# Efficient Circuit-based PSI with Linear Communication

Benny Pinkas[1], Thomas Schneider[2], Oleksandr Tkachenko[2], and Avishay Yanai[1]

[1] Bar-Ilan University, Israel
benny@pinkas.net, ay.yanay@gmail.com
[2] TU Darmstadt, Germany
{schneider,tkachenko}@encrypto.cs.tu-darmstadt.de

**Abstract.** We present a new protocol for computing a circuit which implements the private set intersection functionality (PSI). Using circuits for this task is advantageous over the usage of specific protocols for PSI, since many applications of PSI do not need to compute the intersection itself but rather functions based on the items in the intersection.
Our protocol is the *first circuit-based PSI protocol to achieve linear communication complexity*. It is also concretely more efficient than all previous circuit-based PSI protocols. For example, for sets of size $2^{20}$ it improves the communication of the recent work of Pinkas et al. (EUROCRYPT'18) by more than 10 times, and improves the run time by a factor of 2.8x in the LAN setting, and by a factor of 5.8x in the WAN setting.
Our protocol is based on the usage of a protocol for computing oblivious programmable pseudo-random functions (OPPRF), and more specifically on our technique to amortize the cost of batching together multiple invocations of OPPRF.

**Keywords:** Private Set Intersection, Secure Computation

## 1 Introduction

The functionality of Private Set Intersection (PSI) enables two parties, $P_1$ and $P_2$, with respective input sets $X$ and $Y$ to compute the intersection $X \cap Y$ without revealing any information about the items which are not in the intersection. There exist multiple constructions of secure protocols for computing PSI, which can be split into two categories: (i) constructions that output the intersection itself and (ii) constructions that output the result of a function $f$ computed on the intersection. In this work, we concentrate on the second type of constructions (see §1.2 for motivation). These constructions keep the intersection $X \cap Y$ secret from both parties and allow the function $f$ to be securely computed on top of it, namely, yielding only $f(X \cap Y)$. Formally, denote by $\mathcal{F}_{\mathsf{PSI},f}$ the functionality $(X, Y) \mapsto (f(X \cap Y), f(X \cap Y))$.

A functionality for computing $f(X \cap Y)$ can be naively implemented using generic MPC protocols by expressing the functionality as a circuit. However, naive

protocols for computing $f(X \cap Y)$ have high communication complexity, which is of paramount importance for real-world applications. The difficulty in designing a circuit for computing the intersection is in deciding which pairs of items of the two parties need to be compared. We refer here to the number of comparisons computed by the circuit as the major indicator of the overhead, since it directly affects the amount of communication in the protocol (which is proportional to the number of comparisons, times the length of the representation of the items, times the security parameter). Since the latter factors (input length and security parameter) are typically given, and since the circuit computation mostly involves symmetric key operations, the goal is to minimize the communication overhead as a function of the input size. We typically state this goal as minimizing the number of comparisons computed in the circuit. The protocol presented in this paper is the first to achieve linear communication overhead, which is optimal.

Suppose that each party has an input set of $n$ items. A naive circuit for this task compares all pairs and computes $O(n^2)$ comparisons. More efficient circuits are possible, assuming that the parties first order their respective inputs in specific ways. For example, if each party has sorted its input set then the intersection can be computed using a circuit which first computes, using a merge-sort network, a sorted list of the union of the two sets, and then compares adjacent items [HEK12]. This circuit computes only $O(n \log n)$ comparisons. The protocol of [PSSZ15] (denoted "Circuit-Phasing") has $P_1$ map its items to a table using Cuckoo hashing, and $P_2$ maps its items using simple hashing. The intersection is computed on top of these tables by a circuit with $O(n \log n / \log \log n)$ comparisons. This protocol is the starting point of our work.

A recent circuit-based PSI construction [PSWW18] is based on a new hashing algorithm, denoted "two-dimensional Cuckoo hashing", which uses a table of size $O(n)$ and a stash of size $\omega(1)$. Each party inserts its inputs to a separate table, and the hashing scheme assures that each value in the intersection is mapped by both parties to exactly one mutual bin. Hence, a circuit which compares the items that the two parties mapped to each bin, and also compares all stash items to all items of the other party, computes the intersection in only $\omega(n)$ comparisons (namely, the overhead is slightly more than linear, although it can be made arbitrarily close to being linear).

Our work is based on the usage of an oblivious programmable pseudo-random function (OPPRF), which is a new primitive that was introduced in [KMP+17]. An OPRF — oblivious pseudo-random function (note, this is different than an OPPRF) — is a two-party protocol where one party has a key to a PRF $F$ and the other party can privately query $F$ at specific locations. An OPPRF is an extension of the protocol which lets the key owner "program" $F$ so that it has specific outputs for some specific input values (and is pseudo-random on all other values). The other party which evaluates the OPPRF does not learn whether it learns a "programmed" output of $F$ or just a pseudo-random value.

### 1.1 Overview of our Protocol

The starting point for our protocols is the Circuit-Phasing PSI protocol of [PSSZ15], in which $O(n)$ bins are considered and the circuit computes $O(n \log n / \log\log n)$ comparisons. Party $P_1$ uses Cuckoo hashing to map at most one item to each bin, whereas party $P_2$ maps its items to the bins using simple hashing (two times, once with each of the two functions used in the Cuckoo hashing of the first party). Thus, $P_2$ maps up to $S = O(\log n / \log\log n)$ items to each bin. Since the parties have to hide the number of items that are mapped to each bin, they pad the bins with "dummy" items to the maximum bin size. That is, $P_1$ pads all bins so they all contain exactly one item and $P_2$ pads all bins so they all contain $S$ items.

Both parties use the same hash functions, and therefore for each input element $x$ that is owned by both parties there is exactly one bin to which $x$ is mapped by both parties. Thus, it is only needed to check whether the item that $P_1$ places in a bin is among the items that are placed in this bin by $P_2$. This is essentially a private set membership (PSM) problem: As input, $P_1$ has a single item $x$ and $P_2$ has a set $\Sigma$ with $|\Sigma|$ items, where $S = |\Sigma|$. As for the output, if $x \in \Sigma$ then both parties learn the same random output, otherwise they learn independent random outputs. These outputs can then be fed to a circuit, which computes the intersection. The Circuit-Phasing protocol [PSSZ15] essentially computes the PSM functionality using a sub-circuit of the overall circuit that it computes. Namely, let $S = O(\log n / \log\log n)$ be an upper bound on the number of items mapped by $P_2$ to a single bin. For each bin the sub-circuit receives one input from $P_1$ and $S$ inputs from $P_2$, computes $S$ comparisons, and feeds the result to the main part of the circuit which computes the intersection itself (and possibly some function on top of the intersection). Therefore the communication overhead is $O(nS) = O(n \log n / \log\log n)$. A very recent work in [CO18] uses the same hashing method and computes the PSM using a specific protocol whose output is fed to the circuit. The circuit there computes only $\omega(n)$ comparisons but the PSM protocol itself incurs a communication overhead of $O(\log n / \log\log n)$ and is run $O(n)$ times. Therefore, the communication overhead of [CO18] is also $O(n \log n / \log\log n)$.

We diverge from the protocol of [PSSZ15] in the method for comparing the items mapped to each bin. In our protocol, the parties run an oblivious programmable PRF (OPPRF) protocol for each bin $i$, such that party $P_2$ chooses the PRF key and the programmed values, and the first party learns the output. The function is "programmed" to give the same output $\beta_i$ for each of the $O(\log n / \log\log n)$ items that $P_2$ mapped to this bin. Therefore, if there is any match in this bin then $P_1$ learns the same value $\beta_i$. Then, the parties evaluate a circuit, where for each bin $i$ party $P_1$ inputs its output in the corresponding OPPRF protocol, and $P_2$ inputs $\beta_i$. This circuit therefore needs to compute only a *single* comparison per bin.

The communication overhead of an OPPRF is linear in the number of programmed values. Thus, a stand alone invocation of an OPPRF for every bin incurs an overall overhead of $O(n \log n / \log\log n)$. We achieve linear overhead for comparing the items in all bins, by observing that although each bin is of

maximal size $O(\log n / \log \log n)$ (and therefore naively requires to program this number of values in the OPPRF), the total number of items that need to be programmed in all bins is $O(n)$. We can amortize communication so that the total communication of computing all $O(n)$ OPPRFs is the same as the total number of items, which is $O(n)$.

In addition to comparing the items that are mapped to the hash tables, the protocol must also compare items that are mapped to the stash of the Cuckoo hashing scheme. Fixing a stash size $s = O(1)$, the probability that the stash does not overflow is $O(n^{-(s+1)})$ [KMW09]. It was shown in [GM11] that a stash of size $O(\log n)$ ensures a negligible failure probability (namely, a probability that is asymptotically smaller than any polynomial function). Each item that $P_1$ places in the stash must be compared to all items of $P_2$, and therefore a straightforward implementation of this step requires the circuit to compute $\omega(n)$ comparisons. However, we show an advanced variant of our protocol that computes all comparisons (including elements in the stash) with only $O(n)$ comparisons.

In addition to designing a generic $O(n)$ circuit-based PSI protocol, we also investigate an important and commonly used variant of the problem where each item is associated with some value ("payload"), and it is required to compute a function of the payloads of the items in the intersection. (For example, compute the sum of financial transactions associated with these items.) The challenge is that each of the $S$ items that the second party maps to a bin has a different payload and therefore it is hard to represent them using a single value. (The work in [PSSZ15, CO18], for example, did not consider payloads.) We describe a variant of our PSI protocol which injects the correct payloads to the circuit while keeping the $O(n)$ overhead.

Overall, the work in this paper improves the state of the art in two dimensions:

- With regards to **asymptotic** performance, we show a protocol for circuit-based PSI which has only $O(n)$ communication. This cost is asymptotically smaller than that of all known circuit-based constructions of PSI, and matches the obvious lower bound on the number of comparisons that must be computed.
- With regards to **concrete** overhead, our most efficient protocols improve communication by a factor of 2.6x to 12.8x, and run faster by factor 2.8x to 5.8x compared to the previous best circuit-based PSI protocol of [PSWW18]. We demonstrate this both analytically and experimentally.

### 1.2 Motivation for Circuit-based PSI

Most research on computing PSI focused on computing the intersection itself (see §1.4). On the other hand, many **applications** of PSI are based on computing arbitrary functions of the intersection. For example, Google reported a PSI-based application for measuring the revenues from *online* ad viewers who later perform a related *offline* transaction (namely, ad conversion rates) [Yun15, Kre17]. This computation compares the set of people who were shown an ad with the set of people who have completed a transaction. These sets are held by the advertiser, and by merchants, respectively. A typical use case is where the merchant inputs pairs of the customer-identity and the value of the transactions

made by this customer, and the computation calculates the total revenue from customers who have seen an ad, namely customers in the intersection of the sets known to the advertiser and the merchant. Google reported implementing this computation using a Diffie-Hellman-based PSI cardinality protocol (for computing the cardinality of the intersection) and Paillier encryption (for computing the total revenues) [IKN+17, Kre18]. In fact, it was recently reported that Google is using such a "double-blind encryption" protocol in a beta version of their ads tool.[3] However, their protocol reveals the size of the intersection, and has substantially higher runtimes than our protocol as it uses public key operations, rather than efficient symmetric cryptographic operations (cf. §7.4).

Another motivation for running circuit-based PSI is **adaptability.** A protocol that is specific for computing the intersection, or a specific function such as the cardinality of the intersection, cannot be easily changed to compute another function of the intersection (say, the cardinality plus some noise to preserve differential privacy). Any change to a specialized protocol will require considerable cryptographic know-how, and might not even be possible. On the other hand, the task of writing a new circuit component which computes a different function of the intersection is rather trivial.

Circuit-based protocols also benefit from the **existing code base** for generic secure computation. Users only need to design the circuit to be computed, and can use available libraries of optimized code for secure computation, such as [HEKM11, EFLL12, DSZ15, LWN+15].

### 1.3 Computing Symmetric Functions

We focus in this work on constructing a circuit which computes the intersection. On top of that circuit it is possible to compose a circuit for computing any function that is based on the intersection. In order to preserve privacy, that function must be a symmetric function of the items in the intersection. Namely, the output of the function must not depend on the *order* of its inputs.

If the function that needs to be computed is non-symmetric, then the circuit for computing the intersection must shuffle its output, in order to place each item of the intersection in a location which is independent of the other values. The result is used as the input to the function. The size of this "shuffle" step is $O(n \log n)$, as is described in [HEK12], and it dominates the $O(n)$ size of the intersection circuit. We therefore focus on the symmetric case.[4]

Most interesting functions of the intersection (except for the intersection itself) are symmetric. Examples of symmetric functions include:
– The size of the intersection, i.e., PSI cardinality (PSI-CA).

[3] `https://www.bloomberg.com/news/articles/2018-08-30/google-and-mastercard-cut-a-secret-ad-deal-to-track-retail-sales`

[4] Note that outputting the intersection is a *non-symmetric* function. Therefore in that case the order of the elements must be shuffled. However, it is unclear why a circuit-based protocol should be used for computing the intersection, since there are specialized protocols for this which are much more efficient, e.g. [KKRT16, PSZ18].

– A threshold function that is based on the size of the intersection. For example identifying whether the size of the intersection is greater than some threshold (PSI-CAT). An extension of PSI-CAT, where the intersection is revealed only if the size of the intersection is greater than a threshold, can be used for privacy-preserving ridesharing [HOS17]. Other public-key based protocols for this functionality appear in [ZC17, ZC18].
– A differentially private [Dwo06] value of the size of the intersection, which is computed by adding some noise to the exact count.
– The sum of values associated with the items in the intersection. This is used for measuring ad-generated revenues (cf. §1.2).

The circuits for computing all these functions are of size $O(n)$. Therefore, with our new construction the total size of the circuits for applying these functions to the intersection is $O(n)$.

### 1.4 Related Work

We classify previous works into dedicated protocols for *PSI*, generic protocols for *circuit-based PSI*, and dedicated protocols for *PSI cardinality*.

**PSI.** The first PSI protocols were based on public-key cryptography, e.g., on the Diffie Hellman function (e.g. [Mea86], with an earlier mention in [Sha80]), oblivious polynomial evaluation [FNP04], or blind RSA [DT10]. More recent protocols are based on oblivious transfer (OT) which can be efficiently instantiated using symmetric key cryptography [IKNP03, ALSZ13]: these protocols use either Bloom filters [FNP04] or hashing to bins [PSZ14, PSSZ15, KKRT16, PSZ18]. All these PSI protocols have super-linear complexity and many of them were compared experimentally in [PSZ18]. PSI protocols have also been evaluated on mobile devices, e.g., in [HCE11, ADN+13, CADT14, KLS+17]. PSI protocols with input sets of different sizes were studied in [KLS+17, PSZ18, RA18].

**Circuit-based PSI.** These protocols use secure evaluation of circuits for PSI. A trivial circuit for PSI computes $O(n^2)$ comparisons which result in $O(\sigma n^2)$ gates, where $\sigma$ is the bit-length of the elements.
The sort-compare-shuffle (SCS) PSI circuit of [HEK12] computes $O(n \log n)$ comparisons and is of size $O(\sigma n \log n)$ gates (even without the final shuffle layer).
The Circuit-Phasing PSI circuit of [PSSZ15] uses Cuckoo hashing to $O(n)$ bins by one party and simple hashing by the other party which maps at most $O(\log n / \log \log n)$ elements per bin. Therefore, the Circuit-Phasing circuit has a size of $O(\sigma n \log n / \log \log n)$ gates.
The recent circuit-based PSI protocol of [CO18] applies a protocol based on OT extension to compute private set membership in each bin. The outputs of the invocations of this protocol are input to a comparison circuit. The circuit itself computes a linear number of comparisons, but the total communication complexity of the private set membership protocols is of the same order as that of the Circuit-Phasing circuit [PSSZ15] with $O(\sigma n \log n / \log \log n)$ gates.

Another recent circuit-based PSI protocol of [FNO18, Section 8] has communication complexity $O(\sigma n \log\log n)$. It uses hashing to $O(n)$ bins where each bin has multiple buckets and then runs the SCS circuit of [HEK12] to compute the intersection of the elements in the respective bins.
The two-dimensional Cuckoo hashing circuit of [PSWW18] uses a new variant of Cuckoo hashing in two dimensions and has an almost linear complexity of $\omega(\sigma n)$ gates.

In this work, we present the first circuit-based PSI protocol with a true linear complexity of $O(\sigma n)$ gates.

**PSI Cardinality.** Several protocols for securely computing the cardinality of the intersection, i.e., $|X \cap Y|$, were proposed in the literature. These protocols have linear complexity and are based on public-key cryptography, namely Diffie-Hellman [DGT12], the Goldwasser-Micali cryptosystem [DD15], or additively homomorphic encryption [DC17]. However, these protocols reveal the cardinality of the intersection to one of the parties. In contrast, circuit-based PSI protocols can easily be adapted to efficiently compute the cardinality and even functions of it using mostly symmetric cryptography.

### 1.5 Our Contributions

In summary, in this paper we present the following contributions:
- The first circuit-based PSI protocol with linear asymptotic communication overhead. We remark that achieving a linear overhead is technically hard since hashing to a table of linear size requires a stash of super-linear size in order to guarantee a negligible failure probability. It is hard to achieve linear overhead with objects of super-linear size.
- A circuit-based PSI protocol with small constants and an improved concrete overhead over the state of the art. As a special case, we consider a very common variant of PSI, namely threshold PSI, in which the intersection is revealed only if it is bigger/smaller than some threshold. Surprisingly, our protocol is 1-2 orders of magnitude more efficient than the state-of-the-art [ZC18] and has the same asymptotic communication complexity of $O(n)$, despite the fact that the protocol in [ZC18] is a special purpose protocol for threshold-PSI.
- Our protocol supports associating data ("payload") with each input (from both parties), and compute a function that depends on the data associated with the items in the intersection. This property was not supported by the Phasing circuit-based protocol in [PSSZ15]. It is important for applications that compute some function of data associated with the items in the intersection, e.g., aggregate revenues from common users (cf. §1.2).
- On a technical level, we present a new paradigm for handling $\omega(1)$ stash sizes and obtaining an overall overhead that is linear. This is achieved by running an extremely simple dual-execution variant of the protocol.
- Finally, with regards to concrete efficiency, we introduce a circuit-based PSI protocol with linear complexity. This is achieved by using Cuckoo hashing

with $K = 3$ instead of $K = 2$ hash functions, and no stash. This protocol substantially reduces communication (by a factor of 2.6x to 12.8x) and runtime (by a factor of 2.8x to 5.8x) compared to the best previous circuit-based PSI protocol of [PSWW18].

## 2 Preliminaries

### 2.1 Setting

There are two parties, which we denote as $P_1$ (the "receiver") and $P_2$ (the "sender"). They have input sets, $X$ and $Y$, respectively, each of which contains $n$ items of bitlength $\lambda$. We assume that both parties agree on a function $f$ and wish to securely compute $f(X \cap Y)$. They also agree on a circuit $C$ that receives the items in the intersection as input and computes $f$. That is, $C$ has $O(n\lambda)$ input wires if we consider a computation on the elements themselves or $O(n(\lambda + \rho))$ if we consider a computation on the elements and their associated payloads where the associated payload of each item has bitlength $\rho$. We denote the computational and statistical security parameters by $\kappa$ and $\sigma$, respectively. Denote the set $1, \ldots, c$ by $[c]$. We use the notation $X(i)$ to denote the $i$-th element in the set $X$.

### 2.2 Security Model

This work, similar to most protocols for private set intersection, operates in the semi-honest model, where adversaries may try to learn as much information as possible from a given protocol execution but are not able to deviate from the protocol steps. This is in contrast to malicious adversaries which are able to deviate arbitrarily from the protocol. PSI protocols for the malicious setting exist, but they are less efficient than protocols for the semi-honest setting, e.g., [FNP04, DSMRY09, HN10, DKT10, FHNP16, RR17a, RR17b]. The only circuit-based PSI protocol that can be easily secured against malicious adversaries is the Sort-Compare-Shuffle protocol of [HEK12]: here a circuit of size $O(n)$ can be used to check that the inputs are sorted, resulting in an overall complexity of $O(n \log n)$. For the recent circuit-based PSI protocols that rely on Cuckoo hashing, ensuring that the hashing was done correctly remains the challenge. The semi-honest adversary model is appropriate for scenarios where execution of the intended software is guaranteed via software attestation or business restrictions, and yet an untrusted third party is able to obtain the transcript of the protocol after its execution, by stealing it or by legally enforcing its disclosure.

### 2.3 Secure Two-Party Computation

There are two main approaches for generic secure two-party computation of Boolean circuits with security against semi-honest adversaries: (1) Yao's garbled circuit protocol [Yao86] has a constant round complexity and with today's most efficient optimizations provides free XOR gates [KS08], whereas securely

**FUNCTIONALITY 1 (Two-Party Computation)**

**Parameters.** The Boolean circuit $C$ to be computed, with $I_1, I_2$ inputs and $O_1, O_2$ outputs associated with $P_1$ and $P_2$ resp.
**Inputs.** $P_1$ inputs bits $x_1, \ldots, x_{I_1}$ and $P_2$ inputs bits $y_1, \ldots, y_{I_2}$.
**Outputs.** The functionality computes the circuit $C$ on the parties' inputs and returns the outputs to the parties.

evaluating an AND gate requires sending two ciphertexts [ZRE15]. (2) The GMW protocol [GMW87] also provides free XOR gates and also sends two ciphertexts per AND gate using OT extension [ALSZ13].

The main advantage of the GMW protocol is that *all* symmetric cryptographic operations can be pre-computed in a constant number of rounds in a setup phase, whereas the online phase is very efficient, but requires interaction for each layer of AND gates. In more detail, the setup phase is independent of the actual inputs and precomputes multiplication triples for each AND gate using OT extension in a constant number of rounds. The online phase begins when the inputs are provided and involves a communication round for each layer of AND gates. See [SZ13] for a detailed description and comparison between Yao and GMW.

In our protocol we make use of Functionality 1.

### 2.4 Cuckoo Hashing

Cuckoo hashing [PR01] uses two hash functions $h_0, h_1$ to map $n$ elements to two tables $T_0, T_1$ which each contain $(1+\varepsilon)n$ bins. (It is also possible to use a single table $T$ with $2(1+\varepsilon)n$ bins. The two versions are essentially equivalent.) Each bin accommodates at most a single element. The scheme avoids collisions by relocating elements when a collision is found using the following procedure: Let $b \in \{0,1\}$. An element $x$ is inserted into a bin $h_b(x)$ in table $T_b$. If a prior item $y$ exists in that bin, it is evicted to bin $h_{1-b}(y)$ in $T_{1-b}$. The pointer $b$ is then assigned the value $1-b$. The procedure is repeated until no more evictions are necessary, or until a threshold number of relocations has been reached. In the latter case, the last element is put in a special stash. It was shown in [KMW09] that for a stash of constant size $s$ the probability that the stash overflows is at most $O(n^{-(s+1)})$. It was also shown in [GM11] that this failure probablity is negligilble when the stash is of size $O(\log n)$. An observation in [KM18] shows that this is also the case when $s = O(\omega(1) \cdot \frac{\log n}{\log \log n})$. After insertion, each item can be found in one of two locations or in the stash.

### 2.5 PSI based on Hashing

Existing constructions for circuit-based PSI require the parties to reorder their inputs before inputting them to the circuit. In the sorting network-based circuit of [HEK12], the parties sort their inputs. In the hashing-based construction of [PSSZ15], the parties map their items to bins using a hashing scheme.

It was observed as early as [FNP04] that if the two parties agree on the same hash function and use it to assign their respective input to bins, then the items that one party maps to a specific bin only need to be compared to the items that the other party maps to the same bin. However, the parties must be careful not to reveal to each other the number of items that they mapped to each bin, since this leaks information about their input sets. Therefore, the parties agree beforehand on an upper bound $m$ for the maximum number of items that can be mapped to a bin (such upper bounds are well known for common hashing algorithms, and can also be substantiated using simulation), and pad each bin with random dummy values until it has exactly $m$ items in it. If both parties use the same hash algorithm, then this approach considerably reduces the overhead of the computation from $O(n^2)$ to $O(\beta \cdot m^2)$ where $\beta$ is the number of bins.

When using a random hash function $h$ to map $n$ items to $n$ bins such that $x$ is mapped to bin $h(x)$, the most occupied bin has at most $m = \frac{\ln n}{\ln \ln n}(1 + o(1))$ items with high probability [Gon81]. For instance, for $n = 2^{20}$ and a desired error probability of $2^{-40}$, a careful analysis shows that $m = 20$. Cuckoo hashing is much more promising, since it maps $n$ items to $2n(1 + \varepsilon)$ bins, where each bin stores at most $m = 1$ items.

It is tempting to let both parties, $P_1$ and $P_2$, map their items to bins using Cuckoo hashing, and then only compare the item that $P_1$ maps to a bin with the item that $P_2$ maps to the same bin. The problem is that $P_1$ might map $x$ to $h_0(x)$ whereas $P_2$ might map it to $h_1(x)$. Unfortunately, they cannot use a protocol where $P_1$'s value in bin $h_0(x)$ is compared to the two bins $h_0(x), h_1(x)$ in $P_2$'s input, since this reveals that $P_1$ has an item which is mapped to these two locations. The solution used in [FHNP16, PSZ14, PSSZ15] is to let $P_1$ map its items to bins using Cuckoo hashing, and $P_2$ map its items using simple hashing. Namely, each item of $P_2$ is mapped to both bins $h_0(x), h_1(x)$. Therefore, $P_2$ needs to pad its bins to have exactly $m = O(\log n / \log \log n)$ items in each bin, and the total number of comparisons is $O(n \log n / \log \log n)$.

## 3 OPPRF – Oblivious Programmable PRF

Our protocol builds on a (batched) oblivious programmable pseudorandom function (OPPRF). In this section we gradually present the properties required by that kind of a primitive, by first describing simpler primitives, namely, Programmable PRF (and its batched version) and Oblivious PRF.

### 3.1 Oblivious PRF

An oblivious PRF (OPRF) [FIPR05] is a two-party protocol implementing a functionality between a sender and a receiver. Let $F$ be a pseudo-random function (PRF) such that $F : \{0,1\}^\kappa \times \{0,1\}^\ell \to \{0,1\}^\ell$. The sender inputs a key $k$ to $F$ and the receiver inputs $q_1 \ldots, q_c$. The functionality outputs $F(k, q_1), \ldots, F(k, q_c)$ to the receiver and nothing to the sender. In another variant of oblivious PRF the sender is given a fresh random key $k$ as an output from the functionality rather

than choosing it on its own. In our protocol we will make use of a "one-time" OPRF functionality in which the receiver can query *a single query*, namely, the sender inputs nothing and the receiver inputs a query $q$; the functionality outputs to the sender a key $k$ and to the receiver the result $F_k(q)$. Let us denote that functionality by $\mathcal{F}_{\mathsf{OPRF}}$.

### 3.2 (One-Time) Programmable PRF (PPRF)

A programmable PRF (PPRF) is similar to a PRF, with the additional property that on a certain "programmed" set of inputs the function outputs "programmed" values. Namely, for an arbitrary set $X$ and a "target" multi-set $T$, where $|X| = |T|$ and each $t \in T$ is uniformly distributed[5], it is guaranteed that on input $X(i)$ the function outputs $T(i)$. Let $\mathcal{T}$ be a distribution of such multi-sets, which may be public to both parties.

The restriction of the PPRF to be only one-time comes from the fact that we allow the elements in $T$ to be correlated. If the elements are indeed correlated then by querying it two times (on the correlated positions) it would be easy to distinguish it from a random function.

We capture the above notion by the following formal definition:

**Definition 1.** *An $\ell$-bits PPRF is a pair of algorithms $\hat{F} = (\mathsf{Hint}, F)$ as follows:*

- $\mathsf{Hint}(k, X, T) \to \mathsf{hint}_{k,X,T}$*: Given a uniformly random key $k \in \{0,1\}^\kappa$, the set $X$ where $|X(i)| = \ell$ for all $i \in [|X|]$ and a target multi-set $T$ with $|T| = |X|$ and all elements in $T$ are uniformly distributed (but may be correlated), output the hint $\mathsf{hint}_{k,X,T} \in \{0,1\}^{\kappa \cdot |X|}$.*
- $F(k, \mathsf{hint}, x) \to y^\star$*. Given a key $k \in \{0,1\}^\kappa$, a hint $\mathsf{hint} \in \{0,1\}^{\kappa \cdot |X|}$ and an input $x \in \{0,1\}^\ell$, output $y^\star \in \{0,1\}^\ell$.*

*We consider two properties of a PPRF, correctness and security:*

- **Correctness.** *For every $k, T$ and $X$, and for every $i \in [|X|]$ we have:*

$$F\left(k, \mathsf{hint}, X(i)\right) = T(i).$$

- **Security.** *We say that an interactive machine $M$ is a* PPRF oracle over $\hat{F}$ *if, when interacting with a "caller" $\mathcal{A}$, it works as follows:*
  1. *$M$ is given a set $X$ from $\mathcal{A}$.*
  2. *$M$ samples a uniformly random $k \in \{0,1\}^\kappa$ and $T$ from $\mathcal{T}$, invokes $\mathsf{hint} \leftarrow \mathsf{Hint}(k, X, T)$ and hands $\mathsf{hint}$ to $\mathcal{A}$.*
  3. *$M$ is given an input $x \in \{0,1\}^\ell$ from $\mathcal{A}$ and responds with $F(k, \mathsf{hint}, x)$.*
  4. *$M$ halts (any subsequent queries will be ignored).*

  *The scheme $\hat{F}$ is said to be* secure *if, for every $X$ input by $\mathcal{A}$ (i.e. the caller), the interaction of $\mathcal{A}$ with $M$ is computationally indistinguishable from the interaction with the* PPRF oracle $\mathcal{S}$*, where $\mathcal{S}$ outputs a uniformly random "hint" $\{0,1\}^{\kappa \cdot |X|}$ and a "PRF result" from $\{0,1\}^\ell$.*

[5] We require that each element in $T$ is uniformly random but the elements may be correlated.

**CONSTRUCTION 2 (PPRF)**

Let $F' : \{0,1\}^\kappa \times \{0,1\}^\ell \to \{0,1\}^\ell$ be a PRF.
- $\mathsf{Hint}(k, X, T)$. Interpolate a polynomial $p$ over the points $\left\{\left(X(i)\ ,\ F'_k(X(i)) \oplus T(i)\right)\right\}_{i\in[|X|]}$. Return $p$ as the hint.
- $F(k, \mathsf{hint}, x)$. Interpret $\mathsf{hint}$ as a polynomial, denoted $p$. Return $F'_k(x) \oplus p(x)$.

The definition is reminiscent of a semantically secure encryption scheme. Informally, semantic security means that whatever is efficiently computable about the cleartext given the ciphertext, is also efficiently computable without the ciphertext. Also here, whatever can be efficiently computable given $X$ is also efficiently computable given only $|X|$. That implicitely means that the interaction with a PPRF oracle $M$ over $(\mathsf{KeyGen}, F)$ does not leak the elements in $X$.

Our security definition diverges from that of [KMP+17] in two aspects:

1. In [KMP+17], $\mathcal{A}$ has many queries to $M$ in Step 3 of the interaction, whereas our definition allows only a single query. In the $(n,t)$-security definition in [KMP+17] this corresponds to setting $t = 1$. Our definition is weaker in this sense, but this is sufficient for our protocol as we invoke multiple instances of the one-time PPRF.
2. The definition in [KMP+17] compensates for the fact that $\mathcal{A}$ has many queries, by requiring that the function $F$ outputs an *independent* target value for every $x \in X$. Our definition is stronger as it allows having correlated target elements in $T$. In the most extreme form of correlation all values in $T$ are equal, which makes the task of the adversary "easier". We require the security property to hold even in this case.

We present in Construction 2 a polynomial-based PPRF scheme that is based on the construction in [KMP+17].

**Theorem 3.** *Construction 2 is a PPRF.*

*Proof.* It is easy to see that this construction is correct. For every $k, X$ and $T$, let $p = \mathsf{Hint}(k, X, T)$, then for all $i \in |X|$ it holds that

$$\begin{aligned} F(k, p, X(i)) &= F'(k, X(i)) \oplus p(X(i)) \\ &= F'(k, X(i)) \oplus F'(k, X(i)) \oplus T(i) \\ &= T(i) \end{aligned}$$

as required. We now reduce the security of the scheme to the security of a PRF (i.e., to the standard PRF definition, with many oracle accesses). Let $M$ be a PPRF oracle over $\hat{F}$ of Construction 2. Assume there exists a distinguisher $\mathcal{D}$ and a caller $\mathcal{A}$ such that $\mathcal{D}$ distinguishes between the output of $M$ after interacting with $\mathcal{A}$, when $\mathcal{A}$ chooses $X$ and $x$ as its inputs, and the output of $\mathcal{S}(1^\kappa, |X|)$ (where $\mathcal{S}$ is the simulator described in Definition 1) with probability $\mu$.

We present a distinguisher $\mathcal{D}'$ that has an oracle access to either a truly random function $R(\cdot)$ or a PRF $\tilde{F}(k, \cdot)$. The distinguisher $\mathcal{D}'$ runs as follows:

Given an oracle $\mathcal{O}$ to either $R(\cdot)$ or $\tilde{F}(k,\cdot)$, $\mathcal{D}'$ samples $T$ from $\mathcal{T}$, then, for every $i \in [|X|]$ it queries the oracle on $X(i)$ and obtains $\mathcal{O}(X(i))$. It interpolates the polynomial $p$ using the points $\{(X(i), \mathcal{O}(X(i)) \oplus T(i))\}_{i \in |X|}$ and provides $p$'s coefficients to $\mathcal{D}$. For the query $x$, $\mathcal{D}'$ hands $\mathcal{D}$ the value $\mathcal{O}(x) \oplus p(x)$ and outputs whatever $\mathcal{D}$ outputs.

Observe that if $\mathcal{O}$ is truly random, then the values $\{R(X(i)) \oplus T(i))\}_{i \in [|X|]}$ are uniformly random and thus the polynomial $p$ is uniformly random and independent of $T$. If $x \notin X$ then the value $R(x) \oplus p(x)$ is obviously random since $R(x)$ is independent of $p$. In addition, if $x = X(i)$ for some $i$, then the value $R(x) \oplus p(x)$ equals $T(i)$ for some $i \in [|X|]$, which is uniformly random since $T$ is sampled from $\mathcal{T}$ and every $t \in T$ is distributed uniformly. Therefore, the pair $(p, R(x) \oplus p(x))$ is distributed identically to the output of $\mathcal{S}$. On the other hand, if $\mathcal{O}$ is a pseudorandom function, then the values $\{F_k(X(i)) \oplus T(i))\}_{i \in [|X|]}$ from which the polynomial $p$ is interpolated, along with the second output $F_k(x) \oplus p(x)$, are distributed identically to the output of $M$ upon an interaction with $\mathcal{A}$. This leads to the same distinguishing success probability $\mu$, for both $\mathcal{D}$ and $\mathcal{D}'$, which must be negligible. □

### 3.3 Batch PPRF

Note that the size of the hint generated by algorithm $\mathsf{KeyGen}$ is $\kappa \cdot |X|$ (i.e., the polynomial is represented by $|X|$ coefficients, each of size $\kappa$ bits). In our setting we use an independent PPRF per bin, where each bin contains at most $O(\log n / \log\log n)$ values. Therefore the hint for one bin is of size $O(\kappa \cdot \log n / \log\log n)$, and the size of all hints is $O(\kappa \cdot n \cdot \log n / \log\log n)$. However, we know that the total number of values in all $P_2$'s bins is $2n$, since each value is stored in (at most) two locations of the table[6]. We next show that it is possible to combine the hints of *all* bins to a single hint of length $2n$, thus reducing the total communication for all hints to $O(n)$.

We first present a formal definition of the notion of batch PPRF.

**Definition 2.** *An $\ell$-bits, $\beta$-bins PPRF (or $(\ell, \beta)$-PPRF) is a pair of algorithms $\hat{F} = (\mathsf{KeyGen}, F)$ as follows:*

- $\mathsf{Hint}(k, X, T) \to \mathsf{hint}_{k,X,T}$. *Given a set of uniformly random and independent keys $k = k_1, \ldots, k_\beta \in \{0,1\}^\kappa$, the sets $X = X_1, \ldots, X_\beta$ where $|X_j(i)| = \ell$ for all $j \in [\beta]$ and $i \in [|X|]$ and a target multi-sets $T = T_1, \ldots, T_\beta$ where for every $j \in [\beta]$ it holds that $|T_j| = |X_j|$ and all elements in $T_j$ are uniformly distributed (but, again, may be correlated), output the hint $\mathsf{hint}_{k,X,T} \in \{0,1\}^{\kappa \cdot N}$ where $N = \sum_{j=1}^{\beta} |X_j|$.*
- $F(k, \mathsf{hint}, x) \to y^\star$. *Given a key $k \in \{0,1\}^\kappa$, a hint $\mathsf{hint} \in \{0,1\}^{\kappa \cdot N}$ and an input $x \in \{0,1\}^\ell$, output $y^\star \in \{0,1\}^\ell$.*

*As before, we want a batched PPRF to have the following properties:*

[6] In the actual implementation we use a more general variant of Cuckoo hashing with a parameter $K \in \{2,3\}$ where each item is stored in $K$ locations in the table. The size of the hint will be $K \cdot n$.

**CONSTRUCTION 4 (Batched PPRF)**

Let $F' : \{0,1\}^\kappa \times \{0,1\}^\ell \to \{0,1\}^\ell$ be a PRF.

- $\mathsf{Hint}(k, X, T)$.
  Given the keys $k = k_1, \ldots, k_\beta$, the sets $X = X_1, \ldots, X_\beta$ and the target multi-sets $T = T_1, \ldots, T_\beta$, interpolate the polynomial $p$ using the points $\{(X_j(i), F'(k_j, X_j(i)) \oplus T_j(i))\}_{j \in \beta; i \in [|X_j|]}$. Return $p$ as the hint.
- $F(k, \mathsf{hint}, x)$.
  Interpret $\mathsf{hint}$ as a polynomial, denoted $p$. Return $F'(k, x) \oplus p(x)$. (Same as in Construction 2.)

- **Correctness.** *For every $k = k_1, \ldots, k_\beta$, $T = T_1, \ldots, T_\beta$ and $X = X_1, \ldots, X_\beta$ as above, we have*
$$F\left(k_j, \mathsf{hint}, X_j(i)\right) = T_j(i)$$
*for every $j \in [\beta]$ and $i \in [|X_j|]$.*
- **Security.** *We say that an interactive machine $M$ is a* batched PPRF oracle over $\hat{F}$ *if, when interacting with a "caller" $\mathcal{A}$, it works as follows:*
  1. *$M$ is given $X = X_1, \ldots, X_\beta$ from $\mathcal{A}$.*
  2. *$M$ samples uniformly random keys $k = k_1, \ldots, k_\beta$ and target multi-sets $T = T_1, \ldots, T_\beta$ from $\mathcal{T}$, and invokes $\mathsf{hint} \leftarrow \mathsf{Hint}(k, X, T)$ hands $\mathsf{hint}$ to $\mathcal{A}$.*
  3. *$M$ is given $\beta$ queries $x_1, \ldots, x_\beta$ from $\mathcal{A}$ and responds with $y_1^\star, \ldots, y_\beta^\star$ where $y_j^\star = F(k_j, \mathsf{hint}, x_j)$.*
  4. *$M$ halts.*

  *The scheme $\hat{F}$ is said to be secure if for every disjoint sets $X_1, \ldots, X_\beta$ (where $N = \sum_{j \in [\beta]} |X_j|$) input by a PPT machine $\mathcal{A}$, the output of $M$ is computationally indistinguishable from the output of $\mathcal{S}(1^\kappa, N)$, such that $\mathcal{S}$ outputs a uniformly random $\mathsf{hint} \in \{0,1\}^{\kappa \cdot N}$ and a set of $\beta$ uniformly random values from $\{0,1\}^\ell$.*

Construction 4 is a batched version of Construction 2.

**Theorem 5.** *Construction 4 is a secure $(\ell, \beta)$-PPRF.*

*Proof.* For correctness, note that for every $j \in [\beta]$ and $i \in [|X_j|]$ it holds that

$$\begin{aligned} F(k_j, p, X_j(i)) &= F'(k_j, X_j(i)) \oplus p(X_j(i)) \\ &= F'(k_i, X_j(i)) \oplus F'(k_i, X_j(i)) \oplus T_j(i) \\ &= T_j(i). \end{aligned}$$

The security of the scheme is reduced to the security of a batch PRF $\tilde{F}$. Informally, a batch PRF works as follows: Sample uniform keys $k_1, \ldots, k_\beta \in \{0,1\}^\kappa$ and for a query $(j, x)$ respond with $\tilde{F}(k_j, x)$. One can easily show that a batch PRF is indistinguishable from a set of $\beta$ truly random functions $R_1, \ldots, R_\beta$ where on query $(j, x)$ the output is $R_j(x)$.

Let $M$ be a batched PPRF oracle over $\hat{F}$ of Construction 4. Assume there exists a distinguisher $\mathcal{D}$ and a caller $\mathcal{A}$ such that $\mathcal{D}$ distinguishes between the output of $M$ after interacting with $\mathcal{A}$, when $\mathcal{A}$ chooses $X_1, \ldots, X_\beta$ and $x_1, \ldots, x_\beta$ as its inputs, and the outputs of $\mathcal{S}(1^\kappa, N)$, where $\mathcal{S}$ is the simulator described in Definition 2.

We present a distinguisher $\mathcal{D}'$ that has an oracle access $\mathcal{O}$, to either a batch PRF $\tilde{F}(k_j, \cdot)$ or a set of truly random functions $R_j(\cdot)$ (where $j \in [\beta]$). The distinguisher $\mathcal{D}'$ works as follows: sample $T_1, \ldots, T_\beta$ from $\mathcal{T}$, interpolate a polynomial $p$ with the points $\{(X_j(i), \mathcal{O}(j, X_j(i)) \oplus T_j(i))\}_{j \in [\beta]; i \in [|X_j|]}$ and hand $p$'s coefficients to $\mathcal{D}$ as the hint. Then, for query $x_j$ of $\mathcal{D}$, respond with $y_j^\star = \mathcal{O}(x_j) \oplus p(x_j)$. Finally, $\mathcal{D}'$ outputs whatever $\mathcal{D}$ outputs.

First note that if $\mathcal{O}$ is a set of truly random functions then the polynomial $p$ is uniformly random and independent of $y_1^\star, \ldots, y_\beta^\star$ because all interpolation points are uniformly random. Now, if $x_j \notin X_j$ then the result is obviously uniformly random. Otherwise, if $x_j = X_j(i)$ for some $i$ then note that the result is $T_j(i)$ which is uniformly random as well, since the other elements in $T_j$ are unknown. Thus, this is distributed identically to the output of $\mathcal{S}(1^\kappa, N)$. On the other hand, if $\mathcal{O}$ is a batch PRF then the interpolation points $\{(X_j(i), \mathcal{O}(j, X_j(i)) \oplus T_j(i))\}_{j \in [\beta]; i \in [|X_j|]}$ along with $y_1^\star, \ldots, y_\beta^\star$ are distributed identically to the output of $M$ upon an interaction with $\mathcal{A}$. This leads to the same distinguishing success probability for both $\mathcal{D}$ and $\mathcal{D}'$, which must be negligible. □

### 3.4 Batch *Oblivious* Programmable Pseudorandom Functions

In this section we define a two-party functionality for batched oblivious programmable pseudorandom function (Functionality 6), which is the main building block in our PSI protocols. The functionality is parametrised by a $(\ell, \beta)$-PPRF $\hat{F} = (\mathsf{Hint}, F)$ and interacts with a sender, who programs $\hat{F}$ with $\beta$ sets, and a receiver who queries $\hat{F}$ with $\beta$ queries. The functionality guarantees that the sender does not learn what are the receiver's queries and the receiver does not learn what are the programmed points.

Given a protocol that realizes $\mathcal{F}_{\mathsf{OPRF}}$ and a secure $(\ell, \beta)$-PPRF, the realization of Functionality 6 is simple and described in Protocol 7.

**Theorem 8.** *Given an $(\ell, \beta)$-PPRF, Protocol 7 securely realizes Functionality 6 in the $\mathcal{F}_{\mathsf{OPRF}}$-hybrid model.*

*Proof.* Note that party $P_2$ receives nothing in the functionality but receives $k_1, \ldots, k_\beta$ in the real execution as output from $\mathcal{F}_{\mathsf{OPRF}}$. Therefore, $P_2$'s view can be easily simulated with the simulator of $\mathcal{F}_{\mathsf{OPRF}}$.

As for the view of $P_1$, from the security of the PPRF it follows that it is indistinguishable from the output of $\mathcal{S}(1^\kappa, N)$ where $\mathcal{S}$ is the simulator from Definition 2. □

**FUNCTIONALITY 6 (Batch Oblivious PPRF)**

**Parameters.** A $(\ell, \beta)$-PPRF $\hat{F} = (\mathsf{Hint}, F)$.
**Sender's inputs.** These are the following values:
– Disjoint sets $X = X_1, \ldots, X_\beta$ where $|X_j(i)| \in \{0,1\}^\ell$ for every $j \in [\beta]$ and $i \in [|X_j|]$. Let the total number of elements in all sets be $N = \sum_j |X_j|$.
– The sets $T = T_1, \ldots, T_\beta$ sampled independently from $\mathcal{T}$.
**Receiver's inputs.** The queries $x_1, \ldots, x_\beta \in \{0,1\}^\ell$.

The functionality works as follows:
1. Sample uniformly random and independent keys $k = k_1, \ldots, k_\beta$.
2. Invoke $\mathsf{Hint}(k, X, T) \to \mathsf{hint}$.
3. Output $\mathsf{hint}$ to $P_1$ ($P_2$ can compute it on its own from $k, X, T$).
4. For every $j \in [\beta]$ output $F(k_j, \mathsf{hint}, x_j)$ to the receiver.

**PROTOCOL 7 (Batch Oblivious PPRF)**

The protocol is defined in the $\mathcal{F}_{\mathsf{OPRF}}$-hybrid model and receives an $(\ell, \beta)$-PPRF $\hat{F} = (\mathsf{Hint}, F)$ as a parameter. The underlying PRF in both $\mathcal{F}_{\mathsf{OPRF}}$ and $\hat{F}$ is the same and denoted $F'$. The protocol proceeds as follows:
1. The parties invoke $\beta$ instances of $\mathcal{F}_{\mathsf{OPRF}}$. In the $j \in [\beta]$ instance, $P_2$ inputs nothing and receives the key $k_j$, and $P_1$ inputs $x_j$ and receives $F'(k_j, x_j)$.
2. Party $P_2$ invokes $p \leftarrow \mathsf{Hint}(k, X, T)$ and sends $p$ to $P_1$.
3. For every $j \in [\beta]$, party $P_1$ outputs $F'(k_j, x_j) \oplus p(x_j)$.

# 4 A Super-Linear Communication Protocol

## 4.1 The Basic Construction

Let $C_{a,b}$ be a Boolean circuit that has $2 \cdot a \cdot (b + \lambda)$ input wires, divided to $a$ sections of $2b + \lambda$ inputs wires each. For each section, the first (resp. second) $\beta$ input wires are associated with $P_1$ (resp. $P_2$). The last $\lambda$ input wires are associated with $P_1$ as well. Denote the first (resp. second) $\beta$ bits input to the $i$-th section by $u_{i,1}$ (resp. $v_{i,2}$) and the last $\lambda$ bits by $z_i$. The circuit first compares $u_{i,1}$ to $v_{i,2}$ for every $i \in [\alpha]$ and produces $w_i = 1$ if $u_{i,1} = v_{i,2}$ and 0 otherwise. Then, the circuit computes and outputs $f(Z)$ where $Z = \{z_i \mid w_i = 1\}_{i \in [a]}$ and $f$ is the function required to be computed in the $\mathcal{F}_{\mathsf{PSI},f}$ functionality.

**Correctness.** If $z \in X \cap Y$ then $z$ is mapped to both $\mathsf{Table}_2[H_1(z)]$ and $\mathsf{Table}_2[H_2(z)]$ by $P_2$. There are two cases: (1) $z$ is mapped to $\mathsf{Table}_1[H_b(z)]$ by $P_1$ for $b \in \{1,2\}$. (2) $z$ is mapped to $\mathsf{Stash}$ by $P_1$. In the first case the match is found in section $H_b(z)$ of the circuit; in the second case the match is certainly found since every item in the $\mathsf{Stash}$ is compared to every item in $Y$.

Two items $x \in X$ and $y \in Y$ where $x \neq y$ will not be matched, since by the properties of the PPRF $P_1$ receives a pseudorandom output. Since the parties

**PROTOCOL 9 (Private Set Intersection)**

**Inputs.** $P_1$ has $X = \{x_1, \ldots, x_n\}$ and $P_2$ has $Y = \{y_1, \ldots, y_n\}$.
**Protocol.** The protocol proceeds in 3 steps as follows:

1. **Hashing.** The parties agree on hash functions $H_1, H_2 : \{0,1\}^\ell \to [\beta]$, which are used as follows:
   - $P_1$ uses $H_1, H_2$ in a Cuckoo hashing construction that maps $x_1, \ldots, x_n$ to a table $\mathsf{Table}_1$ of $\beta = 2(1+\varepsilon)n$ entries, where input $x_i$ is mapped to either entry $\mathsf{Table}_1[H_1(x_i)]$ or $\mathsf{Table}_1[H_2(x_i)]$ or the stash $\mathsf{Stash}$ (which is of size $s$)[a]. Since $\beta > n$, $P_1$ fills the empty entries in $\mathsf{Table}_1$ with a uniformly random value.
   - $P_2$ maps $y_1, \ldots, y_n$ to $\mathsf{Table}_2$ of $\beta$ entries using both $H_1$ and $H_2$. That is, $y_i$ is placed in both $\mathsf{Table}_2[H_1(y_i)]$ and $\mathsf{Table}_2[H_2(y_i)]$. (Obviously, some bins will have multiple items mapped to them. This is not an issue, and there is even no need to use a probabilistic upper bound on the occupancy of the bin.)
2. **Computing batch OPPRF.** $P_2$ samples uniformly random and independent target values $t_1, \ldots, t_\beta \in \{0,1\}^\kappa$. The parties invoke an $(\lambda, \beta)$-OPPRF (Functionality 6; recall that $\lambda$ is the bit-length of the items). $P_2$ inputs $Y_1, \ldots, Y_\beta$ and $T_1, \ldots, T_\beta$ where $Y_j = \mathsf{Table}_2[j] = \{y || j \mid y \in Y \wedge j \in \{H_1(y), H_2(y)\}\}$ and $T_j$ has $|Y_j|$ elements, all equal to $t_j$. If, $j = H_1(y) = H_2(y)$ for some $y \in Y$ then $P_2$ adds a uniformly random element to $\mathsf{Table}_2[j]$. $P_1$ inputs $\mathsf{Table}_1[1], \ldots, \mathsf{Table}_1[\beta]$ and receives $y_1^\star, \ldots, y_\beta^\star$. According to the definition of the OPPRF, if $\mathsf{Table}_1[j] \in \mathsf{Table}_2[j]$ then $y_j^\star = t_j$.
3. **Computing the circuit.** The parties use a two-party computation (Functionality 1) with the circuit $C_{\beta+s\cdot n,\gamma}$.[b] For section $j \in [\beta]$ of the circuit, party $P_1$ inputs the first $\gamma$ bits of $y_j^\star$ and $\mathsf{Table}_1[j]$, and $P_2$ inputs the first $\gamma$ bits $t_j$; for the $\beta + j$-th section $P_1$ inputs $\mathsf{Stash}[\lceil j/n \rceil + 1]$ and $P_2$ inputs $\mathsf{Table}[(j \bmod n) + 1]$.

[a] We discuss the value of $s$ in §4.2 and the value of $\varepsilon$ in §7.1.
[b] We discuss the value of $\gamma$ in §4.2.

only input the first $\gamma$ bits of the PPRF results, those values will be matched with probability $2^{-\gamma}$. See §4.2 for a discussion on limiting the failure probability.

**Security.** The security of the protocol follows immediately from the security of the OPPRF and the two-party computation functionalities.

### 4.2 Limiting the Failure Probability

Protocol 9 might fail due to two reasons:

- **Stash size.** For an actual implementation, one needs to fix $s$ and $\varepsilon$ so that the stash failure probability will be smaller than $2^{-\sigma}$. If the stash is overflowed

(i.e., more than $s$ items are mapped to it) then the protocol fails.[7] As discussed in §2, setting $s = O(\log n / \log\log n)$ makes the failure probability negligible.

– **Input encoding.** The circuit compares the first $\gamma$ bits of $y_j^\star$ of $P_1$ to the first $\gamma$ bits of $t_j$ of $P_2$. Thus, the false positive error probability in each comparison equals $2^{-\gamma}$ (due to $F(x)$, for $x \notin Y$, being equal to the programmed output), and therefore the overall probability of a false positive is at most $\beta \cdot 2^{-\gamma} = 2(1+\varepsilon)n \cdot 2^{-\gamma}$.

### 4.3 Reducing Computation

A major computation task of the protocol is interpolating the polynomial which encodes the hint. If we use Cuckoo hashing with $K = O(1)$ hash functions then the polynomial encodes $O(n)$ items and is of degree $O(n)$. This section describes how to reduce the *asymptotic* overhead of *computing* the polynomial and therefore we will use asymptotic notation. The concrete overhead is discussed in §7.2.

The overhead of interpolating a polynomial of degree $O(n)$ over arbitrary points is $O(n^2)$ operations using Lagrange interpolation, or $O(n \log^2 n)$ operations using FFT. The overhead can be reduced by dividing the polynomial to several lower-degree polynomials. In particular, let us divide the $\beta = O(n)$ bins to $B$ "mega-bins", each encompassing $\beta/B$ bins. Suppose that we have an upper bound such that the number of items in a mega-bin is at most $m$, except with negligible probability. Then the protocol can invoke a batch OPPRF for each mega-bin, using a different hint polynomial. Each such polynomial is of degree $m$. Therefore the computation overhead is $O(B \cdot m \log^2 m)$. Ideally, the upper bound on the number of items in a mega-bin, $m$, is of the same order as the expected number of items in a mega-bin, $O(n/B)$. In this case the computation overhead is $O(n/B \cdot B \cdot \log^2(n/B)) = O(n \log^2(n/B))$ and will be minimized when the number of mega-bins $B$ is maximal.

It is known that when mapping $O(n)$ items to $B = n/\log n$ (mega-)bins, then with high probability the most occupied bin has less than $m = O(n/B) = O(\log n)$ items. When interested in concrete efficiency we can use the analysis in [PSZ18] to find the exact number of mega-bins to make the failure probability sufficiently small (see §7.2). When interested in asymptotic analysis, it is easy to deduce from the analysis in [PSZ18] that with $B = n/\log n$ mega-bins, the probability of having more than $\omega(\log n)$ items in a mega-bin is negligible. Therefore when using this number of mega-bins, the computation overhead is only $\omega((n/\log n) \cdot \log^2(n)) = \omega(n \log n)$ using Lagrange interpolation. Using FFT interpolation, the asymptotic overhead is reduced to $\omega((n/\log n)\log n(\log\log n)^2) = \omega(n \cdot (\log\log n)^2)$. But since we map relatively few items to each mega-bin the gain in practice of using FFT is marginal.

[7] In that case either not all items are stored in the stash – resulting in the protocol ignoring part of the input and potentially computing the wrong output, or $P_1$ needs to inform $P_2$ that it uses a stash larger than $s$ – resulting in a privacy breach.

## 5 A Linear Communication Protocol

We describe here a protocol in which the circuit computes only $O(n)$ comparisons. This protocol outperforms the protocols in §4.1 or in [PSWW18, CO18] which have a circuit that computes $\omega(n)$ comparisons. A careful analysis reveals that those protocols require $O(n)$ comparisons to process all items that were mapped to the Cuckoo hash table, and an additional $s \cdot n$ comparisons to process the $s = \omega(1)$ items that were mapped to the stash. We note that the concurrent and independent work of [FNO18] proposes to use a PSI protocol for unbalanced set sizes, such as in the work of [KLS+17], to reduce the complexity of handling the stash from $\omega(n)$ to $O(n)$ in PSI protocols. However, their idea can only be applied when the output is the intersection itself. When the output is a function of the intersection then their protocol has communication complexity $O(n \log \log n)$, cf. §1.4). In contrast, we achieve $O(n)$ communication even when the output is a function of the intersection.

We present two different techniques to achieve a linear communication protocol with failure probability that is negligible in the statistical security parameter $\sigma$. The first technique (see §5.1) is implied by a mathematical analysis of the failure probability (as argued in §1.4). The second technique (see §5.2) is implied by the empirical analysis presented in [PSZ18].

### 5.1 Linear Communication via Dual Execution

We overcome the difficulty of handling the stash by running a modified version of the protocol in three phases. The first phase is similar to the basic protocol, but ignores the items that $P_1$ maps to the stash. Therefore this phase inputs to the circuit the $O(n)$ results of comparing $P_1$'s input items (except those mapped to the stash) with all of $P_2$'s items. The second phase reverses the roles of the parties, and in addition now $P_1$ inputs only the items that it previously mapped to the stash. In this phase $P_2$ uses Cuckoo hashing and might map some items to the stash. The last phase only compares the items that $P_1$ mapped to the stash in the first phase, to the items that $P_2$ mapped to the stash in the second phase, and therefore only needs to handle very few items. Below, we describe our protocol in more detail: In Protocol 10, we describe our protocol in more detail.

**Correctness & Efficiency.** The protocol compares every pair in $X \times Y$ and therefore every item in the intersection is input to the circuit exactly once: Sections $1, \ldots, \beta$ of the circuit cover all pairs in $X_T \times Y$, sections $\beta + 1, \ldots, 2\beta$ cover all pairs in $X_S \times Y_T$ and sections $2\beta + 1, \ldots, 2\beta + s^2$ covers all pairs in $X_S \times Y_S$. This implies that the result of the three-phase construction is exactly the intersection $X \cap Y$. The communication complexity in the first two steps of the protocol is $O(n \cdot \kappa)$ as they involve the execution of a OPPRF with at most $O(n)$ items to the parties. The communication complexity of the third step is $O(n \cdot \gamma)$ since it involves $2n + s^2$ comparisons of $\gamma$-bit elements. Since the stash size is $s = O(\log n)$, overall there are $O(n)$ comparisons.

**PROTOCOL 10 (PSI with Linear Communication)**

**Inputs.** $P_1$ has $X = \{x_1, \ldots, x_n\}$ and $P_2$ has $Y = \{y_1, \ldots, y_n\}$.
**Protocol.** The protocol proceeds in 3 phases as follows:

1. Run steps 1-2 of Protocol 9. Denote the items mapped to $P_1$'s table by $X_T$ (i.e., excluding the items mapped to the stash). In the end of this phase, for every $j \in [\beta]$, $P_1$ holds the OPPRF result $y_j^\star$ and $P_2$ holds the target value $t_j$.
2. Reverse the roles of $P_1$ and $P_2$ and run steps 1-2 of Protocol 9 again, where $P_1$ inputs $X_S = X \setminus X_T$ (i.e., only the items that were previously mapped to the stash) and $P_2$ inputs $Y$. Since the roles are reversed then $P_1$ maps $X_S$ using simple hashing and $P_2$ maps $Y$ using Cuckoo hashing. Denote the items mapped to the table and stash of $P_2$ by $Y_T$ and $Y_S$, respectively. In the end of this phase, for every $j \in [\beta]$, $P_1$ has the target value $\tilde{t}_j$ and $P_2$ has the OPPRF result $\tilde{y}_j^\star$.
3. The parties use secure two-party computation (Functionality 1) with the circuit $C_{2\beta+s^2,\gamma}$ (where $s$ is the stash size). For section $j \in [\beta]$ of the circuit, $P_1$ and $P_2$ input the first $\gamma$ bits of $y_j^\star$ and $t_j$ resp. For section $j \in \{\beta+1, \ldots, 2\beta\}$ of the circuit $P_1$ and $P_2$ input the first $\gamma$ bits of $\tilde{t}_j$ and $\tilde{y}_j^\star$, respectively. Finally, for the rest $s^2$ sections of the circuit, the parties input every combination of $X_S \times Y_S$ (padded with uniformly random items so that $|X_S| = |Y_S| = s$).

**Security.** As in the basic protocol (see §4.1), the security of this protocol is implied by the security of the OPPRF and secure two-party computation.

### 5.2 Linear Communication via Stash-less Cuckoo Hashing

The largest communication cost factor in our protocols is the secure evaluation of the circuit. The asymptotically efficient Protocol 10 requires computing at least two copies of the basic circuit (for Phases 1 and 2), and it is therefore preferable to implement a protocol which has better *concrete* efficiency. We design a protocol that requires no stash (while achieving a small failure probability of less than $2^{-40}$), and hence uses no dual execution.

In order to be able to not use the stash, hashing is done with $K > 2$ hash functions. We take into account the results of [PSZ18], which ran an empirical evaluation for the failure probability of Cuckoo hashing (failure is defined as the event where an item cannot be stored in the table and must be stored in the stash). They run experiments for a failure probability of $2^{-30}$ with $K = 3, 4$ and 5 hash functions, and extrapolated the results to yield the minimum number of bins for achieving a failure probability of less than $2^{-40}$. The results showed that $\beta = 1.27n, 1.09n$, and $1.05n$ bins are required for $K = 3, 4$, and 5, respectively.

The main obstacle in using more than two hash functions in previous works on PSI was that the communication was still linear in $O(\max_b \cdot \beta)$, where $\max_b$ is the maximal number of elements in a bin of the simple hash table. The value of $\max_b$ increases with $K$ since each item is stored $K$ times in the simple hash

table. In our protocol the communication for the circuit is independent of $\max_b$, as it only depends on the number of bins $\beta$. The communication for sending the polynomials, whose size is $O(K \cdot n \cdot \kappa)$, is just a small fraction of the overall communication and was in our experiments always smaller than 3%. In this paper, we therefore use $K = 3$ hash functions for our stash-less protocol.

## 6 PSI with Associated Payload

In many cases, each input item of the parties has some "payload" data associated with it. For example, an input item might include an id which is a credit card number, and a payload which is a transaction that was paid using this credit card. The parties might wish to compute some function of the *payloads* of the items in the intersection (for example, the sum of the financial transactions associated with these items). However, a straightforward application of our techniques does not seem to support this type of computation: Recall that $P_2$ might map multiple items to each bin. The OPPRF associates a single output $\beta$ to all these items, and this value is compared in the circuit with the output $\alpha$ of $P_1$. But if $P_2$ inserts a single item to the circuit, it seems that this item cannot encode the payloads of all items mapped to this bin.

The 2D Cuckoo hashing circuit-based PSI protocol of [PSWW18] handles payloads well, since each comparison involves only a single item from each party. While our basic protocol cannot handle payloads, we show here how it can be adapted to efficiently encode payloads in the input to the circuit.

Let $\mathsf{Table}_1$ and $\mathsf{Stash}$ be $P_1$'s table and stash after mapping its items using Cuckoo hashing and let $\mathsf{Table}_2$ be $P_2$'s table after mapping its items using simple hashing. In addition, denote by $U(x)$ and $V(y)$ the payloads associated with $x \in X$ and $y \in Y$ respectively and assume that all payloads have the same length $\delta$. The parties invoke two instances of batch OPPRF as follows:

1. A batch OPPRF where $P_1$ inputs $\mathsf{Table}_1[1], \ldots, \mathsf{Table}_1[\beta]$ and $P_2$ inputs $\mathsf{Table}_2[1], \ldots, \mathsf{Table}_2[\beta]$ and $T_1, \ldots, T_\beta$ where $T_j$ has $|\mathsf{Table}_2[j]|$ elements, all equal to a uniformly random and independent value $t_j \in \{0,1\}^\lambda$. This is the same invocation of a batch OPPRF as in Protocol 9. At the end, $P_1$ has the OPPRF results $y_1^\star, \ldots, y_\beta^\star$ and $P_2$ has the target values $t_1, \ldots, t_j$.
2. In the second batch OPPRF, $P_2$ chooses the target values such that the elements in the set $T_j$ are not equal. Specifically, $P_1$ inputs $\mathsf{Table}_1[1], \ldots, \mathsf{Table}_1[\beta]$ and $P_2$ samples $\tilde{t}_1, \ldots, \tilde{t}_\beta$ uniformly, and inputs $\mathsf{Table}_2[1], \ldots, \mathsf{Table}_2[\beta]$ and $T_1, \ldots, T_\beta$ where $T_j(i) = \tilde{t}_j \oplus V(\mathsf{Table}_2[j](i))$. Denote the OPPRF results that $P_1$ obtains by $\tilde{y}_1^\star, \ldots, \tilde{y}_\beta^\star$.

Then, the circuit operates in the following way: For the $j$-th section, $P_1$ inputs $\mathsf{Table}_1[j], y_j^\star, \tilde{y}_j^\star$ and $U(\mathsf{Table}_1[j])$, and $P_2$ inputs $t_j$ and $\tilde{t}_j$. The circuit compares $y_j^\star$ to $t_j$. If they are equal then it forwards to the sub-circuit that computes $f$ the item $\mathsf{Table}_1[j]$ itself, $P_1$'s payload $U(\mathsf{Table}_1[j])$ and $P_2$'s payload $\tilde{y}_j^\star \oplus \tilde{t}_j$. This holds since if $\mathsf{Table}_1[j]$ is the $i$-th item in $P_2$'s table, namely, $\mathsf{Table}_2[j](i)$, then the value $\tilde{y}_j^\star$ received by $P_1$ is $\tilde{y}_j^\star = \tilde{t}_j \oplus V(\mathsf{Table}_2[j](i))$. Thus, $\tilde{y}_j^\star \oplus \tilde{t}_j = V(\mathsf{Table}_2[j](i))$ as required.

**Table 1.** The results of [PSZ18] for the required stash sizes $s$ for $K = 2$ hash functions and $\beta = 2.4n$ bins, and the minimum OPPRF output bitlength $\gamma$ to achieve failure probability $< 2^{-40}$ when mapping $n$ elements into $\beta$ bins with Cuckoo hashing. For $K > 2$ hash functions we choose a large enough number of bins $\beta$ to achieve stash failure probability $< 2^{-40}$.

| # Elements $n$ | | $2^8$ | $2^{12}$ | $2^{16}$ | $2^{20}$ | $2^{24}$ |
|---|---|---|---|---|---|---|
| Stash size $s$ for $K = 2$ | | 12 | 6 | 4 | 3 | 2 |
| OPPRF output length $\gamma$ | $K = 2$, $\beta = 2.4n$ | 50 | 54 | 58 | 62 | 66 |
| | $K = 3$, $\beta = 1.27n$, $s = 0$ | 49 | 53 | 57 | 61 | 65 |
| | $K = 4$, $\beta = 1.09n$, $s = 0$ | 49 | 53 | 57 | 61 | 65 |
| | $K = 5$, $\beta = 1.05n$, $s = 0$ | 49 | 53 | 57 | 61 | 65 |

**Efficiency.** The resulting protocol has the same asymptotic complexity as our initial protocols without payloads. The number of comparisons in the circuit is the same as in the basic circuit.

## 7 Concrete Costs

In this section we evaluate the concrete costs of our protocol for concrete values of the security parameters. We set the computational security parameter to $\kappa = 128$, and the statistical security parameter to $\sigma = 40$.

### 7.1 Parameter Choices for Sufficiently Small Failure Probability

For $K = 2$ hash functions, following previous works on PSI (e.g., [PSSZ15, PSWW18]), we set the table size parameter for Cuckoo hashing to $\epsilon = 0.2$, and use a Cuckoo table with $\beta = 2n(1 + \epsilon) = 2.4n$ bins. The resulting stash sizes for mapping $n$ elements into $\beta = 2.4n$ bins, as determined by the experiments in [PSZ18], are summarized in Tab. 1. Note that we use here concrete values for the stash size, and are aiming for a failure probability smaller than $2^{-40}$. This can either be achieved using the basic protocol of §4.1 with the right choice of the stash size, or by running the three rounds $O(n)$ complexity protocol of §5.

Another option is described in §5.2, where we use more than two hash functions (specifically, use $K = 3, 4$, or 5 functions), with the hash table being of size $\beta = 1.27n, 1.09n$, or $1.05n$, respectively. These parameters achieve a failure probability smaller than $2^{-40}$ according to the experimental analysis in [PSZ18].

As described in §4.2, even if there are no stash failures, the scheme can fail due to collisions in the output of the PRF, with probability $\beta \cdot 2^{-\gamma}$, where $\gamma$ is the output bitlength of the OPPRF. To make this failure probability smaller than the statistical security parameter (which we set to 40), the output bitlength of the OPPRF must be $\gamma = 40 + \log_2 \beta$ bits.

### 7.2 Computing Polynomial Interpolation

We implemented interpolation of polynomials of degree $d$ using an $O(d^2)$ algorithm based on Lagrange interpolation in a prime field where the prime is the Mersenne prime $2^{61}-1$. The runtime for interpolating a polynomial of degree $d = 1\,024$ was 7 ms, measured on an Intel Core i7-4770K CPU with a single thread. The runtime for different values of $d$ behaved (very accurately) as a quadratic function of $d$. The actual algorithms are those implemented in NTL v10.0 with field arithmetics replaced with our customized arithmetic operations over the Mersenne prime $2^{61}-1$. Most importantly, this field enables an order of magnitude faster multiplication of field elements: multiplying $x \cdot y$ with $|x|, |y| \leq 61$ is implemented by multiplying $x$ and $y$ over $\mathbb{Z}$ to obtain $z = xy$ with $|z| \leq 122$. Then the result is the sum of the element represented by the lower 61 bits of $z$ with the element represented by the higher 61 bits of $z$ (and therefore no expensive modular reduction is required). The Mersenne prime $2^{61}-1$ allows the use of at least 40-bit statistical security for up to $n = 2^{20}$ elements for all our algorithms using permutation-based hashing (cf. [PSSZ15]). To use larger sets, we see two possible solutions: (i) using a larger Mersenne prime or (ii) reducing the statistical security parameter $\sigma$ (e.g., using $\sigma = 38$ for achieving less than $2^{-\sigma}$ failure probability for $n = 2^{22}$ elements, $K = 3$ hash functions, and $\beta = 1.27n$ bins). The required minimum bit-length of the elements using permutation-based hashing with failure probability $2^{-\sigma}$ is computed as $\ell = \sigma + 2\log_2 n - \log_2 \beta$. The OPPRF output is also $\leq 61$ bits in most cases as shown in Tab. 1.

For reducing the computation complexity of our protocol, we use the approach described in §4.3, where instead of interpolating a polynomial of degree $K \cdot n$, where $K$ is the number of hash functions and $n$ is the number of elements for PSI, we interpolate multiple smaller polynomials of degree at most $d = 1\,024$. We therefore have to determine the minimum number of mega-bins $B$ such that when mapping $N = K \cdot n$ elements to $B$ bins, the probability of having a bin with more than $\max_b = 1\,024$ elements is smaller than $2^{-40}$. As in the analysis for simple hashing in [PSZ18], we use the formula from [MR95]:

$$\begin{aligned} P(\text{“}\exists \text{ bin with } \geq \max_b \text{ elements”}) &\leq \sum_{i=1}^{B} P(\text{“bin } i \text{ has } \geq \max_b \text{ elements”}) \\ &= B \cdot \sum_{i=\max_b}^{N} \binom{N}{i} \cdot \left(\frac{1}{B}\right)^i \cdot \left(1-\frac{1}{B}\right)^{N-i}. \end{aligned}$$

We depict the corresponding numbers in Tab. 2. With these numbers and our experiments for polynomial interpolation described above, the estimated runtimes for the polynomial interpolation are $B \cdot 7$ ms. The hints (polynomials) that need to be sent have size $B \cdot \max_b \cdot \gamma$ bits which is only slightly larger than the ideal communication of $K \cdot n \cdot \gamma$ bits when using one large polynomial as shown in Tab. 2.

Note that in contrast to many PSI solutions whose main run-time bottleneck is already network bandwidth (which cannot be easily improved in many settings

**Table 2.** Parameters for mapping $N = K \cdot n$ elements to $B$ mega-bins s.t. each mega-bin has at most $\max_b \leq 1\,024$ elements with probability smaller than $2^{-40}$. The lower half of the table contains the expected costs for the polynomial interpolations.

| # hash functions | $K = 2$ | | | $K = 3$ | | |
|---|---|---|---|---|---|---|
| Set size | $n = 2^{12}$ | $n = 2^{16}$ | $n = 2^{20}$ | $n = 2^{12}$ | $n = 2^{16}$ | $n = 2^{20}$ |
| # mega-bins $B$ | 11 | 165 | 2 663 | 16 | 248 | 4 002 |
| Maximum number of elements $\max_b$ | 944 | 1 021 | 1 024 | 975 | 1 021 | 1 024 |
| Polynomial interpolation [in milliseconds] | 126 | 1 815 | 29 293 | 183 | 2 809 | 45 335 |
| Size of hints [in bits] | 560 736 | 9 770 970 | 169 068 544 | 826 800 | 14 432 856 | 249 980 928 |
| Ideal size of hints for one polynomial [in bits] | 436 330 | 7 505 580 | 128 477 895 | 651 264 | 11 206 656 | 191 889 408 |

such as over the Internet), the run-time of our protocols can be improved by using multiple threads instead of one thread. Since the interpolation of polynomials for different mega-bins is independent of each other, the computation scales linearly in the number of physical cores and thus can be efficiently parallelized.

### 7.3 Communication and Depth Comparison

We first compute the communication complexity of our basic construction from §4.1. The communication is composed of (a) the OPRF evaluations for each of the $B$ bins, (b) the hints consisting of the polynomials, (c) the circuit for comparing the outputs of the OPPRFs in each bin, and (d) the circuit for comparing the $s$ elements on the stash with the $n$ elements of $P_2$.

With regards to (a), the OPRF protocol of [KKRT16], which was also used in [KMP+17], has an amortized communication of at most 450 bits per OPRF evaluation for set sizes up to $n = 2^{24}$ elements (cf. [KKRT16, Tab. 1]). This amounts to $B \cdot 450$ bits of communication.

With regards to (b), for the size of the hints in the OPPRF construction we use the values given in Tab. 2. These numbers represent the communication when using mega-bins, and are slightly larger than the ideal communication of $K \cdot n$ coefficients of size $\gamma$ bits each, that would have been achieved by using a single polynomial for all values. However, it is preferable to use mega-bins since their usage substantially improves the computation complexity as described in §4.3, while the total communication for the hints is at most 3% of the total communication. (This also shows that any improvements of the size of the hints will have only a negligible effect on the total communication.)

With regards to (c), the circuit compares $B$ elements of bitlength $\gamma$, and hence requires $B \cdot (\gamma - 1)$ AND gates. With 256 bits per AND gate [ALSZ13, ZRE15] this yields $B \cdot (\gamma - 1) \cdot 256$ bits of communication.

With regards to (d), the final circuit consists of $s \cdot n$ comparisons of bitlength $\sigma$. This requires $sn(\sigma - 1) \cdot 256$ bits of communication.

**Table 3.** Communication in MB for circuit-based PSI on $n$ elements of fixed bitlength $\sigma = 32$ (left) and arbitrary bitlength hashed to $\sigma = 40 + 2\log_2(n) - 1$ bits (right). The numbers for previous protocols are based on the circuit sizes given in [PSWW18, Tab. 3] with 256 bit communication per AND gate. The best values are marked in bold.

| | $\sigma = 32$ | | | Arbitrary $\sigma$ | | |
|---|---|---|---|---|---|---|
| **Protocol** $n =$ | $\mathbf{2^{12}}$ | $\mathbf{2^{16}}$ | $\mathbf{2^{20}}$ | $\mathbf{2^{12}}$ | $\mathbf{2^{16}}$ | $\mathbf{2^{20}}$ |
| SCS [HEK12] | 104 | 2 174 | 42 976 | 205 | 4 826 | 106 144 |
| Circuit-Phasing [PSSZ15] | 130 | 1 683 | 21 004 | 320 | 5 552 | 97 708 |
| Hashing + SCS [FNO18] | - | 1 537 | 21 207 | - | 3 998 | 72 140 |
| 2D CH [PSWW18] | 51 | 612 | 6 582 | 115 | 1 751 | 25 532 |
| Ours Basic §4.1 | 41 | 550 | 8 123 | 65 | 870 | 12 731 |
| Ours Advanced §5 | 35 | 604 | 10 277 | 35 | 604 | 10 277 |
| Ours No-Stash §5.2 | **9** | **149** | **2 540** | **9** | **149** | **2 540** |
| Breakdown: | | | | | | |
| OPRF | 0.3 (3%) | 5 (3%) | 72 (3%) | 0.3 (3%) | 5 (3%) | 72 (3%) |
| Sending polynomials | 0.1 (1%) | 2 (1%) | 30 (1%) | 0.1 (1%) | 2 (1%) | 30 (1%) |
| Circuit | 9 (96%) | 142 (96%) | 2 438 (96%) | 9 (96%) | 142 (96%) | 2 438 (96%) |
| Improvement factor | 5.7x | 4.1x | 2.6x | 12.8x | 11.8x | 10.1x |

We now analyze the communication complexity of our $O(n)$ protocol described in §5. The main difference compared to the basic protocol analyzed above is that a different method is used for comparing the elements of the stash, i.e., replacing step (d) above. The new method replaces this step by letting $P_2$ use Cuckoo hashing of its $n$ elements into $B$ bins and then evaluating OPRF for each of these bins. This requires $B \cdot 450$ bits of communication plus $B$ comparisons of $\gamma$ bit values. Overall, this amounts to $B \cdot (450 + (\gamma - 1) \cdot 256)$ bits of communication. For simplicity, we omit the communication for comparing the elements for phase 3 which compares the elements on the two stashes, as it is negligible.

**Comparison to Previous Work.** In Tab. 3, we compare the resulting communication of our protocols to those of previous circuit-based PSI protocols of [HEK12, PSSZ15, PSWW18, FNO18]. As can be seen from this table, our protocols improve communication by an integer factor, where the main advantage of our protocols is that their communication complexity is *independent* of the bitlength of the input elements. Namely, for arbitrary input bitlengths, our no-stash protocol improves the communication over the previous best protocol of [PSWW18] *by a factor of 12.8x for* $n = 2^{12}$ *to a factor of 10.1x for* $n = 2^{20}$. For fixed bitlength of $\sigma = 32$ bits, our no-stash protocol improves communication over [PSWW18] *by a factor of 5.7x for* $n = 2^{12}$ *to a factor of 2.6x for* $n = 2^{20}$.

**Circuit Depth.** For some secure circuit evaluation protocols like GMW [GMW87] the round complexity depends on the depth of the circuit. In Tab. 4, we depict the circuit depths for concrete parameters of our protocols and previous work, and show that our circuits have about the same low depth as the best previous works [PSSZ15, PSWW18]. In more detail, the Sort-Compare-Shuffle (SCS) circuit of [HEK12] has depth $\log_2 \sigma \cdot \log_2 n$ when using depth-optimized

**Table 4.** Circuit depth for circuit-based PSI on $n$ elements of fixed bitlength $\sigma = 32$ (left) and arbitrary bitlength hashed to $\sigma = 40 + 2\log_2(n) - 1$ bits (right).

| | $\sigma = 32$ | | | Arbitrary $\sigma$ | | |
|---|---|---|---|---|---|---|
| **Protocol** $n =$ | $\mathbf{2^{12}}$ | $\mathbf{2^{16}}$ | $\mathbf{2^{20}}$ | $\mathbf{2^{12}}$ | $\mathbf{2^{16}}$ | $\mathbf{2^{20}}$ |
| SCS [HEK12] | 60 | 80 | 100 | 72 | 98 | 126 |
| Circuit-Phasing [PSSZ15] | 5 | 5 | 5 | 6 | 7 | 7 |
| Hashing + SCS [FNO18] | - | 42 | 36 | - | 54 | 51 |
| 2D CH [PSWW18] | 5 | 5 | 5 | 6 | 7 | 7 |
| Our Protocols | 6 | 6 | 6 | 6 | 7 | 7 |

comparison circuits. The protocols of [PSSZ15, PSWW18] have depth $\log_2 \sigma$. A depth-optimized SCS circuit for the construction in [FNO18] has depth $\log_2(\sigma - \log_2(n/b)) \cdot \log_2((1+\delta)b)$, where concrete parameters for $n, \delta, b$ are given in [FNO18, Table 1]. Our protocols consist of circuits for comparing the elements on the stash of bitlength $\sigma$ and the outputs of the OPPRFs of length $\gamma$ and therefore have depth $\max(\log \sigma, \log \gamma) = \max(\log \sigma, \log_2(40 + 2\log_2(n) - 1))$.

### 7.4 Runtime Comparison

In this section we compare the runtimes of different PSI protocols. In §7.2 we conducted experiments for polynomial interpolation, the main new part of our protocol, and we show below that this step takes only a small fraction of the total runtime. We also implemented our most efficient protocol (see §5.2).[8] In addition, we estimate the runtime of our less efficient basic protocol (see §4.1) and the protocol with linear communication overhead (see §5) based on the experiments of the interpolation procedure and rigorous estimations from previous works.

**Previous Work.** As we have seen in the analysis of the communication overhead in §7.3, our protocols provide better improvements to performance in the case of arbitrary bitlengths. The previous work of [PSWW18] gave runtimes only for fixed bitlength of 32 bits in [PSWW18, Tab. 4]. Therefore, we extrapolate the runtimes of the previous protocols from fixed bitlength to arbitrary bitlength based on the circuit sizes given in [PSWW18, Tab. 3]. The estimated runtimes are given in Tab. 5. The LAN setting is a 1 Gbit/s network with round-trip time of 1 ms and the WAN setting is a 100 Mbit/s network with round-trip time of 100 ms. Runtimes were not presented in [FNO18], but since their circuit sizes and depths are substantially larger than those of [PSWW18] (cf. Tab. 3 and Tab. 4), their runtimes will also be substantially higher than those of [PSWW18].

**Our Implementation.** We implemented and benchmarked our most efficient no-stash OPPRF-based PSI protocol (see §5.2) on two commodity PCs with an Intel Core i7-4770K CPU. We instantiated our protocol with security parameter

[8] Our implementation is available at `https://github.com/encryptogroup/OPPRF-PSI`.

**Table 5.** Total run-times in ms for PSI variant protocols on $n$ elements of arbitrary bitlength using GMW [GMW87] for secure circuit evaluation and one thread. Numbers for all but our protocols are based on [PSWW18]. The best values for generic circuit-based PSI protocols are marked in bold.

| Network | LAN | | | WAN | | |
|---|---|---|---|---|---|---|
| **Protocol** **n =** | $\mathbf{2^{12}}$ | $\mathbf{2^{16}}$ | $\mathbf{2^{20}}$ | $\mathbf{2^{12}}$ | $\mathbf{2^{16}}$ | $\mathbf{2^{20}}$ |
| *Special-purpose* PSI protocols (as baseline) | | | | | | |
| DH/ECC PSI [Sha80, Mea86, DGT12] | 3 296 | 49 010 | 7 904 054 | 4 082 | 51 866 | 8 008 771 |
| BaRK-OPRF [KKRT16] | 113 | 295 | 3 882 | 540 | 1 247 | 14 604 |
| *Generic* circuit-based PSI protocols | | | | | | |
| Circuit-Phasing [PSSZ15] | 7 825 | 67 292 | 1 126 848 | 37 380 | 327 976 | 4 850 571 |
| 2D CH [PSWW18] | 5 031 | 25 960 | 336 134 | 22 796 | 129 436 | 1 512 505 |
| Ours Basic §4.1 (estimated) | 2 908 | 13 767 | 182 204 | 12 934 | 63 861 | 752 695 |
| Ours Advanced §5 (estimated) | 1 674 | 9 763 | 148 436 | 7 372 | 43 675 | 597 885 |
| Ours No-Stash §5.2, Total | **1 199** | **8 486** | **120 731** | **5 910** | **22 134** | **261 481** |
| Breakdown: | | | | | | |
| OPRF | 724 (60%) | 1 097 (13%) | 5 844 (5%) | 2 867 (49%) | 4 164 (19%) | 26 121 (10%) |
| Polynomial interpolation | 183 (15%) | 2 809 (33%) | 45 335 (38%) | 183 (3%) | 2 809 (13%) | 45 335 (17%) |
| Polynomial transmission | 16 (1%) | 145 (2%) | 667 (0%) | 816 (13%) | 1 079 (5%) | 4 012 (2%) |
| Polynomial evaluation | 58 (5%) | 1 344 (16%) | 21 768 (18%) | 58 (1%) | 1 344 (6%) | 21 768 (8%) |
| Circuit | 218 (18%) | 3 091 (36%) | 47 117 (39%) | 1 986 (34%) | 12 738 (57%) | 164 245 (63%) |
| Improvement over [PSWW18] | 4.2x | 3.1x | 2.8x | 3.9x | 5.8x | 5.8x |

$\kappa = 128$ bits, $K = 3$ hash functions, $B = 1.27n$ bins, and no stash (see §5.2). Our OPPRF implementation is based on the OPRF protocol of [PSZ18].[9] For the secure circuit evaluation, we used the ABY framework [DSZ15]. The run-times are averaged over 50 executions. The results are described in Tab. 5.

**Comparison with PSI Protocols.** As a baseline, we compare our performance with specific protocols for computing the intersection itself. (However, as is detailed in §1.2, our protocol is circuit-based and therefore has multiple advantages compared to specific PSI protocols.) Our best protocol is slower by a factor of 41x than today's fastest PSI protocol of [KKRT16] for $n = 2^{20}$ elements in the WAN setting (cf. Tab. 5).

**Comparison with Public Key-based PSI Variant Protocols.** Our circuit-based protocol is substantially faster than previous public key-based protocols for computing variants of PSI, although they have similar asymptotic linear complexity. As an example, consider comparing whether the size of the intersection is greater than a threshold (PSI-CAT). In our protocol, we can compute the PSI-CAT functionality by extending the PSI circuit of Tab. 5 with a Hamming distance circuit (which, using the size-optimal construction of [BP06], adds less than $n$ AND gates). The final comparison with the threshold adds another $\log_2 n$ AND gates [BPP00] which are negligible as well. For the PSI-CAT functionality, [ZC17] report runtimes of 779 seconds for $n = 2^{11}$ elements, [HOS17] report runtimes of 728 seconds for $n = 2^{11}$ elements, and [ZC18] report runtimes of at least 138 seconds for $n = 100$ elements, whereas our protocol requires 0.52

[9] This OPRF protocol has communication that is higher by 10% to 15% than the communication of the OPRF protocol of [KKRT16]. But since OPRF requires less than 3 % of the total communication, this additional cost is negligible in our protocol.

seconds for $n = 2^{11}$ elements and 0.34 seconds for $n = 100$ elements. Hence, we improve over [ZC17] by a factor of 1 498x, over [HOS17] by a factor of 1 400x, and over [ZC18] by a factor of 405x. As an example for computing PSI-CAT with larger set sizes, our protocol requires 124 seconds for $n = 2^{20}$ elements.

The protocol described by Google for computing ad revenues [Yun15, Kre17] (see §1.2) is based on the DH-based PSI protocol which is already 65x slower than our protocol for $n = 2^{20}$ elements over a LAN (cf. Tab. 5) and leaks the intersection cardinality as an intermediate result. Here, too, our circuit would be only slightly larger than the PSI circuit of Tab. 5.

**Comparison with Circuit-based PSI Protocols.** As can be seen from Tab. 5, our no-stash protocol from §5.2 is substantially more efficient than our basic protocol and our linear asymptotic overhead protocol from §4.1 and §5, respectively. It improves over the best previous circuit-based PSI protocol from [PSWW18] by factors of 4.2x to 2.8x in the LAN setting, and by factors of 5.8x to 3.9x in the WAN setting. From the micro-benchmarks in Tab. 5, we also observe that the runtimes for the polynomial interpolation are a significant fraction of the total runtimes of our protocols (3% to 33% for the interpolation and 1% to 18% for the evaluation). Since polynomials are independent of each other, the interpolation and evaluation can be trivially parallelized for running with multiple threads, which would give this part of the computation a speed-up that is linear in the number of physical cores of the processor.

**Acknowledgements.** We thank Ben Riva and Udi Wieder for valuable discussions about this work. This work has been co-funded by the DFG within project E4 of the CRC CROSSING and by the BMBF and the HMWK within CRISP, by the BIU Center for Research in Applied Cryptography and Cyber Security in conjunction with the Israel National Cyber Bureau in the Prime Minister's Office, and by a grant from the Israel Science Foundation.

## References

ADN+13. N. Asokan, A. Dmitrienko, M. Nagy, E. Reshetova, A.-R. Sadeghi, T. Schneider, and S. Stelle. CrowdShare: Secure mobile resource sharing. In *ACNS*, 2013.

ALSZ13. G. Asharov, Y. Lindell, T. Schneider, and M. Zohner. More efficient oblivious transfer and extensions for faster secure computation. In *CCS*, 2013.

BP06. J. Boyar and R. Peralta. Concrete multiplicative complexity of symmetric functions. In *MFCS*, 2006.

BPP00. J. Boyar, R. Peralta, and D. Pochuev. On the multiplicative complexity of Boolean functions over the basis $(\wedge, \oplus, 1)$. *TCS*, (1), 2000.

CADT14. H. Carter, C. Amrutkar, I. Dacosta, and P. Traynor. For your phone only: custom protocols for efficient secure function evaluation on mobile devices. *Security and Communication Networks*, 7(7), 2014.

CO18. M. Ciampi and C. Orlandi. Combining private set-intersection with secure two-party computation. In *SCN*, 2018.

DC17. A. Davidson and C. Cid. An efficient toolkit for computing private set operations. In *ACISP*, 2017.

DD15. S. K. Debnath and R. Dutta. Secure and efficient private set intersection cardinality using Bloom filter. In *ISC*, 2015.

DGT12. E. De Cristofaro, P. Gasti, and G. Tsudik. Fast and private computation of cardinality of set intersection and union. In *CANS*, 2012.

DKT10. E. De Cristofaro, J. Kim, and G. Tsudik. Linear-complexity private set intersection protocols secure in malicious model. In *ASIACRYPT*, 2010.

DSMRY09. D. Dachman-Soled, T. Malkin, M. Raykova, and M. Yung. Efficient robust private set intersection. In *ACNS*, 2009.

DSZ15. D. Demmler, T. Schneider, and M. Zohner. ABY – a framework for efficient mixed-protocol secure two-party computation. In *NDSS*, 2015.

DT10. E. De Cristofaro and G. Tsudik. Practical private set intersection protocols with linear complexity. In *FC*, 2010.

Dwo06. C. Dwork. Differential privacy. In *ICALP*, 2006.

EFLL12. Y. Ejgenberg, M. Farbstein, M. Levy, and Y. Lindell. SCAPI: The secure computation application programming interface. Cryptology ePrint Archive, Report 2012/629, 2012.

FHNP16. M. J. Freedman, C. Hazay, K. Nissim, and B. Pinkas. Efficient set intersection with simulation-based security. *Journal of Cryptology*, 29(1), 2016.

FIPR05. M. J. Freedman, Y. Ishai, B. Pinkas, and O. Reingold. Keyword search and oblivious pseudorandom functions. In *TCC*, 2005.

FNO18. B. H. Falk, D. Noble, and R. Ostrovsky. Private set intersection with linear communication from general assumptions. Cryptology ePrint Archive, Report 2018/238, 2018.

FNP04. M. J. Freedman, K. Nissim, and B. Pinkas. Efficient private matching and set intersection. In *EUROCRYPT*, 2004.

GM11. M.T. Goodrich and M. Mitzenmacher. Privacy-preserving access of outsourced data via oblivious ram simulation. In *ICALP*, 2011.

GMW87. O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game or a completeness theorem for protocols with honest majority. In *STOC*, 1987.

Gon81. G. H. Gonnet. Expected length of the longest probe sequence in hash code searching. *Journal of the ACM*, 28(2), 1981.

HCE11. Y. Huang, P. Chapman, and D. Evans. Privacy-preserving applications on smartphones. In *Hot topics in Security (HotSec)*, 2011.

HEK12. Y. Huang, D. Evans, and J. Katz. Private set intersection: Are garbled circuits better than custom protocols? In *NDSS*, 2012.

HEKM11. Y. Huang, D. Evans, J. Katz, and L. Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security*, 2011.

HN10. C. Hazay and K. Nissim. Efficient set operations in the presence of malicious adversaries. In *PKC*, 2010.

HOS17. P. Hallgren, C. Orlandi, and A. Sabelfeld. PrivatePool: Privacy-preserving ridesharing. In *Computer Security Foundations Symposium (CSF)*, 2017.

IKN+17. M. Ion, B. Kreuter, E. Nergiz, S. Patel, S. Saxena, K. Seth, D. Shanahan, and M. Yung. Private intersection-sum protocol with applications to attributing aggregate ad conversions. Cryptology ePrint Archive, Report 2017/738, 2017.

IKNP03. Y. Ishai, J. Kilian, K. Nissim, and E. Petrank. Extending oblivious transfers efficiently. In *CRYPTO*, 2003.

KKRT16. V. Kolesnikov, R. Kumaresan, M. Rosulek, and N. Trieu. Efficient batched oblivious PRF with applications to private set intersection. In *CCS*, 2016.
KLS+17. Á. Kiss, J. Liu, T. Schneider, N. Asokan, and B. Pinkas. Private set intersection for unequal set sizes with mobile applications. *PoPETs*, 2017(4), 2017.
KM18. E. Kushilevitz and T. Mour. Sub-logarithmic distributed oblivious RAM with small block size. *CoRR*, abs/1802.05145, 2018.
KMP+17. V. Kolesnikov, N. Matania, B. Pinkas, M. Rosulek, and N. Trieu. Practical multi-party private set intersection from symmetric-key techniques. In *CCS*, 2017.
KMW09. A. Kirsch, M. Mitzenmacher, and U. Wieder. More robust hashing: Cuckoo hashing with a stash. *SIAM Journal on Computing*, 39(4), 2009.
Kre17. B. Kreuter. Secure multiparty computation at Google. In *RWC*, 2017.
Kre18. Benjamin Kreuter. *Techniques for Scalable Secure Computation Systems*. PhD thesis, Northeastern University, 2018.
KS08. V. Kolesnikov and T. Schneider. Improved garbled circuit: Free XOR gates and applications. In *ICALP*, 2008.
LWN+15. C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi. ObliVM: A programming framework for secure computation. In *S&P*, 2015.
Mea86. C. Meadows. A more efficient cryptographic matchmaking protocol for use in the absence of a continuously available third party. In *S&P*, 1986.
MR95. R. Motwani and P. Raghavan. *Randomized algorithms*. 1995.
PR01. R. Pagh and F. F. Rodler. Cuckoo hashing. In *European Symposium on Algorithms (ESA)*, 2001.
PSSZ15. B. Pinkas, T. Schneider, G. Segev, and M. Zohner. Phasing: Private set intersection using permutation-based hashing. In *USENIX Security*, 2015.
PSWW18. B. Pinkas, T. Schneider, C. Weinert, and U. Wieder. Efficient circuit-based PSI via Cuckoo hashing. In *EUROCRYPT*, 2018.
PSZ14. B. Pinkas, T. Schneider, and M. Zohner. Faster private set intersection based on OT extension. In *USENIX Security*, 2014.
PSZ18. B. Pinkas, T. Schneider, and M. Zohner. Scalable private set intersection based on OT extension. *TOPS*, 21(2), 2018.
RA18. A. C. D. Resende and D. F. Aranha. Faster unbalanced private set intersection. In *FC*, 2018.
RR17a. P. Rindal and M. Rosulek. Improved private set intersection against malicious adversaries. In *EUROCRYPT*, 2017.
RR17b. P. Rindal and M. Rosulek. Malicious-secure private set intersection via dual execution. In *CCS*, 2017.
Sha80. A. Shamir. On the power of commutativity in cryptography. In *ICALP*, 1980.
SZ13. T. Schneider and M. Zohner. GMW vs. Yao? Efficient secure two-party computation with low depth circuits. In *FC*, 2013.
Yao86. A. C. Yao. How to generate and exchange secrets. In *FOCS*, 1986.
Yun15. M. Yung. From mental poker to core business: Why and how to deploy secure computation protocols? In *CCS*, 2015.
ZC17. Y. Zhao and S. S. M. Chow. Are you the one to share? Secret transfer with access structure. *PoPETs*, 2017(1), 2017.
ZC18. Y. Zhao and S. S. M. Chow. Can you find the one for me? Privacy-preserving matchmaking via threshold PSI. In *WPES*, 2018.
ZRE15. S. Zahur, M. Rosulek, and D. Evans. Two halves make a whole: Reducing data transfer in garbled circuits using half gates. In *EUROCRYPT*, 2015.