. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17659-4_6
25. Gilboa, N., Ishai, Y.: Distributed point functions and their applications. In: Nguyen, P.Q., Oswald, E. (eds.) EUROCRYPT 2014. LNCS, vol. 8441, pp. 640–658. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-55220-5_35
26. Ion, M., et al.: Private intersection-sum protocol with applications to attributing aggregate ad conversions. IACR Cryptol. ePrint Arch. **2017**, 738 (2017). <http://eprint.iacr.org/2017/738>
27. Ishai, Y., Kushilevitz, E., Ostrovsky, R., Sahai, A.: Batch codes and their applications. In: Babai, L. (ed.) Proceedings of the 36th Annual ACM Symposium on Theory of Computing, Chicago, IL, USA, 13–16 June 2004, pp. 262–271. ACM (2004). <https://doi.org/10.1145/1007352.1007396>
28. Janssen, A.J., Van Leeuwen, J.: Analytic computation schemes for the discrete-time bulk service queue. Queueing Syst. **50**(2–3), 141–163 (2005)
29. Kiss, Á., Liu, J., Schneider, T., Asokan, N., Pinkas, B.: Private set intersection for unequal set sizes with mobile applications. Proc. Priv. Enhanc. Technol. **2017**(4), 177–197 (2017). <https://doi.org/10.1515/popets-2017-0044>
30. Kolesnikov, V., Kumaresan, R., Rosulek, M., Trieu, N.: Efficient batched oblivious PRF with applications to private set intersection. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna,

- Austria, 24–28 October 2016, pp. 818–829. ACM (2016). <https://doi.org/10.1145/2976749.2978381>
31. Little, J.D.: A proof for the queuing formula: $L = \lambda w$. *Oper. Res.* **9**(3), 383–387 (1961)
 32. Ma, Y., Zhong, K., Rabin, T., Angel, S.: Incremental offline/online PIR (extended version). *IACR Cryptol. ePrint Arch.*, p. 1438 (2021). <https://eprint.iacr.org/2021/1438>
 33. Meadows, C.A.: A more efficient cryptographic matchmaking protocol for use in the absence of a continuously available third party. In: *Proceedings of the 1986 IEEE Symposium on Security and Privacy*, pp. 134–137 (1986)
 34. Menon, S.J., Wu, D.J.: Spiral: fast, high-rate single-server PIR via FHE composition. *IACR Cryptol. ePrint Arch.*, p. 368 (2022). <https://eprint.iacr.org/2022/368>
 35. Pinkas, B., Rosulek, M., Trieu, N., Yanai, A.: SpOT-Light: lightweight private set intersection from sparse OT extension. In: Boldyreva, A., Micciancio, D. (eds.) *CRYPTO 2019*. LNCS, vol. 11694, pp. 401–431. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-26954-8_13
 36. Resende, A.C.D., de Freitas Aranha, D.: Faster unbalanced private set intersection in the semi-honest setting. *J. Cryptogr. Eng.* **11**(1), 21–38 (2021). <https://doi.org/10.1007/s13389-020-00242-7>
 37. Schoppmann, P., Gascón, A., Reichert, L., Raykova, M.: Distributed vector-OLE: improved constructions and implementation. In: *ACM CCS 2019: 26th Conference on Computer and Communications Security*, pp. 1055–1072. ACM Press (2019). <https://doi.org/10.1145/3319535.3363228>
 38. Stewart, C.A., et al.: Jetstream: a self-provisioned, scalable science and engineering cloud environment. In: *Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure*, pp. 1–8 (2015)
 39. Towns, J., et al.: XSEDE: accelerating scientific discovery. *Comput. Sci. Eng.* **16**(5), 62–74 (2014). <https://doi.org/10.1109/MCSE.2014.80>
 40. Trieu, N., Shehata, K., Saxena, P., Shokri, R., Song, D.: Epione: lightweight contact tracing with strong privacy. *IEEE Data Eng. Bull.* **43**(2), 95–107 (2020). <http://sites.computer.org/debull/A20june/p95.pdf>
 41. Wang, F., Yun, C., Goldwasser, S., Vaikuntanathan, V., Zaharia, M.: Splinter: practical private queries on public data. In: *NSDI 2017* (2017)