Efficient Circuit-Based PSI via Cuckoo Hashing

Benny Pinkas[1](✉), Thomas Schneider[2], Christian Weinert[2], and Udi Wieder[3]

[1] Bar-Ilan University, Ramat Gan, Israel
benny@pinkas.net
[2] TU Darmstadt, Darmstadt, Germany
{thomas.schneider,christian.weinert}@crisp-da.de
[3] VMware Research, Palo Alto, USA
udi.wieder@gmail.com

Abstract. While there has been a lot of progress in designing efficient custom protocols for computing Private Set Intersection (PSI), there has been less research on using generic Multi-Party Computation (MPC) protocols for this task. However, there are many variants of the set intersection functionality that are not addressed by the existing custom PSI solutions and are easy to compute with generic MPC protocols (e.g., comparing the cardinality of the intersection with a threshold or measuring ad conversion rates).

Generic PSI protocols work over circuits that compute the intersection. For sets of size n, the best known circuit constructions conduct $O(n \log n)$ or $O(n \log n / \log \log n)$ comparisons (Huang et al., NDSS'12 and Pinkas et al., USENIX Security'15). In this work, we propose new circuit-based protocols for computing *variants of the intersection* with an almost linear number of comparisons. Our constructions are based on new variants of Cuckoo hashing in two dimensions.

We present an asymptotically efficient protocol as well as a protocol with better concrete efficiency. For the latter protocol, we determine the required sizes of tables and circuits experimentally, and show that the run-time is concretely better than that of existing constructions.

The protocol can be extended to a larger number of parties. The proof technique presented in the full version for analyzing Cuckoo hashing in two dimensions is new and can be generalized to analyzing standard Cuckoo hashing as well as other new variants of it.

Keywords: Private set intersection · Secure computation

1 Introduction

Private Set Intersection (PSI) refers to a protocol which enables two parties, holding 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. The PSI functionality is useful for applications where parties need to apply a JOIN operation to private datasets. There are multiple constructions of secure

© International Association for Cryptologic Research 2018
J. B. Nielsen and V. Rijmen (Eds.): EUROCRYPT 2018, LNCS 10822, pp. 125–157, 2018.
https://doi.org/10.1007/978-3-319-78372-7_5

protocols for computing PSI, but there is an advantage for computing PSI by applying a generic Multi-Party Computation (MPC) protocol to a circuit computing the intersection (see Sect. 1.1). The problem is that a naive circuit computes $O(n^2)$ comparisons, and even the most recent circuit-based constructions require $O(n \log n)$ or $O(n \log n / \log\log n)$ comparisons (see Sect. 1.4).

In this work, we present a new circuit-based protocol for computing PSI variants. In our protocol, each party first inserts its input elements into bins according to a new hashing algorithm, and then the intersection is computed by securely computing a Boolean comparison circuit over the bins. The insertion of the items is based on new Cuckoo hashing variants which guarantee that if the two parties have the same input value, then there is exactly one bin to which both parties map this value. Furthermore, the total number of bins is $O(n)$ and there are $O(1)$ items mapped to each bin, plus $\omega(1)$ items which are mapped to a special stash. Hence, the circuit that compares (1) for each bin, the items that the two parties mapped to it, and (2) all stash items to all items of the other party, computes only $\omega(n)$ comparisons.

1.1 Motivation for Circuit-Based PSI

PSI has many applications, as is detailed for example in [42]. Consequently, there has been a lot of research on efficient secure computation of PSI, as we describe in Sect. 1.4. However, most research was focused on computing the intersection itself, while there are interesting applications for the ability to securely compute arbitrary functions of the intersection. We demonstrate the need for efficient computation of PSI using generic protocols through the following arguments:

Adaptability. Assume that you are a cryptographer and were asked to propose and implement a protocol for computing PSI. One approach is to use a specialized protocol for computing PSI. Another possible approach is to use a protocol for generic secure computation, and apply it to a circuit that computes PSI. A trivial circuit performs $O(n^2)$ comparisons, while more efficient circuits, described in [26, 39], perform only $O(n \log n)$ or $O(n \log n / \log\log n)$ comparisons, respectively. The most efficient specialized PSI protocols are faster by about two orders of magnitude than circuit-based constructions (see [39]), and therefore you will probably choose to use a specialized PSI protocol. However, what happens if you are later asked to change the protocol to compute another function of the intersection? For example, output only the size of the intersection, or output 1 iff the size is greater than some threshold, or output the most "representative" item that occurs in the intersection (according to some metric). 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 that computes a different function of the intersection is rather trivial, and can even be performed by undergrad students.

Consider the following function as an example of a variant of the PSI functionality for which we do not know a specialized protocol: Suppose that you want to compute the size of the intersection, but you also wish to preserve the privacy of users by ensuring differential privacy. This is done by adding some noise to the exact count before releasing it. This functionality can easily be computed by a circuit, but it is unclear how to compute it using other PSI protocols. (See [38] for constructions that add noise to the results of MPC computation in order to ensure differential privacy.)

Existing code base. Circuit-based protocols benefit from all the work that was invested in recent years in designing, implementing, and optimizing very efficient systems for generic secure computation. Users can download existing secure computation software, e.g., [13,27], and only need to design the circuit to be computed and implement the appropriate hashing technique.

Existing applications. There are existing applications that need to compute functions over the results of the set intersection. For example, Google reported [34,49] a PSI-based application for measuring ad conversion rates, namely the revenues from ad viewers who later perform a related transaction. This computation can be done by comparing the list of people who have seen an ad with those who have completed a transaction. These lists are held by the advertiser (say, Google or Facebook), and by merchants, respectively. A simple (non-private) solution is for one side to disclose its list of customers to the other side, which computes the necessary statistics. Another option is to run a secure computation over the results of the set intersection. For example, 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) [28]. This protocol reveals the identities of the items in the intersection, and seems less efficient than our protocol as it uses public key operations, rather than efficient symmetric cryptographic operations.[1]

1.2 Our Contributions

This work provides the following contributions:

Circuit-based PSI protocols with almost linear overhead. We show a new circuit-based construction for computing any symmetric function on top of PSI, with an asymptotic overhead of only $\omega(n)$ comparisons. (More accurately, for any function $f \in \omega(n)$, the overhead of the construction is $o(f(n))$.) This construction is based on standard Cuckoo hashing.

[1] Facebook is running a computation of this type with companies that have transaction records for a large part of loyalty card holders in the US. According to the report in https://www.eff.org/deeplinks/2012/09/deep-dive-facebook-and-datalogix-whats-actually-getting-shared-and-how-you-can-opt, the computation is done using an insecure PSI variant based on creating pseudonyms using naive hashing of the items.

Small constants. Standard measures of asymptotic security are not always a good reflection of the actual performance on reasonable parameters. Therefore, in addition to the asymptotic improvement, we also show a concrete circuit-based PSI construction. This construction is based on a new variant of Cuckoo hashing, *two-dimensional Cuckoo hashing*, that we introduce in this work. We carefully handle implementation issues to improve the actual overhead of our protocols, and make sure that all constants are small. In particular, we ran extensive experiments to analyze the failure probabilities of the hashing scheme, and find the exact parameters that reduce this statistical failure probability to an acceptable level (e.g., 2^{-40}). Our analysis of the concrete complexities is backed by extensive experiments, which consumed about 5.5 million core hours on the Lichtenberg high performance computer of the TU Darmstadt and were used to set the parameters of the hashing scheme. Given these parameters we implemented the circuit-based PSI protocol and tested it.

Implementation and experiments. We implemented our protocols using the ABY framework for secure two-party computation [13]. Our experiments show that our protocols are considerably faster than the previously best circuit-based constructions. For example, for input sets of $n = 2^{20}$ elements of arbitrary bitlength, we improve the circuit size over the best previous construction by up to a factor of 3.8x.

New Cuckoo hashing analysis. Our two-dimensional Cuckoo hashing is based on a new Cuckoo hashing scheme that employs two tables and each item is mapped to either *two* locations in the first table, or *two* locations in the second table. This is a new Cuckoo hashing variant that has not been analyzed before. In addition to measuring its performance using simulations, we provide a probabilistic analysis of its performance. Interestingly, this analysis can also be used as a new proof technique for the success probability of standard Cuckoo hashing.

1.3 Computing Symmetric Functions

A trivial circuit for PSI that performs $O(n^2)$ comparisons between all pairs of the input items of the two parties allows the parties to set their inputs in any arbitrary order. On the other hand, there exist more efficient circuit-based PSI constructions where each party first independently orders its inputs according to some predefined algorithm: the sorting network-based construction of [26] requires each party to sort its input to the circuit, while the hashing-based construction of [39] requires the parties to map their inputs to bins using some public hash functions. (These constructions are described in Sect. 1.4.) The location of each input item thus depends on the identity of the other inputs of the input owner, and must therefore be kept hidden from the other party.

In this work, we focus on constructing a circuit that computes the intersection. The outputs of this circuit can be the items in the intersection, or some functions of the items in the intersection: say, a "1" for each intersecting item, or an arbitrary function of some data associated with the item (for example, if the items are transactions, we might want to output a financial value associated with each transaction that appears in the intersection). On top of that circuit it is possible to add circuits for computing any function that is based on the intersection. In order to preserve privacy, the output of 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. There are many examples of interesting symmetric functions of the intersection. (In fact, it is hard to come up with examples for interesting non-symmetric functions of the intersection, except for the intersection itself.) Examples of symmetric functions include:

- Computing the size of the intersection, i.e., PSI cardinality (PSI-CA).
- Computing a threshold function that is based on the size of the intersection. For example, outputting "1" if the size of the intersection is greater than some threshold (PSI-CAT), or outputting a rounded value of the percentage of items that are in the intersection. 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 [23].
- Computing the size of the intersection while preserving the privacy of users by ensuring differential privacy [17]. This can be done by adding some noise to the exact count.
- Computing the sum of values associated with the items in the intersection. This is used for measuring ad-generated revenue (cf. Sect. 1.1). Similarly, there could be settings where each party associates a value with each transaction, and the output is the sum of the differences between these assigned values in the intersection, or the sum of the squares of the differences, etc.

The circuits for computing all these functions are of size $O(n)$. Therefore, with our new construction, the total size of the circuits for computing these functions is $\omega(n)$, whereas circuit-based PSI protocols [26,39] had size $O(n \log n)$.

If one wishes to compute a function that is not symmetric, or wishes to output the intersection itself, then the circuit must first shuffle the values in the intersection (in order to assign a random location to each item in the intersection) and then compute the function over the shuffled values, or output the shuffled intersection. A circuit for this "shuffle" step has size $O(n \log n)$, as described in [26]. (It is unclear, though, why a circuit-based protocol should be used for computing the intersection, since this job can be done much more efficiently by specialized protocols, e.g., [31,42].)

1.4 Related Work

PSI. Work on protocols for private set intersection was presented as early as [35,46], which introduced public key-based protocols using commutative cryptography, namely the Diffie-Hellman function. A survey of PSI protocols appears in [41]. The goal of these protocols is to let one party learn the intersection itself, rather than to enable the secure computation of arbitrary functions of the intersection. Other PSI protocols are based on oblivious polynomial evaluation [20], blind RSA [11], and Bloom filters [16]. Today's most efficient PSI protocols are based on hashing the items to bins and then evaluating an oblivious pseudorandom function per bin, which is implemented using oblivious transfer (OT) extension. These protocols have linear complexity and were all implemented and evaluated, see, e.g., [31,39,41,42]. In cases where communication cost is a crucial and computation cost is a minor factor, recent solutions based on fully homomorphic encryption represent an interesting alternative [6]. PSI protocols have also been adapted to the special requirements of mobile devices [4,25,30].

Circuit-based PSI. Circuit-based PSI protocols compute the set intersection functionality by running a secure evaluation of a Boolean circuit. These protocols can easily be adapted to compute different variants of the PSI functionality. The straightforward solution to the PSI problem requires $O(n^2)$ comparisons – one comparison for each pair of items belonging to the two parties. Huang et al. [26] designed a circuit for computing PSI based on sorting networks, which computes $O(n \log n)$ comparisons and is of size $O(\sigma n \log n)$, where σ is the bitlength of the inputs. A different circuit, based on the usage of Cuckoo hashing by one party and simple hashing by the other party, was proposed in [39]. The size of that circuit is $O(\sigma n \log n / \log \log n)$. In our work we propose efficient circuits for PSI variants with an asymptotic size of $\omega(\sigma n)$ and better concrete efficiency. We give more details and a comparison of the concrete complexities of circuit-based PSI protocols in Sect. 6.2.

PSI Cardinality (PSI-CA). A specific interesting function of the intersection is its cardinality, namely $|X \cap Y|$, and is referred to as PSI-CA. There are several protocols for computing PSI-CA with linear complexity based on public key cryptography, e.g., [9] which is based on Diffie-Hellman and is essentially a variant of the DH-based PSI protocol of [35,46] (see also references given therein for other less efficient public key-based protocols); or [12] which is based on Bloom filters and the public key cryptosystem of Goldwasser-Micali. In these protocols, one of the parties learns the cardinality. As we show in our experiments in Sect. 6.3, these protocols are slower than our constructions already for relatively small set sizes ($n = 2^{12}$) in the LAN setting and for large set sizes ($n = 2^{20}$) in the WAN setting, since they are based on public key cryptography. An advantage of these protocols is that they achieve the lowest amount of communication, but it seems hard to extend them to compute arbitrary functions of the intersection. Protocols for private set intersection and union and their cardinalities with linear complexity are given in [8]. They use Bloom filters and computationally expensive additively homomorphic encryption, whereas our protocols can flexibly be adapted to different variants and are based on efficient symmetric key cryptography.

2 Preliminaries

Setting. We consider two parties, which we denote as Alice and Bob. They have input sets, X and Y, respectively, which are each of size n and each item has bitlength σ. We assume that both parties agree on a symmetric function f and would like to securely compute $f(X \cap Y)$. They also agree on a circuit that receives the items in the intersection as input and computes f.

Security Model. The secure computation literature considers *semi-honest* adversaries, which try to learn as much information as possible from a given protocol execution, but are not able to deviate from the protocol steps, and *malicious* adversaries, which are able to deviate arbitrarily from the protocol. 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, either by stealing it or by legally enforcing its disclosure. Most protocols for private set intersection, as well as this work, focus on solutions that are secure against semi-honest adversaries. PSI protocols for the malicious setting exist, but they are less efficient than protocols for the semi-honest setting (see, e.g., [7,10,19,20,43,44]).

Secure Computation. There are two main approaches for generic secure two-party computation with security against semi-honest adversaries that allow to securely evaluate a function that is represented as a Boolean circuit: (1) Yao's garbled circuit protocol [48] has a constant round complexity and with today's most efficient optimizations provides free XOR gates [33], whereas securely evaluating an AND gate requires sending two ciphertexts [50]. (2) The GMW protocol [21] also provides free XOR gates and requires two ciphertexts of communication per AND gate using OT extension [3]. 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 pre-computes multiplication triples for each AND gate using OT extension in a constant number of rounds (cf. [3]). The online phase runs from the time the inputs are provided until the result is obtained and involves sending one message for each layer of AND gates. A detailed description and a comparison between Yao and GMW is given in [45].

Cuckoo Hashing. In its simplest form, Cuckoo hashing [36] uses two hash functions h_0, h_1 to map n elements to two tables T_0, T_1, each containing $(1 + \varepsilon)n$ bins. 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 performed. In the latter case, the last element is mapped to a special stash. It was shown in [29] that, for any constant s, the probability that the size of the stash is greater than s is at most $O(n^{-(s+1)})$. After inserting all items, each item can be found in one of two locations or in the stash. A lookup therefore requires checking only $O(1)$ locations.

Many variants of Cuckoo hashing were suggested and analyzed. See [47] for a thorough discussion and analysis of different Cuckoo hashing schemes. A variant of Cuckoo hashing that is similar to our constructions was given in [1], although in a different application domain. It considers a setting with three tables, where an item must be placed in two out of three tables. The analysis of this construction uses a different proof technique than the one we present in the full version [40], and we have not attempted to generalize their proof to a general number of item insertions (as we do for our construction). Furthermore, there is no tight analysis of the stash size in [1]. The work in [18] builds on the construction of [1] and proves that the failure probability when using a stash of size s behaves as $\tilde{O}(n^{-s})$. However, the experiments of [18, Fig. 6] reveal that the size of the stash is rather large and actually *increasing* in n within the range of 1 000 to 100 000 elements. For example, for table size $7.1n$, a stash of at least size 4 is required for inserting 10 000 elements, whereas a stash of at least size 11 is required for inserting 100 000 elements. Since each item in the stash must be compared to all items of the other party, and since these comparisons cannot use a shorter representation based on permutation-based hashing, the effect of the stash is substantial, and in the context of circuit-based PSI it is therefore preferable to use constructions that place very few or no items in the stash.

PSI based on Hashing. Some existing constructions of circuits for PSI require the parties to reorder their inputs before inputting them to the circuit: The sorting-network based construction of [26] requires the parties to sort their inputs. The hashing based construction of [39] requires that each party maps its items to bins using a hash function. It was observed as early as [20] that if the two parties agree on the same hash function and use it to map their respective input to bins, then the items that one party maps to a specific bin need to be compared only 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 they mapped to each bin, since this data leaks information about their other items. Therefore, they 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 m is the maximum number of items mapped to any of the β bins.

When a random hash function h is used to map n items to n bins, where x is mapped to bin $h(x)$, the most occupied bin has w.h.p. $m = \frac{\ln n}{\ln \ln n}(1 + o(1))$ items [22] (a careful analysis shows, e.g., that, for $n = 2^{20}$ and an error probability of 2^{-40}, one needs to set $m = 20$). Cuckoo hashing is much more promising, since it maps n items to $2(1+\varepsilon)n$ bins, where each bin stores at most

$m = 1$ items. Cuckoo hashing typically uses two hash functions h_0, h_1, where an item x is mapped to one of the two locations $h_0(x), h_1(x)$, or to a stash of a small size. It is tempting to let both parties, Alice and Bob, map their items to bins using Cuckoo hashing, and then only compare the item that one party maps to a bin with the item that the other party maps to the same bin. The problem is that Alice might map x to $h_0(x)$ whereas Bob might map it to $h_1(x)$. They cannot use a protocol where Alice's value in bin $h_0(x)$ is compared to the two bins $h_0(x), h_1(x)$ in Bob's input, since this reveals that Alice has an item that is mapped to these two locations. The solution used in [19,39,41] is to let Alice map her items to bins using Cuckoo hashing, and Bob map his items using simple hashing. Namely, each item of Bob is mapped to both bins $h_0(x), h_1(x)$. Therefore, Bob needs to pad his bins to have $m = O(\log n / \log\log n)$ items in each bin, and the total number of comparisons is $O(n \log n / \log\log n)$.

3 Analyzing the Failure Probability

Efficient cryptographic protocols that are based on probabilistic constructions are typically secure as long as the underlying probabilistic constructions do not fail. Our work is based on variants of Cuckoo hashing, and the protocols are secure as long as the relevant tables and stashes do not overflow. (Specifically, hashing is computed using random hash functions which are chosen independently of the data. If a party observes that these functions cannot successfully hash its data, it can indeed ask to replace the hash functions, or remove some items from its input. However, the hash functions are then no longer independent of this party's input and might therefore leak some information about the input.)

There are two approaches for arguing about the failure probability of cryptographic protocols:

1. For an **asymptotic analysis**, the failure probability must be negligible in n.
2. For a **concrete analysis**, the failure probability is set to be smaller than some threshold, say $2^{-\lambda}$, where λ is a statistical security parameter.
   In typical experiments, the statistical security parameter is set to $\lambda = 40$. This means that "unfortunate" events that leak information happen with a probability of at most 2^{-40}. In particular, $\lambda = 40$ was used in all PSI constructions which are based on hashing (e.g., [16,19,31,39,41]).

With regards to the probabilistic constructions, there are different levels of analysis of the failure probability:

1. For simple constructions, it is sometimes possible to compute the **exact failure probability**. (For example, suppose that items are hashed to a table using a random hash function, and a failure happens when two items are mapped to the same location. In this case it is trivial to compute the exact failure probability.)

2. For some constructions there are known **asymptotic bounds** for the failure probability, but no concrete expressions. (For example, for Cuckoo hashing with a stash of size s, it was shown in [29] that the overflow probability is $O(n^{-(s+1)})$, but the exact constants are unknown.)[2]
3. For other constructions there is no analysis for the failure probability, even though they **perform very well in practice**. For example, Cuckoo hashing variants where items can be mapped to $d > 2$ locations, or where each bin can hold $k > 1$ items, were known to have better space utilization than standard Cuckoo hashing, but it took several years to theoretically analyze their performance [47]. There are also insertion algorithms for these Cuckoo hashing variants which are known to perform well but which have not yet been fully analyzed.

3.1 Using Probabilistic Constructions for Cryptography

Suppose that one is using a probabilistic construction (e.g., a hash table) in the design of a cryptographic protocol. An asymptotic analysis of the cryptographic protocol can be done if the hash table has either an exact analysis or an asymptotic analysis of its failure probability (items 1 and 2 in the previous list).

If the aim is a concrete analysis of the cryptographic protocol, then exact values for the parameters of the hash construction must be identified. If an exact analysis is known (item 1), then it is easy to plug in the desired failure probability ($2^{-\lambda}$) and compute the values for the different parameters. However, if only an asymptotic analysis or experimental evidence is known (items 2 and 3), then experiments must be run in order to find the parameters that set the failure probability to be smaller than $2^{-\lambda}$.

We stress that a concrete analysis is needed whenever a cryptographic protocol is to be used in practice. In that case, even an asymptotic analysis is insufficient since it does not specify any constants, which are crucial for deriving the exact parameter values.

3.2 Experimental Parameter Analysis

Verifying that the failure probability is smaller than $2^{-\lambda}$ for $\lambda = 40$ requires running many repetitions of the experiments. Furthermore, for large input sizes (large values of n), each single run of the experiment can be rather lengthy. (And one could justifiably argue that the more interesting results are for the larger values of n, since for smaller n we can use less optimal constructions and still get reasonable performance.)

[2] We note though that many probabilistic constructions are analyzed in the algorithms research literature to have a failure probability of $o(1)$, which is fine for many applications, but is typically insufficient for cryptographic applications.

Examining the failure probability for a specific choice of parameters. For a specific choice of parameters, running 2^λ repetitions of an experiment is insufficient to argue about a $2^{-\lambda}$ failure probability, since it might happen that the experiments were very unlucky and resulted in no failure even though the failure probability is somewhat larger than $2^{-\lambda}$. Instead, we can argue about a confidence interval: namely, a confidence interval of $1 - \alpha$ (say, 95%, or 99.9%) states that if the failure probability is greater than $2^{-\lambda}$, then we would have *not* seen the results of the experiment, except with a probability that is smaller than α. Therefore, either the experiment was very unlucky, or the failure probability is sufficiently small. For example, an easy to remember confidence level used in statistics is the "rule of three", which states that if an event has not occurred in $3 \cdot s$ experiments, then the 95% confidence interval for its rate of occurrence in the population is $[0, 1/s]$. For our purposes this means that running $3 \cdot 2^\lambda$ experiments with no failure suffices to state that the failure probability is smaller than $2^{-\lambda}$ with 95% confidence. (We will report experiments in Sect. 6.1 which result in a 99.9% confidence interval for the failure probability.)

Examining the failure probability as a function of *n*. For large values of n (e.g., $n = 2^{20}$), it might be too costly to run sufficiently many (more than 2^{40}) experiments. Suppose that the experiments spend just 10 cycles on each item. This is an extremely small lower bound, which is probably optimistic by orders of magnitude compared to the actual run-time. Then the experiments take at least $10 \cdot 2^{60}$ cycles. This translates to about a million core hours on 3 GHz machines.

In order to be able to argue about the failure probability for large values of n, we can run experiments for progressively increasing values of n and identify how the failure probability behaves as a function of n. If we observe that the failure probability is decreasing, or, better still, identify the dependence on n, we can argue, given experimental results for medium-sized n values, about the failure probabilities for larger values of n.

3.3 Our Constructions

Asymptotic overhead. We present in Sect. 4 a construction of circuit-based PSI that we denote as the "mirror" construction. This construction uses four instances of standard Cuckoo hashing and therefore we know that a stash of size s guarantees a failure probability of $O(n^{-(s+1)})$ [29]. (Actually, the previously known analysis was only stated for $s = O(1)$. We show in the full version [40] that this failure probability also holds for s that is not constant.)

The bound on the failure probability implies that for any constant security parameter λ, a stash of constant size is sufficient to ensure that the failure probability is smaller than $2^{-\lambda}$ for sufficiently large n. In order to achieve a failure probability that is negligible in n, we can set the stash size s to be slightly larger than $O(1)$, e.g., $s = \log\log n$, $s = \log^* n$, or any $s = \omega(1)$. The result is a construction with an overhead of $\omega(n)$. (More accurately, the overhead is as close as desired to being linear: for any $f(n) \in \omega(n)$, the overhead is $o(f(n))$.)

Concrete overhead. In Sect. 5 we present a new variant of Cuckoo hashing that we denote as two-dimensional (or 2D) Cuckoo hashing. We analyze this construction in the full version [40] and show that when no stash is used, then the failure probability (with tables of size $O(n)$) is $O(1/n)$, as in standard Cuckoo hashing.

We only have a sketch of an analysis for the size of the stash of the construction in Sect. 5, but we observed that this construction performed much better than the asymptotic construction. Also, performance was improved with the heuristic of using half as many bins but letting each bin store two items instead of one. (This variant is known to perform much better also in the case of standard Cuckoo hashing, see [47].)

Since we do not have a theoretical analysis of this construction, we ran extensive experiments in order to examine its performance. These experiments follow the analysis paradigm given in Sect. 3.2, and are described in Sect. 6.1. For a specific ratio between the table size and n, we ran 2^{40} experiments for $n = 2^{12}$ and found that the failure probability is at most 2^{-37} with 99.9% confidence. We also ran experiments for increasing values of n, up to $n = 2^{12}$, and found that the failure probability has linear dependence on n^{-3} (an explanation of this behavior appears in the full version [40]). Therefore, we can argue that for $n \geq 2^{13} = 2 \cdot 2^{12}$ the failure probability is at most $2^{-37} \cdot 2^{-3} = 2^{-40}$.

4 An Asymptotic Construction Through Mirror Cuckoo Hashing

We show here a construction for circuit-based PSI that has an $\omega(n)$ asymptotic overhead. The analysis in this section is not intended to be tight, but rather shows the asymptotic behavior of the overhead.

The analysis is based on a construction which we denote as *mirror Cuckoo hashing* (as the placement of the hash functions that are used in one side is a mirror image of the hash functions of the other side). Hashing is computed in a single iteration. The main advantage of this construction is that it is based on four copies of standard Cuckoo hashing. Therefore, we can apply known bounds on the failure probability of Cuckoo hashing. Namely, applying the result of [29] that the failure probability when using a stash of size s is $O(n^{-(s+1)})$. Given this result, a stash of size $\omega(1)$ guarantees that the failure probability is negligible in n (while a constant stash size guarantees that for sufficiently large n the failure probability is smaller than any constant, and in particular smaller than 2^{-40}). We note that while the known results about the size of the stash are only stated for $s = O(1)$, we show in the full version [40] that the $O(n^{-(s+1)})$ bound on the failure probability also applies to a non-constant stash size.

4.1 Mirror Cuckoo Hashing

We describe a hashing scheme that uses two sets of tables. A left set including tables T_{L}, T_{R}, and a right set including tables T'_{L}, T'_{R}. Each table is also denoted

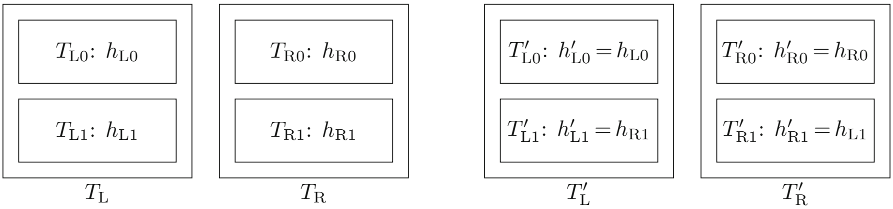
$T_{\mathrm{L}0}$: $h_{\mathrm{L}0}$
$T_{\mathrm{L}1}$: $h_{\mathrm{L}1}$
T_{L}
$T_{\mathrm{R}0}$: $h_{\mathrm{R}0}$
$T_{\mathrm{R}1}$: $h_{\mathrm{R}1}$
T_{R}
$T'_{\mathrm{L}0}$: $h'_{\mathrm{L}0} = h_{\mathrm{L}0}$
$T'_{\mathrm{L}1}$: $h'_{\mathrm{L}1} = h_{\mathrm{R}1}$
T'_{L}
$T'_{\mathrm{R}0}$: $h'_{\mathrm{R}0} = h_{\mathrm{R}0}$
$T'_{\mathrm{R}1}$: $h'_{\mathrm{R}1} = h_{\mathrm{L}1}$
T'_{R}

Fig. 1. The tables T_{L}, T_{R} and T'_{L}, T'_{R}. The hash functions in the upper subtables of T'_{L}, T'_{R} are the same as in T_{L}, T_{R}, and those in the lower subtables are in reverse order.

as a "column". Each table has two subtables, or "rows". So overall there are four tables (columns), each containing two subtables (rows).

Bob maps each of his items to one subtable in each table, namely to one row in each column. Alice maps each of her items to the two subtables in one of the tables, namely to both rows in just one of the columns. These mappings ensure that for any item x that is owned by both Alice and Bob, there is exactly one subtable to which it is mapped by both parties.

The tables. The construction uses two sets of tables, T_{L}, T_{R} and T'_{L}, T'_{R}. Each table is of size $2(1+\varepsilon)n$ and is composed of two subtables of size $(1+\varepsilon)n$ (T_{L} contains the subtables $T_{\mathrm{L}0}$, $T_{\mathrm{L}1}$, etc.). Each subtable is associated with a hash function that will be used by both parties. E.g., function $h_{\mathrm{L}0}$ will be used for subtable $T_{\mathrm{L}0}$, etc. The tables and the hash functions are depicted in Fig. 1.

The hash functions. The hash functions associated with the tables are defined as follows:

- The functions for the left two tables (columns) T_{L}, T_{R}, i.e., $h_{\mathrm{L}0}$, $h_{\mathrm{L}1}$, $h_{\mathrm{R}0}$, $h_{\mathrm{R}1}$, are chosen at random. Each function maps items to the range $[0, (1+\varepsilon)n-1]$, which corresponds to the number of bins in each of $T_{\mathrm{L}0}$, $T_{\mathrm{L}1}$, $T_{\mathrm{R}0}$, $T_{\mathrm{R}1}$.
- The functions for the two right tables T'_{L}, $T'_{\mathrm{R}}$are defined as follows:
  - The two functions of the upper subtables are equal to the functions of the upper subtables on the left. Namely, $h'_{\mathrm{L}0} = h_{\mathrm{L}0}$ and $h'_{\mathrm{R}0} = h_{\mathrm{R}0}$.
  - The two functions of the lower subtables are the *mirror image* of the functions of the lower subtables on the left. Namely, $h'_{\mathrm{L}1}$, $h'_{\mathrm{R}1}$ are defined such that $h'_{\mathrm{L}1} = h_{\mathrm{R}1}$, and $h'_{\mathrm{R}1} = h_{\mathrm{L}1}$.

Bob's insertion algorithm. Bob needs to insert each of his items to one subtable in each of the tables T_{L}, T_{R}, T'_{L}, T'_{R}. He can do so by simply using Cuckoo hashing for each of these tables. For example, for the table T_{L} and its subtables $T_{\mathrm{L}0}$, $T_{\mathrm{L}1}$, Bob uses the functions $h_{\mathrm{L}0}$, $h_{\mathrm{L}1}$ to insert each input x to either $T_{\mathrm{L}0}$ or $T_{\mathrm{L}1}$. The same is applied to T_{R}, T'_{L}, and T'_{R}. In addition, Bob keeps a small stash of size $\omega(1)$ for each of the four tables. Overall, based on known properties of Cuckoo hashing, we can claim that the construction guarantees the following property:

Algorithm 1 (Mirror Cuckoo hashing)

1. Alice uses Cuckoo hashing to insert each item x to one of the subtables $T_{\mathrm{L}0}$, $T_{\mathrm{R}0}$, using the hash functions $h_{\mathrm{L}0}$, $h_{\mathrm{R}0}$.
2. Similarly, Alice uses Cuckoo hashing to insert each item x to one of the subtables $T_{\mathrm{L}1}$, $T_{\mathrm{R}1}$, using the hash functions $h_{\mathrm{L}1}$, $h_{\mathrm{R}1}$.
3. At this point, Alice observes the result of the first two steps. For some inputs x it happened that they were mapped to the same "column" in both of these steps. Namely, x was mapped to both $T_{\mathrm{L}0}$ and $T_{\mathrm{L}1}$, or to both $T_{\mathrm{R}0}$ and $T_{\mathrm{R}1}$. These are the "good" items, since they were mapped to the same column, as is required for all of Alice's inputs.
4. The other inputs of Alice, the "bad" items, were mapped to one column in Step 1 and to the other column in Step 2. Alice applies the following procedure to these items:
   (a) Each "bad" item x is removed from both locations to which it was mapped in Steps 1 and 2.
   (b) x is now inserted in either of $T'_{\mathrm{L}0}$, $T'_{\mathrm{R}0}$ using the hash functions $h'_{\mathrm{L}0} := h_{\mathrm{L}0}$, $h'_{\mathrm{R}0} := h_{\mathrm{R}0}$ with the same mapping as in Step 1.
   (c) x is also inserted in either of $T'_{\mathrm{L}1}$, $T'_{\mathrm{R}1}$ using the hash functions $h'_{\mathrm{L}1} := h_{\mathrm{R}1}$, $h'_{\mathrm{R}1} := h_{\mathrm{L}1}$ with the same mapping as in Step 2.

Claim. With all but negligible probability, it holds that for every input x of Bob, and for each of the four tables T_{L}, T_{R}, T'_{L}, T'_{R}, Bob inserts x to exactly one of the two subtables or to the stash.

Alice's insertion algorithm. Alice's operation is a little more complex and is described in Algorithm 1. Alice considers the two upper subtables on the left, $T_{\mathrm{L}0}$, $T_{\mathrm{R}0}$, as two subtables for standard Cuckoo hashing. Similarly, she considers the two lower subtables on the left, $T_{\mathrm{L}1}$, $T_{\mathrm{R}1}$, as two subtables for standard Cuckoo hashing. In other words, she considers the left top row and the left bottom row as standard Cuckoo hashing tables.

Alice then inserts each input item of hers to each of these two tables using standard Cuckoo hashing. (She also uses stashes of size $\omega(1)$ to store items which cannot be placed in the Cuckoo tables.) For some input items x it happens that x is inserted in the top row to $T_{\mathrm{L}0}$ and in the bottom row to $T_{\mathrm{L}1}$; or x is inserted in the top row to $T_{\mathrm{R}0}$ and in the bottom row to $T_{\mathrm{R}1}$. Therefore, x is inserted in two subtables in the same column. (x is denoted as "good" since this is the outcome that we want.)

Let x' be one of the other, "bad", items. Thus, x' is inserted in the top row to $T_{\mathrm{L}0}$ and in the bottom row to $T_{\mathrm{R}1}$, or vice versa. In this case, Alice removes x' from the tables on the left and inserts it to the tables T'_{L}, T'_{R} on the right. Since the hash functions that are used in T'_{L}, T'_{R} are equal to the functions used on the left side (where in the bottom row the functions are in reverse order), Alice does not need to run a Cuckoo hash insertion algorithm on the right side: Assume that x' was stored in locations $T_{\mathrm{L}0}[h_{\mathrm{L}0}(x')]$ and $T_{\mathrm{R}1}[h_{\mathrm{R}1}(x')]$ on the left. Then Alice inserts it to locations $T'_{\mathrm{L}0}[h'_{\mathrm{L}0}(x')] = T'_{\mathrm{L}0}[h_{\mathrm{L}0}(x')]$ and $T'_{\mathrm{L}1}[h'_{\mathrm{L}1}(x')] = T'_{\mathrm{L}1}[h_{\mathrm{R}1}(x')]$ on the right.

In other words, in a global view, one can see the algorithm as composed of the following steps: (1) First, all items are placed in the left tables. (2) Each subtable is divided in two copies, where one copy contains the good items and the other copy contains the bad items. (3) The subtable copies with the good items are kept on the left, whereas the copies with the bad items are moved to the right, where in the bottom row on the right we replace the order of the subtables.

This algorithm has two important properties: First, all items that were successfully inserted in the first step to the left tables will be placed in tables on either the left or the right hand sides. Moreover, each item will be placed in two subtables in the same column — the good items happened to initially be placed in this way in the left tables; whereas the bad items were in different columns on the left side but were moved to the same column on the right side. Hence, we can state the following claim:

Claim. With all but negligible probability, Alice inserts each of her inputs either to two locations in exactly one of T_{L}, T_{R}, T'_{L}, T'_{R} and to no locations in other tables, or to a stash.

Tables size. The total size of the tables is $8(1+\varepsilon)n$.

Stash size. With regards to stashes, each party needs to keep a stash for each of the Cuckoo hashing tables that it uses. Since Alice runs the Cuckoo hashing insertion algorithm only for the left tables and re-uses the mapping for the right tables, she needs only two stashes. Bob on the other hand runs the Cuckoo hashing insertion algorithm four times and hence needs four stashes. (In order to preserve simplicity, we omitted the stashes in Fig. 1 and Algorithm 1.) Given the result of [29], and our observation in the full version [40] about its applicability to non-constant stash sizes, it holds that a total stash of size $\omega(1)$ elements suffices to successfully map all items, except with negligible probability. We note that the size of the stash can be arbitrarily close to constant, e.g., it can be set to be $O(\log\log n)$ or $O(\log^* n)$. Essentially, for any function $f(n) \in \omega(n)$, the size of the stash can be $o(f(n))$.

4.2 Circuit-Based PSI from Mirror Cuckoo Hashing

Mirror Cuckoo hashing lets the parties map their inputs to tables of size $O(n)$ and stashes of size $\omega(1)$, with negligible failure probability. It is therefore straightforward to construct a PSI protocol based on this hashing scheme:

1. The parties agree on the parameters that define the size of the tables and the stash for mirror Cuckoo hashing. They also agree on the hash functions that will be used in each table.
2. Each party maps its items to the tables using the hash functions that were agreed upon.
3. The parties evaluate a circuit that performs the following operations:
   (a) For each bin in the tables, the circuit compares the item that Alice mapped to the bin to the item that Bob mapped to the same bin.

(b) Each item that Bob mapped to his stashes is compared with all items of Alice. Similarly, each item that Alice mapped to her stashes is compared with all items of Bob.

The properties of mirror Cuckoo hashing ensure: (1) If an item x is in the intersection, then there is exactly one comparison in which x is input by both Alice and Bob. (2) The number of comparisons in Step 3 is $\omega(n)$.

5 A Concretely Efficient Construction Through 2D Cuckoo Hashing

Two-dimensional Cuckoo hashing (a.k.a. 2D Cuckoo hashing) is a new construction with the following properties:

- It uses overall $O(n)$ memory (specifically, $8(1+\varepsilon)n$ in our construction, where we set $\varepsilon = 0.2$ in our experiments).
- Both, Alice and Bob, map each of their items to $O(1)$ memory locations (specifically, to two or four memory locations in our construction).
- If x appears in the input of both parties, then there is exactly one location to which both Alice and Bob map x.

The construction uses two tables, T_{L}, T_{R}, located on the left and the right side, respectively. Each of these tables is of size $4(1+\varepsilon)n$ and is composed of two smaller subtables: T_{L} is composed of the two smaller subtables T_{L0}, T_{L1}, while T_{R} is composed of the two smaller tables T_{R0}, T_{R1}. The hash functions h_{L0}, h_{L1}, h_{R0}, h_{R1} are used to map items to T_{L0}, T_{L1}, T_{R0}, T_{R1}, respectively. The tables are depicted in Fig. 2.

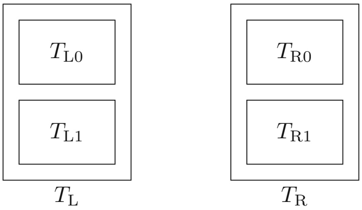

Fig. 2. The tables T_{L} and T_{R}, consisting of T_{L0}, T_{L1} and T_{R0}, T_{R1}, respectively.

Hashing is performed in the following way:

- Alice maps each of her items to all subtables on one of the two sides. Namely, each item x of Alice is either mapped to both bins $T_{\mathrm{L0}}[h_{\mathrm{L0}}(x)]$ and $T_{\mathrm{L1}}[h_{\mathrm{L1}}(x)]$ on the left side, or to bins $T_{\mathrm{R0}}[h_{\mathrm{R0}}(x)]$ and $T_{\mathrm{R1}}[h_{\mathrm{R1}}(x)]$ on the right side. In other words, ALICE maps each item to ALL subtables on one side.

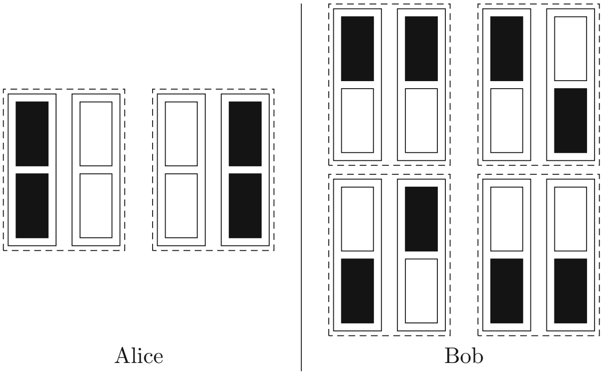

Fig. 3. The possible combinations of locations to which Alice and Bob map their inputs.

- Bob maps each of his items to one subtable on each side. This is done using standard Cuckoo hashing. Namely, each input x of Bob is mapped to one of the locations $T_{\mathrm{L0}}[h_{\mathrm{L0}}(x)]$ or $T_{\mathrm{L1}}[h_{\mathrm{L1}}(x)]$ on the left side, as well as mapped to one of the locations $T_{\mathrm{R0}}[h_{\mathrm{R0}}(x)]$ or $T_{\mathrm{R1}}[h_{\mathrm{R1}}(x)]$ on the right side. In other words, BOB maps each item to one subtable on BOTH sides.

The possible options for hashing an item x by both parties are depicted in Fig. 3. It is straightforward to see that if both parties have the same item x, there is exactly one table out of T_{L0}, T_{L1}, T_{R0}, T_{R1} that is used by both Alice and Bob to store x.

We next describe a construction of 2D Cuckoo hashing, followed by a variant based on a heuristic optimization that stores two items in each table entry. The asymptotic behavior of the basic construction is analyzed in the full version [40]. In Sect. 6.1 we describe simulations for setting the parameters of the heuristic construction in order to reduce the hashing failure probability to below 2^{-40}.

5.1 Iterative 2D Cuckoo Hashing

This construction uses two tables, T_{L}, T_{R}, each of $4(1+\varepsilon)n$ entries. (In this construction, there is no need to assume that each table is composed of two subtables.) The parties associate two hash functions with each table, namely h_{L0}, h_{L1} for T_{L}, and h_{R0}, h_{R1} for T_{R}.

Bob uses Cuckoo hashing to insert each of his items into one location in each of the tables.

Alice inserts each item x either into the two locations $h_{\mathrm{L0}}(x)$ and $h_{\mathrm{L1}}(x)$ in T_{L}, or into the two locations $h_{\mathrm{R0}}(x)$ and $h_{\mathrm{R1}}(x)$ in T_{R}. This is achieved by Alice running a modified Cuckoo insertion algorithm that maps an item to two locations in one table, "kicks out" any item that is currently present in these locations and also removes the other occurrence of this item from the table, and then tries to insert this item into its two locations in the other table, and so on.

Algorithm 2 (Iterative 2D Cuckoo hashing)

1. Alice maps all of her items to table T_{L}, using simple hashing. That is, each item x is inserted in locations $h_{\mathrm{L}0}(x), h_{\mathrm{L}1}(x)$. Obviously, there will be entries in T_{L} that will have more than a single item mapped to them. Denote T_{L} as the active table.
2. For each entry in the active table with more than one item in it: remove all items – except for the item that was mapped to this entry most recently – and move them to the "relocation pool". For each of the removed items, remove the item also from its other appearance in the active table. (At the end of this step, all entries in the active table have at most one entry. However, there might be items in the relocation pool.)
3. If the relocation pool is empty, then stop (found a successful mapping).
4. Change the designation of the active table to point to the other table.
5. Move each item x from the relocation pool to locations $h_0(x), h_1(x)$ in the active table. (For example, if T_{R} is the active table, move x to $h_{\mathrm{R}0}(x), h_{\mathrm{R}1}(x)$.)
6. Go to Step 2.

Algorithm 3 (Iterative 2D Cuckoo hashing with bins of size 2)

The algorithm is identical to Algorithm 2, except for the following change in Step 2:

2. For each entry in the active table with more than *two* items in it: remove all items – except for the *two* items that were mapped to this entry most recently – and move them to the "relocation pool". For each of the removed items, remove the item also from its other appearance in the active table.

This is a new variant of Cuckoo hashing, where inserting an item into a table might result in four elements that need to be stored in the other table: storing x in $h_{\mathrm{L}0}(x), h_{\mathrm{L}1}(x)$ might remove two items, y_0, y_1, one from each location. These items are also removed from their other occurrences in T_{L}. They must now be stored in locations $h_{\mathrm{R}0}(y_0), h_{\mathrm{R}1}(y_0), h_{\mathrm{R}0}(y_1), h_{\mathrm{R}1}(y_1)$ in T_{R}.

It is not initially clear whether such a mapping is possible (with high probability, given random choices of the hash functions). We analyze the construction in the full version [40] and show that it only fails with probability $O(1/n)$. We ran extensive simulations, showing that the algorithm (when using a stash and a certain choice of parameters) fails with very small probability, smaller than 2^{-40}.

The insertion algorithm of Alice is described in Algorithm 2. The choice made in Step 2 of the algorithm, to first remove the oldest items that were mapped to the entry, is motivated by the intuition that it is more likely that the locations to which these items are mapped in the other table are free.

Storing two items per bin. It is known that the space utilization of Cuckoo hashing can be improved by storing more than one item per bin (cf. [15,37] or the review of multiple choice hashing in [47]). We take a similar approach and use two tables of size $2(1+\varepsilon)n$ where each entry can store *two* items. (These tables have half as many entries as before, but each entry can store two items rather than one. The total size of the tables is therefore unchanged.) The change to the insertion algorithm is minimal and affects only Step 2. The new algorithm is defined in Algorithm 3.

Our experiments in Sect. 6.1 show that when using the same amount of space, then this variant of iterative 2D Cuckoo hashing performs better than the basic protocol with bins of size one. That is, it achieves a lower probability of hashing failure, namely of the need to use the stash, and requires less iterations to finish.

5.2 Circuit-Based PSI from 2D Cuckoo Hashing

This section describes how 2D Cuckoo hashing can be used for computing PSI. In addition, we describe two optimizations which substantially improve the efficiency of the protocol. The first optimization has the parties use permutation-based hashing [2] (as was done in [39]) in order to reduce the size of the items that are stored in each bin, and hence reduce the number of gates in the circuit. The second optimization is based on having each party use a single stash instead of using a separate stash for each Cuckoo hashing instance.

The PSI protocol is pretty straightforward given 2D Cuckoo hashing:

First, the parties agree on the hash functions to be used in each table. (These functions must be chosen at random, independently of the inputs, in order not to disclose any information about the inputs. Therefore, a participant cannot change the hash functions if some items cannot be mapped, and thus we seek parameter values that make the hashing failure probability negligible, e.g., smaller than 2^{-40}.)

Then, each party maps its items to bins using 2D Cuckoo hashing and the chosen hash functions. The important property is that if Alice and Bob have the same input item then there exists exactly one bin into which both parties map this item (or, alternatively, at least one of them places this item in a stash). Empty bins are padded with dummy elements. This ensures that no information is leaked by how empty the tables and stashes are.

Afterwards, the parties construct a circuit that compares, for each bin, the items that both parties stored in it. In addition, this circuit compares each item that Alice mapped to the stash with all of Bob's items, and vice versa. Since the number of bins is $O(n)$, the number of items in each bin is $O(1)$, and the number of items in the stash is $\omega(1)$, the total size of this circuit is $\omega(n)$. The parties can define another circuit that takes the output of this circuit and computes a desired function of it, e.g., the number of items in the intersection.

Finally, the parties run a generic MPC protocol that securely evaluates this circuit (cf. Sect. 6.3 for a concrete implementation and benchmarks).

Permutation-based Hashing. The protocol uses permutation-based hashing to reduce the bitlength of the elements that are stored in the bins and thus reduces the size of the circuit comparing them. This idea was introduced in [2] and used for PSI in [39]. It is implemented in the following way. The hash function h that is used to map an item x to one of the β bins is constructed as follows: Let $x = x_L|x_R$ where $|x_L| = \log \beta$. We first assume that β is a power of 2 and then describe the general case. Let f be a random function with range $[0, \beta - 1]$. Then h maps an element x to bin $x_L \oplus f(x_R)$ and the value stored in the bin is x_R. The important property is that the stored value has a reduced bitlength of only $|x| - \log \beta$, yet there are no collisions (since if x, y are mapped to the same bin and store the same value, then $x_R = y_R$ and $x_L \oplus f(x_R) = y_L \oplus f(y_R)$ and therefore $x = y$).

In the general case, where β is not a power of two, the output of h is reduced modulo β and a stored extra bit indicates if the output was reduced or not.

For Cuckoo hashing the protocol uses two hash functions to map the elements to the bins in one table. To avoid collisions among the two hash functions, a stored extra bit indicates which hash function was used.

Using a Combined Stash. Recall that Alice uses 2D Cuckoo hashing, for which we show experimentally in Sect. 6.1 that no stash is needed. Bob, on the other hand, uses two invocations of standard Cuckoo hashing, and therefore when he does not succeed in mapping an item to a table, he must store it in a stash and compare it with all items of Alice. In this case, the parties cannot encode their items using permutation-based hashing, and therefore these comparisons must be of the full-length original values and not of the shorter values computed using permutation-based hashing as described before. Therefore, the size of the circuits that handle the stash values have a considerable effect on the total overhead of the protocol.

We observe that, instead of keeping several stashes, Bob can collect all the values that he did not manage to map to any of the tables in a *combined* stash. Suppose that he maps items to c tables and that we have an upper bound s which holds w.h.p. on the size of each stash. A naive approach would use c stashes of that size, resulting in a total stash size of $c \cdot s$. A better approach would be to use a single stash for all these items, since it is very unlikely that all stashes will be of maximal size, and therefore we can show that with the same probability, the size s' of the combined stash is much smaller than $c \cdot s$. To do so, we determine the upper bounds for the combined stash for $c = 2$: The probability of having a combined stash of size s' is $\sum_{i=0}^{s'} P(i) \cdot P(s' - i)$, where $P(i)$ denotes the probability of having a single stash of size i. The value of $P(i)$ is $O(n^{-i}) - O(n^{-(i+1)}) \approx O(n^{-i})$ [29]. We can estimate the exact values of these probabilities based on the experiments conducted by [39]: they performed 2^{30} Cuckoo hashing experiments for each $n \in \{2^{11}, 2^{12}, 2^{13}, 2^{14}\}$ and counted the required stash sizes. Using linear regression, we extrapolated the results for larger sets of 2^{16} and 2^{20} elements. Table 1 shows the required stash sizes when binding the probability to be below 2^{-40}: it turns out that for 2^{12} and 2^{16} elements the combined stash should include only one more element compared to the upper bound for a single stash, whereas for 2^{20} even the same stash size is sufficient.

Table 1. Stash sizes required for binding the error probability to be below 2^{-40} when inserting $n \in \{2^{12}, 2^{16}, 2^{20}\}$ elements into $2.4n$ bins using Cuckoo hashing.

Number of elements n	2^{12}	2^{16}	2^{20}
Single stash size s (from [39, Table 4])	6	4	3
Stash size for two separate stashes $s' = 2s$	12	8	6
Combined stash size s'	7	5	3

All in all, when comparing to the naive solution with two separate stashes, the combined stash size is reduced by almost a factor of 2x.

5.3 Extension to a Larger Number of Parties

Computing PSI between the inputs of more than two parties has received relatively little interest. (The challenge is to compute the intersection of the inputs of all parties, without disclosing information about the intersection of the inputs of any subset of the parties.) Specific protocols for this task were given, e.g., in [20,24,32]. We note that our 2D Cuckoo hashing can be generalized to m dimensions in order to obtain a circuit-based protocol for computing the intersection of the inputs of m parties. The caveat is that the number of tables grows to 2^m and therefore the solution is only relevant for a small number of parties.

We describe the case of three parties: The hashing will be to a set of eight tables $T_{x,y,z}$, where $x, y, z \in \{0, 1\}$. Any input item of P_1 is mapped to either all tables $T_{0,0,0}, T_{0,0,1}, T_{0,1,0}, T_{0,1,1}$, or to all tables $T_{1,0,0}, T_{1,0,1}, T_{1,1,0}, T_{1,1,1}$. Namely, the index x is set to either 0 or 1, and the input item is mapped to all tables with that value of x. Every input of P_2 is mapped either to all tables whose y index is 0, or to all tables where $y = 1$. Every input of P_3 is mapped either to all tables whose z index is 0, or to all tables where $z = 1$.

It is easy to see that regardless of the choices of the values of x, y, z, the sets of tables to which all parties map an item intersect in exactly one table. Therefore, the parties can evaluate a simple circuit that checks every bin for equality of the values that were mapped to it by the three parties. It is guaranteed that if the same value is in the input sets of all parties, then there is exactly one bin to which this value is mapped by all three parties. If some items are mapped to a stash by one of the parties, they must be compared with all items of the other parties, but the overhead of this comparison is $\omega(n)$ if the stash is of size $\omega(1)$.

The remaining issue is the required size of the tables. In the full version [40] we show that inserting an item into one of two (big) tables, such that the item is mapped to k locations in that table, requires tables of size greater than $k^2(1+\varepsilon)n$. When computing PSI between three parties using the method described above, we have eight (small) tables, where each party must insert its items to four tables in one plane or to four tables in the other plane. Each such set of four small tables corresponds to a big table in the analysis and is therefore of size $16(1 + \varepsilon)n$. The total size of the tables is therefore $32(1 + \varepsilon)n$.

5.4 No Extension to Security Against Malicious Adversaries

We currently do not see how to extend our hashing-based protocols to achieve security against malicious adversaries. As pointed out by [44], it is inherently hard to extend protocols based on Cuckoo hashing to obtain security against malicious adversaries. The reason is that the placement of items depends on the exact composition of the input set, and therefore a malicious party might learn the placement used by the other party.

Coming up with a similar argument as in [44], assume that in our construction in Fig. 3, Bob maps an item x to the two upper subtables and Alice maps x to the two left subtables. Now assume Alice maliciously deviates from the protocol and places x only in the upper left subtable, but not in the lower left one. This deviation may allow Alice to learn whether Bob placed x in the upper or lower subtables: For example, in a PSI-CA protocol Alice could use only dummy elements and x as an input set and if the cardinality turns out to be 1, then she knows that Bob placed x in the upper left subtable. However, the locations in which Bob places an item cannot be simulated in the ideal world as they depend on other items in his input set. Therefore, we see no trivial way to provide security against malicious adversaries based on 2D Cuckoo hashing.

6 Evaluation

This section describes extensive experiments that set the parameters for the hashing schemes, the resulting circuit sizes, and the results of experiments evaluating PSI using these circuits.

6.1 Simulations for Setting the Parameters of 2D Cuckoo Hashing

We experimented with the iterative 2D Cuckoo hashing scheme described in Sect. 5.1, set concrete sizes for the tables, and examined the failure probabilities of hashing to the tables.

Our implementation is written in C and available online at http://encrypto.de/code/2DCuckooHashing. It repeatedly inserts a set of random elements into two tables using random hash functions. The insertion algorithm is very simple: All elements are first inserted into the two locations to which they are mapped (by the hash functions) in the first table. Obviously, many table entries will contain multiple items. Afterwards, the implementation iteratively moves items between the tables, in order to reduce the maximum bin occupancy below a certain threshold (cf. Algorithms 2 and 3 in Sect. 5.1).

Run-time. We report in Sect. 6.3 the results of experiments analyzing the run-time of the 2D Cuckoo hashing insertion algorithm. Overall, the insertion time (a few milliseconds) is negligible compared to the run-time of the entire PSI protocol.

Hashing to bins of size 1. First, we checked if it is possible to use a maximum bin occupation of 1. For this, we set the sizes of each of the two tables to be $4.8n$ (corresponding to the threshold size of $4(1+\varepsilon)n$ in the analysis available in the full version [40], as well as twice the recommended size for Cuckoo hashing, since all elements are inserted twice). We ran the experiment 100 000 times with input size $n = 2^{12}$ and bitlength 32. For all except 828 executions it was possible to reduce the maximum bin occupation to 1 after at least 7 and at most 129 iterations of the insertion algorithm. On average, 20 iterations of the insertion algorithm were necessary to achieve the desired result. In said 828 cases there remained at least one bin with more than one item even after 500 iterations of the insertion algorithm. This implies that iterative 2D Cuckoo hashing works in principle, but, as standard Cuckoo hashing, requires a stash for storing the elements of overfull bins.

Hashing to bins of size 2. For PSI protocols it would be desirable to avoid having an additional stash on Alice's side. In standard Cuckoo hashing it is possible to achieve better memory utilization and less usage of the stash by using fewer bins, where each bin can store two items [47]. Therefore, we changed the parameters as follows: the table size is halved and reduced to $2.4n$, but each bin is allowed to contain two elements. This way, while consuming the same amount of memory as before, we try to achieve better utilization. We followed the paradigm that was described in Sect. 3.2 for the experimental analysis of the failure probability. Namely, we ran massive sets of experiments to measure the number of failures for several values of n and several table sizes, and given this data we (1) found confidence intervals for the failure probability for specific values of the parameters, and (2) found how the failure probability behaves as a function of n.

Our first experiment ran 2^{40} tests within ~2 million core hours on the Lichtenberg[3] high performance computer of the TU Darmstadt for input size $n = 2^{12}$. We chose input size 2^{12} (instead of larger sizes like 2^{16} or 2^{20}) since running experiments with larger values of n would have taken even more time and would have simply been impractical. It turned out that the insertion algorithm was successful in reducing the maximum bin size to 2 (after at most 18 iterations) in all but one test.

Given this data, we calculated the confidence interval of the failure probability p. The probability of observing one failure in N experiments is $N \cdot p \cdot (1-p)^{N-1}$, where in our experiments $N = 2^{40}$. We checked the values of p for which the probability of this observation is greater than 0.001 and concluded that with 99.9% confidence, the failure probability for iterative 2D Cuckoo hashing with set size $n = 2^{12}$ and table size $2.4n$ lies within $\left[2^{-50}, 2^{-37}\right]$. (Namely, there is at most a 0.001 probability that we would have seen one failure in 2^{40} runs if p was greater than 2^{-37} or smaller than 2^{-50}.)

[3] See http://www.hhlr.tu-darmstadt.de/hhlr/index.en.jsp for details on the hardware configuration.

Measuring the dependence on the parameters. To get a better understanding on how the failure probability behaves for different input and table sizes, we performed a set of experiments that required another ~3.5 million core hours. Concretely, we ran 2^{40} tests for each set size $n \in \{2^6, 2^8, 2^{10}\}$ and each table size in the range $2.2n$, $2.4n$, and $2.6n$. We also tested the table size $3.6n$ for $n \in \{2^6, 2^8\}$ as well as table sizes $3.0n$ and $3.2n$ for $n = 2^{10}$. The results for all experiments are given in Table 2 and are depicted in Fig. 4.

The results demonstrate that, w.r.t. the dependence on n, for set sizes $n \in \{2^6, 2^8, 2^{10}\}$ it can be observed that increasing the set size by factor 4x reduces the failure probability by factor 64x. (For larger set sizes, the number of failures

Table 2. Number of observed stashes for different table sizes and set sizes n when performing 2^{40} tests of iterative 2D Cuckoo hashing.

Table size	Stash size	$n = 2^6$	$n = 2^8$	$n = 2^{10}$	$n = 2^{12}$
$2.2n$	1	64 020	1 021	16	—
	2	154	1	0	—
	3	4	0	0	—
$2.4n$	1	31 033	499	8	1
	2	65	0	0	0
$2.6n$	1	16 014	270	5	—
	2	33	0	0	—
$3.0n$	1	—	—	0	—
$3.2n$	1	—	—	0	—
$3.6n$	1	1 202	17	—	—

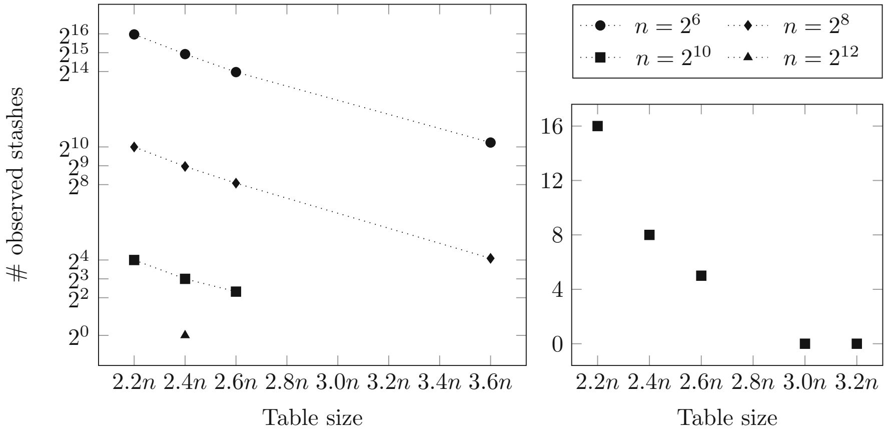

Fig. 4. Number of observed stashes for different table and set sizes when performing 2^{40} tests of iterative 2D Cuckoo hashing.

is too small to be meaningful.) These experiments also demonstrate that the dependence of the failure probability on n is $O(n^{-3})$. An intuitive theoretical explanation why the probability behaves this way is given in the full version [40]. As for the dependence on the table size, the failure probability decreases by a factor of 2x when increasing the table size in steps of $0.2n$ within the tested range $2.2n$ to $3.6n$.

From these results (a failure probability of at most 2^{-37} for $n = 2^{12}$ with table size $2.4n$ and a dependence of $O(n^{-3})$ of the failure probability on n) we conclude that the failure probability for $n \geq 2^{13}$ and table size $2.4n$ is at most 2^{-40}.

In total we spent about 5.5 million core hours on our experiments on the Lichtenberg high performance computer of the TU Darmstadt.

6.2 Circuit Complexities

We compare the complexities of the different circuit-based PSI constructions for two sets, each with n elements that have bitlength σ. We consider two possible bitlengths:

1. **Fixed bitlength:** Here, the elements have fixed bitlength $\sigma = 32$ bits (e.g., for IPv4 addresses).
2. **Arbitrary bitlength:** Here, the elements have arbitrary bitlength and are hashed to values of length $\sigma = 40 + 2\log_2(n) - 1$ bits, with a collision probability that is bounded by 2^{-40}. (See Appendix A of the full version of [41] for an analysis.) Therefore, we set the bitlength to $\sigma = 40 + 2\log_2(n) - 1$ bits.

For all protocols we report the circuit size where we count only the number of AND gates, since many secure computation protocols provide free computation of XOR gates. We compute the size of the circuits up to the step where single-bit wires indicate if a match was found for the respective element. We note that for many circuits computing functions of the intersection, this part of the circuit consumes the bulk of the total size. For example, computing the Hamming weight of these bits is equal to computing the cardinality of the intersection (PSI-CA). The size-optimal Hamming weight circuit of [5] has size $x - w_H(x)$ and depth $\log_2 x$, where x is the number of inputs and $w_H(\cdot)$ is the Hamming weight. The size of the Hamming weight circuit is negligible compared to the rest of the circuit. As another example, if the cardinality is compared with a threshold (yielding a PSI-CAT protocol), this only adds $3\log_2 n$ AND gates and depth $\log_2 \log_2 n$ using the depth-optimized construction described in [45], which is also negligible.

The size of the Sort-Compare-Shuffle circuit. The Sort-Compare-Shuffle circuit [26] has three phases. In the SORT phase, the two sorted lists of inputs are merged into one sorted list, which takes $2\sigma n \log_2(2n)$ AND gates. In the COMPARE phase, neighboring elements are compared to find the elements in the intersection, which takes $\sigma(3n - 1) - n$ AND gates. The SHUFFLE phase randomly permutes these values and takes $\sigma(n\log_2(n) - n + 1)$ AND gates. To have a fair comparison with our protocols, we remove the SHUFFLE phase and let the COMPARE phase output only a single bit that indicates if a match was found for the respective element or not; this removes n multiplexers of σ-bit values from the COMPARE phase, i.e., σn AND gates. Hence, the total size is $2\sigma n\log_2(n) + 2\sigma n - n - \sigma + 2$ AND gates.

The size of the Circuit-Phasing circuit. The Circuit-Phasing circuit [39] has $2.4nm(\sigma - \log_2(2.4n) + 1) + sn(\sigma - 1)$ AND gates where m is the maximum occupancy of a bin for simple hashing and s is the size of the stash.

The size of our iterative 2D Cuckoo hashing construction of Sect. 5.2*.* Each of the following operations is performed twice for the left and right side: (1) For each of the $2.4n$ bins the shortened representation (cf. Sect. 5.2) of the single item in Bob's bin is compared with the two elements in the corresponding bin of Alice. (2) Bob has a stash of size s'. Each item in the stash is compared to all of Alice's items (using the full bitlength representation). Hence, the overall complexity is $4 \cdot 2.4n(\sigma - \log_2(2.4n) + 1) + s'n(\sigma - 1)$ AND gates, where s' is the size of the combined stash.

Concrete Circuit Sizes. The Sort-Compare-Shuffle construction [26] has a circuit of size $O(\sigma n \log n)$. The Circuit-Phasing construction [39] has circuit size $O(\sigma n \log n / \log\log n)$, while the asymptotic construction we present in this paper has a size of $\omega(\sigma n)$ and the iterative 2D Cuckoo hashing construction has an even smaller size.

For a comparison of the concrete circuit sizes, we use the parameters from the analysis in [39]: For $n = 2^{12}$ elements the maximum bin size for simple hashing is $m = 18$, for $n = 2^{16}$ we set $m = 19$, and for $n = 2^{20}$ we set $m = 20$. We set the stash size s and the combined stash size s' according to Table 1 (on page 21).

On the left side of Table 3 we compare the concrete circuit sizes for *fixed* bitlength $\sigma = 32$ bit. Our best protocol ("Ours Iterative Combined") improves over the best previous protocol by factor 2.0x for $n = 2^{12}$ (over [26]), by factor 2.7x for $n = 2^{16}$ (over [39]), and by factor 3.2x for $n = 2^{20}$ (over [39]).

On the right side of Table 3 we compare the concrete circuit sizes for *arbitrary* bitlength σ. Our best protocol (Ours Iterative Combined) improves over the best previous protocol by factor 1.8x for $n = 2^{12}$ (over [26]), by factor 2.8x for $n = 2^{16}$ (over [26]), and by factor 3.8x for $n = 2^{20}$ (over [39]).

Our constructions always have smaller circuits than both former constructions, and, due to our better asymptotic size, the savings become greater as n increases.

Circuit Depths. For some protocols, the circuit depth is a relevant metric (e.g., for the GMW protocol the depth determines the round complexity of the online phase). Our constructions have the same depth as the Circuit-Phasing protocol of [39], i.e., $\log_2 \sigma$. This is much more efficient than the depth of the

Table 3. Concrete circuit sizes in #ANDs for PSI variants on n elements of fixed bitlength $\sigma = 32$ (left) and arbitrary bitlength hashed to $\sigma = 40 + 2\log_2(n) - 1$ bits (right).

Protocol	Fixed bitlength $\sigma = 32$			Arbitrary bitlength		
	$n = 2^{12}$	$n = 2^{16}$	$n = 2^{20}$	$n = 2^{12}$	$n = 2^{16}$	$n = 2^{20}$
Sort-Compare-Shuffle [26]	3 403 746	71 237 602	1 408 237 538	6 705 091	158 138 299	3 478 126 515
Circuit-Phasing [39]	4 254 256	55 155 466	688 258 388	10 501 475	181 928 305	3 201 695 060
Separate stashes $s' = 2s$						
Ours iterative separate	2 299 801	26 153 770	313 183 300	5 042 482	71 137 681	1 081 999 223
Combined stash s' (cf. Table 1)						
Ours iterative combined	1 664 921	20 058 922	215 665 732	3 772 722	57 375 121	836 632 439

Sort-Compare-Shuffle circuit of [26] which is $O(\log\sigma \cdot \log n)$ when using depth-optimized comparison circuits.

Further Optimizations. So far, we computed the comparisons with a Boolean circuit consisting of 2-input gates: For elements of bitlength ℓ, the circuit XORs the elements and afterwards computes a tree of $\ell - 1$ non-XOR gates s.t. the final output is 1 if the elements are equal or 0 otherwise. This circuit allows to use an arbitrary secure computation protocol based on Boolean gates, e.g., Yao or GMW. The recent approach of [14] shows that for security against semi-honest adversaries the communication can be improved by using multi-input lookup tables (LUTs). Their best LUT has 7 inputs and requires only 372 bits of total communication (cf. [14, Table 4]). For computing equality, 6 of the non-XOR gates in the tree can be combined into one 7-input LUT. This improves communication of the Circuit-Phasing protocol of [39] and our protocols by factor $6 \cdot 256/372 = 4.1$x.

6.3 Performance

We empirically compare the performance of our iterative 2D Cuckoo hashing PSI-CAT protocol with a combined stash described in Sect. 5.2 with the Circuit-Phasing PSI-CAT protocol of [39]. As a baseline, we also compare with the public key-based PSI-CA protocol of [9,35,46] that leaks the cardinality to one party, and the currently best specialized PSI protocol of [31] that cannot be easily modified to compute variants of the set intersection functionality.

Implementation. Pinkas et al. [39] provide the implementation of their Circuit-Phasing PSI protocol as part of the ABY framework [13]. This framework allows to securely evaluate the PSI circuit using either Yao's garbled circuit or the GMW protocol, both implemented with most recent optimizations (cf. Sect. 2). However, since the evaluation in [39] showed that using the GMW protocol yields much better run-times, we focus only on GMW. ABY also implements the LUT-based evaluation of [14] (cf. Sect. 6.2), which we compare to GMW evaluation.

For the Circuit-Phasing PSI-CAT protocol, we extended the existing codebase with the Hamming weight circuit of [5] and the depth-optimized comparison circuit of [45] to compare the Hamming weight with a threshold. Based on this, we implemented our iterative 2D Cuckoo hashing PSI-CAT protocol by duplicating the code for simple hashing and Cuckoo hashing, combining the stashes, and implementing the iterative insertion algorithm. Our implementation is available online as part of the ABY framework at http://encrypto.de/code/ABY. For the DH/ECC-based protocol of Shamir/Meadows/De Cristofaro et al. [9,35,46], we use the ECC-based implementation of [39] available online at http://encrypto.de/code/PSI that already supports computing the cardinality (PSI-CA). The implementation of the special purpose BaRK-OPRF PSI protocol of [31] is taken from https://github.com/osu-crypto/BaRK-OPRF.

Benchmarking Environment. For our benchmarks we use two machines, each equipped with an Intel Core i7-4790 CPU @ 3.6 GHz and 16 GB of RAM. The CPUs support the AES-NI instruction set for fast AES evaluations. We distinguish two network settings: a LAN setting and a WAN setting. For the LAN setting, we restrict the bandwidth of the network interfaces to 1 Gbit/s and enforce a round-trip time of 1 ms. For the WAN setting, we limit the bandwidth to 100 Mbit/s and set a round-trip time of 100 ms. We instantiate all protocols corresponding to a computational security parameter of 128 bit and a statistical security parameter of 40 bit. All reported run-times are the average of 10 executions with less than 10% variance.

Benchmarking Results. In Table 4, we give the run-times for $n \in \{2^{12}, 2^{16}, 2^{20}\}$ elements[4] of bitlength $\sigma = 32$ (suitable, e.g., for IPv4 addresses). The corresponding communication is given in Table 6. We do not use the LUT-based evaluation in the LAN setting since there is little need for better communication while the run-times are not competitive. However, to demonstrate the advantages of the LUT-based evaluation in the WAN setting, we compare the protocols when running with a single thread and four threads.[5]

Run-times (Tables 4 and 5). In comparison with the Circuit-Phasing PSI-CAT protocol of [39] in Table 4, our iterative combined PSI-CAT protocol is faster by factor 1.4x for $n = 2^{12}$ and up to factor 2.8x for $n = 2^{20}$. This holds when the circuit is evaluated with GMW in both network settings and for both 1 and 4 threads. With LUT-based evaluation [14], we observe a further improvement for the circuit-based protocols by about 13% in the WAN setting, but only for medium set sizes of $n = 2^{16}$ and 4 threads due to the higher computation complexity.

The circuit-based protocols have two steps: mapping the input items to the tables, and securely evaluating the circuit. The run-times of the hashing step are

[4] Unfortunately, the LUT-based implementation of [14] was not capable of evaluating the PSI circuits for $n = 2^{20}$ elements.

[5] We do not provide benchmarks with multiple threads for the DH/ECC PSI-CA protocol since the implementation of [39] does not support multi-threading.

shown in Table 5. The times for Cuckoo hashing into two tables in our PSI-CAT protocol are exactly twice of those for Cuckoo hashing into one table in [39]. Compared to simple hashing, our 2D Cuckoo hashing is slower by factor 1.6x up to factor 2.1x due to the additional iterations. However, all in all, the hashing procedures are by 2–3 orders of magnitude faster than the times for securely evaluating the circuit, and therefore negligible w.r.t. the overall run-time.

In comparison with the DH-based PSI-CA protocol of [9,35,46], our iterative combined PSI-CAT protocol is faster by factor 1.5x for $n = 2^{12}$ up to factor 91x for $n = 2^{20}$ in the LAN setting with a single thread. Also in the WAN setting with a single thread, our protocol is faster (except for small sets with $n = 2^{12}$), despite the substantially lower communication of the DH-based protocol described below. In both network settings even the best measured run-times of our PSI-CAT protocol are between 19x to 36x slower than the BaRK-OPRF specialized PSI protocol of [31], but our protocols are generic.

Communication (Table 6). The communication given in Table 6 is measured on the network interface, so these numbers are slightly larger than the theoretical communication (derived from the number of AND gates on the left side in Table 3) due to TCP/IP headers and padding of messages. The lowest communication is

Table 4. Total run-times in ms for PSI variants on n elements of bitlength $\sigma = 32$ bit.

Protocol	Network setting	LAN			WAN				
	Circuit evaluation protocol	GMW [21]			GMW [21]			LUT [14]	
	Set size n	2^{12}	2^{16}	2^{20}	2^{12}	2^{16}	2^{20}	2^{12}	2^{16}
DH/ECC PSI-CA [9,35,46]		3 296	49 010	7 904 054	4 082	51 866	8 008 771	4 082	51 866
BaRK-OPRF PSI [31]		113	295	3 882	540	1 247	14 604	540	1 247
1 Thread									
Circuit-Phasing PSI-CAT [39]		3 170	20 401	242 235	15 143	99 433	1 042 712	19 951	117 438
Ours iterative separate PSI-CAT		2 433	11 251	122 008	11 210	57 474	547 950	15 656	70 545
Ours iterative combined PSI-CAT		2 220	9 076	86 648	10 060	45 252	389 891	12 999	56 179
4 Threads									
Circuit-Phasing PSI-CAT [39]		2 333	10 600	123 765	12 492	97 480	987 459	15 471	76 184
Ours iterative separate PSI-CAT		1 903	6 273	64 324	9 361	56 141	541 677	11 946	46 797
Ours iterative combined PSI-CAT		1 694	5 177	49 417	8 793	44 596	376 591	9 413	39 272

Table 5. Run-times in ms for hashing n elements of bitlength $\sigma = 32$ bit.

Hashing procedure	Set size n	2^{12}	2^{16}	2^{20}
Circuit-Phasing PSI-CAT [39]				
Simple hashing		3.50	27.96	557.54
Cuckoo hashing		2.43	15.87	391.16
Ours iterative PSI-CAT				
2D Cuckoo hashing		6.23	58.90	873.19
Cuckoo hashing (for two tables with a combined stash)		4.85	31.75	782.32

Table 6. Communication in MB for PSI variants on n elements of bitlength $\sigma = 32$ bit.

Protocol Set size n	2^{12}	2^{16}	2^{20}
DH/ECC PSI-CA [9,35,46]	0.4	6.6	106.0
BaRK-OPRF PSI [31]	0.53	8.06	127.20
GMW [21]			
Circuit-Phasing PSI-CAT [39]	121.9	1 588.9	20 028.5
Ours iterative separate PSI-CAT	72.3	826.1	9 971.4
Ours iterative combined PSI-CAT	52.7	638.8	6 950.6
LUT [14]			
Circuit-Phasing PSI-CAT [39]	32.6	418.1	—
Ours iterative separate PSI-CAT	19.4	221.3	—
Ours iterative combined PSI-CAT	14.3	171.3	—

achieved by the DH-based PSI-CA protocol of [9,35,46] which is in line with the experiments in [39]. Our best protocol for PSI-CAT has between 132x (for $n = 2^{12}$) and 66x (for $n = 2^{20}$) more communication than the DH-based PSI-CA protocol when evaluated with GMW. Recall, however, that our protocol does not leak the cardinality. Our best protocol improves the communication over the PSI-CAT protocol of [39] by factor 2.3x (for $n = 2^{12}$) to 2.9x (for $n = 2^{20}$). When using LUT-based evaluation of [14], we observe that the communication of all circuit-based PSI-CAT protocols improves over GMW by factor 3.7x which is close to the theoretical upper bound of 4.1x (cf. Sect. 6.2). Still, our best LUT-based protocol has more than 20x higher communication than the BaRK-OPRF specialized PSI protocol of [31], but it is generic.

Application to privacy-preserving ridesharing. Our PSI-CAT protocol can easily be extended for the privacy-preserving ridesharing functionality of [23], where the intersection is revealed only if the size of the intersection is larger than a threshold. The authors of [23] give a protocol that securely computes this functionality, but has quadratic computation complexity. By slightly extending our circuit for PSI-CAT to encapsulate a key that is released only if the size of the intersection is larger than the threshold and using this key to symmetrically encrypt the last message in any linear complexity PSI protocol (e.g., [31,39,41,42]), we get a protocol with almost linear complexity. Our key encapsulation would take less than 3 s for $n = 2^{12}$ elements (cf. our results for PSI-CAT in Table 4), whereas the solution of [23] takes 5 627 s, i.e., we improve by factor 1 876x and also asymptotically.

Acknowledgments. We thank Oleksandr Tkachenko for his invaluable help with the implementation and benchmarking. We also thank Moni Naor for suggesting the application to achieve differential privacy. This work has been co-funded by the DFG as part of project E4 within the CRC 1119 CROSSING and by the German Federal

Ministry of Education and Research (BMBF), the Hessen State Ministry for Higher Education, Research and the Arts (HMWK) within CRISP, and 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. Calculations for this research were conducted on the Lichtenberg high performance computer of the TU Darmstadt.

References

1. Amossen, R.R., Pagh, R.: A new data layout for set intersection on GPUs. In: International Symposium on Parallel and Distributed Processing (IPDPS) (2011)
2. Arbitman, Y., Naor, M., Segev, G.: Backyard cuckoo hashing: constant worst-case operations with a succinct representation. In: FOCS (2010)
3. Asharov, G., Lindell, Y., Schneider, T., Zohner, M.: More efficient oblivious transfer and extensions for faster secure computation. In: CCS (2013)
4. Asokan, N., Dmitrienko, A., Nagy, M., Reshetova, E., Sadeghi, A.-R., Schneider, T., Stelle, S.: CrowdShare: secure mobile resource sharing. In: Jacobson, M., Locasto, M., Mohassel, P., Safavi-Naini, R. (eds.) ACNS 2013. LNCS, vol. 7954, pp. 432–440. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38980-1_27
5. Boyar, J., Peralta, R.: Concrete multiplicative complexity of symmetric functions. In: Královič, R., Urzyczyn, P. (eds.) MFCS 2006. LNCS, vol. 4162, pp. 179–189. Springer, Heidelberg (2006). https://doi.org/10.1007/11821069_16
6. Chen, H., Laine, K., Rindal, P.: Fast private set intersection from homomorphic encryption. In: CCS (2017)
7. Dachman-Soled, D., Malkin, T., Raykova, M., Yung, M.: Efficient robust private set intersection. In: Abdalla, M., Pointcheval, D., Fouque, P.-A., Vergnaud, D. (eds.) ACNS 2009. LNCS, vol. 5536, pp. 125–142. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-01957-9_8
8. Davidson, A., Cid, C.: An efficient toolkit for computing private set operations. In: Pieprzyk, J., Suriadi, S. (eds.) ACISP 2017. LNCS, vol. 10343, pp. 261–278. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-59870-3_15
9. De Cristofaro, E., Gasti, P., Tsudik, G.: Fast and private computation of cardinality of set intersection and union. In: Pieprzyk, J., Sadeghi, A.-R., Manulis, M. (eds.) CANS 2012. LNCS, vol. 7712, pp. 218–231. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-35404-5_17
10. De Cristofaro, E., Kim, J., Tsudik, G.: Linear-complexity private set intersection protocols secure in malicious model. In: Abe, M. (ed.) ASIACRYPT 2010. LNCS, vol. 6477, pp. 213–231. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17373-8_13
11. De Cristofaro, E., Tsudik, G.: Practical private set intersection protocols with linear complexity. In: Sion, R. (ed.) FC 2010. LNCS, vol. 6052, pp. 143–159. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14577-3_13
12. Debnath, S.K., Dutta, R.: Secure and efficient private set intersection cardinality using bloom filter. In: Lopez, J., Mitchell, C.J. (eds.) ISC 2015. LNCS, vol. 9290, pp. 209–226. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23318-5_12
13. Demmler, D., Schneider, T., Zohner, M.: ABY – a framework for efficient mixed-protocol secure two-party computation. In: NDSS (2015)
14. Dessouky, G., Koushanfar, F., Sadeghi, A.-R., Schneider, T., Zeitouni, S., Zohner, M.: Pushing the communication barrier in secure computation using lookup tables. In: NDSS (2017)

15. Dietzfelbinger, M., Weidling, C.: Balanced allocation and dictionaries with tightly packed constant size bins. Theoret. Comput. Sci. **380**(1–2), 47–68 (2007)
16. Dong, C., Chen, L., Wen, Z.: When private set intersection meets big data: an efficient and scalable protocol. In: CCS (2013)
17. Dwork, C.: Differential privacy. In: Bugliesi, M., Preneel, B., Sassone, V., Wegener, I. (eds.) ICALP 2006. LNCS, vol. 4052, pp. 1–12. Springer, Heidelberg (2006). https://doi.org/10.1007/11787006_1
18. Eppstein, D., Goodrich, M., Mitzenmacher, M., Torres, M.: 2–3 cuckoo filters for faster triangle listing and set intersection. In: Symposium on Principles of Database Systems (PODS) (2017)
19. Freedman, M.J., Hazay, C., Nissim, K., Pinkas, B.: Efficient set intersection with simulation-based security. J. Cryptol. **29**(1), 115–155 (2016)
20. Freedman, M.J., Nissim, K., Pinkas, B.: Efficient private matching and set intersection. In: Cachin, C., Camenisch, J.L. (eds.) EUROCRYPT 2004. LNCS, vol. 3027, pp. 1–19. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24676-3_1
21. Goldreich, O., Micali, S., Wigderson, A.: How to play any mental game or a completeness theorem for protocols with honest majority. In: STOC (1987)
22. Gonnet, G.H.: Expected length of the longest probe sequence in hash code searching. J. ACM **28**(2), 289–304 (1981)
23. Hallgren, P., Orlandi, C., Sabelfeld, A.: PrivatePool: privacy-preserving ridesharing. In: Computer Security Foundations Symposium (CSF) (2017)
24. Hazay, C., Venkitasubramaniam, M.: Scalable multi-party private set-intersection. In: Fehr, S. (ed.) PKC 2017. LNCS, vol. 10174, pp. 175–203. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54365-8_8
25. Huang, Y., Chapman, P., Evans, D.: Privacy-preserving applications on smartphones. In: Hot Topics in Security (HotSec) (2011)
26. Huang, Y., Evans, D., Katz, J.: Private set intersection: Are garbled circuits better than custom protocols? In: NDSS (2012)
27. Huang, Y., Evans, D., Katz, J., Malka, L.: Faster secure two-party computation using garbled circuits. In: USENIX Security (2011)
28. Ion, M., Kreuter, B., Nergiz, E., Patel, S., Saxena, S., Seth, K., Shanahan, D., Yung, M.: Private intersection-sum protocol with applications to attributing aggregate ad conversions. Cryptology ePrint Archive, Report 2017/738 (2017)
29. Kirsch, A., Mitzenmacher, M., Wieder, U.: More robust hashing: cuckoo hashing with a stash. SIAM J. Comput. **39**(4), 1543–1561 (2009)
30. Kiss, Á., Liu, J., Schneider, T., Asokan, N., Pinkas, B.: Private set intersection for unequal set sizes with mobile applications. In: PoPETs, vol. 2017(4) (2017)
31. Kolesnikov, V., Kumaresan, R., Rosulek, M., Trieu, N.: Efficient batched oblivious PRF with applications to private set intersection. In: CCS (2016)
32. Kolesnikov, V., Matania, N., Pinkas, B., Rosulek, M., Trieu, N.: Practical multi-party private set intersection from symmetric-key techniques. In: CCS (2017)
33. Kolesnikov, V., Schneider, T.: Improved garbled circuit: free XOR gates and applications. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008. LNCS, vol. 5126, pp. 486–498. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70583-3_40
34. Kreuter, B.: Secure multiparty computation at Google. In: Real World Crypto Conference (RWC) (2017)
35. Meadows, C.: A more efficient cryptographic matchmaking protocol for use in the absence of a continuously available third party. In: S&P (1986)

36. Pagh, R., Rodler, F.F.: Cuckoo hashing. In: European Symposium on Algorithms (ESA) (2001)
37. Panigrahy, R.: Efficient hashing with lookups in two memory accesses. In: ACM-SIAM Symposium on Discrete Algorithms (SODA) (2005)
38. Pettai, M., Laud, P.: Combining differential privacy and secure multiparty computation. In: ACSAC (2015)
39. Pinkas, B., Schneider, T., Segev, G., Zohner, M.: Phasing: private set intersection using permutation-based hashing. In: USENIX Security (2015)
40. Pinkas, B., Schneider, T., Weinert, C., Wieder, U.: Efficient circuit-based PSI via cuckoo hashing. In: Cryptology ePrint Archive, Report 2018/120 (2018)
41. Pinkas, B., Schneider, T., Zohner, M.: Faster private set intersection based on OT extension. In: USENIX Security (2014)
42. Pinkas, B., Schneider, T., Zohner, M.: Scalable private set intersection based on OT extension. ACM Trans. Priv. Secur. (TOPS) **21**(2) (2018)
43. Rindal, P., Rosulek, M.: Improved private set intersection against malicious adversaries. In: Coron, J.-S., Nielsen, J.B. (eds.) EUROCRYPT 2017. LNCS, vol. 10210, pp. 235–259. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-56620-7_9
44. Rindal, P., Rosulek, M.: Malicious-secure private set intersection via dual execution. In: CCS (2017)
45. Schneider, T., Zohner, M.: GMW vs. Yao? Efficient secure two-party computation with low depth circuits. In: Sadeghi, A.-R. (ed.) FC 2013. LNCS, vol. 7859, pp. 275–292. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39884-1_23
46. Shamir, A.: On the power of commutativity in cryptography. In: de Bakker, J., van Leeuwen, J. (eds.) ICALP 1980. LNCS, vol. 85, pp. 582–595. Springer, Heidelberg (1980). https://doi.org/10.1007/3-540-10003-2_100
47. Wieder, U.: Hashing, load balancing and multiple choice. Found. Trends Theoret. Comput. Sci. **12**(3–4), 275–379 (2017)
48. Yao, A.C.: How to generate and exchange secrets. In: FOCS (1986)
49. Yung, M.: From mental poker to core business: why and how to deploy secure computation protocols? In: CCS (2015)
50. Zahur, S., Rosulek, M., Evans, D.: Two halves make a whole: reducing data transfer in garbled circuits using half gates. In: Oswald, E., Fischlin, M. (eds.) EUROCRYPT 2015. LNCS, vol. 9057, pp. 220–250. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46803-6_8