Phasing: Private Set Intersection using Permutation-based Hashing

Benny Pinkas, *Bar-Ilan University;* **Thomas Schneider,** *Technische Universität Darmstadt;*
Gil Segev, *The Hebrew University of Jerusalem;* **Michael Zohner,**
Technische Universität Darmstadt

https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/pinkas

This paper is included in the Proceedings of the 24th USENIX Security Symposium

August 12–14, 2015 • Washington, D.C.

ISBN 978-1-939133-11-3

Open access to the Proceedings of the 24th USENIX Security Symposium is sponsored by USENIX

Phasing: Private Set Intersection using Permutation-based Hashing

Benny Pinkas
Bar-Ilan University, Israel
benny@pinkas.net

Thomas Schneider
TU Darmstadt, Germany
thomas.schneider@ec-spride.de

Gil Segev
Hebrew University, Israel
segev@cs.huji.ac.il

Michael Zohner
TU Darmstadt, Germany
michael.zohner@ec-spride.de

Abstract

Private Set Intersection (PSI) allows two parties to compute the intersection of private sets while revealing nothing more than the intersection itself. PSI needs to be applied to large data sets in scenarios such as measurement of ad conversion rates, data sharing, or contact discovery. Existing PSI protocols do not scale up well, and therefore some applications use insecure solutions instead.

We describe a new approach for designing PSI protocols based on permutation-based hashing, which enables to reduce the length of items mapped to bins while ensuring that no collisions occur. We denote this approach as Phasing, for Permutation-based Hashing Set Intersection. Phasing can dramatically improve the performance of PSI protocols whose overhead depends on the length of the representations of input items.

We apply Phasing to design a new approach for circuit-based PSI protocols. The resulting protocol is up to 5 times faster than the previously best Sort-Compare-Shuffle circuit of Huang et al. (NDSS 2012). We also apply Phasing to the OT-based PSI protocol of Pinkas et al. (USENIX Security 2014), which is the fastest PSI protocol to date. Together with additional improvements that reduce the computation complexity by a logarithmic factor, the resulting protocol improves run-time by a factor of up to 20 and can also have similar communication overhead as the previously best PSI protocol in that respect. The new protocol is only moderately less efficient than an *insecure* PSI protocol that is currently used by real-world applications, and is therefore the first secure PSI protocol that is scalable to the demands and the constraints of current real-world settings.

1 Introduction

Private set intersection (PSI) allows two parties P_1 and P_2 with respective input sets X and Y to compute the intersection $X \cap Y$ of their sets without revealing any information but the intersection itself. Although PSI has been widely studied in the literature, many real-world applications today use an insecure hash-based protocol instead of a secure PSI protocol, mainly because of the insufficient efficiency of current PSI protocols.

In this work we present *Phasing*, Permutation-based Hashing Set Intersection, which is a new approach for constructing PSI protocols based on a hashing technique that ensures that hashed elements can be represented by short strings without any collisions. The overhead of recent PSI protocols depends on the length of these representations, and this new structure of construction, together with other improvements, results in very efficient performance that is only moderately larger than that of the *insecure* protocol that is in current real-world usage.

1.1 Motivating Scenarios

The motivation for this work comes from scenarios where PSI must be applied quite frequently to large sets of data, and therefore performance becomes critical. Moreover, the communication overhead might be even more important than the computation overhead, since in large data centers it is often easier to add computing power than to improve the outgoing communication infrastructure. We describe here three scenarios which require large-scale PSI implementations.

Measuring ad conversion rates Online advertising, which is a huge business, typically measures the success of ad campaigns by measuring the success of converting viewers into customers. A popular way of measuring this value is by computing the conversion rate, which is the percentage of ad viewers who later visit the advertised site or perform a transaction there. For banner ads or services like Google Adwords it is easy to approximate this value by measuring ad click-throughs. However, measuring click-throughs is insufficient in other online advertising settings. One such setting is *mobile advertising*, which is becoming a dominating part of online ad-

vertising. Even though mobile ads have a great effect, click-throughs are an insufficient measure of their utility, since it is unlikely, due to small displays and the casual nature of mobile browsing, that a user will click on an ad and, say, purchase a car using his mobile device. Another setting where click rate measurement is unsatisfactory is advertising of offline goods, like groceries, where the purchase itself is done offline.[1]

An alternative method of measuring ad performance is to compare 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. It is often possible to identify users on both ends, using identifiers such as credit card numbers, email addresses, etc. A simple solution, which ignores privacy, is for one side to disclose its list of customers to the other side, which then computes the necessary statistics. Another option is to run a PSI protocol between the two parties. (The protocol should probably be a variant of PSI, e.g. compute total revenues from customers who have seen an ad. Such protocols can be derived from basic PSI protocols.) In fact, Facebook is running a service of this type with Datalogix, Epsilon and Acxiom, companies which have transaction records for a large part of loyalty card holders in the US. According to reports[2], the computation is done using a variant of the *insecure* naive hashing PSI protocol that we describe in §3.1. Our results show that it can be computed using secure protocols even for large data sets.

Security incident information sharing Security incident handlers can benefit from information sharing since it provides them with a global view during incidents. However, incident data is often sensitive and potentially embarrassing. The shared information might reveal information about the business of the company that provided it, or of its customers. Therefore, information is typically shared rather sparsely and protected using legal agreements. Automated large scale sharing will improve security, and there is in fact work to that end, such as the IETF Managed Incident Lightweight Exchange (MILE) effort. Many computations that are applied to the shared data compute the intersection and its variants. Applying PSI to perform these computations can simplify the legal issues of information sharing. Efficient PSI protocols will enable it to be run often and in large scale.

Private contact discovery When a new user registers to a service it is often essential to identify current registered users who are also contacts of the new user. This operation can be done by simply revealing the user's contact list to the service, but can also be done in a privacy preserving manner by running a PSI protocol between the user's contact list and the registered users of the service. This latter approach is used by the TextSecure and Secret applications, but for performance reasons they use the insecure naive hashing PSI protocol described in §3.1.[3]

In these cases each user has a small number of records n_2, e.g., $n_2 = 256$, whereas the service has millions of registered users (in our experiments we use $n_1 = 2^{24}$). It therefore holds that $n_2 \ll n_1$. In our best PSI protocol, the client needs only $O(n_2 \log n_1)$ memory, $O(n_2)$ symmetric cryptographic operations and $O(n_1)$ cheap hash table lookups, and the communication is $O(n_1 \log n_1)$. (The communication overhead is indeed high as it depends on n_1, but this seems inevitable if brute force searches are to be prevented.)

1.2 Our Contributions

Our goal in this work is to enable PSI computations for large scale sets that were previously beyond the capabilities of state-of-the-art protocols. The constructions that we design in this work improve performance by more than an order of magnitude. We obtain these improvements by generalizing the hashing approach of [22] and applying it to generic secure computation-based PSI protocols. We replace the hash function in [22] by a permutation which enables us to reduce the bit-length of internal representations. Moreover, we suggest several improvements to the OT-based PSI protocol of [22]. We explain our contributions in more detail next:

Phasing: Using permutation-based hashing to reduce the bit-length of representations. The overhead of the best current PSI protocol [22] is linear in the length of the representations of items in the sets (i.e., the ids of items in the sets). The protocol maps items into bins, and since each bin has very few items in it, it is tempting to hash the ids to shorter values and trust the birthday paradox to ensure that no two items in the same bin are hashed to the same representation. However, a closer examination shows that to ensure that the collision probability is smaller than $2^{-\lambda}$, the length of the representation must be at least λ bits, which is too long.

In this work we utilize the permutation-based hashing techniques of [1] to reduce the bit-length of the ids of items that are mapped to bins. These ideas were suggested in an algorithmic setting to reduce memory us-

[1] See, e.g., `http://www.reuters.com/article/2012/10/01/us-facebook-ads-idUSBRE8900I120121001` .

[2] See, e.g., `https://www.eff.org/deeplinks/2012/09/deep-dive-facebook-and-datalogix-whats-actually-getting-shared-and-how-you-can-opt`.

[3] See `https://whispersystems.org/blog/contact-discovery/` and `https://medium.com/@davidbyttow/demystifying-secret-12ab82fda29f` , respectively.

age, and as far as we know this is the first time that they are used in a cryptographic or security setting to improve performance. Essentially, when using β bins the first $\log\beta$ bits in an item's hashed representation define the bin to which the item is mapped, and the other bits are used in a way which provably prevents collisions. This approach reduces the bit-length of the values used in the PSI protocol by $\log\beta$ bits, and this yields reduced overhead by up to 60%-75% for the settings we examined.

Circuit-Phasing: Improved circuit-based PSI. As we discuss in §3.4 there is a great advantage in using generic secure computation for computing PSI, since this enables to easily compute variants of the basic PSI functionality. Generic secure computation protocols evaluate Boolean circuits computing the desired functionality. The best known circuit for computing PSI was based on the Sort-Compare-Shuffle circuit of [12]. We describe Circuit-Phasing, a new generic protocol that uses hashing (specifically, Cuckoo hashing and simple hashing) and secure circuit evaluation. In comparison with the previous approach, our circuits have a smaller number of AND gates, a lower depth of the circuit (which affects the number of communication rounds in some protocols), and a much smaller memory footprint. These factors lead to a significantly better performance.

OT-Phasing: Improved OT-based PSI. We introduce the OT-Phasing protocol which improves the OT-based PSI protocol of [22] as follows:

- **Improved computation and memory.** We reduce the length of the strings that are processed in the OT from $\mathsf{O}(\log^2 n)$ to $\mathsf{O}(\log n)$, which results in a reduction of computation and memory complexity for the client from $\mathsf{O}(n\log^2 n)$ to $\mathsf{O}(n\log n)$.
- **3-way Cuckoo hashing.** We use 3 instead of 2 hash functions to generate a more densely populated Cuckoo table and thus decrease the overall number of bins and hence OTs.

OT-Phasing improves over state-of-the-art PSI both in terms of run-time and communication. Compared to the previously fastest PSI protocol of [22], our protocol improves run-time by up to factor 10 in the WAN setting and by up to factor 20 in the LAN setting. Furthermore, our OT-Phasing protocol in some cases achieves similar communication as [18], which was shown to achieve the lowest communication of all PSI protocols [22].

1.3 Outline

We give preliminary information in §2 and summarize related work in §3. In §4 we describe Phasing, our optimization for permutation-based hashing that reduces the bit-length of elements in PSI. Afterwards, we apply Phasing to generic secure computation protocols, and present Circuit-Phasing, our new approach for circuit-based PSI §5. Thereafter, we apply Phasing to the previously fastest OT-based PSI protocol of [22] and present several optimizations in §6. In §7 we analyze the hashing failure probability of Circuit- and OT-Phasing. Finally, we provide an evaluation of our PSI protocols in §8.

2 Preliminaries

2.1 Notation

We denote the parties as P_1 and P_2. For all protocols we assume that P_2 obtains the output. The respective input sets are denoted as X and Y, with sizes $n_1 = |X|$ and $n_2 = |Y|$. Often $n_1 = n_2$ and we use the notation $n = n_1 = n_2$. We assume that elements are of bit-length σ.

We call the symmetric security parameter κ, the bit-length of the elliptic curves φ, and the statistical security parameter λ. Throughout the paper we assume 128-bit security, i.e., $\kappa = 128$, $\varphi = 283$ (using Koblitz-curves), and $\lambda = 40$. For symmetric encryption we use AES-128.

We refer to the concatenation of bit-strings by $||$, to the exclusive-OR (XOR) operation by $\oplus$, and to the i-th element in a sequence S by $S[i]$. In many protocols, we shorten the size of hash values that are sent to $\ell = \lambda + \log_2(n_1) + \log_2(n_2)$ instead of 2κ. This yields collision probability $2^{-\lambda}$, which is suited for most applications.

2.2 Security

Two types of adversaries are typically discussed in the secure computation literature: A *semi-honest* adversary is trusted to follow the protocol, but attempts to learn as much information as possible from the messages it receives. This adversary model is appropriate for scenarios where execution of the correct software is enforced by software attestation or where an attacker might obtain the transcript of the protocol *after* its execution, either by stealing it or by legally enforcing its disclosure. In contrast, a *malicious* adversary can behave arbitrarily. Most work on PSI was in the semi-honest setting. Protocols that are secure against malicious adversaries, e.g., [9, 10, 14], are considerably less efficient. We focus on optimal performance and therefore design protocols secure against semi-honest adversaries only. Furthermore, the security of the protocols is proven in the random oracle model, as is justified in the full version [21].

2.3 Hashing to Bins

Our protocols hash the input items to bins and then operate on each bin separately. In general, our hashing

schemes use a table T consisting of β bins. An element e is mapped to the table by computing an address $a = H(e)$ using a hash function H that is modeled as a random function. A value related to e is then stored in bin $T[a]$.

There is a rich literature on hashing schemes, which differ in the methods for coping with collisions, the complexity for insertion/deletion/look-up, and the utilization of storage space. In [9, 10, 22], hashing to bins was used to improve the number of comparisons that are performed in PSI protocols. In the following, we detail the two most promising hashing schemes for use in PSI, according to [22]: *simple hashing* and *Cuckoo hashing*. For the OT-based PSI protocol of [22] it was shown that a combination of simple hashing (for P_1) and Cuckoo hashing (for P_2) results in the best performance.

2.3.1 Simple Hashing

Simple hashing builds the table T by mapping each element e to bin $T[H(e)]$ and appending e to the bin. Each bin must, of course, be able to store more than one element. The size of the most populated bin was analyzed in [23], and depends on the relation between the number of bins and the total number of elements. Most importantly for our application, when hashing n elements into $\beta = n$ bins, it was shown that the maximum number of elements in a bin is $\frac{\ln n}{\ln \ln n}(1 + o(1))$. In §7.1 we give a theoretical and an empirical analysis of the maximum number of elements in a bin.

2.3.2 Cuckoo Hashing

Cuckoo hashing [19] uses h hash functions $H_1, ..., H_h$ to map an element e to a bin using either one of the h hash functions. (Typically, h is set to be $h = 2$; we also use $h = 3$.) In contrast to simple hashing, it allows at most one element to be stored in a bin. If a collision occurs, Cuckoo hashing evicts the element in the bin and performs the insertion again for the evicted element. This process is repeated until an empty bin is found for the evicted element. If the resulting sequence of insertion attempts fails a certain number of times, the current evicted element is placed in a special bin called stash. In [16] it was shown that for $h = 2$ hash functions, $\beta = 2(1+\varepsilon)n$ bins, and a stash of size $s \leq \ln n$, the insertion of elements fails with small probability of $O(n^{-s})$, which is smaller than $n^{-(s-1)}$ for sufficiently large values of n (cf. §7.2).

2.4 Oblivious Transfer

1-ouf-of-2 oblivious transfer (OT) [8] is a protocol where the receiver with choice bit c, chooses one of two strings (x_0, x_1) held by the sender. The receiver receives x_c but gains no information about x_{1-c}, while the sender gains no information about c.

OT extension protocols [2, 17] precompute a small number (say, $\kappa = 128$) of "real" public-key-based OTs, and then compute any polynomial number of OTs using symmetric-key cryptography alone. The most efficient OT variant that we use computes *random OT*. In that protocol the sender has no input but obtains random (x_0, x_1) as output, while the receiver with input c obtains x_c [2]. The advantage of this protocol is that the sender does not need to send messages based on its inputs, as it does not have any inputs, and instead computes them on-the-fly during the OT extension protocol. As a result, the communication overhead of the protocol is greatly reduced.

An additional improvement that we use, described in [17], efficiently computes 1-out-of-N OT for short strings. The communication for a random 1-out-of-N OT (for $3 \leq N \leq 256$) is only 2κ-bits, whereas the communication for a random 1-out-of-2 OT is κ-bits. The computation for a random 1-out-of-N OT amounts to four pseudo-random generator (PRG) and one correlation-robust function (CRF) evaluations for the receiver and two PRG and N CRF evaluations for the sender. In addition, if the sender only requires $i \leq N$ outputs of the OT, it only needs to perform i CRF evaluations.

We use 1-out-of-N OT since we have to perform OTs for every bit of an element. By using 1-out-of-N OT for $N = 2^\mu$, we process μ bits in parallel with communication equal to that of processing two bits. We denote m 1-out-of-N OTs on ℓ-bit strings by $\binom{N}{1}$-OT_ℓ^m.

2.5 Generic Secure Computation

Generic secure two-party computation protocols allow two parties to securely evaluate any function that can be expressed as a Boolean circuit. The communication overhead and the number of cryptographic operations that are computed are linear in the number of non-linear (AND) gates in the circuit, since linear (XOR) gates can be evaluated "for free" in current protocols. Furthermore, some protocols require a number of interaction rounds that are linear in the AND depth of the circuit. The two main approaches for generic secure two-party computation on Boolean circuits are *Yao's garbled circuits* [25] and the protocol by *Goldreich-Micali-Wigderson* [11]. We give a summary of these protocols in the full version [21].

3 Related Work

We reflect on existing PSI protocols by following the classification of PSI protocols in [22]: the *naive hashing* protocol (§3.1), *server-aided* PSI protocols (§3.2), *public-key cryptography-based* PSI protocols (§3.3), *generic secure computation-based* PSI protocols (§3.4), and *OT-based* PSI protocols (§3.5). For each category,

we review existing work and outline the best performing protocol, according to [22].

3.1 (Insecure) Naive Hashing

In the naive hashing protocol, detailed in the full version [21], P_1 permutes and hashes its elements, and sends the results to P_2 which compares these values to the hashes of its elements. This approach is very efficient and is currently employed in practice, but it allows P_2 to brute-force the elements of P_1 if they do not have high entropy. Furthermore, even if inputs elements have high entropy, forward-secrecy is not provided since P_2 can check at any later time whether an element was in X.

3.2 Server-Aided PSI

To increase the efficiency of PSI, protocols that use a semi-trusted third party were proposed [15]. These protocols are secure as long as the third party does not collude with any of the participants. We mention this set of protocols here for completeness, as they require different trust assumptions as protocols involving no third party.

The protocol of [15] has only a slightly higher overhead than the naive hashing PSI solution described in §3.1. In that protocol, P_1 samples a random κ-bit key k and sends it to P_2. Both parties compute $h_i = F_k(x_i)$ (resp. $h'_j = F_k(y_j)$), where F_k is a pseudo-random permutation that is parametrized by k. Both parties then send the hashes to the third party (in randomly permuted order) who then computes $I = h_i \cap h'_j$, for all $1 \leq i \leq n_1$ and $1 \leq j \leq n_2$ and sends I to P_2. P_2 obtains the intersection by computing $F_k^{-1}(e)$ for each $e \in I$.

3.3 Public-Key Cryptography based PSI

The first protocols for PSI were outlined in [13, 18] and were based on the Diffie-Hellmann (DH) key exchange. The overhead of these protocols is $\mathsf{O}(n)$ exponentiations. In [9, 10], a PSI protocol based on El-Gamal encryption was introduced that uses oblivious polynomial evaluation and requires $\mathsf{O}(n \log\log(n))$ public-key encryptions (the advantage of that protocol was that its security was not based on the random oracle model). A PSI protocol that uses blind-RSA was introduced in [3].

We implement the DH-based protocol of [13, 18] based on elliptic-curve-cryptography, which was shown to achieve lowest communication in [22]. We describe the protocol in the full version [21].

3.4 PSI based on Generic Protocols

Generic secure computation can be used to perform PSI by encoding the intersection functionality as a Boolean circuit. The most straightforward method for this encoding is to perform a pairwise-comparison which compares each element of one party to all elements of the other party. However, this circuit uses $\mathsf{O}(n^2)$ comparisons and hence scales very poorly for larger set sizes [12]. The *Sort-Compare-Shuffle (SCS)* circuit of [12] is much more efficient. As indicated by its name, the circuit first *sorts* the union of the elements of both parties, then *compares* adjacent elements for equality, and finally *shuffles* the result to avoid information leakage. The sort and shuffle operations are implemented using a sorting network of only $\mathsf{O}(n \log n)$ comparisons, and the comparison step requires only $\mathsf{O}(n)$ comparisons.

The work of [12] describes a size-optimized version of this circuit for use in Yao's garbled circuits; [22] describes a depth-optimized version for use in the GMW protocol. The size-optimized SCS circuit has $\sigma(3n\log_2 n + 4n)$ AND gates[4] and AND depth $(\sigma + 2)\log_2(2n) + \log_2(\sigma) + 1$ while the depth-optimized SCS circuit has about the same number of gates and AND depth of $(\log_2(\sigma) + 4)\log_2(2n)$, for $n = (n_1 + n_2)/2$.

PSI protocols based on generic secure computation have higher run-time and communication complexity than most special-purpose PSI protocols [4, 22]. Yet, these protocols are of great importance since they enable to easily compute any functionality that is based on basic PSI. Consider, for example, an application that needs to find if the size of the intersection is greater than some threshold, or compute the sum of revenues from items in the intersection. Computing these functionalities using specialized PSI protocols requires to change the protocols, whereas a PSI protocol based on generic computation can be adapted to compute these functionalities by using a slightly modified circuit. In other words, changing specialized protocols to have a new functionality requires to employ a cryptographer to design a new protocol variant, whereas changing the functionality of a generic protocol only requires to design a new circuit computing the new functionality. The latter task is of course much simpler. An approximate PSI protocol that uses generic secure computation protocols in combination with Bloom filters was given in [24].

3.5 OT-based PSI

OT-based PSI protocols are the most recent category of PSI protocols. Their research has been motivated by recent efficiency improvements in OT extension. The garbled Bloom filter protocol of [7] was the first OT-based PSI protocol and was improved in [22]. A novel OT-based PSI protocol, which we denote *OT-PSI* protocol,

[4] The original description of the SCS circuit in [12] embedded input keys into AND gates in the sort circuit to reduce communication. We did not use this optimization in our implementation.

was introduced in [22], combining OT and hashing to achieve the best run-time among all analyzed PSI protocols. We next summarize the OT-PSI protocol of [22] and give a detailed description in the full version [21].

The abstract idea of the OT-PSI protocol is to have both parties hash their elements into bins using the same hash function (Step 1, cf. §3.5.1) and compare the elements mapped to the same bin. The comparison is done using OTs that generate random masks from the elements (Step 2, cf. §3.5.2), such that the intersection of the random masks corresponds to the intersection of the original inputs (Step 3, cf. §3.5.3). Finally, the intersection of the elements in the stash is computed (§3.5.4). We give the overhead of the protocol in §3.5.5.

3.5.1 PSI via Hashing to Bins

In the first step of the protocol, the parties map their elements into their respective hash tables T_1 and T_2, consisting of $\beta = h(1+\varepsilon)n_2$ bins (cf. §7). P_2 uses Cuckoo hashing with h hash functions (with $h = 2$), and obtains a one-dimensional hash table T_2. P_1 hashes each item h times (once for each hash function) using *simple hashing* and obtains a two-dimensional hash table T_1 (where the first dimension addresses the bin and the second dimension the elements in the bin). Each party then pads all bins in its table to the maximum size using respective dummy elements: P_1 pads each bin to max_β elements using a dummy element d_1 (where max_β is computed using β and n_1 as detailed in §7 to set the probability of mapping more items to a bin to be negligible), while P_2 fills each empty bin with dummy element d_2 (different than d_1). The padding is performed to hide the number of elements that were mapped to a specific bin, which would leak information about the input.

3.5.2 Masking via OT

After the hashing, the parties use OT to generate an ℓ-bit random mask for each element in their hash table.

Naively, for each bin, and for each item that P_2 mapped to the bin, the parties run a 1-out-of-2 OT for each bit of this item. P_2 is the receiver and its input to the OT is the value of the corresponding bit in the single item that it mapped to the bin. P_1's input is two random ℓ-bit strings. After running these OTs for all σ bits of the item, P_1 sends to P_2 the XOR of the strings corresponding to the bits of P_1's item. Note that if P_1's item is equal to that of P_2 then the sent value is equal to the XOR of the output strings that P_2 received in the OTs. Otherwise the values are different with high probability, which depends on the length ℓ of the output strings.

This basic protocol was improved upon in [22] in several ways:

- Recall that OT extension is more efficient when applied to 1-out-of-N OT [17]. Therefore, the protocol uses μ-bit characters instead of a binary representation. It splits the elements into t μ-bit characters, and uses t invocations of 1-out-of-N OT where $N = 2^\mu$, instead of $t\mu$ invocations of 1-out-of-2 OT.
- In each bin the parties run OTs for all max_β items that P_1 mapped to the bin, and to all characters in these items. P_2's inputs are the same for all max_β OTs corresponding to the same character. Thus, the parties could replace them with a single OT, where the output string of the OT has max_β longer size.
- Recall that random OT, where the protocol randomly defines the inputs of P_1, is more efficient than an OT where P_1 chooses these inputs by itself. For the purpose of PSI the protocol can use random OT. It is also important to note that if P_1 mapped $m < max_\beta$ elements to a bin, it only needs to evaluate inputs for m random OTs in this bin and not for all max_β random OTs that are taking place. This improves the overhead of the protocol.

3.5.3 Intersection

The parties compute the intersection of their elements using the random masks (XOR values) generated during Step 2: P_1 generates a set V as the masks for all of its non-dummy elements. P_1 then randomly permutes the set V to hide information about the number of elements in each bin, and sends V to P_2. P_2 computes the intersection $X \cap Y$ by computing the plaintext intersection between V and the set of XOR values that it computed.

3.5.4 Including a Stash

The OT-based PSI protocol of [22] uses Cuckoo hashing with a stash of size s. The intersection of P_2's elements with P_1's elements is done by running the masking procedure of Step 2 for all s items in the stash, comparing them with all n_1 items in P_1's input. Finally, P_1 sends the masks it computed to P_2 (in randomly permuted order) which can then check the intersection as in Step 3.

3.5.5 Overhead

The overhead of this protocol is linear in the bit-length of the input elements. Therefore, any reduction in the bit-length of the inputs directly results in a similar improvement in the overhead.

For readers interested in the exact overhead of the protocol, we describe here the details of the overhead. In total, the parties have to evaluate random $\binom{N}{1}$-$\mathrm{OT}^{\beta t}_{max_\beta \ell}$ +$\binom{N}{1}$-$\mathrm{OT}^{st}_{n_1 \ell}$ and send $(h+s)n_1$ masks of ℓ-bit length, where $\beta = h(n_2+\varepsilon)$, $N = 2^\mu$, $t = \lceil \sigma/\mu \rceil$, $\ell = \lambda +$

$\log_2(n_1)+\log_2(n_2)$, and s is the size of the stash. To be exact, the server has to perform $2t(\beta+s)$ pseudo-random generator evaluations during OT extension, $(h+s)n_1t$ correlation-robust function evaluations to generate the random masks, and send $(2+s)n_1\ell$ bits. The client has to perform $4t(\beta+s)$ pseudo-random generator evaluations during OT extension, $n_2 t max_\beta \ell/o + sn_1t\ell/o$ correlation-robust function evaluations to generate the random masks, and send $2(\beta+s)t\kappa$ bits during OT extension, where o is the output length of the correlation-robust function. Note especially that the client has to evaluate the correlation-robust function $\mathsf{O}(n\log^2 n)$ times to generate the random bits which represent the masks of the server's elements. This cost can become prohibitive for larger sets, as we will show in our evaluation in §8.

4 Permutation-based Hashing

The overhead of the OT-based PSI protocol of [22] and of the circuit-based PSI protocols we describe in §5 depends on the bit-lengths of the items that the parties map to bins. The bit-length of the stored items can be reduced based on a permutation-based hashing technique that was suggested in [1] for reducing the memory usage of Cuckoo hashing. That construction was presented in an algorithmic setting to improve memory usage. As far as we know this is the first time that it is used in secure computation or in a cryptographic context.

The construction uses a Feistel-like structure. Let $x = x_L|x_R$ be the bit representation of an input item, where $|x_L| = \log\beta$, i.e. is equal to the bit-length of an index of an entry in the hash table. (We assume here that the number of bins β in the hash table is a power of 2. It was shown in [1] how to handle the general case.) Let $f()$ be a random function whose range is $[0,\beta-1]$. Then item x is mapped to bin $x_L \oplus f(x_R)$. The value that is stored in the bin is x_R, which has a length that is shorter by $\log\beta$ bits than the length of the original item. This is a great improvement, since the length of the stored data is significantly reduced, especially if $|x|$ is not much greater than $\log\beta$. As for the security, it can be shown based on the results in [1] that if the function f is k-wise independent, where $k = \text{polylog}\, n$, then the maximum load of a bin is $\log n$ with high probability.

The structure of the mapping function ensures that if two items x, x' store the same value in the same bin then it must hold that $x = x'$: if the two items are mapped to the same bin, then $x_L \oplus f(x_R) = x'_L \oplus f(x'_R)$. Since the stored values satisfy $x_R = x'_R$ it must also hold that $x_L = x'_L$, and therefore $x = x'$.

As a concrete example, assume that $|x| = 32$ and that the table has $\beta = 2^{20}$ bins. Then the values that are stored in each bin are only 12 bits long, instead of 32 bits in the original scheme. Note also that the computation of the bin location requires a single instantiation of f, which can be implemented with a medium-size lookup table.

A comment about an alternative approach An alternative, and more straightforward approach for reducing the bit-length could map x using a random *permutation* $p()$ to a random $|x|$-bit string $p(x)$. The first $\log\beta$ bits of $p(x)$ are used to define the bin to which x is mapped, and the value stored in that bin holds the remaining $|x| - \log\beta$ bits of $p(x)$. This construction, too, has a shorter length for the values that are stored in the bins, but it suffers from two drawbacks: From a performance perspective, this construction requires the usage of a random *permutation* on $|x|$ bits, which is harder to compute than a random *function*. From a theoretical perspective, it is impossible to have efficient constructions of k-wise independent permutations, and therefore we only know how to prove the $\log n$ maximum load of the bins under the stronger assumption that the permutation is random.

5 Circuit-Phasing

PSI protocols that are based on generic secure computation are of great importance due to their flexibility (cf. §3.4 for details). The best known construction of a circuit computing the intersection (of σ-bit elements) is the SCS circuit of [12] with about $3n\sigma\log_2 n$ AND gates and an AND depth of $\Theta(\log_2\sigma\cdot\log_2 n)$. We describe a new construction of circuits with the same order of AND gates (but with smaller constants), and a much smaller depth. Our experiments, detailed in §8.1, demonstrate that the new circuits result in much better performance.

The new protocol, which we denote as Circuit-Phasing, is based on the two parties mapping their inputs to hash tables before applying the circuit. The idea is similar to the OT-based PSI protocol of [22] described in §3.5, but instead of using OTs for the comparisons, the protocol evaluates a pairwise-comparison circuit between each bin of P_1 and P_2 in parallel:

- Both parties use a table of size $\beta = \mathsf{O}(n)$ to store their elements. Our analysis (§7) shows that setting $\beta = 2.4n$ reduces the error probability to be negligible for reasonable input sizes ($2^8 \le n \le 2^{24}$) when setting the stash size according to Tab. 4.
- P_2 maps its input elements to β bins using Cuckoo hashing with two hash functions and a stash; empty bins are padded with a dummy element d_2.
- P_1 maps its input elements into β bins using simple hashing. The size of the bins is set to be max_β, a parameter that is set to ensure that no bin overflows (see §7.1). The remaining slots in each bin are padded with a dummy element $d_1 \neq d_2$. The analy-

sis described in §7.1 shows how max_β is computed and is set to a value smaller than $\log_2 n$.
- The parties securely evaluate a circuit that compares the element that was mapped to a bin by P_2 to each of the max_β elements mapped to it by P_1.
- Finally, each element in P_2's stash is checked for equality with all n_1 input elements of P_1 by securely evaluating a circuit computing this functionality.
- To reduce the bit-length of the elements in the bins, and respectively the circuit size, the protocol uses permutation-based hashing as described in §4. (Note that using this technique is impossible with SCS circuits of [12].)

A detailed analysis of the circuit size and depth Let m be the size of P_1's input to the circuit with $m = \beta max_\beta + sn_1$, i.e., for each of the β bins, P_1 inputs max_β items as well as n_1 items for each of the s positions in the stash. The circuit computes a total of m comparisons between the elements of the two parties. Each element is of length σ' bits, which is the reduced length of the elements after being mapped to bins using permutation-based hashing, i.e. $\sigma' = \sigma - \log_2 \beta$.

A comparison of two σ'-bit elements is done by computing the bitwise XOR of the elements and then a tree of $\sigma' - 1$ OR gates, with depth $\lceil \log_2 \sigma' \rceil$. The topmost gate of this tree is a NOR gate. Afterwards, the circuit computes the XOR of the results of all comparisons involving each item of P_2. (Note that at most one of the comparisons results in a match, therefore the circuit can compute the XOR, rather than the OR, of the results of the comparisons.) Overall, the circuit consists of about $m \cdot (\sigma' - 1) \approx n_1 \cdot (max_\beta + s) \cdot (\sigma' - 1)$ non-linear gates and has an AND depth of $\lceil \log_2 \sigma \rceil$.

Advantages Circuit-Phasing has several advantages over the SCS circuit:
- Compared to the number of AND gates in the SCS circuit, which is $3n\sigma \log n$, and recalling that $\sigma' < \sigma$, and that max_β was shown in our experiments to be no greater than $\log n$, the number of non-linear gates in Circuit-Phasing is smaller by a factor greater than 3 compared to the number of non-linear gates in the SCS circuit (even though both circuits have the same big "O" asymptotic sizes).
- The main advantage of Circuit-Phasing is the low AND depth of $\log_2(\sigma)$, which is also independent of the number of elements n. This affects the overhead of the GMW protocol that requires a round of interaction for every level in the circuit.
- Another advantage of Circuit-Phasing is its simple structure: the same small comparison circuit is evaluated for each bin. This property allows for a SIMD (Single Instruction Multiple Data) evaluation with a very low memory footprint and easy parallelization.

Hashing failures: The correct performance of the protocol depends on the successful completion of the hashing operations: The Cuckoo hashing must succeed, and the simple hashing must not place more than max_β elements in each bin. Tables of size $2(1+\varepsilon)n$ and $max_\beta = \mathsf{O}(\log n)$ guarantee these properties with high probability. We analyze the exactly required table sizes in §7 and set them to be negligible in the statistical security parameter λ.

6 OT-Phasing

We improve the OT-PSI protocol of [22] by applying the following changes to the protocol:
- Reducing the bit-length of the items using the permutation-based hashing technique described in §4. This improvement reduces the length of the items from $|x|$ bits to $|x| - \beta$ bits, where β is the size of the tables, and consequently reduces the number of OTs by a factor of $\beta/|x|$.
- Using OTs on a single mask instead of on $\mathsf{O}(\log n)$ masks before. This improvement is detailed in §6.1.
- Improving the utilization of bins by using 3-way Cuckoo hashing (§6.2).

We call the resulting PSI protocol that combines all these optimizations OT-Phasing. In the full version [21], we evaluate the performance gain of each optimization individually and micro-benchmark the resulting protocol.

6.1 A Single Mask per Bin

In order to hide information about the number of items that were mapped to a bin, the original OT-PSI protocol of [22] (cf. §3.5) padded all bins to a maximum size of $max_\beta = \mathsf{O}(\log n)$. The protocol then ran OTs on max_β masks of ℓ-bit length where the parties had to generate and process all of the max_β masks. We describe here a new construction that enables the parties to compute only a constant number of masks per element, regardless of the number of elements that were mapped to the bin by P_1. While this change seems only small, it greatly increases the performance and scalability of the protocol (cf. iterative performance improvements in the full version [21]). In particular, this change results in two improvements to the protocol:
- The number of symmetric cryptographic operations to generate the masks is reduced from $\mathsf{O}(\log^2 n)$ to $\mathsf{O}(\log n)$. Furthermore, note that P_2 had to compute the plaintext intersection between his $n_2 max_\beta$ generated masks and the $2n_1$ masks sent by P_1. This

also greatly improves the memory footprint and plaintext intersection.

- In the previous OT-based protocol, a larger value of the parameter max_β reduced the failure probability of the simple hashing procedure used by P_1, but increased the string size in the OTs. In the new protocol the value of max_β does not affect the overhead. Therefore P_1 can use arbitrarily large bins and ensure that the mapping that it performs never fails.

Recall that in the OT-based PSI protocol of [22] (cf. §3.5) the parties had inputs of t characters, where each character was μ bits long, and we used the notation $N = 2^\mu$. The parties performed OTs on strings of max_β masks per bin. Each mask had length $\ell = \lambda + \log_2(n_1) + \log_2(n_2)$ bits, corresponded to an element that P_1 mapped to the bin, and included a 1-out-of-N random-OT for each of the t characters of this element. P_1 was the sender, received all the N sender input-strings of each OT, and chose from them the one corresponding to the value of the character in its own element. P_2 was the receiver and received the string corresponding to the value of the character in its own element. Then P_1 computed the XOR of the t strings corresponding to the t characters of its element and sent this XOR value to P_2, which compared it to the XOR of its t outputs from OT.

The protocol can be improved by running the t 1-out-of-N OTs on a *single* mask per bin. Denote by u the actual number of items mapped by P_1 to a bin. The value of u is not revealed to P_2 in the new protocol and therefore there is no need to pad the bin with dummy items. Denote the single item that P_2 mapped to the bin as $y = y_1, \ldots, y_t$, and the u items that P_1 mapped to the bin as $x^1, \ldots, x^u$, where each x^i is defined as $x^i = x^i_1, \ldots, x^i_t$.

Define the input strings to the j-th OT as $\{s_{j,\ell}\}_{\ell=1\ldots N}$. The protocol that is executed is a random OT and therefore these strings are chosen by the protocol and not by P_1. The parties run a single set of t OTs and P_2 learns the t strings $s_{1,y_1}, \ldots, s_{t,y_t}$. It computes their XOR $S_{P_2} = s_{1,y_1} \oplus \cdots \oplus s_{t,y_t}$, and the value $H(S_{P_2})$, where $H()$ is a hash function modeled as a random oracle.

P_1 learns all the Nt strings generated in the random-OT protocols. For each input element x^i that P_1 mapped to the bin, it computes the XOR of the strings corresponding to the characters of the input, namely $S^i_{P_1} = s_{1,x^i_1} \oplus \cdots \oplus s_{t,x^i_t}$, and then computes the value $H(S^i_{P_1})$. Note that over all bins, P_1 needs to perform this computation only $O(n_1)$ times and compute $O(n_1)$ hash values. P_1 then sends all these values to P_2 in randomly permuted order. P_2 computes the intersection between these values and the $H(S_{P_2})$ values that it computed in the protocol.

Efficiency: P_2 computes only a single set of t OTs per bin on one mask, compared to t OTs on max_β masks in the OT-based protocol of [22]. As for P_1's work, it computes a single set of OTs per bin, and in addition computes a XOR of strings and a hash for each of its $O(n_1)$ input elements. This is a factor of $max_\beta = O(\log n_1)$ less work as before. Communication is only $O(n\sigma)$ strings, as before.

Security: Assuming that the OT protocols are secure and that the parties are semi-honest, the only information that is received by any party in the protocol is the $H(S^i_{P_1})$ values that are sent from P_1 to P_2. For all values in the intersection of the input sets of the two parties, P_1 sends to P_2 the same hash values as those computed by P_2. Consider the set of input elements $\bar{X}$ that are part of P_1's input and are not in P_2's input, and the set of XOR values corresponding to $\bar{X}$. There might be linear dependencies between the XOR values of $\bar{X}$, but it holds with overwhelming probability that all these values are different, and they are also all different from the XOR values computed by P_2. Therefore, the result of applying a random hash function $H()$ to these values is a set of random elements in the range of the hash function. This property enables to easily provide a simulation based proof of security for the protocol.

6.2 3-Way Cuckoo Hashing

The original OT-based PSI protocol of [22] uses Cuckoo hashing which employs two hash functions to map elements into bins. It was shown in [20] that if n elements are mapped to $2(1+\varepsilon)n$ bins, Cuckoo hashing succeeds with high probability for $\varepsilon > 0$. This means that Cuckoo hashing achieves around 50% utilization of the bins. If the number of hash functions h is increased to $h > 2$, a much better utilization of bins can be achieved [6]. However, using h hash functions in our protocol requires P_1 to map each element h times into its bins using simple hashing and requires P_1 to send hn_1 masks in the intersection step of the protocol.

We detail in Tab. 1 the utilization and total communication of our PSI protocol for $n_1 = n_2 = 2^{20}$ and $n_2 = 2^8 \ll n_1 = 2^{20}$, for $\sigma = 32$-bit elements with different numbers of hash functions. We observe that there is a tradeoff between the communication for the OTs and the communication for the masks that are sent by P_1. Our goal is to minimize the total communication, and this is achieved for $h = 3$ hash functions in the setting of $n_1 = n_2$ and for $h = 2$ in the setting of $n_2 \ll n_1$. For $n_1 = n_2$ using $h = 3$ instead of $h = 2$, as in the original protocol of [22], reduces the overall communication by 33%.

Hashing failures: We observe that with OT-Phasing, there is essentially no bound on the number of items that the server can map to each specific bin, since the client does not observe this value in any way (the message that the client receives only depends on the total number of items that the server has). However, the parameters used

h	Util. [%]	#OTs	#Masks	Comm. [MB]	
				$n_1 = n_2$	$n_2 \ll n_1$
2	50.0	$2.00n_2$ t	$\mathbf{2}n_1\ell$	148.0	**17.0**
3	91.8	$1.09n_2$ t	$3n_1\ell$	**99.8**	25.5
4	97.7	$1.02n_2$ t	$4n_1\ell$	105.3	34.0
5	**99.2**	$\mathbf{1.01}n_2$ **t**	$5n_1\ell$	114.6	42.5

Table 1: Overall communication for a larger number of hash functions h. Communication is given for a) $n_1 = n_2 = 2^{20}$ and b) $n_2 = 2^8 \ll n_1 = 2^{20}$ elements of $\sigma = 32$-bit length. Utilization according to [6].

in the protocol do need to ensure that the Cuckoo hashing procedure does not fail. The analysis appears in §7.

7 Hashing Failures

The PSI schemes we presented use simple hashing (by P_1), and Cuckoo hashing (by P_2). In both hashing schemes, the usage of bins (or a stash) of constant size, might result in hashing failures if the number of items mapped to a bin (or the stash) exceeds its capacity.

When hashing fails, the party which performed the hashing has two options: (1) Ignore the item that cannot be mapped by the hashing scheme. This essentially means that this item is removed from the party's input to the PSI protocol. Consequently, the output of the computation might not be correct (although, if this type of event happens rarely, the effect on correctness is likely to be marginal). (2) Attempt to use a different set of hash functions, and recompute the hash of all items. In this case the other party must be informed that new hash functions are used. This is essentially a privacy leak: for example, the other party can check if the input set S of the first party might be equal to a set S' (if a hashing failure does not occur for S' then clearly $S' \neq S$). The effect of this leak is likely to be weak, too, but it is hard to quantify.

The effect of hashing failures is likely to be marginal, and might be acceptable in many usage settings (for example, when measuring ad conversion rates it typically does not matter if the revenue from a single ad view is ignored). However, it is preferable to set the probability of hashing failures to be negligibly small.

In OT-Phasing, P_2 does not learn the number of items that P_1 maps to each bin, and therefore P_1 can set the size of the bins to be arbitrarily large. However, in that PSI protocol P_1 knows the size of the stash that is used in the Cuckoo hashing done by P_2. In Circuit-Phasing, each party knows the size of the bins (or stash) that is used by the other party. We are therefore interested in learning the failures probabilities of the following schemes, and bound them to be negligible, i.e., at most 2^{-40}:

- §7.1: Simple hashing in the Circuit-Phasing scheme, where n items are mapped using two independent functions to $2.4n$ bins. This is equivalent to mapping $2n$ items to $2.4n$ bins.
- §7.2: Cuckoo hashing, using $2.4n$ bins and either 2-way hashing (for Circuit-Phasing), or 3-way hashing (for OT-Phasing). The failure probability for 3-way hashing is smaller than for 2-way hashing (since there is an additional bin to which each item can be mapped), and therefore we will only examine the failure probability of 2-way Cuckoo hashing.

7.1 Simple Hashing

It was shown in [23] that when n balls are mapped at random to n bins then the maximum number of elements in a bin is with high probability $\frac{\ln n}{\ln\ln n}(1+o(1))$. Let us examine in more detail the probability of the following event, "$2n$ balls are mapped at random to $2.4n$ bins, and the most occupied bin has at least k balls":

$$\begin{aligned}
&\Pr(\exists\text{bin with} \geq k \text{ balls}) && (1)\\
&\leq 2.4n \cdot \Pr(\text{bin \#1 has} \geq k \text{ balls}) && (2)\\
&\leq 2.4n \binom{2n}{k} \left(\frac{1}{2.4n}\right)^k && (3)\\
&\leq \left(\frac{2ne}{k}\right)^k \left(\frac{1}{2.4n}\right)^{k-1} && (4)\\
&= n\left(\frac{2e}{k}\right)^k \left(\frac{1}{2.4}\right)^{k-1}. && (5)
\end{aligned}$$

It is straightforward to see that this probability can be bounded to be at most 2^{-40} by setting

$$k \quad \geq \quad \max(6, 2e\log n/\log\log n). \qquad (6)$$

We calculated for some values of n the desired bin sizes based on the upper bound of Eq. (6) and the tighter calculation of Eq. (5), and chose the minimal value of k that reduces the failure probability to below 2^{-40}. The results are in Table 2. It is clear that Eq. (5) results in smaller bins for sufficiently large n, and therefore the maximal bin size should be set according to Eq. (5).

n	2^{12}	2^{16}	2^{20}	2^{24}
Eq. (5)	18	19	20	21
Eq. (6)	19	22	26	29

Table 2: The bin sizes max_β that are required to ensure that no overflow occurs when mapping $2n$ items to $2.4n$ bins, according to Eq. (5) and Eq. (6).

7.2 Cuckoo Hashing

It was shown in [16] that Cuckoo hashing with a stash of size s fails with probability $O(n^{-s})$. The constants in the big "O" notation are unclear, but it is obvious that $O(n^{-s}) \leq n^{-(s-1)}$ for sufficiently large values of n.

s	2^{11}	2^{12}	2^{13}	2^{14}
0	1,068,592,289	1,070,826,935	1,072,132,187	1,072,845,430
1	4,994,200	2,861,137	1,592,951	891,497
2	147,893	52,038	16,404	4,840
3	7,005	1,647	274	56
4	407	62	8	1
5	28	5	0	0
6	2	0	0	0

Table 3: Required stash sizes s accumulated over 2^{30} Cuckoo hashing repetitions mapping $n \in \{2^{11}, 2^{12}, 2^{13}, 2^{14}\}$ elements to $2.4n$ bins.

We would like to find the exact size of the stash that ensures that the failure probability is smaller than 2^{-40}. We ran 2^{30} repetitions of Cuckoo hashing, mapping n items to $2.4n$ bins, for $n \in \{2^{11}, 2^{12}, 2^{13}, 2^{14}\}$, and recorded the stash size s that was needed for Cuckoo hashing to be successful. Tab. 3 depicts the number of repetitions where we required a stash of size s. From the results we can observe that, to achieve 2^{-30} failure probability of Cuckoo hashing, we would require a stash of size $s = 6$ for $n = 2^{11}$, $s = 5$ for $n = 2^{12}$, and $s = 4$ for both $n = 2^{13}$ and $n = 2^{14}$ elements.

However, in our experiments we need the stash sizes for larger values of $n \geq 2^{14}$ to achieve a Cuckoo hashing failure probability of 2^{-40}. To obtain the failure probabilities for larger values of n, we extrapolate the results from Tab. 3 using linear regression and illustrate the results in Fig. 1. We observe that the stash size for achieving a failure probability of 2^{-40} is drastically reduced for higher values of n: for $n = 2^{16}$ we need a stash of size $s = 4$, for $n = 2^{20}$ we need $s = 3$, and for $n = 2^{24}$ we need $s = 2$. This observation is in line with the asymptotic failure probability of $O(n^{-s})$.

Finally, we extrapolate the required stash sizes s to achieve a failure probability of 2^{-40} for smaller values of $n \in \{2^8, 2^{12}\}$ and give the results together with the stash sizes for $n \in \{2^{16}, 2^{20}, 2^{24}\}$ in Tab. 4.

number of elements n	2^8	2^{12}	2^{16}	2^{20}	2^{24}
stash size s	12	6	4	3	2

Table 4: Required stash sizes s to achieve 2^{-40} error probability when mapping n elements into $2.4n$ bins.

8 Evaluation

We report on our empirical performance evaluation of Circuit-Phasing (§5) and OT-Phasing (§6) next. We evaluate their performance separately (§8.1 and §8.2), since special purpose protocols for set intersection were shown to greatly outperform circuit-based solutions in [22]. (The latter are nevertheless of independent interest because their functionality can be easily modified.)

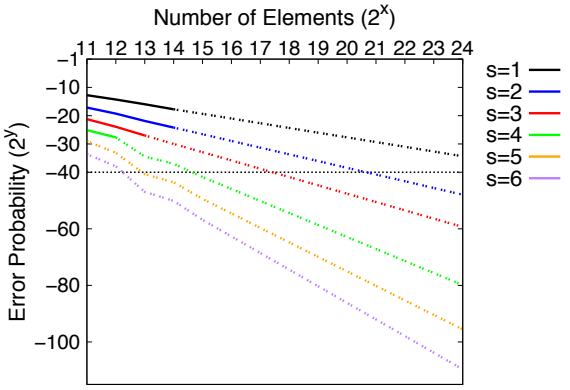

Figure 1: Error probability when mapping n elements to $2.4n$ bins using 2-way Cuckoo hashing for stash sizes $1 \leq s \leq 6$. The solid lines correspond to actual measurements, the dashed lines were extrapolated using linear regression. Both axes are in logarithmic scale.

Benchmarking Environment We consider two benchmark settings: a *LAN* setting and a *WAN* setting. The LAN setting consists of two desktop PCs (Intel Haswell i7-4770K with 3.5 GHz and 16GB RAM) connected by Gigabit LAN. The WAN setting consists of two Amazon EC2 m3.medium instances (Intel Xeon E5-2670 CPU with 2.6 GHz and 3.75 GB RAM) located in the US east coast (North Virginia) and Europe (Frankfurt) with an average bandwidth of 50 MB/s and average latency (round-trip time) of 96 ms.

We perform all experiments for a symmetric security parameter $\kappa = 128$-bit and statistical security parameter $\lambda = 40$ (cf. §2.1), using a single thread (except for GMW, where we use two threads to compute OT extension), and average the results over 10 executions. In our experiments, we frequently encountered outliers in the WAN setting with more than twice of the average run-time, for which we repeated the execution. The resulting variance decreased with increasing input set size; it was between $0.5\% - 8.0\%$ in the LAN setting and between $4\% - 16\%$ in the WAN setting. Note that all machines that we perform our experiments on are equipped with the AES-NI extensions which allows for very fast AES evaluation.

Implementation Details We instantiate the random oracle, the function for hashing into smaller domains, and the correlation-robust function in OT extension with SHA256. We instantiate the pseudo-random generator using AES-CTR and the pseudo-random permutation in the server-aided protocol of [15] using AES. To compute the $\binom{2^\mu}{1}$-OT^t_ℓ functionality, we use the random 1-

out-of-N OT extension of [17] and set $\mu = 8$, i.e., use $N = 256$, since this was shown to result in minimal overhead in [22]. We measure the times for the function evaluation including the cost for precomputing the OT extension protocol and build on the OT extension implementation of [2]. Our OT-Phasing implementation is available online at `https://github.com/encryptogroup/PSI` and our Circuit-Phasing implementation is available as part of the ABY framework of [5] at `https://github.com/encryptogroup/ABY`.

For simple hashing we use the maximum bin sizes that were computed using Equation 5 in §7.1 (cf. Tab. 2). For Cuckoo hashing, we set $\varepsilon = 0.2$ and map n elements to $2(1+\varepsilon)n$ bins for 2-way Cuckoo hashing and to $(1+\varepsilon)n$ bins for 3-way Cuckoo hashing with a stash size according to Tab. 4. The only exception for the stash size are the experiments with different set sizes in §8.2.2, where we use no stash for our OT-Phasing protocol.

For OT-based PSI [22] and OT-Phasing, where the performance depends on the bit-length of elements, we hash the σ-bit input elements into a $\ell = \lambda + \log_2(n_1) + \log_2(n_2)$-bit representation using SHA256 if $\sigma > \ell$.

We use a garbled circuits implementation with most recent optimizations (cf. full version [21] for details).

We emphasize that all implementations are done in the same programming language (C++), use the same underlying libraries for evaluating cryptographic operations (OpenSSL for symmetric cryptography and Miracl for elliptic curve cryptography), perform the plaintext-intersection of elements using a standard hash map, are all executed using a single thread (except for the GMW implementation which uses two threads), and run in the same benchmarking environment.

8.1 Generic Secure Computation-based PSI Protocols

For the generic secure computation-based PSI protocols, we perform the evaluation on a number of elements varying from 2^8 to 2^{20} and a fixed bit-length of $\sigma = 32$-bit. For $n = 2^{20}$ all implementations, except Circuit-Phasing with GMW, exceeded the available memory, which is due to the large number of AND gates in the SCS circuit (estimated 2 billion AND gates) and the requirement to represent bits as keys for Circuit-Phasing with Yao, where storing only the input wire labels to the circuit requires 1 GB. A more careful implementation, however, could allow the evaluation of these circuits. We compare the sort-compare-shuffle (SCS) circuit of [12] and its depth-optimized version of [22], with Circuit-Phasing (§5), by evaluating both constructions using Yao's garbled circuits protocol [25] and the GMW protocol [11] in the LAN and WAN setting. We use the size-optimized version of the SCS circuit in Yao's garbled circuit and the depth-optimized version of the circuit in the GMW protocol (cf. §3.4). For the evaluation in Circuit-Phasing, we set the maximum bin size in simple hashing according to Equation 5 (cf. Tab. 2, set $\varepsilon = 0.2$, set the stash size according to Tab. 4, and assume $n = n_1 = n_2$. The run-time of Circuit-Phasing would increase linear in the bin size max_β, while the stash size s would have a smaller impact on the total run-time as the concrete factors are smaller.

Run-Time (Tab. 5) Our main observation is that Circuit-Phasing outperforms the SCS circuit of [12] for all parameters except Yao's garbled circuits with small set sizes $n = 2^8$. In this case, the high stash size of $s = 12$ greatly impacts the run-time of Circuit-Phasing. When evaluated using Yao's garbled circuits, Circuit-Phasing outperforms the SCS circuit by a factor of 1-2, and when evaluated using GMW it outperforms SCS by a factor of 2-5. Furthermore, the run-time for Circuit-Phasing grows slower with n than for the SCS circuit for all settings except for GMW in the WAN setting. There, the run-time of the SCS circuit grows slower than that of Circuit-Phasing. This can be explained by the high number of communication rounds of the SCS based protocol, which are slowly being amortized with increasing values of n. The slower increase of the run-time of Circuit-Phasing with increasing n is due to the smaller increase of the bin size $max_\beta \in O(\frac{\ln n}{\ln \ln n})$ vs. $O(\log n)$ for the SCS circuit, and the use of permutation-based hashing, which reduces the bit-length of the inputs to the circuit. Note that our Yao's garbled circuits implementation suffers from similar performance drawbacks in the WAN setting as our GMW implementation, although being a constant round protocol. This can be explained by the pipelining optimization we implement, where the parties pipeline the garbled circuits generation and evaluation. The performance drawback could be reduced by using an implementation that uses independent threads for sending / receiving.

Communication (Tab. 6) Analogously to the run-time results, Circuit-Phasing improves the communication of the SCS circuit by factor of 1-4 and grows slower with increasing values of n. The improvement of the round complexity, which is mostly important for GMW, is even more drastic. Here, Circuit-Phasing outperforms the SCS circuit by a factor of 16-38. Note that the round complexity of Circuit-Phasing only depends on the bit-length of items and is independent of the number of elements.

8.2 Special Purpose PSI Protocols

For the special purpose PSI protocols we perform the experimental evaluation for equally sized sets $n_1 =$

Protocol	LAN				WAN			
	$n=2^8$	$n=2^{12}$	$n=2^{16}$	$n=2^{20}$	$n=2^8$	$n=2^{12}$	$n=2^{16}$	$n=2^{20}$
Yao's garbled circuits [25]								
SCS [12]	309	3,464	63,857	—	2,878	20,184	301,512	—
Circuit-Phasing §5	376	3,154	39,785	—	3,004	17,133	178,865	—
Goldreich-Micali-Wigderson [11]								
SCS [12]	626	2,175	38,727	—	11,870	21,030	218,378	—
Circuit-Phasing §5	280	1,290	14,149	168,397	2,681	8,681	81,534	846,510

Table 5: Run-time in ms for generic secure PSI protocols in the LAN and WAN setting on $\sigma = 32$-bit elements.

Protocol	$n=2^8$	$n=2^{12}$	$n=2^{16}$	$n=2^{20}$	Asymptotic
Number of AND gates					
SCS [12]	229,120	5,238,784	107,479,009	*2,000,000,000	$\sigma(3n\log_2(n)+4n)$
Circuit-Phasing §5	297,852	3,946,776	49,964,540	600,833,968	$(\sigma-\log_2(n)-2)(6(1+\varepsilon)n\frac{\ln n}{\ln\ln n}+sn)$
Communication in MB for Yao's garbled circuits [25] and GMW [11]					
SCS [12]	7	169	3,485	*64,850	$2\kappa\sigma(3n\log_2(n)+4n)$
Circuit-Phasing §5	9	122	1,550	18,736	$2\kappa(\sigma-\log_2(n)-2)(6(1+\varepsilon)n\frac{\ln n}{\ln\ln n}+sn)$
Number of communication rounds for GMW [11]					
SCS [12]	85	121	157	193	$(\log_2(\sigma)+4)\log_2(2n)+4$
Circuit-Phasing §5	5	5	5	5	$\log_2(\sigma)$

Table 6: Number of AND gates, concrete communication in MB, round complexity, and failure probability for generic secure PSI protocols on $\sigma = 32$-bit elements. Numbers with * are estimated.

n_2 (§8.2.1) and differently sized sets $n_2 \ll n_1$ (§8.2.2), for set sizes ranging from 2^8 to 2^{24} in the LAN setting and from 2^8 to 2^{20} in the WAN setting.

We compare OT-Phasing (§6) to the original OT-based PSI protocol of [22], the naive hashing solution (§3.1), the semi-honest server-aided protocol of [15] (§3.2), and the Diffie-Hellmann (DH)-based protocol of [18] (§3.3) using elliptic curves. Note that the naive hashing protocol and the server-aided protocol of [15] have different security assumptions and cannot directly be compared to the remaining protocols. We nevertheless included them in our comparison to serve as a base-line on the efficiency of PSI. For the protocol of [15], we run the server routine that computes the intersection between the sets on the machine located at the US east coast (North Virginia) and the server and client routine on the machine in Europe (Frankfurt). For the original OT-based PSI and OT-Phasing, we give the run-time and communication for three bit-lengths: *short* $\sigma = 32$ (e.g., for IPv4 addresses), *medium* $\sigma = 64$ (e.g., for credit card numbers), and *long* $\sigma = 128$ (for set intersection between arbitrary inputs).

Note that the OT-based PSI protocol of [22] and our OT-Phasing protocol both evaluate public-key cryptography during the base-OTs, which dominates the run-time for small sets. However, these base-OTs only need to be computed once and can be re-used over multiple sessions. In the LAN setting, the average run-time for computing the 256 base-OTs was 125 ms while in the WAN setting the run-time was 245 ms. Nevertheless, our results all contain the time for the base-OTs to provide an estimation of the total run-time.

8.2.1 Experiments with Equal Input Sizes

In the experiments for input sets of equal size $n = n_1 = n_2$ we set $n \in \{2^8, 2^{12}, 2^{16}, 2^{20}, 2^{24}\}$ in the LAN setting and $n \in \{2^8, 2^{12}, 2^{16}, 2^{20}\}$ in the WAN setting. Note that for larger bit-lengths $\sigma \geq 64$ and for $n = 2^{24}$ elements, the memory needed for the OT-based PSI protocol of [22] exceeded the available memory.

Run-Time (Tab. 7) As expected, the lowest run-time for the equal set-size experiments is achieved by the (insecure) naive hashing protocol followed by the server-aided protocol of [15], which has around twice the run-time. In the LAN setting, however, for short bit-length $\sigma = 32$, our OT-Phasing protocol nearly achieves the same run-time as both of these solutions (which are in a different security model). In particular, when computing the intersection for $n = 2^{24}$ elements, our OT-Phasing protocol requires only 3.5 more time than the naive hashing protocol and 2.5 more time than the server-aided protocol. In comparison, for the same parameters, the original OT-based PSI protocol of [22] has a 68 times higher run-time than the naive hashing protocol, and the DH-based ECC protocol of [18] has a four orders of magnitude higher run-time compared to naive hashing.

While the run-time of our OT-Phasing protocol increases with the bit-length of elements, for $\sigma = 128$-bit its run-time is only 15 times higher than the naive hashing protocol, and is still nearly two orders of magnitude better than the DH-based ECC protocol.

Overall, in the LAN setting and for larger sets (e.g., $n = 2^{24}$), the run time of OT-Phasing is 20x better than that of the original OT-based PSI protocol of [22], and

Setting	LAN					WAN			
Protocol	$n=2^8$	$n=2^{12}$	$n=2^{16}$	$n=2^{20}$	$n=2^{24}$	$n=2^8$	$n=2^{12}$	$n=2^{16}$	$n=2^{20}$
Naive Hashing$^{(*)}$ §3.1	1	4	48	712	13,665	97	111	558	3,538
Server-Aided$^{(*)}$ [15]	1	5	78	1,250	20,053	198	548	2,024	7,737
DH-based ECC [18]	231	3,238	51,380	818,318	13,065,904	628	10,158	161,850	2,584,212
Bit-length $\sigma = 32$*-bit*									
OT PSI [22]	184	216	3,681	62,048	929,685	957	1,820	9,556	157,332
OT-Phasing §6	179	202	437	4,260	46,631	912	1,590	3,065	14,567
Bit-length $\sigma = 64$*-bit*									
OT PSI [22]	201	485	7,302	125,697	—	977	1,873	18,998	315,115
OT-Phasing §6	180	240	865	10,128	137,036	1,010	1,780	5,009	29,387
Bit-length $\sigma = 128$*-bit*									
OT PSI [22]	201	485	8,478	155,051	—	980	1,879	21,273	392,265
OT-Phasing §6	181	240	915	13,485	204,593	1,010	1,780	5,536	37,422

Table 7: Run-time in ms for protocols with $n = n_1 = n_2$ elements. (Protocols with $^{(*)}$ are in a different security model.)

Protocol	$n=2^8$	$n=2^{12}$	$n=2^{16}$	$n=2^{20}$	$n=2^{24}$	**Asymptotic [bit]**
Naive Hashing$^{(*)}$ §3.1	0.01	0.03	0.56	10.0	176.0	$n_1\ell$
Server-Aided$^{(*)}$ [15]	0.01	0.16	2.5	40.0	640.0	$(n_1+n_2+\lvert X\cap Y\rvert)\kappa$
DH-based ECC [18]	0.02	0.28	4.56	74.0	1,200.0	$(n_1+n_2)\varphi+n_1\ell$
Bit-length $\sigma = 32$*-bit*						
OT PSI [22]	0.09	1.39	22.58	367.20	5,971.20	$0.6n_2\sigma\kappa+6n_1\ell$
OT-Phasing §6	0.06	0.73	8.74	136.8	1,494.4	$2.4n_2\kappa(\lceil\frac{\sigma-\lfloor\log_2(1.2n_2)\rfloor}{8}\rceil)+(3+s)n_1\ell$
Bit-length $\sigma = 64$*-bit*						
OT PSI [22]	0.14	2.59	41.78	674.4	10,886.4	$0.6n_2\kappa*min(\ell,\sigma)+6n_1\ell$
OT-Phasing §6	0.09	1.34	18.34	290.4	3,952.0	$2.4n_2\kappa(\lceil\frac{min(\ell,\sigma)-\log_2(n_2)}{8}\rceil)+(3+s)n_1\ell$
Bit-length $\sigma = 128$*-bit*						
OT PSI [22]	0.14	2.59	46.58	828.0	14,572.8	$0.6n_2\ell\kappa+6n_1\ell$
OT-Phasing §6	0.09	1.34	20.74	367.2	5,795.2	$2.4n_2\kappa(\lceil\frac{\ell-\log_2(n_2)}{8}\rceil)+(3+s)n_1\ell$

Table 8: Communication in MB for PSI protocols with $n = n_1 = n_2$ elements. $\ell = \lambda + \log_2(n_1) + \log_2(n_2)$. Assuming intersection of size $1/2 \cdot n$ for TTP-based protocol. (Protocols with $^{(*)}$ are in a different security model.)

60-278x better than that of the DH-ECC protocol of [18].

When switching to the WAN setting, the run-times of the protocols are all increased by a factor of 2-6. Note that the faster protocols suffer from a greater performance loss (factors of 5 and 6 for 2^{20} elements, for the naive hashing protocol and server-aided protocol) than the slower protocols (factor 3 for the DH-based and our OT-Phasing protocol and 2.5 for the OT-based PSI protocol of [22]). This difference can be explained by the greater impact of the high latency of 97 ms on the runtime of the protocols. The relative performance among the protocols remains similar to the LAN setting.

Communication (Tab. 8) The amount of communication performed during protocol execution is often more limiting than the required computation power, since the latter can be scaled up more easily by using more machines. The naive hashing approach has the lowest communication among all protocols, followed by the server-aided solution of [15]. Among the secure two-party PSI protocols, the DH-based ECC protocol of [18] has the lowest communication. In the setting for $n = 2^{24}$ elements of short bit-length $\sigma = 32$ bit, our OT-Phasing protocol nearly achieves the same complexity as the DH-based ECC protocol, which is due to the use of permutation-based hashing. This is quite surprising, as protocols that use public-key cryptography, in particular elliptic curves, were believed to have much lower communication complexity than protocols based on other cryptographic techniques.

In comparison to the original OT-based PSI protocol of [22], OT-Phasing reduces the communication for all combinations of elements and bit-lengths by factor 2.5 - 4. We also observe that OT-Phasing reduces the impact when performing PSI on elements of longer bit-length. In fact, it even has a lower communication for $\sigma = 128$ than the original OT-based PSI protocol has for $\sigma = 32$.

8.2.2 Experiments with Different Input Sizes

For examining the setting where the two parties have different input sizes, we set $n_1 \in \{2^{16}, 2^{20}, 2^{24}\}$ and $n_2 \in \{2^8, 2^{12}\}$ and run the protocols on all combinations such that $n_2 \ll n_1$. Note that we excluded the original OT-based PSI protocol of [22] from the comparison, since the bin size max_β becomes large when $\beta \ll n$ and the memory requirement when padding all bins to max_β elements quickly exceeded the available memory. In this setting, unlike the equal input sizes experiments in §8.2.1, we use $h = 2$ hash functions instead of $h = 3$,

Setting	LAN						WAN			
Protocol	$n_2 = 2^8$			$n_2 = 2^{12}$			$n_2 = 2^8$		$n_2 = 2^{12}$	
	$n_1 = 2^{16}$	$n_1 = 2^{20}$	$n_1 = 2^{24}$	$n_1 = 2^{16}$	$n_1 = 2^{20}$	$n_1 = 2^{24}$	$n_1 = 2^{16}$	$n_1 = 2^{20}$	$n_1 = 2^{16}$	$n_1 = 2^{20}$
Naive Hashing$^{(*)}$ §3.1	33	464	7,739	35	466	7,836	560	2,775	562	2,797
Server-Aided$^{(*)}$ [15]	74	680	8,935	75	696	8,965	629	2,923	731	2,951
DH-based ECC [18]	28,387	421,115	6,848,215	29,810	422,712	6,849,534	112,336	1,743,400	111,642	1,753,595
OT-Phasing §6										
Bit-length $\sigma = 32$	360	906	9,465	369	2,949	12,634	2,139	4,780	3,143	11,399
Bit-length $\sigma = 64$	555	1,506	15,789	581	6,146	22,368	3,349	6,879	3,923	20,345
Bit-length $\sigma = 128$	571	1,942	21,843	649	7,291	31,932	3,352	7,999	4,391	23,209

Table 9: Run-time in ms for PSI protocols with $n_2 \ll n_1$ elements. (Protocols with $^{(*)}$ are in a different security model.)

Protocol	$n_2 = 2^8$			$n_2 = 2^{12}$			Asymptotic [bit]
	$n_1 = 2^{16}$	$n_1 = 2^{20}$	$n_1 = 2^{24}$	$n_1 = 2^{16}$	$n_1 = 2^{20}$	$n_1 = 2^{24}$	
Naive Hashing$^{(*)}$ §3.1	0.5	8.5	144.0	0.5	9.0	152.0	$n_1\ell$
Server-Aided$^{(*)}$ [15]	1.0	16.0	256.0	1.1	16.1	256.1	$(n_1+n_2+\lvert X\cap Y\rvert)\kappa$
DH-based ECC [18]	2.5	40.5	656.0	2.7	41.1	664.1	$(n_1+n_2)\varphi+n_1\ell$
OT-Phasing §6							
Bit-length $\sigma = 32$	1.1	18.1	288.1	2.0	18.9	320.9	$4.8n_2\kappa(\lceil\frac{\sigma-\lceil\log_2(2.4n_2)\rceil}{8}\rceil)+2n_1\ell$
Bit-length $\sigma = 64$	1.1	18.1	288.1	3.2	20.1	322.1	$4.8n_2\kappa(\lceil\frac{\sigma-\lceil\log_2(2.4n_2)\rceil}{8}\rceil)+2n_1\ell$
Bit-length $\sigma = 128$	1.1	18.2	288.2	3.5	20.4	322.7	$4.8n_2\kappa(\lceil\frac{\sigma-\lceil\log_2(2.4n_2)\rceil}{8}\rceil)+2n_1\ell$

Table 10: Communication in MB for special purpose PSI protocols with $n_2 \ll n_1$ elements. $\ell = \lambda + \log_2(n_1) + \log_2(n_2)$. Assuming intersection of size $1/2 \cdot n_2$ for the TTP-based protocol. (Protocols with $^{(*)}$ are in a different security model.)

since this results in less total computation and communication (cf. §6.2). Since we use $h = 2$ hash functions, we also increase the number of bins from $1.2n_2$ to $2.4n_2$. Furthermore, we do not use a stash for our OT-Phasing protocol with different input sizes, since the stash would greatly increase the overall communication. However, not using a stash reveals some information on P_2's set (cf. §7). We show how to secure our protocol at a much lower cost by increasing the number of bins in the full version [21].

Run-Time (Tab. 9) Similar to the results for equal set sizes, the naive hashing protocol is the fastest protocol for all parameters. The server-aided protocol of [15] is the second fastest protocol but it scales better than the naive hashing protocol for increasing number of elements. The best scaling protocol is our OT-Phasing protocol. It achieves the same performance as the server-aided protocol for $n_2 = 2^8$, $n_1 = 2^{24}$ with short bit-length $\sigma = 32$. For $n_1 = 2^{24}$ its run-time is at most twice that of the server-aided protocol in both network settings.

When switching to the WAN setting, the run-times of all protocols are increased by a factor 4-6 while the relative performance between the protocols remains similar, analogously to the equal set size experiments.

Communication (Tab. 10) As expected, the naive hashing solution again has the lowest communication overhead. Surprisingly, our OT-Phasing protocol achieves nearly the same communication as the server-aided protocol of [15] and has only two times the communication of the naive hashing protocol for all bit-lengths. Furthermore, our OT-Phasing protocol requires a factor of 2-3 less communication than the DH-based ECC protocol of [18] for nearly all parameters. The low communication of our OT-Phasing protocol for unequal set sizes is due to the low number of OTs performed.

Acknowledgements: We thank Elaine Shi and the anonymous reviewers of USENIX Security 2015 for their helpful comments. This work was supported by the European Union's 7th Framework Program (FP7/2007-2013) under grant agreement n. 609611 (PRACTICE) and via a Marie Curie Career Integration Grant, by the DFG as part of project E3 within the CRC 1119 CROSSING, by the German Federal Ministry of Education and Research (BMBF) within EC SPRIDE, by the Hessian LOEWE excellence initiative within CASED, by a grant from the Israel Ministry of Science and Technology (grant 3-9094), by a Magneton grant of the Israeli Ministry of Economy, by the Israel Science Foundation (Grant No. 483/13), and by the Israeli Centers of Research Excellence (I-CORE) Program (Center No. 4/11).

References

[1] Y. Arbitman, M. Naor, and G. Segev. Backyard cuckoo hashing: Constant worst-case operations with a succinct representation. In *FOCS'10*, pages 787–796. IEEE, 2010.

[2] G. Asharov, Y. Lindell, T. Schneider, and M. Zohner. More efficient oblivious transfer and extensions for faster secure computation. In *CCS'13*, pages 535–548. ACM, 2013.

[3] E. De Cristofaro and G. Tsudik. Practical private set intersection protocols with linear complexity. In *FC'10*, volume 6052 of *LNCS*, pages 143–159. Springer, 2010.

[4] E. De Cristofaro and G. Tsudik. Experimenting with fast private set intersection. In *TRUST'12*, volume 7344 of *LNCS*, pages 55–73. Springer, 2012.

[5] D. Demmler, T. Schneider, and M. Zohner. ABY - A framework for efficient mixed-protocol secure two-party computation. In *NDSS'15*. The Internet Society, 2015.

[6] M. Dietzfelbinger, A. Goerdt, M. Mitzenmacher, A. Montanari, R. Pagh, and M. Rink. Tight thresholds for cuckoo hashing via XORSAT. In *ICALP'10*, volume 6198 of *LNCS*, pages 213–225. Springer, 2010.

[7] C. Dong, L. Chen, and Z. Wen. When private set intersection meets big data: An efficient and scalable protocol. In *CCS'13*, pages 789–800. ACM, 2013.

[8] S. Even, O. Goldreich, and A. Lempel. A randomized protocol for signing contracts. *Communmunications of the ACM*, 28(6):637–647, 1985.

[9] M. J. Freedman, C. Hazay, K. Nissim, and B. Pinkas. Efficient set-intersection with simulation-based security. In *Journal of Cryptology*, pages 1–41. Springer, October 2014.

[10] M. J. Freedman, K. Nissim, and B. Pinkas. Efficient private matching and set intersection. In *EUROCRYPT'04*, volume 3027 of *LNCS*, pages 1–19. Springer, 2004.

[11] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game or a completeness theorem for protocols with honest majority. In *STOC'87*, pages 218–229. ACM, 1987.

[12] Y. Huang, D. Evans, and J. Katz. Private set intersection: Are garbled circuits better than custom protocols? In *NDSS'12*. The Internet Society, 2012.

[13] B. A. Huberman, M. Franklin, and T. Hogg. Enhancing privacy and trust in electronic communities. In *EC'99*, pages 78–86. ACM, 1999.

[14] S. Jarecki and X. Liu. Efficient oblivious pseudorandom function with applications to adaptive OT and secure computation of set intersection. In *TCC'09*, volume 5444 of *LNCS*, pages 577–594. Springer, 2009.

[15] S. Kamara, P. Mohassel, M. Raykova, and S. Sadeghian. Scaling private set intersection to billion-element sets. In *FC'14*, volume 8437 of *LNCS*, pages 195–215. Springer, 2014.

[16] A. Kirsch, M. Mitzenmacher, and U. Wieder. More robust hashing: Cuckoo hashing with a stash. *SIAM Journal of Computing*, 39(4):1543–1561, 2009.

[17] V. Kolesnikov and R. Kumaresan. Improved OT extension for transferring short secrets. In *CRYPTO'13 (2)*, volume 8043 of *LNCS*, pages 54–70. Springer, 2013.

[18] C. Meadows. A more efficient cryptographic matchmaking protocol for use in the absence of a continuously available third party. In *S&P'86*, pages 134–137. IEEE, 1986.

[19] R. Pagh and F. F. Rodler. Cuckoo hashing. In *European Symposium on Algorithms (ESA'01)*, volume 2161 of *LNCS*, pages 121–133. Springer, 2001.

[20] R. Pagh and F. F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[21] B. Pinkas, T. Schneider, G. Segev, and M. Zohner. Phasing: Private set intersection using permutation-based hashing. Cryptology ePrint Archive, Report 2015/634, 2015. `http://eprint.iacr.org/2015/634`.

[22] B. Pinkas, T. Schneider, and M. Zohner. Faster private set intersection based on OT extension. In *USENIX Security Symposium*, pages 797–812. USENIX, 2014.

[23] M. Raab and A. Steger. "Balls into bins" - a simple and tight analysis. In *RANDOM'98*, volume 1518 of *LNCS*, pages 159–170. Springer, 1998.

[24] X. Shaun Wang, C. Liu, K. Nayak, Y. Huang, and E. Shi. iDASH secure genome analysis competition using ObliVM. Cryptology ePrint Archive, Report 2015/191, 2015. `http://eprint.iacr.org/2015/191`.

[25] A. C. Yao. How to generate and exchange secrets. In *FOCS'86*, pages 162–167. IEEE, 1986.