A Plug-n-Play Framework for Scaling Private Set Intersection to Billion-sized Sets

Saikrishna Badrinarayanan, Ranjit Kumaresan, Mihai Christodorescu, Vinjith Nagaraja, Karan Patel, Srinivasan Raghuraman, Peter Rindal, Wei Sun, Minghua Xu

Visa Research

Abstract

Motivated by the recent advances in practical secure computation, we design and implement a framework for scaling solutions for the problem of private set intersection (PSI) into the realm of big data. A protocol for PSI enables two parties each holding a set of elements to jointly compute the intersection of these sets without revealing the elements that are not in the intersection. Following a long line of research, recent protocols for PSI only have $\approx 5\times$ computation and communication overhead over an insecure set intersection. However, this performance is typically demonstrated for set sizes in the order of ten million. In this work, we aim to scale these protocols to efficiently handle set sizes of one billion elements or more.

We achieve this via a careful application of a *binning* approach that enables parallelizing any arbitrary PSI protocol. Building on this idea, we designed and implemented a framework that takes a pair of PSI executables (i.e., for each of the two parties) that typically works for million-sized sets, and then scales it to billion-sized sets (and beyond). For example, our framework can perform a join of billion-sized sets in 83 minutes compared to 2000 minutes of Pinkas et al. (ACM TPS 2018), an improvement of $25\times$. Furthermore, we present an *end-to-end* Spark application where two enterprises, each possessing private databases, can perform a restricted class of database join operations (specifically, join operations with only an *on* clause which is a conjunction of equality checks involving attributes from both parties, followed by a *where* clause which can be split into conjunctive clauses where each conjunction is a function of a single table) without revealing any data that is not part of the output. [1]

1 Introduction

Private set intersection (PSI) enables two parties, each holding a private set of elements to compute the intersection of the two sets while revealing nothing more than the intersection itself. PSI is an extremely well-motivated problem and has found applications in a variety of settings. For instance, PSI has been used to measure the effectiveness of online advertising [IKN+17], private contact discovery [CLR17, RA17, DRRT18], privacy-preserving location sharing [NTL+11], privacy-preserving ride sharing [HOS17], remote diagnostics [BPSW07] and botnet detection [NMH+10]. In the last few years, PSI has become truly practical with extremely fast implementations [FNP04,

[1]The authors grant IACR a non-exclusive and irrevocable license to distribute the article under the https://creativecommons.org/licenses/by-nc/3.0/

KS05] [HN10, DCT10, CKT10, DCW13, PSZ14, PSSZ15, KKRT16, OOS16, RR17a, RR17b, KMP+17, HV17]
[PSWW18, PRTY19, DRRT18, PSZ18, FNO19, PRTY20]. In terms of performance, the most computationally efficient PSI protocol [KKRT16] can privately compute the intersection of two million size sets in about 4 seconds. On the other hand, and for settings where only low bandwidth communication is available, one can employ the communication-optimal PSI protocol [ACT11] whose communication is only marginally more than an insecure set intersection protocol. Several recent works, most notably [PRTY19, PRTY20, CM20] have studied the balance between computation and communication, and even optimize for *monetary cost* of running PSI protocols in the cloud.

While significant progress has been made in advancing the efficiency of PSI protocols, almost all documented research in this area has so far focused on settings with set sizes of at most $2^{24} \approx 16$ million.[2] One notable exception is the work of [SK14] who demonstrate the feasibility of PSI over billion sized sets albeit in the non-standard *server-aided* model where a mutually trusted third party server aids in the computation. Another notable exception is the recent work of [PSZ16, PSZ18] whose implementation on 2 servers each with < 16 GB memory takes 34.2 hours to compute the intersection of two billion-element sets. Clearly, this leaves a lot of room for improvement. This is the gap we aspire to fill in this paper.

(Issues in) scaling existing PSI protocols. Broadly speaking, memory consumption is a big problem when implementing cryptographic schemes that operate on large amounts of data. In fact, many if not all implemented PSI protocols (e.g., those based on garbled circuits, or bloom filters, or cuckoo hashing) quickly exceed the main memory, thereby requiring more engineering effort. Even computing the plaintext intersection for billions of elements becomes a nontrivial problem. That said, many of the PSI protocols somewhat benefit from thread-level parallelism (e.g., for preprocessing OTs, generating garbled circuits) and hardware support (e.g., AES-NI). Some of the steps that do not parallelize well are those dealing with data structures (such as cuckoo hashing or bloom filters), however these may be preprocessed since only one party's input is required.

Concretely, we discuss the implementation of the OT-based PSI protocol of [PSZ18] running on billion-sized sets containing 128-bit elements. The work of [PSZ18] makes use of solid state drives in their PSI execution on billion-sized sets. As documented in [PSZ18], the total execution time is 34.2 hours.[3] In comparison, the (insecure) naive hashing protocol for set intersection required 74 mins, of which 19 min (26%) are for hashing and transferring data and 55 min (74%) are for computing the plaintext intersection.

1.1 Our Contributions

We study the possibility of parallelizing PSI protocols by distributing a party's workload into *multiple worker nodes* running within its premises. Towards this, we propose a simple technique that can parallelize any PSI protocol in a blackbox way. Finally, we build a framework to test out the feasibility of our technique in scaling PSI via Spark in a practical use case involving private database join operations. Comparing to the work of [PSZ18], our protocols for the same setting

[2]This does not necessarily apply to the setting of unbalanced PSI where the set sizes can be orders of magnitude apart [AK17, PSWW18, BP19, HCR18, HCR17]. For instance, [DK19] do unbalanced PSI with 2^{28} elements on one side and say 1024 elements on the other side.

[3]For a further breakdown of this number, [PSZ18] note that 30.0 hours (88%) are for simple hashing (cuckoo hashing runs in parallel and requires 16.3 hours), 3 hours (9%) are for computing the OTs, and 1.2 hours (4%) are for computing the plaintext intersection.

of billion-sized sets containing 128-bit elements, we require a total execution time of 83 minutes in total, a 25× improvement compared to [PSZ18]. We explain these in more detail below.

1.1.1 Techniques to parallelize PSI

We describe a few approaches at a high level and analyze their security and generality.

Self-reduction. A natural approach is to reduce an instance of PSI on large sets to multiple instances of PSI on smaller sets. Some care needs to be taken to ensure that privacy is still preserved. Specifically, note that PSI protocols are not guaranteed to hide the size of the input sets. For instance, if the sets are partitioned based on lexicographic ordering of the elements, then this would likely result in partitions being of unequal size, and thus either party could learn how the elements of the other party's set are distributed. We avoid such issues by proposing a natural random self-reduction which we refer to as our *binning technique* (see Section 3 for a formal description). Loosely speaking, our binning technique proceeds by asking each party to (1) locally randomize its input set (by applying a random oracle), (2) locally partition the randomized set, say lexicographically, into smaller sets, (3) locally pad each of these smaller sets with dummy elements so that they are all of the same size, (4) feed each small set into an independent PSI instance with the other party, and (5) finally use each PSI instance's output to recover the intersection in the original input set.

Important note. Partitioning elements into bins is a standard technique that appears several times and in several forms in the PSI literature. For instance, in [PSWW18] such a binning strategy is used to enable a reduction from PSI to *private set membership* by partitioning n elements into m bins for $m \approx n$. Among other similar works, the protocols of [KKRT16, PSZ14] enjoy high efficiency by employing cuckoo hashing which partitions n elements into $m \approx 1.2n$ bins. A similar partitioning approach is also used in the case of unbalanced sets of sizes n_0 and n_1 with $n_0 \gg n_1$ and even there (cf. [PSZ18]) the set of size n_0 is partitioned into $\approx 2.4n_1$ bins.

Where our approach differs from prior work is that we perform a *self-reduction* (i.e., PSI to itself) with a choice of parameters that differs from prior works mentioned above. In particular, for large n, we will be partitioning a set of size n into m bins for $n \gg m$ (e.g., $n = 10^9$ and $m = 64$).[4] While our PSI self-reduction is very simple and straightforward, to the best of our knowledge, we are not aware of any prior work documenting or implementing the self-reduction for the parameters that we employ in this paper. In particular, while the binning technique that we described above appears (nearly) verbatim in Section 3.1.1 of [PSZ18], the corresponding analysis in Section 3.1.2 of [PSZ18] focuses on $m = n$ resulting in n instances of PSI each of size $\frac{\ln n}{\ln \ln n}(1 + o(1))$ (see Table 3 in [PSZ18] for exact numbers).[5] On the other hand, we provide a hybrid approach where we employ the binning technique (referred to as *simple hashing* in Section 3.1.1 of [PSZ18]) to set up input sets (for independent PSI instances) which are large enough to enable application of a fast PSI protocol [KKRT16] (for independent PSI instances) that employs cuckoo hashing. To see how this affects performance, note that in Section 6.2.4 of [PSZ18], which details the performance of their best PSI protocol on billion element sets, the authors note that the cuckoo hashing step requires 16.3 hours. In contrast, applying our binning technique with our choice of parameters, i.e., $m = 64$ for $n = 10^9$, even **serially** would likely result in significant improvements since the best known

[4] Using $m \approx n$ in our self-reduction would incur an unacceptable overhead due to padding. Please see Subsection 3.1 on how to choose the optimum value of m.

[5] In that Section, they also analyze the choice of m for PSI with unbalanced sets.

PSI protocols on instances of size $n/m \approx 2^{24}$ use cuckoo hashing and still complete in under 2 minutes [KKRT16].

Big data frameworks. Another approach to parallelize a PSI protocol Π is to implement it in a big data framework like Spark which will distribute the work among many nodes. The downside is the lack of generality, in that each protocol must be rewritten in Scala to scale it. For instance, there exist efficient PSI protocols based on a variety of techniques and assumptions. Choice of what protocol to implement may also depend on the setting (e.g., client-server), set sizes (balanced or unbalanced), network bandwidth, or whether the PSI output needs to be kept secret-shared in order to pipeline it into other MPC protocols. Also, recent PSI protocols rely on data structures such as cuckoo hashing whose efficient scaling may be nontrivial [YS17] and may depend on the underlying big data framework.

In this paper, we show how our binning technique allows us to leverage a big data framework like Spark in a protocol agnostic way. The high level idea is to express (PSI) protocols in terms of its *round functions* aka *next message functions*.[6] These round functions are to be executed by a designated party at a particular round to determine the next message that needs to be sent by that party. More concretely, a round function takes as input the current state of the protocol, and the inputs and randomness of the designated party, and outputs the next message that the designated party sends to to the other party.[7] Expressing PSI protocols in this way, allows a protocol agnostic[8] way of orchestrating on each Spark cluster. Please see Section 5 for specific implementation of KKRT protocol [KKRT16]. Furthermore, such an orchestration does not require reconfiguring the clusters or modifying the internals of Spark (c.f. unlike [ZDB+17]).

1.1.2 Private database joins

Building on our techniques to parallelize PSI protocols, we describe how to implement an *end-to-end private database join application*. We consider a setting where two enterprises wish to perform a *data exchange*. That is, each of these two parties have databases storing sensitive information, and they wish to enrich their data based on information from the database tables of the other party. More concretely, the operation they wish to perform can be expressed succinctly as a join operation (inner, outer, left, or right) which specifies the attributes that need to be matched, and additional attributes that need to be fetched on the matched rows. A necessary privacy requirement in this setting is that either enterprise wishes to not reveal any information other than what is revealed by the output of the join operation.

Since enterprises often have (access to) dedicated clusters supporting their big data frameworks, an import design goal is to leverage these frameworks to (1) increase the efficiency of the join operation, and also to (2) integrate with existing data pipelines for pre- or post-processing. In this work, we focus specifically on Spark. We picked Spark because it is open-source and widely adopted big data analytics engine for large-scale data processing. Additionally, it comes with higher-level libraries and extensions which makes it an ideal choice for various use cases beyond PSI. We assume that the two enterprises each employ a Spark cluster consisting of multiple nodes co-ordinated by an *orchestrator* (that may be on either side). Note that each Spark cluster has complete access to that

[6] This is a standard technique to capture protocols in cryptography, for example while designing zero-knowledge compilers that transform a semi-honest secure protocol into a maliciously secure protocol.

[7] Most PSI protocols have very few rounds (exceptions include circuit PSI protocols that rely on the GMW compiler).

[8] We support any PSI protocol irrespective of the underlying cryptographic assumptions or algorithmic techniques.

party's input dataset only. Communication between the clusters that is required for private database join will be facilitated via dedicated edge servers. More details about our system architecture can be found in Subsection 4.2. Functionally, an analyst may connect to the orchestrator and use, for instance, a JupyterLab interface to issue private database join instructions to initiate and run our protocol on the specified datasets.

Plug-n-Play framework. By itself, Spark does not provide any privacy guarantees for computations that cross data boundaries. In Section 4, we describe a natural transformation of the private database join problem into a PSI problem. (The transformation itself can be carried out locally, and additionally admits parallelization via Spark.) Then, to solve the resultant PSI problem, we implement a generic framework that can apply our binning technique on top of any existing PSI implementation. Our framework is generic in that one could plug in any C/C++ PSI implementation (say from [Rin]) to our framework. Using the Java Native Interface (JNI) [Wik20] technology, our framework integrates the native implementation with the rest of our Spark pipeline. We refer to Subsection 4.3 for additional details.

Important note. Our system Spark-PSI described in Section 4, 4.3 supports a restricted class of join operations, namely join operations with only an *on* clause which is a conjunction of equality checks involving attributes from both parties, followed by a *where* clause which can be split into conjunctive clauses where each conjunction is a function of a single table. Parts of this restriction may be removed say if the underlying PSI protocol supports (the nonstandard but useful notion of) "PSI with computation" (PSIC) or "Circuit PSI" [BP19, MC18, PR] (where the PSI outputs are secret shared or in a form that can be fed into a different secure computation protocol) and is amenable to scaling via the binning approach. Post-join operations involving grouping, aggregation, and ordering can be enabled if these (nonstandard but useful) variants are used as the underlying PSI protocol. A circuit-PSI protocol realizing the circuit-PSI functionality in [PR] (that allows set elements along with associated data as input) can be scaled via our binning approach to efficiently enable operations mentioned above. Despite this, we chose not to do this since the fastest circuit-PSI protocols are significantly slower than the fastest (standard) PSI protocols, and furthermore, supporting arbitrary (post-)join operations requires generic MPC techniques on the secret shared PSI output (plus associated data), which is beyond the scope of this work.

1.2 Organization

We give a brief overview of PSI and Spark along with the associated threat models in Section 2. Then, we describe and analyze the binning technique in Section 3. After this, we describe the architecture and functionality of our private database join application in Subsection 4.2. Implementation details of the application can be found in Subsection 4.3, and our experimental evaluation is described in Section 5.

2 Preliminaries

2.1 Private Set Intersection

In the problem of private set intersection (PSI), two parties (sometimes referred to as "sender" and "receiver") each hold a set of items and wish to learn nothing more than the intersection of these sets. In this paper, we present generic techniques to *securely* parallelize any PSI protocol,

with security against semihonest (aka honest-but-curious) adversaries. For our experiments, we apply our parallelization technique on the KKRT PSI protocol [KKRT16]. The KKRT protocol is an OT-based PSI (like [PSZ14, PSZ18, OOS16]) and relies heavily on modern OT extension protocols [YI03, VK13, GA13]. We chose KKRT because it is currently the fastest PSI protocol against semihonest adversaries.

2.2 Apache Spark

Apache Spark is an open-source, fast, and distributed computing framework used for large-scale data workloads. It utilizes in-memory caching and optimizes query execution for any size of data. It is faster and more flexible than other systems such as Google's MapReduce [DG04], as it runs in memory, which makes processing much faster than disk [ZCF+10], and allows for complex processing schemes, instead of MapReduce's linear model. On top of Spark, there are libraries for running distributed computations ranging from SQL queries, to machine-learning algorithms, to graph analytics, and to data streaming.

A Spark application consists of a *driver program* that translates user-provided data processing pipelines into individual tasks and distributes these tasks to *worker nodes*. The basic abstractions available in Spark are built on a distributed data structure called *resilient distributed dataset* (RDD) [ZCD+12] and these abstraction offer distributed data processing operators such as map, filter, reduce, broadcast, etc. Higher-level abstractions expose popular APIs such as SQL, streaming, and graph processing.

Implementing state-of-art PSI protocols on top of Spark holds the promise of using the demonstrated capabilities of Spark and similar data platforms to achieve significant performance gains. Unfortunately Spark lacks any multi-tenant concepts, running all applications and scheduling all tasks in one security domain. This is incompatible with the basic settings of PSI protocols which involve two or more untrusted parties, which require multiple security domains with strong isolation between them. We address this problem by assigning each party to one Spark cluster, thus achieving isolation by physically separating each party's computation, and then introducing an *orchestrator* component that coordinates multiple independent Spark clusters in different data centers to jointly perform the PSI tasks. Section 4 describes our multi-cluster architecture and motivates our design.

A second security challenge in Apache Spark is the default data-partitioning scheme, which can reveal information about a party's dataset. For example, if data partitioning relies on the first byte in each record to distribute data records to nodes, an adversary can learn how many records start with `0x00`, how many with `0x01`, and so on. This leaks information about the data distribution in a dataset and undermines the security guarantees offered by a PSI protocol. We address this problem by introducing a *secure binning approach* (described in Section 3) that makes such leaks statistically inconsequential while still allowing each Spark cluster to partition data and distribute tasks as is locally optimal.

Finally, adding an orchestrator outside of the Spark clusters and treating individual Spark clusters' schedulers as black boxes, which are convenient for operational purposes, can lead to sub-optimal execution plans. In particular, the local optimization of schedules at each cluster may contradict with desired performance from collaborative computing across multiple clusters with different data sizes and hardware configurations. We take advantage of Spark's lazy evaluation capability, which can be used to delay the execution of a task until a certain action is triggered. Subsection 4.3 presents how we effectively use lazy evaluation to loosely and efficiently coordinate across clusters.

2.3 Threat Model

We consider a semi-honest adversary and detail its capabilities with respect to the PSI protocol we wish to deploy on Spark, to the Spark framework, and to our overall Spark-PSI system.

2.3.1 Threat Model of the PSI Protocol

In standard cryptography terminology, we assume that the PSI protocol is secure against *semi-honest* (aka honest-but-curious) adversaries. That is, we expect the participants to faithfully follow the instructions of the protocol but allow the parties to learn as much as they can from the protocol messages. We believe that this assumption fits many use cases, where parties are likely already under certain agreements to participate honestly. We further assume that all cryptographic primitives are secure. Finally, we note that the PSI protocol does reveal the sizes of the sets to both parties, as well as the final outputs in the clear (see [GA11] for size-hiding PSI, and [PSWW18, MC18] for protecting the outputs).

2.3.2 Threat Model of the Spark Framework

We assume that every Spark cluster's built-in security features are enabled and that the Spark implementation is free of vulnerabilities. These features include data-at-rest encryption, access management, quota management, queue management, etc. We further assume that these features guarantee a locally secure computing environment at each local cluster, such that an attacker cannot gain access to a Spark cluster unless authorized.

2.3.3 Threat Model of Spark-PSI

We assume that only authorized users can issue commands to the orchestrator and we further assume that the orchestrator is operated by one of the two parties. We note that it could be operated by some (semi-honest) third party without impacting security.

The adversary can observe the network communication between different parties during execution of the protocol. It may also control some of the parties to observe data present in the storage and memory of their clusters, as well as the order of memory accesses.

Our semi-honest adversary model implies that we expect participants to supply correct inputs to the PSI protocol. While in practice input validity is important, it is outside the scope of this work as we believe it can be tackled as a future, separate layer on top of Spark-PSI.

3 Parallelizing PSI via Binning

We describe an efficient technique to scale any PSI protocol Π. For simplicity, we assume that both parties have equal sized sets, say of size n. Each set contains elements of length κ bits (typically and wlog $\kappa = 128$). Then, for a given parameter m, we show how to solve PSI on instances of size n, via m invocations of Π on set sizes $\approx n/m$ with minimal overhead. Our parallelization technique will be statistically secure. To aid in the analysis, we use σ to denote the statistical security parameter (typically, set to 80). Our PSI protocol, denoted by Π_{PPSI} (stands for Parallel PSI) is illustrated in Fig. 1.

Our idea is to first let each party to locally partition their set into $m > 1$ subsets. That is, the parties first locally sample a random hash function $h : \{0,1\}^* \to \{1,\dots,m\}$. Each party P

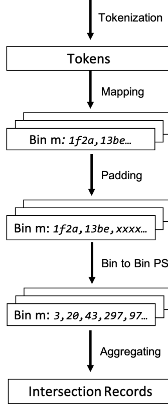

Figure 1: Binning Pipeline

transforms its set $S = \{s_1, \ldots, s_n\}$ into subsets $T_1, \ldots, T_m$ such that for all $s \in S$ it holds that $s \in T_{h(s)}$. Modeling h as a random function ensures that the elements $\{h(s) \mid s \in S\}$ are all distributed uniformly. This directly implies that $\mathbb{E}[\text{size of } T_i] = n/m$.

However, observe that the number of elements in any given bin T_i does in fact leak information about the distribution of the input set. For example, say there are no items in T_i, then this implies that the set S does not contain any element s that s.t. $h(s) = i$. To maintain the security guarantees of PSI, it is critical that this information is not leaked.

Now given that the parties have locally partitioned their sets into $T_1, \ldots, T_m$, next they pad each T_j with uniformly random dummy elements to ensure that the size of each padded set equals $(1 + \delta_0)n/m$ for some parameter δ_0. Since $S \subset \{0, 1\}^\kappa$, there are 2^κ possible elements and the probability of a dummy item being in the intersection is negligible. Alternatively, if κ is large enough, we can ensure that no dummy item is in the intersection by asking each party to pad its j-th bin with dummy items s' sampled from non-overlapping subsets of $\{0, 1\}^\kappa$ such that $h(s') = j' \neq j$. Then, the two parties engage in m *parallel* instances of Π, where in the i-th instance π_i, parties input their respective i-th padded tokenized set. Once all m instances of Π deliver output, parties then by simply combining these m individual outputs to obtain the final output.

In summary, our binning technique proceeds by asking each party to (1) locally tokenize the sets, (2) locally map the set elements into m bins, (3) locally pad each bin with dummy elements to ensure that each bin contains exactly $(1 + \delta_0)n/m$ elements, (4) execute m instances of a PSI protocol with the other party, (5) finally combine the outputs of the individual PSI protocols to get the final output.

3.1 Analysis

We compute the value of the parameter δ_0 that ensures that the binning step does not fail except with negligible probability. For a fixed $i \in [n]$ and $j \in \{1, \ldots, m\}$, suppose $X_{i,j}$ is the indicator variable that equals 1 iff the i-th element s_i ended up in T_j, and suppose $X_j = \sum_{i \in [n]} X_{i,j}$ denotes the size of T_j. For a fixed j, since $X_{i,j}$ variables are independent of each other (since h is modeled as a random function), a Chernoff bound yields $\mathbb{P}[X_j > (1 + \delta)\mu] \leq e^{-\delta^2 \mu/3}$ where $\mu = \mathbb{E}[X_j] = n/m$

and $0 \leq \delta \leq 1$. The above analysis was for a single bin T_j. By union bound, we see that the probability that any of the m bins have greater than $(1+\delta)\mu$ elements is $\leq me^{-\delta^2\mu/3}$. Thus if we set the failure probability $me^{-\delta^2 n/3m} = 2^{-\sigma}$, we get $\delta = \sqrt{3m/n \cdot (\sigma \ln 2 + \ln m)} \stackrel{\text{def}}{=} \delta_0$. That is, the above binning technique requires only max bin size at most $(1+\delta_0)n/m$ w.h.p. More concretely, suppose set size $n = 10^9$ and statistical parameter $\sigma = 80$, then choosing parameter $m = 64$, we see that the max bin size of any of the 64 bins is at most $n' \approx 15.68 \times 10^6$ (with $\delta_0 = 0.0034$) with probability $(1 - 2^{-80})$.

Note that existing PSI protocols [KKRT16, PRTY20, CM20] can already efficiently handle set sizes of n'. Therefore, in principle, we can use 64 instances of PSI protocol of say [KKRT16] to implement a PSI protocol that operates on 1 billion sized sets.

3.2 Simulation

Formally, we prove security in the so called *simulation paradigm* that is standard in cryptography. In short, this enables proving that all attacks that can be carried by an adversary in the designed protocol can be *simulated* in an ideal world where parties only interact with an imaginary trusted third party $\mathcal{F}_{\mathsf{PSI}}$ that accepts inputs from the parties, computes the intersection locally, and returns only the intersection to the parties. As our binning technique self-reduces PSI, we inherit the security properties of the underlying PSI protocol Π (i.e., that operates on smaller set sizes). For the reduction, we assume the h is statistically close to a random function (alternatively, a nonprogrammable random oracle), and this enables us to prove that our reduction is *statistically secure*. However, our final protocol Π_{PPSI} (where the underlying PSI instances Π are instantiated with the real PSI protocols) will be computationally secure. Concretely say if the underlying PSI protocol Π relies on DDH, then our protocol Π_{PPSI} remains secure assuming DDH holds. This is the case, for instance, when the underlying PSI protocol is [KKRT16] assuming the OTs are instantiated via DDH [MN01].

We provide a short sketch of the simulation of our protocol Π_{PPSI} in the $\mathcal{F}_{\mathsf{PSI}}$ ideal world. Note that we are in the semi-honest model, so the simulator has access to the input tape of the corrupt party. In addition, we are in the $\mathcal{F}_{\mathsf{PSI}}$-hybrid model where our protocol Π_{PPSI} can call the PSI functionality $\mathcal{F}_{\mathsf{PSI}}$ as a subroutine. We note however these calls will be on subsets of the overall data. For the sake of readability, we denote this functionality as $\mathcal{F}'_{\mathsf{PSI}}$.

Since our protocol is effectively symmetric, w.l.o.g., we assume P_0 to be the corrupt party. The simulator begins by feeding the input S_0 to the ideal PSI functionality $\mathcal{F}_{\mathsf{PSI}}$ to obtain the PSI output $I' = S_0 \cap S_1$. Next it partitions I' into m bins as specified by h, i.e. $I_j = \{i \in I' \mid h(i) = j\}$ denotes bin j. If any bin I_j has more than $(1+\delta_0)n/m$ elements, then the simulator aborts. Then for each bin j, the simulator emulating $\mathcal{F}'_{\mathsf{PSI}}$ in the hybrid world receives from P_0 a padded set of size $(1+\delta_0)n/m$, and returns I_j as the output of the call to $\mathcal{F}'_{\mathsf{PSI}}$. Finally, the simulator outputs I'. This completes the description of the simulation.

Note that the simulation fails if (1) simulator encounters binning failure (i.e., bin size exceeds $(1+\delta_0)n/m$), or (2) dummy item added by one party matches an item from the other party. Thus from the analysis in the previous section, we conclude that the ideal world simulation is *statistically indistinguishable* from the hybrid-world protocol.

3.3 Applying our binning technique

We emphasize that our technique works for any PSI protocol (no matter what assumption it is based on) for all settings including cases where the sets are unbalanced. Furthermore, the PSI instances operating on different bins could in principle use different PSI protocols or implementations (which can be useful if the underlying infrastructure is heterogeneous).

By design our technique is highly conducive for an efficient Spark implementation (or in any other big data framework). Also, large input sets may already be distributed across several nodes in a Spark cluster. We provide a quick overview of how our protocol would operate in such a setting. At the beginning of the protocol the hash function h is sampled and distributed to the nodes in both clusters. Within each cluster, each node uses h as a mapping function to define the new partitions $T_1, ..., T_m$ which are each assigned to some worker node in the same cluster. The main phase of the protocol proceeds as described by running m parallel instances[9] of the PSI protocol across the two clusters which outputs the intersection sets $I_1, ..., I_m$ such that the final output is defined as $I' = \cup_i I_i$. In the next sections, we describe our system Spark-PSI that applies our binning technique in a real-world application.

4 Scalable Private Database Joins

In this section, we describe how to perform SQL styled join queries with the use of our parallel PSI protocol implemented via Spark. In Subsection 4.1, we describe the problem of private database joins across different data domains, and outline a solution which leverages our binning technique for parallelizing PSI. Then, in Subsection 4.2, we describe the architecture of our system Spark-PSI that solves the database join problem. Finally, in Subsection 4.3, we describe the various techniques we employ to efficiently implement our binning technique in Spark.

4.1 Database Joins Across Data Domains

In the problem of private database joins, we have two distinct parties A, B, who wish to perform a join operation on their private data. To model the problem, we denote Domain A as the data domain of party A, and likewise Domain B for party B. We assume that one of the parties hosts an *orchestrator* which is essentially a server that exposes metadata such as schemas of the data sets that are available for the join operation. (For more details, see Subsection 4.2). This way parties discover the available types of queries and can submit them via the orchestrator API. When a query is submitted, the orchestrator will validate the correctness of the query and forward to request to the other party for approval. While many types of query languages could be supported, we have chosen to implement a subset of SQL.

More precisely we support any query which can be divided into the following. A *select* clause which specifies one more columns among the two tables. A *join on* clause which compares one or more columns for equality between the two sets. A *where* clause which can be split into conjunctive clauses where each conjunction is a function of a single table.

For example, we support the following query:

```
SELECT DomainB.table0.col4, DomainA.table0.col3
```

[9] If we have k worker nodes on each side, then we can run k instances of Π in parallel, and repeat this m/k times to complete the PSI portion of the execution.

```
FROM DomainA.table0
JOIN DomainB.table0
ON DomainA.table0.col1 = DomainB.table0.col2
AND DomainA.table0.col2 = DomainB.table0.col6
WHERE DomainA.table0.col3 > 23.
```

In this example a column from both parties is being selected where they are being joined on the equality of the join keys

```
DomainA.table0.col1 = DomainB.table0.col2
DomainA.table0.col2 = DomainB.table0.col6
```

along with the added constraint

```
DomainA.table0.col3 > 23.
```

Our framework transforms this query by first filtering the local data sets based on the `WHERE` clause. We require that each of the where clauses be a function of a single table. For example, we do not support a where clause such as

```
WHERE DomainA.table0.col0 > DomainB.table0.col7
```

because this predicate compares across the two data sets.[10]

Once the local *where* clauses have filtered the input tables, the parties *tokenize* the join key columns. The join key columns refer to the columns which appear in the `JOIN ON` clause. In the example above these are `DomainA.table0.col1`, `DomainA.table0.col2` from the first party (Domain A) and `DomainB.table0.col2`, `DomainB.table0.col6` from the second party (Domain B). For each row of the respective data sets, the parties generate a set of tokens by hashing together their join keys. For example, Domain A can generate their set A as

$$\begin{aligned} A = \Big\{ H\big(& \texttt{DomainA.table0.col1}[i], \\ & \texttt{DomainA.table0.col2}[i]\big) \mid \\ i \in & \{1, ..., n\} \Big\} \end{aligned}$$

Let B denote the analogous set of tokens for Domain B. We note that rows with the same join keys will have the same token and that the A, B sets will contain only a single copy of that token. Later we will need to map elements of A, B back to the rows which they correspond to. For this task we will logically add an additional column to each input table which we label as `token` and stores that row's token value. That is, for the example above we have

$$\begin{aligned} \texttt{DomainA.table0.token} = H\big(& \texttt{DomainA.table0.col1}[i], \\ & \texttt{DomainA.table0.col2}[i]\big) \end{aligned}$$

Now the parties can execute a PSI protocols on their respective A, B sets as inputs. The protocol outputs the intersection $I = A \cap B$ to both parties. As described in the previous section, this phase is parallelized with the use of our binning technique.

[10] Restricting clauses this way enables us to reduce the above problem to the PSI problem. We note that the restriction above can be lifted if we use more sophisticated PSI protocols that can keep the PSI output in secret shared form without revealing it. We leave this for future work.

In the final phase the parties use the intersection I to construct the output table. Here we will assume that only Domain A should obtain the output table but note that this general procedure can provide output to both. Both parties take subsets of their tables such that only rows which have a token value in I remain. This can efficiently be implemented using the `token` column that was appended to the input tables. From this subset, Domain B sends their columns which appear in the select clause along with the token value. Let this table be denoted as `table`*. Domain A then joins their table with `table`* to construct the final output table.

In summary, the private database join operation can be performed via the following three phases. The first phase, referred to as tokenization, translates a possibly complex join query into a set intersection problem. In the second phase our parallel PSI protocol runs and outputs the intersection to both parties. The final phase is referred to as reverse-lookup which instructs the parties to use the intersection to construct the final join output which may contain significantly more information than the intersection alone, e.g. additional attributes being selected. In the next section, we will see the design of an architecture that enables us to efficiently execute these several phases in a setting where parties have Spark clusters.

4.2 System Architecture

Figure 2 describes the overall architecture of our system where we connect two distinct parties (or data domains) each having a Spark cluster. To solve the private database join problem, we need to co-ordinate the two Spark clusters to implement the various phases described in the previous section.

This co-ordination is carried out by an *orchestrator* that exposes an interface (such as UI application/portal/Jupyter Lab) to specify the database join operation and to receive the results. Our orchestrator interfaces with each party's Spark cluster via Apache Livy [Liv17].

In more detail, the orchestrator is responsible for storing various metadata such as the schemas of the data sets. We assume that these schemas are made available to the orchestrator by the parties in an initialization phase. Following this, either party can authenticate itself to the orchestrator and submit a SQL styled query. The orchestrator is then responsible for parsing the query and compiling Spark jobs for two clusters for different phases of the private database join operation, including the PSI protocols. The orchestrator then initiates the protocol by sending both clusters the relevant parameters for different phases of the protocol, e.g. data sets identifiers, join columns, network configuration, etc. Once the database join protocol completes, the orchestrator will record audit logs and potentially facilitate access to the output of the join.

The Apache Livy [Liv17] interface helps internally to manage Spark session and submission of Spark code for PSI computation. Communication between the two clusters for various phases of the join protocol (e.g., for the PSI subprotocols) is facilitated via dedicated edge servers which work as Kafka brokers to establish a secure data transmission channel. While we have chosen Apache Kafka for implementing the communication pipeline, our architecture allows the parties to plug their own communication channel of choice to read/write data back and forth. Additionally, our architecture doesn't change any internals of Spark that makes easier to adopt and deploy at scale.

Security implications. We discuss some security considerations and highlight some security implications that are a consequence of our architecture described above. While the theoretical security of the database join protocol is guaranteed by employing a secure PSI protocol, we now discuss other security features provided by our architecture. More concretely, we highlight that in

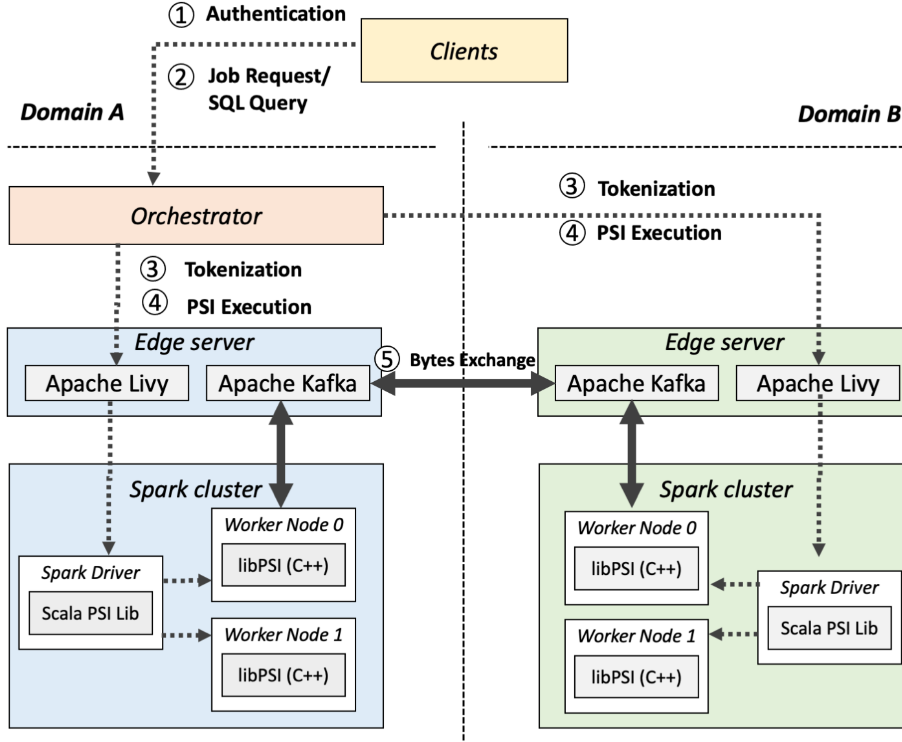

Figure 2: Spark-PSI Framework.① Clients need authentication first so that they can talk to the orchestrator ② Clients have two modes to start a job: a JSON style request or a SQL style query. ③ The orchestrator parses the query for tokenization of related columns. ④ Execute Spark PSI pipelines ⑤ Intermediate bytes are exchanged through Kafka brokers deployed on edge servers.

addition to the built-in security features of Spark cluster, our design ensures *cluster isolation* and *session isolation* which we describe next.

The orchestrator provides a protected virtual computing environment for each database join job thereby guaranteeing *session isolation*. While standard TLS is used to protect the communication between different Spark clusters, the orchestrator provides additional communication protection such as session specific encryption and authentication keys, randomizing and anonymizing the endpoints, managing allow and deny lists, and monitoring/preventing DOS/DDOS attack to the environments. The orchestrator also provides an additional layer of user authentication and authorization. All of the computing resources, including tasks, cached data, communication channels, and metadata are protected within this session. No foreign user or job may peek or alter the internal state of the session. Each parties' Spark session is isolated from each other and only reports execution state back to orchestrator.

On the other hand, *cluster isolation* aims at protecting computing resources from each parties from misuse or abuse in the database join jobs. To accomplish this, the orchestrator is the only node in the environment that controls and is visible to the end-to-end processing flows. It is also the only party that has the metadata for Spark clusters involved in the session. Recall that a separate secure

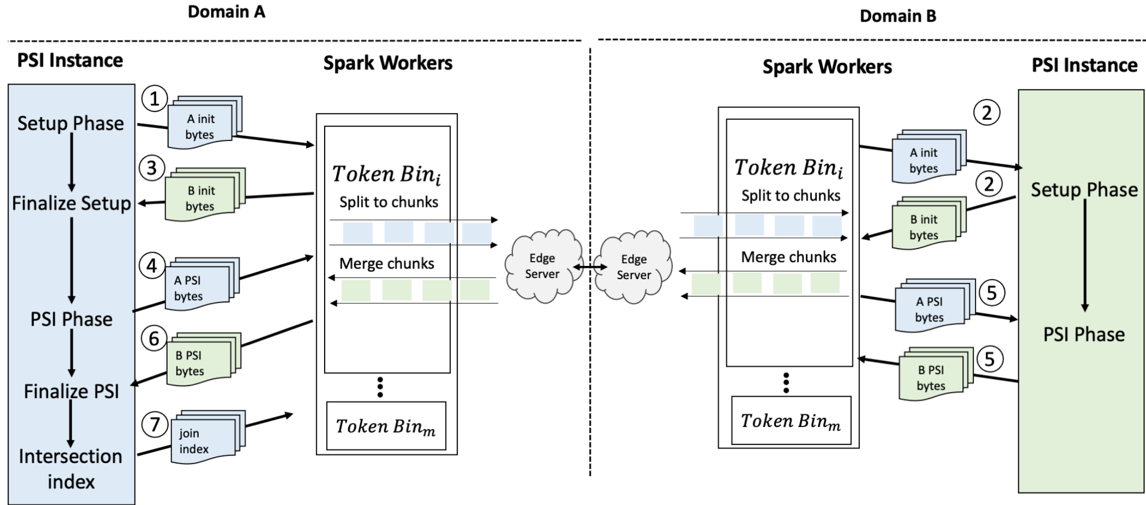

Figure 3: KKRT Implementation Workflow. ① [DomainA.setup1] Domain A in its setup phase generates encrypted data and transmits them to Domain B. ② [DomainB.setup1] Domain B performs its setup phase with data received from Domain A and in turn generates encrypted data and transmits them to Domain A. ③ [DomainA.setup2] Domain A finalizes setup phase. ④ [DomainA.psi1] and ⑤ [DomainB.psi1] Domains A and B execute the online PSI phase, which proceeds over multiple rounds. ⑥ [DomainA.psi2] Domain A enters the Finalize PSI phase and computes the intersection with Domain B as row indices. ⑦ Domain A retrieves matching records by doing a reverse lookup into its dataset using the computed indices.

communication channel is employed via Livy and Kafka that limits the parties from accessing each other's Spark cluster. This keeps the orchestrator out of the data flow pipeline thereby preventing the party operating the orchestrator from gaining advantages over other parties involved. It also ensures that each Spark cluster is self-autonomous and requires little or no changes to participate in a database join protocol with other parties. The orchestrator also takes care of job failures or uneven computing speed to ensure out-of-the-box reusability of Spark clusters that typically already exist in enterprise organizations.

Finally, we remark that the low level APIs calling cryptographic libraries and exchanging data between C++ instances and Spark dataframes, lie in each party's data cluster and thus do not introduce any information leakage. The high level APIs package the secure Spark execution pipeline as a service, and are responsible for mapping independent jobs to each executor and collecting the results from them. See Subsection 4.3 for more details.

Taken together, our architecture essentially provides the theoretical security that is guaranteed by the underlying PSI subprotocol. More concretely, when one party is compromised by the adversary, the other party's data remains completely private except whatever is revealed by the output of the computation.

4.3 Spark-PSI Implementation

We provide details on how we leverage Spark to implement the binning technique. Our underlying PSI protocol is the KKRT protocol implemented in C++.

KKRT Workflow. Figure 3 shows the detailed data flows in our Spark-PSI framework instantiated with the KKRT protocol. All of the phases shown in Figure 3 are invoked by the orchestrator sequentially. The orchestrator starts the native KKRT execution by submitting metadata information about the datasets to both parties. Based on the request, both parties start executing their Spark code which creates new dataframes by loading the required data set using the supported JDBC driver. This dataframe is then hashed into the tokens of fixed length by both parties. This *token dataframe* is then mapped to m number of bins (in Spark terminology partitions) using the custom partitioner by both parties, which is basically distributing tokens evenly on both side. Then the final intersection is obtained by taking the union of each bin intersection. Note that tokenization and binning are generic functionalities in our framework and can be used by any other PSI algorithm. This way Spark achieves parallel execution of multiple bins on both sides.

The native KKRT protocol is executed via a generic JNI interface that connects to the Spark code. Specifically, the JNI interface is in terms of round functions, and therefore is agnostic about internal protocol implementation. Note that there is a one-time setup phase for KKRT (this setup is required only once for a pair of parties). This is described in Steps ①, ②, ③ in Figure 3. Later, the online PSI phase that actually computes the intersection between the bins is shown in Steps ④, ⑤, ⑥, and ⑦. The parties use edge servers to mirror data whenever there is a write operation on any of the Kafka brokers. Note that the main PSI phase consists of sending the encrypted data sets and can be a performance bottleneck as Kafka is optimized for small size messages. To overcome this issue, we are chunking encrypted data sets into smaller partitions on both sides so that we can utilize Kafka's capability efficiently. We also keep the intermediate data retention period very short on Kafka broker to overcome storage and security concerns.

The above strategy also has the benefit of enabling streaming of the underlying PSI protocol messages. Note that the native KKRT implementation is designed to send and receive data as soon as it is generated. As such, our Spark implementation continually forwards the protocol messages to and from Kafka the moment they become available. This effectively results in additional parallelization due to the Spark worker not needing to block for slow network I/O. Note that we also explicitly cache *token dataframe* and *instance address dataframe* which are used in multiple phases to avoid any re-computation. This way we take advantage of Spark's lazy evaluation that optimizes execution plan based on DAG and RDD persistence.

Reusable components for parallelizing other PSI protocols. Our code is packaged as a Spark-Scala library which includes an end-to-end example implementation of native KKRT protocol. This library itself has many useful reusable components such as JDBC connectors to work with multiple data sources, methods for tokenization and binning approach, general C++ interface to link other native PSI algorithms and a generic JNI interface between Scala and C++ interface. All these functions are implemented in base class of the library, which may be reused for other native PSI implementations. Additionally, our library decouples networking methods from actual PSI computation which adds flexibility to the framework to support other networking channels if required.

Any PSI implementation can be plugged into Spark-PSI by exposing a C/C++ API that can be invoked by the framework. The API is structured around the concept of setup rounds and online

rounds and does not make assumptions about the cryptographic protocol executed in these rounds. The following functions are part of the API:

- `get-setup-round-count() -> count` – retrieves the total number of setup rounds required by this PSI implementation;
- `setup( id, in-data ) -> out-data` – invokes round `id` on the appropriate party with data received from the other party in the previous round of the setup and returns the data to be sent;
- `get-online-round-count() -> count` – retrieves the total number of online rounds required by this PSI implementation;
- `psi-round( id, in-data ) -> out-data` – invokes the online round `id` on the appropriate party with data received from the other party in the previous round of the PSI protocol and returns the data to be sent.

The data passed to an invocation of `psi-round` is the data from a single bin, and SPARK-PSI orchestrates the parallel invocations of this API over all of the bins. For example, KKRT has three setup rounds (which we label for clarity in the rest of the paper as DomainA.setup1, DomainB.setup1, and DomainA.setup2) and three online rounds (labeled DomainA.psi1, DomainB.psi1, and DomainA.psi2). When running KKRT with 256 bins (as done in one of the experiments detailed in Section 5), the setup rounds DomainA.setup1, DomainB.setup1, and DomainA.setup2 each invoke `setup` once with the appropriate round id, and the online rounds DomainA.psi1, DomainB.psi1, and DomainA.psi2 each invoke `psi-round` with the appropriate round id 256 times.

5 Experimental Evaluation

In this section, we describe the performance of our SPARK-PSI implementation, and provide detailed benchmarks for various steps. Then, we provide our end-to-end performance numbers and study the impact of number of bins on the running time. The highlight of this section is our running time of 82.88 minutes for sets of size 1 billion. We obtain this when we use $m = 2048$ bins.[11]

5.1 System Setup

Our experiments are evaluated on a setup similar to the one described in Figure 2. Each party runs an independent standalone six-node Spark (v2.4.5) cluster with 1 server for driver and 5 servers for workers. Additionally we have an independent Kafka (v2.12-2.5.0) VM on each side for inter-cluster communication. The orchestrator server, which triggers the PSI computation, is on Domain A. All servers have 8 vCPUs (2.6 GHz), 64 GB RAM and run Ubuntu 18.04.4 LTS.

5.2 Microbenchmarking

We first benchmark the performance of the various steps in the binning pipeline (cf. Figure 1) and the KKRT implementation workflow (cf. Figure 3). For these experiments, we assume that each party uses a dataset of size 100M as input.

[11] This corresponds to $\delta_0 = 0.019$ for a bin size of $\approx$ 500K (cf. Section 3.1).

SPARK-PSI step	Time (s) by dataset size		
	$10M$	$50M$	$100M$
DomainA.tokenize	47.21	91.20	124.68
DomainB.tokenize	45.90	92.89	121.64
DomainA.psi1	8.40	20.64	31.55
DomainB.psi1	40.83	121.73	247.30
DomainA.psi2	14.92	47.49	88.05

Table 1: Microbenchmark of SPARK-PSI when using KKRT PSI and 2048 bins.

KKRT PSI round	Time (s) by number of bins	
	256	2048
DomainA.psi1.write	36.44	**13.36**
DomainB.psi1.read	178.76	**98.77**
DomainB.psi1.write	15.24	**8.12**
DomainA.psi2.read	25.61	**21.35**

Table 2: Network latency for a dataset of size 100M.

Table 1 describes the total time required for the individual phases of our protocol when the number of bins equals 2048. For example, DomainA.tokenize (resp. DomainB.tokenize) denotes the time taken for tokenizing A's input (resp. B's input) and mapping these tokens into different bins and padding each bin to be of the same size (cf. Figure 1). Note that the tokenization step is done in parallel. DomainA.psi1 denotes the time taken for executing Step ④ for all the bins. In this step, Domain A generates and transfers approximately $60n$ bytes for dataset size n (i.e., 100M) to Domain B. Likewise, DomainB.psi1 denotes the time taken for executing Step ⑤ for all the bins. In this step, Domain B generates and transfers approximately $22n$ bytes back to Domain A. Finally, DomainA.psi2 denotes the time taken for executing Steps ⑥ and ⑦, where the intersection is determined for all the bins. Note that we have excluded benchmarking Steps ①, ②, ③ in Figure 3 as these correspond to the setup functions which have a constant cost, and more importantly these functions need to be executed only once between a pair of parties (and can be reused for subsequent PSI executions).

Communication vs. number of bins. Table 2 describes the impact of bin size on the time taken for reading and writing data via Kafka (i.e., inter-cluster communication). (Note that the numbers in Table 1 include the time taken for reading and writing data.) Here, DomainA.psi1 produces intermediate data of size 9.1GB which is sent to Doman B, while DomainB.psi1 produces 3.03GB intermediate data that is sent to A. As evident from the benchmarks in Table 2, more bins improve networking performance as the message chunks become smaller. In more detail, when we go with 256 bins, individual messages of size 35.55MB are sent over Kafka for DomainA.psi1. With 2048 bins, the corresponding individual message size is only 4.44MB.

Number of bins	Time (m)		
	Insecure single-cluster Spark join	Insecure cross-cluster Spark join	SPARK-PSI
256	**3.76**	7.60	11.41
4,096	5.62	**4.90**	**8.71**
8,192	10.83	10.26	9.79

Table 3: Total execution time for different joins over datasets of size 100M. Fastest times in each column are highlighted.

5.3 End-to-End Performance

Shuffle overhead of our protocol. In Table 3, we compare the performance of our protocol with the performance of insecure joins on datasets of size 100M. To evaluate and compare with the performance of insecure joins, we consider two variants. In the first variant, which we call *single-cluster Spark join*, we employ a single cluster with six nodes (one server for driver and five servers for workers) to perform the join on two datasets each of size 100M. Here, we assume that both datasets reside on the same cluster. The join computation then proceeds by partitioning the data into multiple bins and then computing the intersection directly using a *single* Spark join call. In the second variant, which we call *cross-cluster Spark join*, we employ two clusters each with six nodes, and each containing only one 100M tokenized dataset. Now, to perform the join, each cluster partitions its dataset into multiple bins. Then one of the clusters sends the partitioned dataset to the other cluster, which then aggregates the received data into one dataset, and then computed the final join using a *single* Spark join call.

In the case of insecure join on a single cluster, we observe that increasing the number of bins leads to an increase in the number of data shuffling operations (shuffle read/write), which ends up slowing down the execution. When we split the insecure join across two clusters, we incur the overhead of network communication across clusters and the additional shuffling on the destination cluster, but gain a parallelism because we have twice the compute resources.

When we switch to SPARK-PSI, we maintain the overhead of cross-cluster communication and incur additional overhead of the PSI computation, but we avoid the extra data shuffling (as we employ broadcast join). We believe the effect of the broadcast join appears most significant when we have smaller per-bin data (as is the case with 8,192 bins) making SPARK-PSI faster than the insecure cross-cluster join in some cases. Our secure system introduces an overhead of up to 77% in the worst case on top of the insecure cross-cluster join.

Choosing the optimal bin size. In Table 4 we report the running time of the PSI as a function of the number of bins and dataset size, and plot the same in figure 4. The highlight of this table is our running time of 82.88 minutes for dataset size 1B, roughly a 25× speedup over the prior work of Pinkas et al. [PSZ18]. As evident from the table, we obtain this running time when we set the number of bins $m = 2048$. Also as evident from the table and from the corresponding plot in Figure 4, the performance of our protocol on datasets of a given size first begins to improve as we increase the number of bins, and then hits an inflection point after which the performance degrades. The initial improvement is a result of parallelization. Higher number of bins results in smaller bin

Number of bins	Time (m) by dataset size			
	$1M$	$10M$	$100M$	$1B$
1	1.07	12.04	–	–
16	**0.75**	2.00	–	–
64	0.78	1.66	15.27	154.10
256	0.99	**1.47**	11.41	116.89
1,024	1.03	1.63	8.57	86.54
2,048	1.11	1.86	**8.12**	**82.88**
4,096	1.40	1.94	8.71	90.46
8,192	2.45	3.07	9.79	94.74

Table 4: Total execution time for PSI with various dataset sizes and bin sizes. Fastest times in each column are highlighted.

size on Spark and this is ideal especially for larger datasets, but the strategy of increasing the number of bins doesn't continue to work as the task scheduling overhead in Spark (and the padding overhead of the binning technique itself) slows down the execution. Also, we believe that better performance is possible if we use more executor cores (i.e., a larger cluster) as this is likely to allow better parallelization.

6 Related work

Private Set Intersection. Several protocols have been proposed to realize PSI such as the efficient but insecure naive hashing solution, public key cryptography based protocols [Mea86, HFH99, FNP04, FHNP16, DCT10, GA11, ACT11, RR17a, CLR17], those based on oblivious transfer [DCW13, PSZ14, KKRT16, PRTY19, PRTY20, CM20] and other circuit-based solutions [HEKM11, PSSZ15, PSWW18, BP19]. Another popular model for PSI is to introduce a semi-trusted third party that aids in efficiently computing the intersection [SK14, ATD15, ATD16]. We refer to [PSZ16] for a more detailed overview on the various approaches taken to solve PSI. In addition, other variants of PSI have also been extensively studied such as multi-party PSI [KMP+17, HV17], PSI cardinality [CGT12, IKN+17], PSI sum [IKN+17, IKN+19], threshold PSI [GS19, BMR20] to name a few. Apart from PSI, there is also a line of work on performing other set operations such as union privately [KS05, BA16, DC17, KRTW19].

Privacy-Preserving Frameworks. Modern big data systems have demonstrated unprecedented scalability and performance since the MapReduce programming model [DG08]. This introduces both opportunities and challenges alike for secure distributed computing over massive data sets and cloud computing.

Dong et al. [DCW13] introduce garbled Bloom filters to design an efficient PSI protocol over big data, which is implemented using the MapReduce framework. PSJoin [DYC+19] makes use of differential privacy to build a MapReduce-based privacy-preserving similarity join. Hahn et al. [HLK19] use searchable encryption and key-policy attribute-based encryption to design a protcol for secure joins that leak the fine granular access pattern and frequency of elements selected for the join.

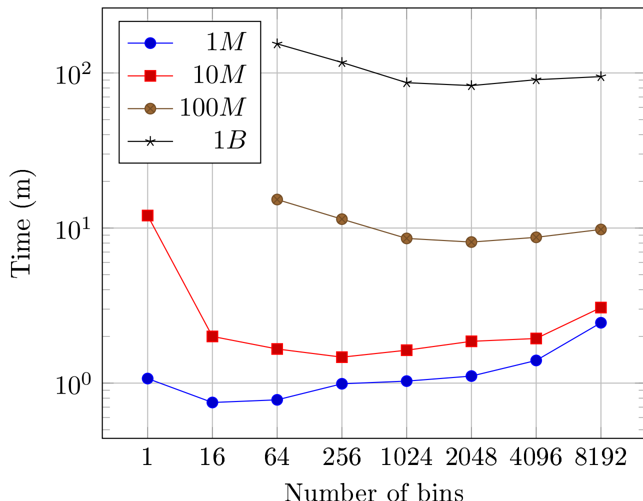

Figure 4: Different input sizes achieve optimal execution time for different number of bins.

SMCQL [BEE+17] uses the garbled-circuit based backend ObliVM [LWN+15] to compute query results over the union of several source databases without revealing sensitive information about individual tuples. Although optimized, it introduces a prohibitive overhead. ConClave [VSG+19] builds a secure query compiler based on ShareMind [BLW08] and Obliv-C [ZE15] to improve scalability. ConClave works in the server-aided model in order to decrease computational overhead. However, these systems still leave much to be desired in terms of performing efficient secure computation over big data. Furthermore, existing works are tailor-made to meet specific requirements and hence would not offer the same performance gains for arbitrary secure computation.

Another set of privacy-preserving frameworks makes use of hardware enclaves. Opaque [ZDB+17] is an oblivious distributed data analytics platform which utilized Intel SGX hardware enclaves to provide strong security guarantees. OCQ [DLP+20] further decreases communication and computation costs of Opaque via an oblivious planner. Unlike these methods, Spark-PSI does not depend on hardware. Other recent works include CryptDB [PRZB11] and Seabed [PBC+16] which provide protocols for the secure execution of analytical queries over encrypted big data. Senate [RP20] describes a framework for enabling privacy preserving database queries in a multiparty setting.

7 Conclusion

In this paper, we described the analysis and application of a simple technique to parallelize any PSI protocol. Using this, we studied how different PSI protocols in the literature can be easily scaled to set sizes $\approx 60\times$ greater than standard benchmarks. We then described a Spark framework and architecture to implement our technique in a private database join application. Our experiments show that this framework is ready for use in real-world scenarios. Additionally, our framework provides reusable components that enable cryptographers to scale novel PSI protocols to sets of size one billion.

We see several threads for future work. One is to explore novel optimizations to our Spark framework including use of alternative architectural components. Another is to generalize our Spark framework to secure computation workloads (as opposed to just PSI). For instance, secure private evaluation of *random forests* (a collection of decision trees) can be naturally parallelized with Spark. Finally, we mention the study of the generality of the binning technique to malicious settings and to variants of PSI such as threshold PSI, PSI-sum, and PSI-cardinality.

Disclaimer

All trademarks are the property of their respective owners, are used for identification purposes only, and do not necessarily imply product endorsement or affiliation with the author(s).

References

[ACT11] Giuseppe Ateniese, Emiliano De Cristofaro, and Gene Tsudik. (if) size matters: Size-hiding private set intersection. In Dario Catalano, Nelly Fazio, Rosario Gennaro, and Antonio Nicolosi, editors, *Public Key Cryptography - PKC 2011 - 14th International Conference on Practice and Theory in Public Key Cryptography, Taormina, Italy, March 6-9, 2011. Proceedings*, volume 6571 of *Lecture Notes in Computer Science*, pages 156–173. Springer, 2011.

[AK17] Thomas Schneider N. Asokan Benny Pinkas Agnes Kiss, Jian Liu. Private set intersection for unequal set sizes with mobile applications. In *Proc. Priv. Enhancing Technol. (4)*, pages 177–197, 2017.

[ATD15] Aydin Abadi, Sotirios Terzis, and Changyu Dong. O-PSI: delegated private set intersection on outsourced datasets. In Hannes Federrath and Dieter Gollmann, editors, *ICT Systems Security and Privacy Protection - 30th IFIP TC 11 International Conference, SEC 2015, Hamburg, Germany, May 26-28, 2015, Proceedings*, volume 455 of *IFIP Advances in Information and Communication Technology*, pages 3–17. Springer, 2015.

[ATD16] Aydin Abadi, Sotirios Terzis, and Changyu Dong. VD-PSI: verifiable delegated private set intersection on outsourced private datasets. In Jens Grossklags and Bart Preneel, editors, *Financial Cryptography and Data Security - 20th International Conference, FC 2016, Christ Church, Barbados, February 22-26, 2016, Revised Selected Papers*, volume 9603 of *Lecture Notes in Computer Science*, pages 149–168. Springer, 2016.

[BA16] Marina Blanton and Everaldo Aguiar. Private and oblivious set and multiset operations. *Int. J. Inf. Sec.*, 15(5):493–518, 2016.

[BEE+17] Johes Bater, Gregory Elliott, Craig Eggen, Satyender Goel, Abel Kho, and Jennie Rogers. Smcql: secure querying for federated databases. *Proceedings of the VLDB Endowment*, 10(6):673–684, 2017.

[BLW08] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In *European Symposium on Research in Computer Security*, pages 192–206. Springer, 2008.

[BMR20] Saikrishna Badrinarayanan, Peihan Miao, and Peter Rindal. Multi-party threshold private set intersection with sublinear communication. *IACR Cryptol. ePrint Arch.*, 2020:600, 2020.

[BP19] Oleksandr Tkachenko Avishay Yanai Benny Pinkas, Thomas Schneider. Efficient circuit-based psi with linear communication. In *Eurocrypt* 3, pages 122–153, 2019.

[BPSW07] Justin Brickell, Donald E Porter, Vitaly Shmatikov, and Emmett Witchel. Privacy-preserving remote diagnostics. In *CCS*, 2007.

[CGT12] Emiliano De Cristofaro, Paolo Gasti, and Gene Tsudik. Fast and private computation of cardinality of set intersection and union. In Josef Pieprzyk, Ahmad-Reza Sadeghi, and Mark Manulis, editors, *Cryptology and Network Security, 11th International Conference, CANS 2012, Darmstadt, Germany, December 12-14, 2012. Proceedings*, volume 7712, pages 218–231. Springer, 2012.

[CKT10] Emiliano De Cristofaro, Jihye Kim, and Gene Tsudik. Linear-complexity private set intersection protocols secure in malicious model. In Masayuki Abe, editor, *Advances in Cryptology - ASIACRYPT 2010 - 16th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 5-9, 2010. Proceedings*, volume 6477 of *Lecture Notes in Computer Science*, pages 213–231. Springer, 2010.

[CLR17] Hao Chen, Kim Laine, and Peter Rindal. Fast private set intersection from homomorphic encryption. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1243–1255. ACM, 2017.

[CM20] Melissa Chase and Peihan Miao. Private set intersection in the internet setting from lightweight oblivious PRF. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology - CRYPTO 2020 - 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17-21, 2020, Proceedings, Part III*, volume 12172 of *Lecture Notes in Computer Science*, pages 34–63. Springer, 2020.

[DC17] Alex Davidson and Carlos Cid. An efficient toolkit for computing private set operations. In Josef Pieprzyk and Suriadi Suriadi, editors, *Information Security and Privacy - 22nd Australasian Conference, ACISP 2017, Auckland, New Zealand, July 3-5, 2017, Proceedings, Part II*, volume 10343 of *Lecture Notes in Computer Science*, pages 261–278. Springer, 2017.

[DCT10] Emiliano De Cristofaro and Gene Tsudik. Practical private set intersection protocols with linear complexity. In *FC*, 2010.

[DCW13] Changyu Dong, Liqun Chen, and Zikai Wen. When private set intersection meets big data: an efficient and scalable protocol. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 789–800, 2013.

[DG04] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, San Francisco, CA, 2004.

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[DK19] Thomas Schneider Matthias Senker Christian Weinert Daniel Kales, Christian Rechberger. Mobile private contact discovery at scale. In *USENIX Annual Technical Conference*, pages 1447–1464, 2019.

[DLP+20] Ankur Dave, Chester Leung, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. Oblivious coopetitive analytics using hardware enclaves. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–17, 2020.

[DRRT18] Daniel Demmler, Peter Rindal, Mike Rosulek, and Ni Trieu. PIR-PSI: scaling private contact discovery. *Proc. Priv. Enhancing Technol.*, 2018(4):159–178, 2018.

[DYC+19] Xiaofeng Ding, Wanlu Yang, Kim-Kwang Raymond Choo, Xiaoli Wang, and Hai Jin. Privacy preserving similarity joins using mapreduce. *Inf. Sci.*, 493:20–33, 2019.

[FHNP16] Michael J. Freedman, Carmit Hazay, Kobbi Nissim, and Benny Pinkas. Efficient set intersection with simulation-based security. *J. Cryptology*, 29(1):115–155, 2016.

[FNO19] Brett Hemenway Falk, Daniel Noble, and Rafail Ostrovsky. Private set intersection with linear communication from general assumptions. In Lorenzo Cavallaro, Johannes Kinder, and Josep Domingo-Ferrer, editors, *Proceedings of the 18th ACM Workshop on Privacy in the Electronic Society, WPES@CCS 2019, London, UK, November 11, 2019*, pages 14–25. ACM, 2019.

[FNP04] Michael J. Freedman, Kobbi Nissim, and Benny Pinkas. Efficient private matching and set intersection. In *EUROCRYPT*, 2004.

[GA11] Gene Tsudik Giuseppe Ateniese, Emiliano De Cristofaro. (if) size matters: Size-hiding private set intersection. In *PKC*, pages 156–173, 2011.

[GA13] Thomas Schneider Michael Zohner Gilad Asharov, Yehuda Lindell. More efficient oblivious transfer and extensions for faster secure computation. In *CCS*, pages 535–548, 2013.

[GS19] Satrajit Ghosh and Mark Simkin. The communication complexity of threshold private set intersection. In Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology - CRYPTO 2019 - 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2019, Proceedings, Part II*, volume 11693 of *Lecture Notes in Computer Science*, pages 3–29. Springer, 2019.

[HCR17] Kim Laine Hao Chen and Peter Rindal. Fast private set intersection from homomorphic encryption. In *CCS*, pages 1243–1255, 2017.

[HCR18] Kim Laine Hao Chen, Zhicong Huang and Peter Rindal. Labeled psi from fully homomorphic encryption with malicious security. In *CCS*, pages 1223–1237, 2018.

[HEKM11] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. Faster secure two-party computation using garbled circuits. In *20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings*. USENIX Association, 2011.

[HFH99] Bernardo A. Huberman, Matthew K. Franklin, and Tad Hogg. Enhancing privacy and trust in electronic communities. In Stuart I. Feldman and Michael P. Wellman, editors, *Proceedings of the First ACM Conference on Electronic Commerce (EC-99), Denver, CO, USA, November 3-5, 1999*, pages 78–86. ACM, 1999.

[HLK19] Florian Hahn, Nicolas Loza, and Florian Kerschbaum. Joins over encrypted data with fine granular security. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*, pages 674–685. IEEE, 2019.

[HN10] Carmit Hazay and Kobbi Nissim. Efficient set operations in the presence of malicious adversaries. In *PKC*, 2010.

[HOS17] Per A. Hallgren, Claudio Orlandi, and Andrei Sabelfeld. Privatepool: Privacy-preserving ridesharing. In *CSF*, 2017.

[HV17] Carmit Hazay and Muthuramakrishnan Venkitasubramaniam. Scalable multi-party private set-intersection. In Serge Fehr, editor, *PKC*, 2017.

[IKN+17] Mihaela Ion, Ben Kreuter, Erhan Nergiz, Sarvar Patel, Shobhit Saxena, Karn Seth, David Shanahan, and Moti Yung. Private intersection-sum protocol with applications to attributing aggregate ad conversions. 2017. ia.cr/2017/735.

[IKN+19] Mihaela Ion, Ben Kreuter, Ahmet Erhan Nergiz, Sarvar Patel, Mariana Raykova, Shobhit Saxena, Karn Seth, David Shanahan, and Moti Yung. On deploying secure computing commercially: Private intersection-sum protocols and their business applications. *IACR Cryptol. ePrint Arch.*, 2019:723, 2019.

[KKRT16] Vladimir Kolesnikov, Ranjit Kumaresan, Mike Rosulek, and Ni Trieu. Efficient batched oblivious prf with applications to private set intersection. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 818–829, 2016.

[KMP+17] Vladimir Kolesnikov, Naor Matania, Benny Pinkas, Mike Rosulek, and Ni Trieu. Practical multi-party private set intersection from symmetric-key techniques. In *CCS*, 2017.

[KRTW19] Vladimir Kolesnikov, Mike Rosulek, Ni Trieu, and Xiao Wang. Scalable private set union from symmetric-key techniques. In Steven D. Galbraith and Shiho Moriai, editors, *Advances in Cryptology - ASIACRYPT 2019 - 25th International Conference on the Theory and Application of Cryptology and Information Security, Kobe, Japan, December 8-12, 2019, Proceedings, Part II*, volume 11922 of *Lecture Notes in Computer Science*, pages 636–666. Springer, 2019.

[KS05] Lea Kissner and Dawn Song. Privacy-preserving set operations. In *CRYPTO*, 2005.

[Liv17] Apache Livy. Apache livy, 2017.

[LWN$^+$15] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. Oblivm: A programming framework for secure computation. In *2015 IEEE Symposium on Security and Privacy*, pages 359–376. IEEE, 2015.

[MC18] Claudio Orlandi Michele Ciampi. Combining private set-intersection with secure two-party computation. In *SCN*, pages 464–482, 2018.

[Mea86] Catherine A. Meadows. 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, Oakland, California, USA, April 7-9, 1986*, pages 134–137. IEEE Computer Society, 1986.

[MN01] Benny Pinkas Moni Naor. Efficient oblivious transfer protocols. In *SODA*, pages 448–457, 2001.

[NMH$^+$10] Shishir Nagaraja, Prateek Mittal, Chi-Yao Hong, Matthew Caesar, and Nikita Borisov. Botgrep: Finding p2p bots with structured graph analysis. In *USENIX security symposium*, 2010.

[NTL$^+$11] Arvind Narayanan, Narendran Thiagarajan, Mugdha Lakhani, Michael Hamburg, and Dan Boneh. Location privacy via private proximity testing. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*. The Internet Society, 2011.

[OOS16] Michele Orrù, Emmanuela Orsini, and Peter Scholl. Actively secure 1-out-of-n OT extension with application to private set intersection. In *CT-RSA*, 2016.

[PBC$^+$16] Antonis Papadimitriou, Ranjita Bhagwan, Nishanth Chandran, Ramachandran Ramjee, Andreas Haeberlen, Harmeet Singh, Abhishek Modi, and Saikrishna Badrinarayanan. Big data analytics over encrypted datasets with seabed. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 587–602, 2016.

[PR] Phillipp Schoppmann Peter Rindal. Vole-psi: Fast oprf and circuit-psi from vector-ole. In *Eurocrypt*.

[PRTY19] Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. Spot-light: Lightweight private set intersection from sparse ot extension. In *CRYPTO*, 2019.

[PRTY20] Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. PSI from paxos: Fast, malicious private set intersection. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT*, 2020.

[PRZB11] Raluca Ada Popa, Catherine MS Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 85–100, 2011.

[PSSZ15] Benny Pinkas, Thomas Schneider, Gil Segev, and Michael Zohner. Phasing: Private set intersection using permutation-based hashing. In *USENIX*, 2015.

[PSWW18] Benny Pinkas, Thomas Schneider, Christian Weinert, and Udi Wieder. Efficient circuit-based PSI via cuckoo hashing. In *EUROCRYPT*, 2018.

[PSZ14] Benny Pinkas, Thomas Schneider, and Michael Zohner. Faster private set intersection based on ot extension. In *USENIX*, 2014.

[PSZ16] Benny Pinkas, Thomas Schneider, and Michael Zohner. Scalable private set intersection based on OT extension. *IACR Cryptol. ePrint Arch.*, 2016:930, 2016.

[PSZ18] Benny Pinkas, Thomas Schneider, and Michael Zohner. Scalable private set intersection based on OT extension. *ACM Trans. Priv. Secur.*, 21(2):7:1–7:35, 2018.

[RA17] Amanda C. Davi Resende and Diego F. Aranha. Unbalanced approximate private set intersection. *IACR Cryptol. ePrint Arch.*, 2017:677, 2017.

[Rin] Peter Rindal. libPSI: an efficient, portable, and easy to use Private Set Intersection Library. https://github.com/osu-crypto/libPSI.

[RP20] Avishay Yanai Ryan Deng Raluca Ada Popa Joseph M. Hellerstein Rishabh Poddar, Sukrit Kalra. Senate: A maliciously-secure mpc platform for collaborative analytics. *IACR Cryptol. ePrint Arch.*, 2020:1350, 2020.

[RR17a] Peter Rindal and Mike Rosulek. Improved private set intersection against malicious adversaries. In *EUROCRYPT*, 2017.

[RR17b] Peter Rindal and Mike Rosulek. Malicious-secure private set intersection via dual execution. In *CCS*, 2017.

[SK14] Mariana Raykova Saeed Sadeghian Seny Kamara, Payman Mohassel. Scaling private set intersection to billion-element sets. In *Financial Cryptography and Data Security*, pages 195–215, 2014.

[VK13] Ranjit Kumaresan Vladimir Kolesnikov. Improved ot extension for transferring short secrets. In *Crypto (2)*, pages 54–70, 2013.

[VSG+19] Nikolaj Volgushev, Malte Schwarzkopf, Ben Getchell, Mayank Varia, Andrei Lapets, and Azer Bestavros. Conclave: secure multi-party computation on big data. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–18, 2019.

[Wik20] Wikipedia. Java native interface - wikipedia, 2020.

[YI03] Kobbi Nissim Erez Petrank Yuval Ishai, Joe Kilian. Extending oblivious transfers efficiently. In *Crypto*, pages 145–161, 2003.

[YS17] Song Jiang Qiuyu Li Shunde Cao Pengfei Zuo Yuanyuan Sun, Yu Hua. Smartcuckoo: A fast and cost-efficient hashing index scheme for cloud storage systems. In *USENIX Annual Technical Conference*, pages 553–565, 2017.

[ZCD+12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*, pages 15–28, 2012.

[ZCF+10] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, page 10, USA, 2010. USENIX Association.

[ZDB+17] Wenting Zheng, Ankur Dave, Jethro G Beekman, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pages 283–298, 2017.

[ZE15] Samee Zahur and David Evans. Obliv-c: A language for extensible data-oblivious computation. *IACR Cryptol. ePrint Arch.*, 2015:1153, 2015.