Scalable Private Set Intersection Based on OT Extension*

Benny Pinkas† Thomas Schneider‡ Michael Zohner‡

September 26, 2016

Abstract

Private set intersection (PSI) allows two parties to compute the intersection of their sets without revealing any information about items that are not in the intersection. It is one of the best studied applications of secure computation and many PSI protocols have been proposed. However, the variety of existing PSI protocols makes it difficult to identify the solution that performs best in a respective scenario, especially since they were not compared in the same setting. In addition, existing PSI protocols are several orders of magnitude slower than an insecure naive hashing solution which is used in practice.

In this work, we review the progress made on PSI protocols and give an overview of existing protocols in various security models. We then focus on PSI protocols that are secure against semi-honest adversaries and take advantage of the most recent efficiency improvements in OT extension and propose significant optimizations to previous PSI protocols and to suggest a new PSI protocol whose run-time is superior to that of existing protocols. We compare the performance of the protocols both theoretically and experimentally, by implementing all protocols on the same platform, give recommendations on which protocol to use in a particular setting, and evaluate the progress on PSI protocols by comparing them to the currently employed insecure naive hashing protocol. We demonstrate the feasibility of our new PSI protocol by processing two sets with a billion elements each.

Keywords: PSI, Secure Computation

1 Introduction

Private set intersection (PSI) allows two parties P_1 and P_2 holding sets X and Y, respectively, to identify the intersection $X \cap Y$ without revealing any information about elements that are not in the intersection. The basic PSI functionality can be used in applications where two parties want to perform JOIN operations over database tables that they must keep private, e.g., private lists of preferences, properties, or personal records of clients or patients. PSI was used in several research projects for privacy-preserving computation of functionalities such as relationship path discovery in social networks [47], botnet detection [51], testing of fully-sequenced human genomes [4], proximity testing [54], or cheater detection in online games [11].

*This paper is a combined and extended version of [61] (USENIX 2014) and [59] (USENIX 2015) with substantial improvements summarized in §1.4. This work was supported by the European Union's 7th Framework Program (FP7/2007-2013) under grant agreement n. 609611 (PRACTICE). We thank Oleksandr Tkachenko for extending our implementation to allow processing sets with a billion elements.

†Department of Computer Science, Bar-Ilan University, Ramat Gan, Israel. Supported by the Israel Ministry of Science and Technology (grant 3-10883), and by a Magneton grant of the Israeli Ministry of Economy. benny@pinkas.net.

‡Department of Computer Science, TU Darmstadt, Darmstadt, Germany. Supported 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 and CRISP, and by the Hessian LOEWE excellence initiative within CASED. {thomas.schneider,michael.zohner}@crisp-da.de.

PSI has been a very active research field, and there have been many suggestions for PSI protocols. The large number of proposed protocols makes it non-trivial to perform comprehensive cross-evaluations. This is further complicated by the fact that many protocol designs have not been implemented and evaluated, were analyzed under different assumptions and observations, and were often optimized w.r.t. overall run-time while neglecting other relevant factors such as communication. Furthermore, even though several PSI protocols have been introduced, practical applications that need to compute the intersection of privacy-sensitive lists often use insecure solutions. The reason for the poor acceptance of secure solutions is, among others, the poor efficiency of existing schemes, which have more than two orders of magnitude more overhead than insecure solutions.

In this paper, we give an overview on existing efficient PSI protocols, optimize exiting PSI protocols, and describe a new PSI protocol based on efficient oblivious transfer extensions. We compare both the theoretical and empirical performance of all protocols on the same platform and evaluate their cost compared to the insecure hash-based solution used in practice. We show that our new PSI protocol achieves a reasonable overhead compared to solutions used in practice.

1.1 Motivating Applications

The motivation for our work comes from several practical applications which require the PSI functionality. In the following, we list three of these applications:

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 advertising. 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. Our results show that it can be computed using secure protocols even for large data sets.

[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`.

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.[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 $n_1 = 16{,}777{,}216$). It therefore holds that $n_1 \gg n_2$. In our best PSI protocol, the client needs only $\mathsf{O}(n_2 \log n_1)$ memory, $\mathsf{O}(n_2)$ symmetric cryptographic operations and $\mathsf{O}(n_1)$ cheap hash table lookups, and the communication is $\mathsf{O}(n_1 \log n_1)$. (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 Classification of PSI Protocols

Securely intersecting two sets without leaking any information but the result of the intersection is one of the most prominent problems in secure computation. Several techniques have been proposed that realize the PSI functionality, such as the efficient but insecure *naive hashing* solution, protocols that require a *semi-trusted third party*, or *two-party PSI* protocols. The earliest proposed two-party PSI protocols were special-purpose solutions based on *public-key* cryptography. Later, solutions were proposed using *circuit*-based generic techniques for secure computation, that are mostly based on symmetric cryptography. The most recent development are PSI protocols that are based on *oblivious transfer (OT)* alone, and combine the efficiency of symmetric cryptographic primitives with special purpose optimizations.

A naive solution When confronted with the PSI problem, most novices come up with a solution where both parties apply a cryptographic hash function to their inputs and then compare the resulting hashes. Although this protocol is very efficient, it is insecure if the input domain is small or does not have high entropy, since one party could easily run a brute force attack that applies the hash function to all items that are likely to be in the input set and compare the results to the received hashes. (When inputs to PSI have high entropy, a protocol that compares hashes of the inputs can be used [52].)

Third Party-Based PSI Several PSI protocols have been proposed that utilize additional parties, e.g., [5]. In [30], a trusted hardware token is used to evaluate an oblivious pseudo-random function. This approach

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

was extended to multiple untrusted hardware tokens in [21]. Several efficient server-aided protocols for PSI were presented and benchmarked in [36].

Public-Key-Based PSI A PSI protocol based on the Diffie-Hellmann (DH) key agreement scheme was presented in [46] (related ideas were presented in [67, 34]). Their protocol is based on the commutative properties of the DH function and was used for private preference matching, which allows two parties to verify if their preferences match to some degree.

Freedman et al. [25] introduced PSI protocols secure against semi-honest and malicious adversaries in the standard model (rather than in the random oracle model assumed in the DH-based protocol). This protocol was based on polynomial interpolation, and was extended in [23], which presents protocols with simulation-based security against malicious adversaries, and evaluates the practical efficiency of the proposed hashing schemes. We discuss the proposed hashing schemes in §3. A similar approach that uses oblivious pseudo-random functions to perform PSI was presented in [24]. A protocol that uses polynomial interpolation and differentiation for finding intersections between multi-sets was presented in [38].

Another PSI protocol that uses public-key cryptography (more specifically, blind-RSA operations) and scales linearly in the number of elements was presented in [15] and efficiently implemented and benchmarked in [16]. In [17], a family of Bloom filter-based PSI protocols was introduced that realize PSI, PSI cardinality and authenticated PSI functionalities. These protocols also use public-key operations, linear in the number of elements.

Circuit-Based PSI Generic secure computation protocols have been subject to substantial efficiency improvements in the last decade. They allow the secure evaluation of arbitrary functions, expressed as Arithmetic or Boolean circuits. Several Boolean circuits for PSI were proposed in [32] and evaluated using the Yao's garbled circuits framework of [33]. The authors showed that their Java implementation scales very well with increasing security parameter and outperforms the blind-RSA protocol of [15] for larger security parameter. We reflect on and present new optimizations for circuit-based PSI in §4.

OT-Based PSI A recent PSI protocol of [20] uses Bloom filters [9] and OT extension [35] to obtain very efficient PSI protocols with security against semi-honest and malicious adversaries. Recently in [43, 64], it was shown that the Bloom filter-based protocol is insecure with respect to malicious adversaries. The authors of [64] showed how to fix the malicious secure Bloom filter-based protocol and gave the first implementation of a malicious secure PSI protocol, which computes the intersection of two sets with a million elements each in ~ 200 s.

In [63], our OT-based PSI protocol of [61] was extended to security against weakly malicious adversaries and used as a building block in a batch dual-execution Yao's garbled circuits protocol. In [43], our OT-based PSI protocol of [59] was secured against a semi-honest P_1 and malicious P_2. An improved version of our OT-based PSI protocol in [59] is given in [40], which presents an efficient construction of an oblivious pseudo-random function (OPRF) using the OT extension protocol of [39] (cf. §2.2.3). The main observation of the authors is that the [39] OT extension does not require an error correcting code but can instead use a pseudo-random code, which can be generated from a pseudo-random generator. The authors then apply their efficient OPRF construction to our [59] protocol and thereby achieve performance independent of the bit-length of elements σ. The OPRF construction of [40] is similar to our new OPRF construction described in §5. The idea of both works is to instantiate the OPRF that is implicitly used in the [59] OT-based PSI protocol using larger codes. However, while [40] replace the error correcting code with a pseudo-random code, we keep the error correcting code. Thereby, our OPRF construction achieves better communication for smaller σ when using custom-tailored error correcting codes but does not achieve performance independent of σ. In addition, the work of [40] can also benefit from our improved hashing analysis of §3.

1.3 Our Contributions

We survey existing PSI protocols that are secure against semi-honest adversaries as well as solutions that employ a trusted third party. We then describe in detail the semi-honest secure PSI protocols and suggest how to improve the performance of some protocols using carefully analyzed features of OT extension. We introduce a new OT-based PSI protocol, perform a detailed experimental comparison of the most promising semi-honest secure PSI protocols, and evaluate their overhead compared to the insecure naive hashing protocol that is currently used in real-world applications. Our implementations for selected protocols are available online at `http://github.com/encryptogroup/PSI` and `http://github.com/encryptogroup/ABY`. In the following, we detail our contributions.

Concrete Parameter Estimation for Hashing In [25] the use of hashing-to-bins was suggested in the context of PSI to reduce the overhead for pairwise-comparisons. However, their analysis of the involved parameters was only asymptotic. In §3, we empirically analyze the hashing-to-bins techniques that were suggested in [25] and identify concrete parameters for the schemes. In addition, in §3.3 we utilize the permutation-based hashing techniques of [2] to reduce the bit-length of the representations that are stored in the bins. This improves the performance of PSI protocols that require an overhead linear in the bit-length of elements, e.g., the protocols in §4.3 and §5.

Optimizations of Existing Protocols We improve the circuit protocols using recent optimizations for OT extension [3]. In particular, in §4 we evaluate the circuit-based solution of [32] on a secure evaluation of the GMW protocol, and utilize features of random OT (cf. §2.2) to optimize the performance of multiplexer gates (which form about two thirds of the circuit). Furthermore, in §4.3 we outline how to use the permutation-based hashing techniques to improve the performance of circuit-based PSI even further.

A Novel OT-Based PSI Protocol We present a new PSI protocol that is based on OT (§5) and directly benefits from improvements in efficient OT extensions [39, 3]. Our PSI protocol uses an efficient oblivious pseudo-random function that is instantiated based on the $\binom{N}{1}$-OT extension protocol of [39] and uses the hashing techniques from §3 to reduce the communication overhead from $O(n^2)$ to $O(n)$. The resulting protocol has very low computation complexity since it mostly requires symmetric key operations and has even less communication than some public-key-based PSI protocols, which had the lowest communication before.

A Detailed Comparison of PSI Protocols We implement the most promising candidate PSI protocols using state-of-the-art cryptographic techniques and compare their performance on one platform. As far as we know, this is the first time that such a wide comparison has been made, since previous comparisons were either theoretical, compared implementations on different platforms or programming languages, or used implementations without state-of-the-art optimizations. Our implementations and experiments are described in detail in §6. Certain experimental results were unexpected. We give a partial summary of our results in Tab. 1. We briefly describe our most prominent findings next.

PSI Protocol	Hashing	Server-Aided [36]	Public Key [46]	Circuit PWC §4.3 / OPRF §4.4	OT+Hashing §3+§5
Equal set sizes $n_1 = n_2 = 2^{20}$					
Runtime (s)	0.7	1.3	818.3	83.9	5.6
Comm. (MB)	10	20	74	9,170	107
Unequal set sizes $2^{24} = n_1 \gg n_2 = 2^{12}$					
Runtime (s)	6.1	7.6	12,712.3	7.3	35.1
Comm. (MB)	160	160	593	947	362

Table 1: Runtime and transferred data for private set intersection protocols on sets with 2^{20} $\sigma = 32$-bit elements and 128-bit security with a single thread over Gigabit LAN.

- The Diffie-Hellman-based protocol [46], which was the first PSI protocol, is actually the most efficient w.r.t. communication (when implemented using elliptic-curve crypto). Therefore it is suitable for settings with distant parties which have strong computation capabilities but limited connectivity.
- Generic circuit-based protocols [32] are less efficient than the newer, OT-based constructions, but they are more flexible and can easily be adapted for computing variants of the set intersection functionality (e.g., computing whether the size of the intersection exceeds some threshold). Our experiments also support the claim of [32] that circuit-based PSI protocols are faster than the blind-RSA-based PSI protocol of [15] for larger security parameters and given sufficient bandwidth.
- Compared to the insecure naive hashing solution, previous PSI protocols are at least two orders of magnitude less efficient in run-time or communication. Our OT-based PSI protocol reduces this overhead to only one order of magnitude in both run-time and communication.
- When intersecting sets with unequal sizes ($n_1 \gg n_2$), the run-time difference between most protocols remains similar to the run-time difference for equal set sizes ($n_1 = n_2$). The only exception is the circuit-based oblivious pseudo-random function protocol (§4.4), which achieves similar performance as the naive hashing and server-aided solutions.

1.4 Additions to Conference Versions

This journal article is a significantly extended and improved version of the conference publications [61] and [59]. Compared to the conference papers, we add the following contributions:

Broader scope We broadened the scope of the work by describing and benchmarking the circuit-based OPRF protocol of [60] in §4.4.

Extended Hashing Parameter Analysis We extend the hashing parameter analysis for schemes that are using pairwise comparison. In our previous works, we only bounded the hashing failure for one particular set of parameters that is tailored to one use-case. However, the hashing parameters for which PSI protocols perform well change depending on the settings (unequal set sizes, different networks, etc.). We show a trade-off between different parameters, resulting in a large variety of parameters which perform well for different settings.

Optimizations In previous works, our OT-based PSI protocol scaled linear in the bit-length of the inputs, which decreased its performance on arbitrary input data. We now outline how to achieve performance *independent* of the bit-length in §5 by using more efficient instantiations of underlying primitives (cf. §2.2.3).

Comparison We extend the theoretical and empirical comparison between the protocols. In §6.1 we perform a broader theoretical comparison between the protocols and discuss their implementation features. Furthermore, we extended our analysis to cover multi-threading (§6.3).

Unequal Set Sizes We extend the focus of the work to unequal set sizes where $n_1 \gg n_2$. This setting is relevant for use-cases where, e.g., an end user wants to compare its data (few hundred elements) to a company's database (several million elements). We show how to modify the circuit-based protocols (§4.4) as well as our OT-based protocol (§3.2.2) to efficiently extend to this setting, and perform experiments for the protocols (§6.2.3).

Scalability The largest sets on which secure two-party PSI protocols were evaluated until now were of size 2^{24} [59]. We demonstrate the scalability of our novel OT-based PSI protocol by processing two sets of a billion $\sigma = 128$-bit elements each (§6.4).

2 Preliminaries

We give our notation and security definitions in §2.1, review recent relevant work on oblivious transfer in §2.2, and outline how to hash large inputs into smaller domains in §2.3.

2.1 Notation and Security Definitions

We denote the parties as P_1 and P_2, and their respective input sets as X and Y with $|X| = n_1$ and $|Y| = n_2$. We refer to elements from X as x and elements from Y as y and each element has bit-length σ (cf. §2.3 for the relation between n and σ). We write $b[i]$ for the i-th element of a list b, denote the bitwise-AND between two bit strings a and b of equal length as $a \wedge b$ and the bitwise-XOR as $a \oplus b$. We denote a constant string of m zeros (or ones) as 0^m (or 1^m). We refer to a correlation resistant one-way function as CRF, to a pseudo-random generator as PRG, to a pseudo-random permutation as PRP, and to an oblivious pseudo-random function as OPRF (see definitions below). We write $\binom{N}{1}$-OT_ℓ^m for m parallel 1-out-of-N oblivious transfers on ℓ-bit strings, and write OT_ℓ^m for $\binom{2}{1}$-OT_ℓ^m. In a similar fashion, we denote the random OT functionality (cf. §2.2.2), where the functionality chooses m N-tuples of random ℓ-bit strings as $\binom{N}{1}$-ROT_ℓ^m. We fix the key sizes according to the NIST guideline [56]: the symmetric security parameter as $\kappa = 128$, the asymmetric security parameter as $\rho = 3{,}072$, the statistical security parameter as $\lambda = 40$, and the elliptic curve size $\varphi = 284$ for Koblitz curve K-283 when using point compression (this is the number of bits for one coordinate and a sign-bit). We denote and fix the hashing failure parameter which affects the correctness of some protocols as $\eta = 30$, meaning that hashing failures occur with probability $< 2^{-30}$.

Adversary definition The secure computation literature considers two types of adversaries with different strengths: A *semi-honest adversary* tries to learn as much information as possible from a given protocol execution but is not able to deviate from the protocol steps. The semi-honest adversary model is appropriate for scenarios where execution of the intended software is guaranteed via software attestation or where 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. The stronger, *malicious adversary* extends the semi-honest adversary by being able to deviate arbitrarily from the protocol.

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 considerably less efficient than protocols for the semi-honest setting (see, e.g., [25, 14, 64]).

The random oracle model As most previous works on efficient PSI, we use the random oracle model to achieve more efficient implementations [8]. The security of cryptographic constructions can be proven in the standard model, or in the "random oracle model", which is based on modeling a hash function as a random function [8]. There are many criticisms about the random oracle model, and in the theory of cryptography proofs in this model are considered heuristic. Yet, protocols in the random oracle are often more efficient than protocols that are proven in the standard model.

The efficiency gain in using the random oracle model is particularly true with regards to protocols for private set intersection. The only semi-honest protocol that we describe that is in the standard model is the protocol based on oblivious polynomial evaluation by [25, 23], but that protocol is less efficient than the other protocols that we present. The public-key-based protocols (based on Diffie-Hellman and blind-RSA) use a hash function $H()$ that must be modeled as a random oracle, or modeled using another non-standard assumption. The other protocols (the generic protocol, as well as the protocol based on Bloom filters and the new OT-based protocol) can be implemented without a random oracle assumption, but in order to speedup the computation of OT in these protocols we must use random OT extension, whose efficient implementation relies on a function that must be modeled as a random oracle.

Correlation-robustness A correlation robust one-way function (CRF) $H : \{0,1\}^\kappa \mapsto \{0,1\}^\ell$ is a function for which, given uniformly and randomly chosen $t_1, ..., t_m, s$, an adversary is unable to computationally distinguish the outputs $H(t_1 \oplus s)$, ..., $H(t_m \oplus s)$ from uniform distribution. It is a weaker assumption than the random oracle model and is used in OT extension as well as Yao's garbled circuit protocol. Traditionally, many implementations use a hash function (e.g., SHA) to increase the performance. An instantiation of the CRF in Yao's garbled circuit protocol which uses fixed-key AES and greatly improves performance was proposed in [7] and refined in [69] for use in the half-gates scheme. In this paper, we use both optimizations.

Oblivious Pseudo-Random Functions An oblivious pseudo-random function (OPRF) [24] is a function $F : (\{0,1\}^\kappa, \{0,1\}^\sigma) \mapsto (\perp, \{0,1\}^\ell)$ that, given a key k from P_1 and an input element e from P_2, computes and outputs $F_k(e)$ to P_2. P_1 obtains no output and learns no information about e while P_2 learns no information about k. OPRFs can be used for PSI by first evaluating the OPRF protocol on the set of P_2 and then having P_1, who knows the secret key k, evaluate the OPRF locally on its own set, and send the OPRF output to P_2, who computes a plaintext intersection. There exist several instantiations for OPRFs, described in [24]: based on generic secure computation techniques (using an AES circuit [60]), based on the Diffie-Hellman assumption, or based on OT. In §4.4 we analyze the efficiency of generic secure computation-based OPRF instantiations and in §5 we give a more efficient OT-based instantiation.

2.2 Oblivious Transfer

Oblivious transfer (OT) is a major building block for secure computation. When executing m invocations of 1-out-of-2 OT on ℓ-bit strings (denoted $\binom{2}{1}$-OT$^m_\ell$), the sender S holds m message pairs (x_0^i, x_1^i) with $x_0^i, x_1^i \in \{0,1\}^\ell$, while the receiver R holds an m-bit choice vector b. At the end of the protocol, R receives $x_{b[i]}^i$ but learns nothing about $x_{1-b[i]}^i$, and S learns nothing about b. Many OT protocols have been proposed, most notably (for the semi-honest model) the Naor-Pinkas OT [53], which uses public-key operations and has amortized complexity of $3m$ public-key operations when performing m OTs.

OT extension [6, 35] reduces the number of expensive public-key operations for OT_{ℓ}^{m} to that of only $\mathrm{OT}_{\kappa}^{\kappa}$, and computes the rest of the protocol using more efficient symmetric cryptographic operations which faster by are orders of magnitude. The security parameter κ is essentially independent of the number of OTs m, and can be as small as 128. Thereby, the computational complexity for performing OT is reduced to such an extent, that the network bandwidth becomes the main bottleneck [22, 3].

Recently, the efficiency of OT extension protocols has gained a lot of attention. In [39], an efficient $\binom{N}{1}$-OT extension protocol was shown, that has sub-linear communication in κ for short messages. Another protocol improvement is outlined in [3, 39], which decreases the communication from R to S by half. Additionally, several works [3, 55] improve the efficiency of OT by using an OT variant, called *random OT*. In random OT, (x_0^i, x_1^i) are chosen uniformly and randomly within the OT and are output to S, thereby removing the final message from S to R. Random OT is useful for many applications, and we show how it can reduce the overhead of PSI.

We describe the OT extension protocol of [3, 35], the random OT functionality, and the $\binom{N}{1}$-OT extension protocol of [39] in more detail next.

2.2.1 1-out-of-2 OT Extension

In [35], a $\binom{2}{1}$-OT extension protocol was outlined that extends $\mathrm{OT}_{\kappa}^{\kappa}$ (κ OTs of κ bits) to OT_{ℓ}^{m} (m OTs of ℓ bits). We describe the OT extension protocol of [35] with optimizations of [3, 39] in Prot. 1.

PROTOCOL 1 (OT Extension Protocol of [35])

- **Input of P_1:** m pairs of ℓ-bit strings (x_0^i, x_1^i), $1 \leq i \leq m$.
- **Input of P_2:** A choice vector $b \in \{0,1\}^m$.
- **Common Input:** Symmetric security parameter κ.
- **Oracles and cryptographic primitives:** Both parties have access to an $\mathrm{OT}_{\kappa}^{\kappa}$ oracle, a PRG $G : \{0,1\}^{\kappa} \mapsto \{0,1\}^m$, and a CRF $H : \{0,1\}^{\kappa} \mapsto \{0,1\}^{\ell}$.

1. P_1 initializes a random vector $s \in \{0,1\}^{\kappa}$ and P_2 chooses κ random key pairs $(k_0^i, k_1^i) \in \{0,1\}^{2\kappa}$, for $1 \leq i \leq \kappa$.
2. The parties invoke the $\mathrm{OT}_{\kappa}^{\kappa}$ oracle where, in the i-th OT, P_1 plays the receiver with input $s[i]$ and P_2 plays the sender with inputs (k_0^i, k_1^i).
3. P_2 computes $t^i = G(k_i^0)$ and $u^i = t^i \oplus G(k_i^1) \oplus b$, and sends u^i to P_1, for $1 \leq i \leq \kappa$. Let $T = [t^1|...|t^{\ell}]$ denote a random $m \times \ell$ bit matrix that is generated by P_2 where the i-th column is t^i and the j-th row is t_j, for $1 \leq i \leq \ell$ and $1 \leq j \leq m$.
4. P_1 defines $q^i = (s[i] \cdot u^i) \oplus G(k_i^{s[i]})$ (note that $q^i = (s[i] \cdot b \oplus t^i)$.
5. Let $Q = [q^1|...|q^{\kappa}]$ denote the $m \times \kappa$ bit matrix where the i-th column is q^i. Let q_j denote the j-th row of the matrix Q (note that $q_j = (b[j] \cdot s) \oplus t_j$.
6. P_1 sends (y_j^0, y_j^1) for every $1 \leq j \leq m$, where:
$$y_j^0 = x_j^0 \oplus H(q_j) \text{ and } y_j^1 = x_j^1 \oplus H(q_j \oplus s)$$
7. P_2 computes $x_j^{b[j]} = y_j^{b[j]} \oplus H(t_j)$, for $1 \leq j \leq m$.
8. **Output:** P_1 has no output; P_2 outputs $(x_{b[1]}^1, ..., x_{b[m]}^m)$.

Efficiency Overall, when using OT extension, the sender in OT^m_ℓ has to evaluate $2m$ CRFs and m PRGs, and send $2m\ell$ bits, while the receiver has to evaluate m CRFs and $2m$ PRGs, and send $m\kappa$ bits. (In addition, there is a preprocessing cost of OT^κ_κ public-key-based OTs, which is negligible compared to the main protocol if $\kappa \ll m$.)

2.2.2 Random OT Extension

To improve efficiency of OT extension protocols in specific settings, several works [55, 3] use a special purpose OT functionality, called random OT. In a random OT, (x^i_0, x^i_1) are chosen uniformly and randomly during the OT and are output to P_1. A random OT extension protocol can be obtained by leaving out the last message from P_1 to P_2, containing (y^i_0, y^i_1). More detailed, P_1 has no input to the protocol and sets $(x^j_0 = H(q_j), x^j_1 = H(q_j \oplus s))$ in Step 6 in Prot. 1 while P_2 sets $x^i_{b[j]} = H(t_j)$ in Step 7. P_1 then outputs m pairs of ℓ-bit strings (x^j_0, x^j_1) and P_2 outputs $x^j_{b[j]}$. This random OT extension protocol reduces the bits that P_1 has to send from $2m\ell$ to 0 at the expense of the stronger assumption that H is modeled as a RO instead of a CRF.

2.2.3 1-out-of-N OT Extension

In [39], an efficient $\binom{N}{1}$-OT extension protocol was introduced which allows to transfer short messages with sublinear communication in κ. The protocol builds on the original OT extension protocol of [35] and encodes the choices of R using an error correcting code $C^N = c_0, ..., c_{N-1}$, which encodes $\lceil \log_2 N \rceil$-bit words with p-bit codewords that have at least κ Hamming distance from each other. More detailed, in the i-th $\binom{N}{1}$-OT, S inputs $x^i_0, ..., x^i_{N-1}$ and R inputs $b_i \in [0...N-1]$. The parties perform p base-OTs such that S holds $s \in_R \{0,1\}^p$ and $k^j_{s[j]}$ and R holds k^j_0 and k^j_1 (p is a security parameter, see below). R then computes $m \times p$ matrices T and U as $t^j = G(k^j_0)$ and $u^j = G(k^j_1)$ and transfers $v_i = t_i \oplus u_i \oplus c_{b[i]}$ (note that we address v and t row-wise instead of column-wise as in the original OT extension protocol). As in the original protocol, S then generates a $m \times p$ bit-matrix Q as $q^j = v^j_{s[j]} \oplus G(k^j_{s[j]})$ and transfers $y^i_w = x^i_w \oplus H(q_i \oplus (s \wedge c_w))$ to R, where $\wedge$ is the bitwise-AND and $0 \leq w < N$. Finally, R obtains his output $x^i_{b[i]} = y^i_{b[i]} \oplus H(t_i)$.

Two things are noteworthy in this $\binom{N}{1}$-OT extension protocol. Firstly, we can also use the random OT extension functionality by having S set $x^i_w = H(q_i \oplus (s \wedge c_w))$ and R set $x^i_{b[i]} = H(t_i)$. Secondly, in order to achieve the same computational security level $\kappa = 128$ as in the original $\binom{2}{1}$-OT extension protocol of [35], the parties have to increase the number of base-OTs to the codeword length p of the underlying code, which depends on N (cf. [39]). The reason for the increase in base-OTs is that the Hamming distance between the codewords has to be at least κ. For $2 < N \leq 2\kappa$, [39] proposes to use the Walsh-Hadamard code, which encodes up to 2κ words to codewords of length 2κ with relative Hamming distance κ. However, in our OT-based PSI protocol, we use σ-bit elements as input to the $\binom{N}{1}$-OT protocol of [39] and hence need to handle $2^\sigma = N \gg 2\kappa$. In order to process such a σ-bit element, we need to find an error correcting code that processes 2^σ input elements with codewords of relative Hamming distance 128-bit and short codesize. As an example, when processing $\sigma = 13$-bit elements, we could use a code of size 271-bit, as given in [66].

In the remainder of the paper and for ease of presentation, we fix a linear BCH code, generated from [49], which encodes up to 2^{77} words to codewords of length 512 with relative Hamming distance κ, which is denoted as a [2^{77}, 512, 129] code. Using the permutation-based hashing techniques, outlined in §3.3, and assuming a statistical security of $\lambda = 40$ bit, this allows us to process sets with up to 100 billion (2^{37}) elements independently of their bit-length σ, which suffices for most applications.

Efficiency Evaluating one $\binom{N}{1}$-ROT using the $\binom{N}{1}$-OT extension protocol of [39] and our linear BCH code requires 512 bits of communication and N CRF evaluations. Note that, although the high number of CRF evaluations for the $\binom{N}{1}$-OT seems prohibitive for large N, we only need to perform a constant number (say 3) of CRF evaluations in our protocol. In comparison, naively building $\binom{N}{1}$-ROT from $\binom{2}{1}$-OT extension would require $\log N$ $\binom{2}{1}$-OT invocations and hence require $128 \log N$ bits of communication and $2 \log N$ CRF evaluations. More concretely, when computing the intersection between two million element sets using our OT-based PSI protocol in §5, we would have $N \approx 60$ and hence would require 512 bit communication using the $\binom{N}{1}$-OT extension protocol of [39] and 7,680 bits communication using the regular $\binom{2}{1}$-OT extension protocol of [35] with most recent optimizations of [3, 39].

2.3 Hashing Inputs to a Smaller Domain

The performance of some PSI protocols depends on the length of the representation of their inputs. This is particularly true for protocols that run an OT for each bit of the input representation, e.g., the protocols described in §4.3 and §5.

When the original input representation is sparse, we can first use a hash function to map the identities of the input items to identities from a smaller domain with a shorter representation. We then run the original protocol on that representation, resulting in a more efficient execution. The size of the new domain should be large enough so that no two different input items are mapped to the same value. The theoretical analysis of this mapping, related to the birthday paradox, shows that when n items are mapped to a domain of size D using a random hash function, the probability of experiencing a collision is $p = 1 - e^{-n\cdot(n-1)/(2D)}$, and can be approximated as $p \approx n^2/(2D)$ (see [50], p. 45).

Let us denote the length of the representation of items in D as $d = \log D$. Then $p \approx n^2/(2 \cdot 2^d)$, and therefore

$$d = 2\log(n) - 1 - \log(p).$$

3 Hashing Schemes and PSI

Computing the plaintext intersection between two sets is often done using *hashing techniques*. The parties agree on a publicly known random hash function to map elements to a *hash table*, which consists of multiple *bins*. If an input element is in the intersection, both parties map it to the same bin. Hence, the parties only need to compare the elements that are in the same bin to identify intersecting elements. Thereby, the average number of comparisons between elements can be reduced from $O(n^2)$ to $O(n)$ for pairwise comparisons.

In a similar fashion, PSI protocols that privately compute the equality between values can use hashing techniques in order to reduce the number of comparisons [25, 23]. Examples for such private equality test protocols are [25, 32, 13], the circuit-based protocol in §4.3 or our OT-based protocol in §5. When naively using hashing techniques, if n items are mapped to n bins then the average number of items in a bin is $O(1)$, checking for an intersection in a bin takes $O(1)$ work, and hence the total number of comparisons is $O(n)$. However, privacy requires that the parties hide from each other how many of their inputs were mapped to each bin.[4] As a result, we must calculate in advance the number of items that will be mapped to the *most populated* bin (w.h.p.), and then set all bins to be of that size. (This can be done by storing dummy items in bins which are not fully occupied.) This change hides the bin sizes but also increases the overhead of the

[4]Otherwise, and since the hash function is public, some information is leaked about the input. For example, if no items of P_1 were mapped to the first bin by the hash function h, then P_2 learns that P_1 has no inputs in the set $h^{-1}(1)$, which covers about $1/n$ of the input range.

protocol, since the number of comparisons per bin now depends on the size of the most populated bin (worst case) rather than on the actual number of items in the bin (average case).

In fact, this worst case analysis is key to balancing security and efficiency. On the one hand, if the estimation is too optimistic, the probability of a party failing to perform the mapping becomes intolerable. As a result, the output might be inaccurate (since not all items can be mapped to bins), or one party needs to request a new hash function (a request that leaks information about the input set of that party). On the other hand, the number of performed comparisons and hence the protocol overhead can become prohibitive if the analysis is too pessimistic. The work of [25, 23] gave asymptotic values for this analysis and of the resulting overhead. They left the task of setting appropriate parameters for the hashing schemes to future work.

In this section, we revisit the simple hashing (§3.1) and Cuckoo hashing (§3.2) schemes, used in [25, 23]. We describe how to use both hashing schemes in the context of PSI and give a concrete parameter analysis that balances security and efficiency. Finally, we show how the bit-length of the representations that are stored in the bins can be reduced using permutation-based mapping, which improves the performance of some PSI protocols (§3.3).

Note that, for our hashing failure analysis, we use a dedicated hashing failure parameter η, which is different from the statistical security parameter λ. We use a dedicated parameter since our analysis requires running empirical experiments for determining concrete numbers, which would have cost several hundred thousand USD for 2^{40} iterations in the Amazon EC2 cloud. Hence, we perform the experiments and give concrete numbers for $\eta = 30$ and interpolate from these results to $\eta = 40$.

3.1 Simple Hashing

In the simplest hashing scheme, the hash table consists of b bins $B_1...B_b$. Hashing is done by mapping each input element e to a bin $B_{h(e)}$ using a hash function $h : \{0,1\}^\sigma \mapsto [1, b]$ that was chosen uniformly at random and independently of the input elements. An element is always added to the bin to which it is mapped, regardless of whether other elements are already stored in that bin.

3.1.1 Simple Hashing for PSI

To apply simple hashing in the context of PSI, both parties map their elements to b bins. The intersection is then computed by having both parties separately compare the items mapped to bin $i \in [1, ..., b]$. In order to hide the number of elements that were mapped to a bin, the parties need to pad their bins using dummy elements to contain max_b elements. This maximum bin size must ensure that except with probability $< 2^{-\eta}$, no bin will contain more than max_b real elements.

3.1.2 Simple Hashing Parameter Analysis

Estimating max_b has been subject to extensive research [28, 62, 48]. When hashing n elements to $b = n$ bins, [28] showed that $max_b = \frac{\ln n}{\ln \ln n}(1 + o(1))$ w.h.p. In this case, there is a difference between the expected and the maximum number of elements mapped to a bin, which are 1 and $O(\frac{\ln n}{\ln \ln n})$, respectively. Let us examine in more detail the probability of the following event, "n balls are mapped at random to b

bins, and the most occupied bin has at least k balls":

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

Case $n = b$ We calculate max_b when mapping $n \in \{2^8, 2^{12}, 2^{16}, 2^{20}, 2^{24}\}$ elements to $b = n$ bins using Eq. (5), choose the minimal value of k that reduces the failure probability to below 2^{-30} and 2^{-40} and depict the results in Tab. 2.

Hash Failure Parameter ϕ	**30**					**40**				
Set Size n	2^8	2^{12}	2^{16}	2^{20}	2^{24}	2^8	2^{12}	2^{16}	2^{20}	2^{24}
max$_b$ (Eq. 5)	16	17	18	19	20	17	18	19	20	21

Table 2: The bin sizes max_b that are required to ensure that no overflow occurs when mapping n items to $b = n$ bins, according to Eq. (5).

Case $n \gg b$ In certain settings, the server P_1 has a much larger set than the client P_2. For simple hashing, this translates to the number of elements n being much larger than the number of bins b. Later in the paper, we perform experiments for this setting (cf. §6.2.3), where P_2 has a set of size $n_2 \in \{2^8, 2^{12}\}$, while P_1 has a set of size $n_1 \in \{2^{16}, 2^{20}, 2^{24}\}$ and both map $n = 2n_1$ elements into $b = 2.4n_2$ bins. To determine max_b in this setting, we evaluate Eq. 5 with these set sizes and depict max_b for hashing failure probabilities 2^{-30} and 2^{-40} in Tab. 3. From the results, we can observe that as the fraction n_1/n_2 grows, the maximum number of bin grows closer to the expected number of bins.

Set Size n_2	2^8			2^{12}		
Set Size n_1	2^{16}	2^{20}	2^{24}	2^{16}	2^{20}	2^{24}
Hash Failure Parameter $\eta = 30$						
max$_b$ (Eq. 5)	607	9,306	148,482	60	610	9,309
Hash Failure Parameter $\eta = 40$						
max$_b$ (Eq. 5)	614	9,313	148,489	65	616	9,316

Table 3: The bin sizes max_b that are required to ensure that no overflow occurs when mapping $n = 2n_1$ items to $b = 2.4n_2$ bins for $n_1 \gg n_2$, according to Eq. (5).

3.2 Cuckoo Hashing

Cuckoo hashing [57] uses k hash functions $h_1, ..., h_k : \{0,1\}^\sigma \mapsto [1, b]$ to map m elements to $b = \epsilon n$ bins. The scheme avoids collisions by relocating elements when a collision is found using the following procedure: An element e is inserted into a bin $B_{h_1(e)}$. Any prior contents o of $B_{h_1(e)}$ are evicted to a new bin $B_{h_i(o)}$, using h_i to determine the new bin location, where $h_i(o) \neq h_1(e)$ for $i \in [1...k]$. 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 put in a special stash. A lookup in this scheme is very efficient as it only compares e to the k items in $B_{h_1(e)}, ..., B_{h_k(e)}$ and to the s items in the stash. The size of the hash table depends on the number of hash functions k as well as on the stash size s. The higher k is chosen, the more likely it is that the insertion process succeeds and hence the smaller the number of bins b becomes. On the other hand, the higher s is chosen, the more insertion failures can be tolerated.

3.2.1 Cuckoo Hashing for PSI

A major problem occurs when using Cuckoo hashing for PSI: every item can be mapped to one of k bins, and therefore it is unclear with which of P_1's bins should P_2 compare its own input elements. Furthermore, the protocol must hide from each party the choice of bins made by the other party to store an element, since that choice depends on other input elements and might reveal information about them. The solution to this is that P_2 uses Cuckoo hashing whereas P_1 maps each of its elements using simple hashing with each of the k hash functions. In addition, for Cuckoo hashing, we must ensure that the hashing succeeds except with probability $< 2^{-\eta}$, since a hashing error on the side of P_2 reveals information about its set or results in an incorrect result. As in PSI with simple hashing (cf. §3.1), P_1 will need to pad its bins to size max_b using dummy elements $d_1 \neq d_2$.

3.2.2 Cuckoo Hashing Parameter Analysis

Cuckoo hashing has three parameters that affect the hashing failure probability: the stash size s, the number of hash functions k, and the number of bins $b = \epsilon n$ [37]. It was shown in [37] that Cuckoo hashing of n elements into $(1+\zeta)n$ bins with $\zeta \in (0,1)$ for any $k \geq 2(1+\zeta)\log(\frac{e}{\zeta})$ and $s \geq 0$ fails with probability $O(n^{(1-k)(s+1)})$. The constants in the big "O" notation are unclear, which makes it hard to compute a concrete failure probability given a set of parameters.

In the following, we empirically determine the failure probability given the stash size s, the number of hash functions k, and the number of bins b. We analyze the effect of all three parameters separately. We first fix the number of bins $b = 2.4n$ and hash functions $k = 2$ (as was done in [37]) and determine the necessary stash sizes s. In order to improve performance, we increase the number of hash functions k and determine the number of bins b for which no stash is required (i.e., $s = 0$). While both approaches achieve good overhead when $n_1 = n_2$, they perform poorly when the parties have unequal set sizes $n_1 \gg n_2$. Hence, in the last step, we show how to obtain a low values for the stash size s and a low number of hash functions k by increasing the number of bins $b = \epsilon n$, which results in a collection of trade-offs for unequal set size applications.

Adjusting the Stash Size s In the following, we identify the exact stash size s that ensures that the hashing failure probability is smaller than a given $2^{-\eta}$. To obtain concrete numbers, we ran 2^{30} repetitions of Cuckoo hashing, where we mapped n items to $b = \epsilon n = 2.4n$ bins, for $n \in \{2^{11}, 2^{12}, 2^{13}, 2^{14}\}$, using $k = 2$ hash functions and recorded the stash size s that was needed for Cuckoo hashing to be successful. We fix $\epsilon = 2.4$ as was done in the original Cuckoo hashing with a stash paper [37]. The solid lines in Fig. 1 depict the probability that a stash of size s prevented a hashing failure.

From the results we can observe that, to achieve 2^{-30} failure probability of Cuckoo hashing, we 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 smaller as well as larger values of n to achieve a Cuckoo hashing failure probability of 2^{-30}. To obtain the failure probabilities for larger values of n, we extrapolate the results using linear regression and illustrate the results as dotted lines in Fig. 1. We give the extrapolated stash sizes for achieving a hashing failure probability of 2^{-30} and 2^{-40} for $n \in \{2^8, 2^{12}, 2^{16}, 2^{20}, 2^{24}\}$ in Tab. 4. We observe that the stash size for achieving a failure probability of 2^{-30} 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 $O(n^{-s})$.

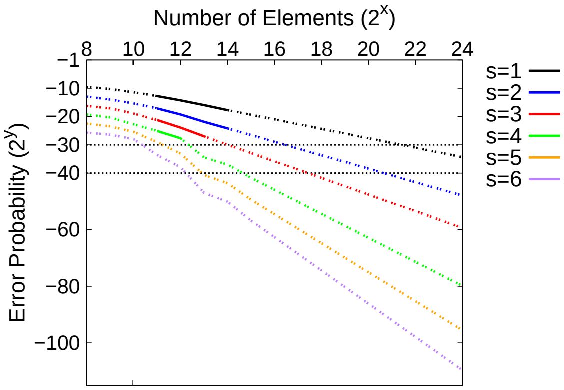

Figure 1: Error probability when mapping n elements to $2.4n$ bins using Cuckoo hashing with $k = 2$ hash functions for stash sizes $1 \leq s \leq 6$. The solid lines correspond to actual measurements, the dashed lines were extrapolated using linear regression.

# Elements n	2^8	2^{12}	2^{16}	2^{20}	2^{24}
Stash s ($\eta = 30$)	8	5	3	2	1
Stash s ($\eta = 40$)	12	6	4	3	2

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

Adjusting the Number of Hash Functions k The original Cuckoo hashing procedure [58] fixed the number of hash functions $k = 2$. It was later shown in [19] that increasing the number of hash functions $k > 2$ achieves much better utilization of bins in the hash table. I.e., while the average utilization for $k = 2$ hash functions is around 50%, the utilization increases to 91.8% for $k = 3$, 97.7% for $k = 4$, and 99.2% for $k = 5$. Hence, higher values of k allow us to drastically decrease the number of bins. However, similar to the previous stash allocation, the analysis in [19] was only asymptotic and does not allow to compute the concrete hashing failure probability.

In order to determine the concrete failure probability, we again perform 2^{30} iterations of Cuckoo hashing on $n = 1{,}024$ elements using $k \in \{3, 4, 5\}$ hash functions. Our goal in this analysis is to determine the minimum number of bins $b_{min} = \epsilon_{min} n$ for which the hashing procedure succeeds without a stash except with probability 2^{-30}. In order to determine the value of b_{min}, we run Cuckoo hashing on an initialization value $\epsilon_{min} = 1.0$ and increase ϵ_{min} by 0.1 each time more than one hashing failure has occurred. An interesting observation that we made during the experiments with multiple hash functions was that after a certain threshold value, the hashing failure probability decreased drastically. E.g., only increasing ϵ by as little as 0.1 when using $k = 5$ hash functions could reduce the required stash size from $s = 2$ to $s = 0$. Overall, we determined the following bin sizes that resulted in a hashing failure probability of $< 2^{-30}$: $\epsilon_{min} = 1.20$ for $k = 3$, $\epsilon_{min} = 1.07$ for $k = 4$, and $\epsilon_{min} = 1.04$ for $k = 5$.

A consequence of increasing the number of hash functions is that the party P_1, who uses simple hashing, needs to increase the maximum bin size max_b. This is due to two factors: on the one hand P_1 needs to map each element k times to its hash table. On the other hand, the parties decrease the number of bins due to the reduced ϵ. We re-compute the maximum bin size of P_1 given the increased number of hash functions using Eq. 5 and give the results in Tab. 5. Given these results, we can compute the total number of comparisons by multiplying the number of bins b with max_b. From these results, we observe that $k = 3$ achieves the best performance.

Hash Failure Parameter η	**30**					**40**				
Set Sizes n_1 =n_2	2^8	2^{12}	2^{16}	2^{20}	2^{24}	2^8	2^{12}	2^{16}	2^{20}	2^{24}
max$_b$ for *k*=2 ($b = 2.4n_2, n = 2n_1$)	15	16	17	18	19	17	18	19	20	21
max$_b$ for *k*=3 ($b = 1.2n_2, n = 3n_1$)	23	24	25	26	28	26	27	28	29	30
max$_b$ for *k*=4 ($b = 1.07n_2, n = 4n_1$)	28	29	30	32	33	31	32	34	35	36
max$_b$ for *k*=5 ($b = 1.04n_2, n = 5n_1$)	31	33	34	36	37	35	36	38	39	40

Table 5: The bin sizes max_b that are required to ensure that no overflow occurs when mapping n items to b bins using k hash functions, according to Eq. (5).

Adjusting the Number of Bins b The required stash sizes for $b = 2.4n$ bins and $k = 2$ hash functions are relatively large for small set sizes (e.g., $s = 8$ for $n = 256$). In case of equal set sizes $n_1 = n_2$, this does not impact the performance of the protocols much. In the case of unequal set sizes $n_1 \gg n_2$, however, large stash sizes will greatly decrease the performance, since each element in the stash needs to be compared with each item in the large set with possibly millions of elements. Furthermore, even when increasing the number of hash functions $k > 2$ to remove the stash, P_1 would need to map each of its million elements k times into its hash table, which increases max_b and hence incurs a great overhead.

To improve the performance for unequal set sizes, we fix the stash sizes $s \in \{0, 1, 2, 3, 4\}$ and the number of hash functions to $k = 2$ and try to identify the number of bins $b = \epsilon n$ such that the hashing failure probability is less than $2^{-\eta}$. Similarly to the previous experiments, we ran 2^{30} repetitions of Cuckoo hashing, mapping n items to $b = \epsilon n$ bins, for $n = 256$ and $\epsilon = \{2.4, 3, 4, 5, 6, 7, 8, 9, 10, 20, 100, 200\}$, and recorded the stash size s that was needed for Cuckoo hashing to be successful. We chose $n = 256$ since it is a good approximation of the number of contacts in a user's addressbook and it is used in our experiments in §6.2.3.

The results of our experiments are depicted as solid lines in Fig. 2. From the results, we can observe that the probability of requiring a stash size of s decreases logarithmically with growing ϵ: while for small ϵ the probabilities decrease quickly, they decrease slower for large ϵ. E.g., when increasing ϵ from 2.4 to 4, the hashing failure probability for a stash of size $s = 0$ decreases from 2^{-6} to 2^{-12}. If, on the other hand, ϵ is increased from 20 to 100, the hashing failure probability for $s = 0$ only decreases from 2^{-21} to 2^{-28}. Since we are interested in identifying ϵ such that the probability of requiring a stash of size s decreases below $2^{-\eta}$, we use regression via a logarithmic function to extrapolate the probabilities. These estimated probabilities are depicted as dotted lines in Fig. 2 and the smallest ϵ for which the hashing failure probability decreases below 2^{-30} and 2^{-40} is given in Tab. 6.

The estimations indicate that, in order to reduce the stash size to $s = 0$, we would need to set $\epsilon = 166$ to guarantee 2^{-30} hashing failure probability and to $\epsilon =$ 2,500 to guarantee 2^{-40} hashing failure probability. When allowing a bigger stash size $s = 1$, ϵ decreases drastically, allowing us to set $\epsilon = 7.8$ for 2^{-30} hashing failure probability and $\epsilon = 16$ for 2^{-40} hashing failure probability. In our experiments, the exact choice of ϵ and s depends on the difference between the set sizes n_1 and n_2 as well as the protocol that is used (cf. §6.2.3). I.e., if n_2 is only a few hundred while n_1 is several million, it can be more efficient to choose $\epsilon = 166$ to achieve stash size $s = 0$.

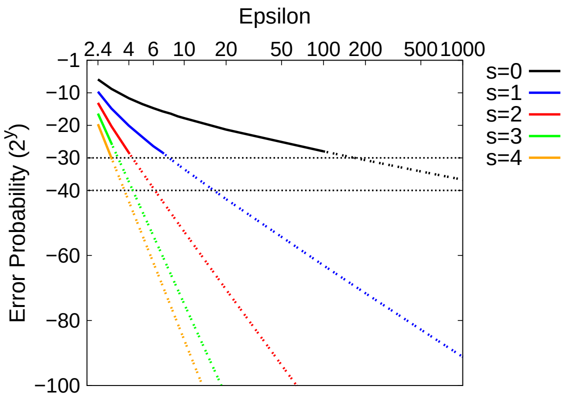

Figure 2: Error probability when mapping 256 elements to $b = 256\epsilon$ bins using Cuckoo hashing with $k = 2$ hash functions for stash sizes $0 \leq s \leq 4$. The solid lines correspond to actual measurements, the dashed lines were extrapolated using logarithmic regression.

Stash Size s	**0**	**1**	**2**	**3**	**4**
ϵ ($\eta = 30$)	166	7.8	4.2	3.4	3
ϵ ($\eta = 40$)	2,500	16	6.2	4.4	3.8

Table 6: Required number of bins $b = 256\epsilon$ to achieve $< 2^{-\eta}$ hashing failure probability given a fixed stash size s.

3.3 Permutation-based Hashing

The overhead of our circuit-based PSI protocols in §4 and of the OT-based PSI protocol in §5 depends on the bit-length σ 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 [2] 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 $|x_L| = \log b$, 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 b in the hash table is a power of 2. It was shown in [2] how to handle the general case.) Let $f()$ be a random function whose range is $[0, b-1]$. 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 $\log b$ 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 $\log b$. As for the security, it can be shown based on the results in [2] 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 $b = 2^{20}$ 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. Note that, when mapping an element into a bin using multiple hash functions, e.g., when using Cuckoo hashing, the index of the hash function needs to be added to the representation in the bin to preserve uniqueness. This observation was also pointed out in [43].

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 $\log b$ 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 $|x| - \log b$ 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.

4 Circuit-Based PSI

Unlike special purpose PSI protocols, the protocols that we describe in this section are based on *generic* secure computation techniques that can be used for computing arbitrary functionalities. We first briefly outline the two most prominent generic secure computation protocols in the semi-honest model: the Goldreich-Micali-Wigderson protocol [27] and Yao's garbled circuits protocol [68] (§4.1). We outline the sort-compare-shuffle (SCS) circuit of [32] a Boolean circuit of size $\mathcal{O}(n \log n)$ for computing the PSI functionality (§4.2). We then show how to use the hashing methods described in §3 to achieve better complexity than the SCS circuit using a naive pairwise-comparison circuit (§4.3). Finally, we revisit the method of [60] where generic secure computation techniques are used to instantiate an OPRF (cf. §2.1), which is used to process the input elements of one party (§4.4).

The usage of generic protocols has the advantage that the functionality of the protocol can easily be extended, without having to change the protocol or the security of the resulting protocol. For example, it is straightforward to change the SCS and PWC protocols to compute the size of the intersection, or a function that outputs if the intersection is greater than some threshold, or compute a summation of values (e.g., revenues) associated with the items that are in the intersection. Computing these variants using other PSI protocols is non-trivial.

4.1 A comparison between GMW and Yao

In the following, we give a high level comparison between the GMW protocol and Yao's garbled circuits protocol. We first outline the differences in the pre-computing setup phase and then detail differences in the online phase, where the circuit is evaluated.

Setup Phase The pre-computation complexity of GMW and Yao's protocol is measured by the circuit's *size*, i.e., the number of AND gates. Both protocols require 2κ bits communication per AND gate using OT extension for GMW [3] and the Half-Gates optimization for Yao's protocol [69]. The computational workload in GMW is dominated by the OT extension routine, where each party performs six symmetric key operations per AND gate and which can be pre-computed in parallel and independently of the function being evaluated. In contrast, in Yao's protocol, the party that generates the garbled circuit performs four symmetric key operations per AND gate, cf. [7]. To pre-compute the garbled circuit, the circuit garbler has to know the specific function and the size of the inputs in advance. Using fixed-key AES garbling [7] or other optimized instantiations of the CRF [29], the time for evaluating these symmetric cryptographic operations for both protocols can be significantly decreased. GMW allows efficient evaluation of multiplexer circuits using the vector multiplication triple optimization [18].

Online Phase In the online phase of the GMW protocol, the parties only evaluate one-time-pad operations and the main bottleneck of the protocol is its round complexity, which is linear in the circuit's *depth*, i.e., the highest number of AND gates on a path from any input to any output. In [65] it was shown that using circuits with smaller depth and larger size can be more efficient for GMW. In contrast, the round complexity of Yao's protocol is constant, but the evaluator has to perform two symmetric cryptographic operations per gate in the online phase [69].

Overall The GMW protocol is suited for use in the pre-processing model due to its function-independent pre-processing but requires multiple communication rounds in the online phase. Yao's garbled circuits protocol has a constant round online phase but requires the function and input sizes to be in the setup phase. A more detailed comparison between both protocols can be found in [18].

4.2 Sort-Compare-Shuffle Circuit for PSI

A Boolean circuit for PSI that has $O(n \log n)$ size is the *sort-compare-shuffle (SCS)* circuit described in [32]. (We refer here to the SCS circuit that uses the Waksman permutation for shuffling). The SCS circuit computes the intersection between two sets by first *sorting* both sets into a single sorted list, then *comparing* all neighboring elements for equality, and finally *shuffling* the intersecting elements to hide any information that could be obtained from the resulting order.

The overall size of the SCS circuit for inputs of bit-length σ is $\sigma(3n \log_2 n + 4n) - n$ AND gates, which is the sum of $2\sigma n \log_2(2n)$ AND gates for the sort circuit, $\sigma(3n-1)-n$ AND gates for the compare circuit, and $\sigma(n \log_2 n - n + 1)$ for the shuffle circuit. It is important to note that approximately $2/3$ of the AND gates in the circuit are due to multiplexers. These multiplexer gates can be efficiently evaluated in GMW using vector multiplication triples [18], reducing the pre-computation cost in GMW from σ AND gates to the equivalent of 1 AND gate for a σ-bit multiplexer.

Instantiation For our experiments in §6, we used GMW to evaluate a depth-optimized variant of the SCS circuit, where the comparison gates have $3\sigma - \log_2(\sigma) - 2$ AND gates instead of σ but have a depth of $\log_2 \sigma$ instead of σ for σ-bit values (cf. [65]). Consequently, the size of the SCS circuit is increased from approximately $3n\sigma \log_2 n$ to $5n\sigma \log_2 n$, but its depth is decreased from $\sigma \log_2 n$ to $\log_2(n) \log_2(\sigma)$. Using the vector multiplication-triple optimization of [18], the size of the depth-optimized SCS circuit is again decreased back to approximately $3n\sigma \log_2 n$.

4.3 Pairwise Comparison (PWC) and Hashing

A simpler circuit for performing the PSI functionality is a pairwise-comparison (PWC) circuit, where each element in the set of P_1 is compared to each element in the set of P_2. However, this circuit would scale with $O(n_1 n_2)$, making it impractical for larger sets. Using the hashing methods from §3, we can drastically reduce the number of comparisons. The circuit processes elements as follows:

- Both parties use a table of size $b = \mathsf{O}(n_2)$ to store their elements. Our analysis (§3.2.2) shows that setting $b = \epsilon n_2$ reduces the error probability to be negligible for reasonable input sizes ($2^8 \leq n_2 \leq 2^{24}$) when setting the stash size accordingly (cf. §3.2).
- P_2 maps its input elements to b bins using Cuckoo hashing with k hash functions and a stash; empty bins are padded with a dummy element d_2.
- P_1 maps its input elements into b bins using simple hashing. The size of the bins is set to be max_b, a parameter that is set to ensure that no bin overflows (cf. §3.1.2). The remaining slots in each bin are padded with a dummy element $d_1 \neq d_2$. The analysis described in §3.1.2 shows how max_b is computed and is set to a value smaller than $\log n_2$.
- The parties securely evaluate a circuit that compares the element that was mapped to a bin by P_2 to each of the max_b 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 §3.3. (Note that using this technique is impossible with SCS circuits of [32].)

Efficiency Let m be the number of element comparisons that are performed in the circuit with $m = b \cdot max_b + sn_1$, i.e., for each of the b bins, the parties perform max_b comparisons per bin as well as n_1 comparisons for each of the s positions in the stash. 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. $\sigma' = \sigma - \log_2 b$. 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 $m \cdot (\sigma' - 1) \approx n_1 \cdot (max_b + s) \cdot (\sigma' - 1)$ non-linear gates and has an AND depth of $\lceil \log_2 \sigma \rceil$.

Advantages The PWC circuit offers 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_b was shown in our experiments to be no greater than $2 \log n$ (and not greater than $\log n$ asymptotically), the number of non-linear gates in the PWC circuit is smaller by more than a factor 1.5 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 the PWC circuit 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 the PWC circuit 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.
- Finally, the efficiency of the SCS circuit is tailored for equal set sizes. For unequal set sizes, the circuit size does not scale well. The PWC circuit, on the other hand, scales much better for unequal set sizes.

4.4 Secure Evaluation of an OPRF

Another method for circuit-based PSI was outlined in [24, 60] and uses an OPRF (cf. §2.1). In this protocol, the parties use secure computation to evaluate a pseudo-random function $F_k(y) = z$, which takes as input a random key k from P_1 and an element y from P_2 and returns the output z to P_2. The use of secure computation guarantees the obliviousness, i.e., that P_1 learns no information about y or z while P_2 learns no information about k. The PSI functionality can then be achieved by evaluating the OPRF on each element in the set of P_2 and having P_1 locally evaluate and send $F_k(x_i)$ for all elements $x_i \in X$. P_2 can then identify the intersection by computing the plaintext intersection between his output of the OPRF with the elements sent by P_1.

Efficiency The efficiency of the circuit-based OPRF construction depends mainly on the instantiation of the pseudo-random function F. While it is possible to instantiate F with a cipher that is optimized for use in secure computation such as [1], we consider an AES-based instantiation in our efficiency analysis, since the security of AES is better established. The number of AND gates in the AES circuit is 5,120 and its multiplicative depth is 60 [10]. In total, we have to perform n_2 parallel oblivious AES evaluations, resulting in a total of $5{,}120n_2$ AND gates and a depth of 60. P_1, on the other hand, can perform a plaintext AES evaluation on his elements and only needs to send n_1 collision-resistant strings length of $\ell = \lambda + \log(n_1) + \log(n_2)$ bit. Hence, due to the large constants, the OPRF-based approach is less efficient in concrete terms than the SCS or PWC circuits, even though it scales with $O(n)$ while both other circuits scale with $O(n \log n)$. However, if the set sizes of the parties greatly differ, i.e., for the mobile messenger application where $n_1 \gg n_2$, the OPRF-based approach can be more efficient than other circuit constructions and in fact more efficient than even all other PSI protocols, since the elements in the much larger set of P_1 can be processed at very low cost (cf. §6.2.3).

5 Private Set Intersection via OT

In this section, we describe our new OT-based PSI protocol, of which an earlier version appeared in [61, 59]. In contrast to the conference versions, we improve our protocol such that its complexity is now independent of the bit length σ for realistic set sizes. The core of our OT-based PSI protocol is an efficient OPRF (cf. §2.1) instantiation using recent OT extension techniques, in particular the random OT functionality [55, 3] and the $\binom{N}{1}$-OT of [39]. Our protocol operates in three steps: the parties *hash* their elements into hash tables, mask their elements using the *OPRF,* and compute the *plaintext intersection* of these masks to identify the intersecting elements. In the hashing step we use the methods from §3 for hashing the elements to bins. In the following, we describe the OPRF construction (§5) in more detail.

In the first step of our OT-based PSI protocol, the parties have mapped their elements into hash tables T_1 and T_2 where the elements in the tables have bit-length $\mu = \sigma - \log_2 b + \log_2 k$ due to permutation-based mapping (cf. §3.3). P_1 has used simple hashing and hence its hash table T_1 has two dimensions, where the first dimension addresses the bins and the second dimension addresses the elements in the bins. P_2 has used Cuckoo hashing and hence its hash table T_2 has only one dimension, which addresses the bins. Our OT-based PSI protocol then evaluates an OPRF F (cf. §2.1) where, for each bin, P_1 samples a random key and P_2 inputs the μ-bit element in bin $T_2[i]$ and obtains the resulting mask $M_2[i] = F_{k_i}(T_2[i])$, for $1 \leq i \leq b$. The OPRF must ensure that P_1 learns no information on the input of P_2 and that P_2 learns no information except the outputs that correspond to its elements.

The main observation is that we can instantiate an OPRF for μ-bit inputs using one random 1-out-of-2^μ

PROTOCOL 2 (Our OT-based PSI Protocol)

- **Input of P_1:** $X = \{x_1, ..., x_{n_1}\}$.
- **Input of P_2:** $Y = \{y_1, ..., y_{n_2}\}$.
- **Common Input:** Bit-length of elements σ; number of bins $b = \epsilon n_2$ (cf. §3.2.2); k random hash functions $\{h_1, ..., h_k\} : \{0,1\}^\sigma \mapsto [1...b]$; reduced bit-length of items in the hash table $\mu = \sigma - \log_2 b + \log_2 k$ (cf. §3.3); symmetric security parameter κ; statistical security parameter λ; mask-length $\ell = \lambda + \log_2(kn_1) + \log_2(n_2)$; $N = 2^\mu$; dummy element d_2; stash size s.
- **Oracles and cryptographic primitives:** Both parties have access to a $\binom{N}{1}$-ROT$^1_\ell$ functionality.

1. **Hashing:**

   (a) P_1 maps the elements in its set X into a two-dimensional hash table $T_1[\,][\,]$ using simple hashing and k hash functions $\{h_1, ..., h_k\}$. The first dimension has size b and addresses the bin in the table while the second dimension addresses the elements in the bins.

   (b) P_2 maps the elements in its set Y into a one-dimensional hash table $T_2[\,]$ and stash $S[\,]$ using Cuckoo hashing and k hash functions $\{h_1, ..., h_k\}$. The hash table has size b and the stash has size s. P_2 then fills all empty entries in T_2 and S with d_2.

   Let $|T_1[i]|$ be the number of elements that are stored in the i-th bin of the hash table T_1 and μ be the bit-length of these elements for $1 \le i \le b$.

2. **OPRF evaluation (via OT):**
   For each bin $1 \le i \le b$, the parties perform the following steps:

   (a) Let $v_j = T_1[i][j]$ and $w = T_2[i]$ for $1 \le j \le |T_1[i]|$.

   (b) The parties evaluate an OPRF using the $\binom{N}{1}$-ROT$^1_\ell$ functionality, where P_1 has no inputs and obtains a random N-entry look-up table L and P_2 inputs w as choice bits and obtains a random mask $L[w]$.

   (c) P_1 computes $M_1[i][j] = L[v_j]$ and P_2 computes $M_2[i] = L[w]$.

   Stash: For each element in the stash S, the parties repeat the same steps where, for the i-th stash position, P_1 evaluates the OPRF on his whole input set X and obtains n_1 masks $M_{S_1}[i]$ while P_2 evaluates the OPRF on $S[i]$ and obtains one masks $M_{S_2}[i]$.

3. **Plaintext Intersection**

   (a) Let $\bigcup_{1\le i\le b, 1\le j\le |T_1[i]|} M_1[i][j]$. P_1 randomly permutes V and sends it to P_2.

   (b) P_2 computes the intersection $Z = \{T_2[i] | M_2[i] \in V\}$.

   Stash: The parties perform the same operation to identify whether an element on the stash is in the intersection: P_1 permutes and sends $M_{S_1}[i]$ to P_2, who adds $S[i]$ to the intersection Z if $M_{S_2}[i] \in M_{S_1}[i]$.

- **Output:** P_1 has no output; P_2 outputs $Z = X \cap Y$.

random OT on ℓ-bit strings ($\binom{2^\mu}{1}$-ROT$^1_\ell$), where P_1 plays the sender and obtains a 2^μ-dimensional lookup-table $L : \{0,1\}^\mu \mapsto \{0,1\}^\ell$ while P_2 plays the receiver who inputs $T_2[i]$ and obtains $L[T_2[i]]$. P_1 can then evaluate the OPRF on the elements in its bin $T_1[i]$ locally by computing $M_1[i][j] = L[T[i][j]]$, for $1 \le i \le b$ and $1 \le j \le |T_1[i]|$. After P_1 has evaluated the OPRF for all bins i, it collects the OPRF outputs $M_1[i]$ for all $|T_1[i]|$ elements in a bin to a set V and permutes and sends V. P_2 identifies whether $T_2[i]$ is in the intersection by checking whether $M_2[i]$ matches any element in V. If the element $T_2[i]$ matches any element in $T_1[i]$, their OPRF outputs will be equal. If $T_2[i]$ matches no element in $T_1[i]$, their OPRF outputs will differ except with probability $|T_1[i]| \cdot 2^{-\ell}$. The elements in the stash of P_2 are processed independently in a similar fashion: both parties evaluate the OPRF, P_2 obtains the output for the elements in its stash, and P_1 evaluates the OPRF locally on each element of its set and sends the permuted outputs to P_2, who identifies the intersection.

Efficiency The main computation and communication overhead comes from the OPRF evaluation. The efficiency of the OPRF depends greatly on the underlying instantiation. We instantiate the OPRF that maps μ-bit inputs to ℓ-bit outputs using the $\binom{2^\mu}{1}$-ROT$^1_\ell$ protocol of [39] with the linear BCH code [2^{77}, 512, 129], generated by [49] (cf. §2.2.3). Overall, the parties perform $s + b$ OPRF evaluations, which correspond to $\binom{2^\mu}{1}$-ROT$^{s+b}_\ell$, where the stash size s and the number of bins $b = \epsilon n_2$ are chosen to achieve negligible Cuckoo hashing error probability (cf. §3.2.2). Regarding the communication, P_2 sends $512(s{+}b)$ bits for the $\binom{2^\mu}{1}$-ROT, while P_1 sends $k\ell n_1$ bits for the permuted OPRF output, where k is the number of hash functions used for Cuckoo hashing (cf. §3.2.2) and $\ell = \log_2(kn_1) + \log_2(n_2) + \lambda$. Regarding the computation, note that in a naive $\binom{2^\mu}{1}$-OT evaluation the sender P_1 would need to perform 2^μ CRF evaluations, one for each message. However, since P_1 only needs to obtain the output for actual elements in its bins, it only needs to perform $(k + s)n_1$ CRF evaluations, which is independent of μ.

Correctness In the following, we analyze the correctness of the scheme. We assume that in Step 1 in Prot. 2, P_1 has used simple hashing to map each element k times into the hash table T_1 while P_2 has used Cuckoo hashing to map each element once into the hash table T_2.

If $x = y$ then P_1 and P_2 will have the same item in a bin in their hash tables (P_2 has mapped the item to one of k bins while P_1 has mapped the item to all k bins). For this bin, P_2 obtains $M_x = L[x]$ as output of the OPRF and P_1 can locally compute $M_y = L[y]$ with $M_x = M_y$, and P_2 successfully identifies equality.

If $x \neq y$ then the probability that $M_x = M_y$ is $2^{-\ell}$. However, we require that *all* OPRF outputs M_2 for elements in the hash table T_2 of P_2 are distinct from *all* outputs M_1 for elements in the hash table T_1 of P_1, which happens with probability $kn_1n_2 2^{-\ell}$. Thus, to achieve correctness with probability 1-$2^{-\lambda}$, we must increase the bit-length of the OTs to $\ell = \lambda + \log_2(kn_1) + \log_2(n_2)$.

Security P_2's security is obvious, since the only information that P_1 learns are the random values chosen in the random OT, which are independent of P_2's input.

As for P_1's security, note that P_2's view in the protocol consists of its outputs M_2 of the $\binom{N}{1}$-ROT protocols, and of the values M_1 sent by P_1. If there are two elements $x \in X$ and $y \in Y$ with $x = y$, then there are outputs $M_x = M_y$. Otherwise, for $x \neq y$, these values are uniformly distributed and P_2 can gain no information about M_x, which is guaranteed by the properties of the $\binom{N}{1}$-ROT protocol. In both cases, the view of P_2 can be easily simulated given the output of the protocol (i.e., knowledge whether $x = y$). The protocol is therefore secure according to the common security definitions of secure computation [26].

6 Experimental Evaluation

In the following, we experimentally evaluate the most promising PSI protocols that were outlined before. We first discuss their implementational features and compare them theoretically (§6.1). We then give an empirical performance comparison between the protocols for different settings (§6.2). Throughout the evaluation, we divide the PSI protocols into four categories, depending on whether the protocol is based on *public-key* operations, *circuits*, *OT*, or provides *limited security* and mark the best result of each category in bold.

6.1 Theoretical Evaluation

Before evaluating the empirical performance of the PSI protocols, we discuss implementational features of the protocols such as their suitability for large-scale PSI on sets with several million elements (§6.1.1) or the ability of the schemes for parallelization (§6.1.2), and give their asymptotic computation and communication complexities (§6.1.3).

6.1.1 Suitability for Large-Scale PSI

Although hardly discussed, memory consumption poses a very big problem when implementing cryptographic schemes that operate on large amounts of data. As such, many of the implemented PSI protocols quickly exceeded the main memory, requiring more engineering effort and a more careful implementation to allow for PSI on larger sets. In fact, even computing the plaintext intersection for sets of billions of elements becomes a tedious problem, since at least one set needs to be fully stored at one point during the execution. In this case, one can store the data on disk, which decreases performance greatly when arbitrary look ups are performed.

Limited Security & Public-Key-Based PSI The naive-hashing, server-aided, and public-key-based PSI schemes are very memory efficient, since they operate only on single elements and can be easily pipelined, allowing PSI on millions of elements even on standard PCs.

Circuit-Based PSI The circuit-based PSI schemes have a very high memory consumption. In our implementations we evaluate and delete gates if they will not be used anymore to decrease the memory consumption. Yao's garbled circuits has a higher memory consumption than GMW, since κ-bit keys have to be stored for each wire instead of single bits. A pipelined circuit generation and evaluation, as is done in VMCRYPT [44], FastGC [33, 31], or PCF [42] would allow us to perform PSI on larger sets. The main memory limitation of our Yao and GMW implementation comes from the circuit having to be fully built and stored in memory. To decrease the memory footprint of the circuit, we build circuits that are evaluated many times in parallel in a SIMD fashion, which evaluates the circuit on multiple values in parallel. This SIMD evaluation especially benefits the PWC (§4.3) and OPRF (§4.4) circuits, since the same circuit is evaluated on all elements in parallel.

OT-Based PSI The garbled Bloom filter and random garbled Bloom filter PSI protocols of [20, 61] have to store the full Bloom filter in memory to identify the intersecting elements. The garbled Bloom filter holds $1.44n\kappa$ entries of at least λ-bit shares, resulting in at least 875 MB for sets of one million elements. In addition, the parties have to perform arbitrary element look ups, which greatly decrease the performance if the Bloom filter is outsourced to the hard disk.

The main memory limitation of our OT-based PSI protocol (§5) are the hash tables, in particular the Cuckoo hash table. While the hash table for simple hashing can be easily stored on disk, the Cuckoo hash table needs to perform arbitrary look ups when evicting elements. The Cuckoo hash table holds $1.2n$ elements of at most $\ell = \lambda + \log(n_1) + \log(n_2)$-bit length, resulting in 12 MB for sets of one million elements and hence scales much better than the Bloom filter-based protocols.

6.1.2 Parallelizability of Schemes

The experiments we perform in the empirical evaluation only consider execution using a single thread. However, if more computational resources are available, the schemes can be run using multiple threads in order to improve their performance. Note, however, that the bottleneck for many protocols (i.e., all except the public-key-based protocols) quickly shifts from computation to communication, since symmetric cryptographic operations can be evaluated very efficiently using AES-NI. In the following we discuss the ability of the schemes to be parallelized.

Limited Security & Public-Key-Based PSI The naive-hashing, server-aided, and public-key-based PSI schemes can easily be parallelized since the elements are processed independently of each other. The main bottleneck for parallelization in all these schemes is the plaintext intersection of hash values that is done at the end of each protocol.

Circuit-Based PSI The circuit-based PSI protocols parallelize differently depending on the underlying secure computation protocol. The GMW protocol uses OT extension to pre-compute multiplication triples. This step presents the main computational workload and can be parallelized well. However, the circuit evaluation of GMW requires a number of sequential interactions between the parties that is linear in the depth of the circuit and which cannot be parallelized. Yao's garbled circuits, on the other hand, is a constant round protocol. Its ability to parallelize depends on the underlying circuit structure. Circuits that can be split into many sub-circuits that are independent of each other, such as the PWC and OPRF circuits, can be parallelized easily and efficiently while circuits where all gates are connected, such as the sort-compare-shuffle circuit, require circuit-dependent methods for parallelization. For such circuits, an automatically parallelizing compiler could be used [12].

OT-based PSI For all OT-based PSI protocols it holds that the underlying OT extension protocol can be parallelized well. The main differences in parallelizability are due to the hashing scheme that is used to map the elements into the corresponding structure. In the garbled Bloom filter-based PSI protocol of [20], P_1 has to generate the garbled Bloom filter in advance, and this step does not parallelize well. This is improved on by the random garbled Bloom filter protocol of [61], where the garbled Bloom filter is generated as an output of OT extension and can hence be fully parallelized. In our OT-PSI protocol, the main bottleneck for parallelization is the Cuckoo-hashing procedure. However, Cuckoo hashing can be pre-processed since no input of the other party is required.

6.1.3 Asymptotic Performance Comparison

We depict the asymptotic computation complexity for the party with the majority of the workload and total communication complexity of the PSI protocols in Tab. 7. The computation complexity is expressed as the number of symmetric cryptographic primitive evaluations (sym) and the number of asymmetric cryptographic primitive operations (pk). We assume 3 sym per OT (2.5 sym for the Bloom filter-based protocols), 4 sym per AND gate in Yao's protocol, and 6 sym per AND gate in the GMW protocol.

The most crucial observation we make from the asymptotic complexities is that, asymptotically, the performance amongst the schemes with the same type is nearly equal. The naive hashing and server-aided protocol both require 1 sym operation per element, the public-key-based protocols all require 2 pk operations per element and need to send two ciphertexts and a hash value, the circuit-based protocols all have to perform work linear in the number of AND gates in the circuit, and the Bloom filter-based protocols both have to perform work linear in the size of the Bloom filter. The main discrepancy can be seen among the OT-based protocols, where the communication complexity of the Bloom filter-based protocols scales quadratically with the symmetric security parameter κ while our OT-based PSI protocol scales only linear in the security parameter κ (we need 512-bit codewords to achieve relative Hamming distance κ, cf. §2.2.3).

Type	Protocol	Computation [#Ops sym/pk]	Communication [bit]
Limited Security	Naive Hashing	m sym	$n_1\ell$
	Server-aided [36]	m sym	$t + \|X \cap Y\|$
Public-Key	DH FFC [46]	$2t$ pk	$t\rho + n_1\ell$
	DH ECC [46]	$2t$ pk	$t\varphi + n_1\ell$
	RSA [15]	$2t$ pk	$t\rho + n_1\ell$
Circuit	Yao SCS [32]	$12m\sigma \log m + 3m\sigma$ sym	$6m\kappa\sigma \log m + 2m\kappa\sigma$
	GMW SCS [32]	$18m\sigma \log m$ sym	$6m(\kappa+2)\sigma \log m$
	Yao PWC (§4.3)	$\sigma(4\epsilon n_2 max_b + 4sn_1 + 3\epsilon n_2)$ sym	$2\epsilon n_2 \kappa max_b \sigma + 3sn_1\kappa\sigma + 2\epsilon n_2\sigma$
	GMW PWC (§4.3)	$6\sigma(\epsilon n_2 max_b + sn_1)$ sym	$2(2+\kappa)\sigma(\epsilon n_2 max_b + sn_1)$
	Yao OPRF (§4.4)	$21{,}760 n_2 + 3\sigma n_2$ sym	$10{,}880 n_2\kappa + 2n_2\kappa\sigma + n_1\ell$
	GMW OPRF (§4.4)	$32{,}640 n_2$ sym	$10{,}880 n_2(\kappa+2) + n_1\ell$
OT	Bloom Filter [20]	$3.6m\kappa$ sym	$1.44m\kappa(\kappa+\lambda)$
	OT (§5) + Hashing (§3)	$3\epsilon n_2 + (k+s)n_1$ sym	$512\epsilon n_2 + (k+s)n_1\ell$

Table 7: Asymptotic complexities for PSI protocols (σ: bit size of set elements; $t = n_1 + n_2$; $m = max(n_1, n_2)$; pk: public-key operations; sym: symmetric cryptographic operations; $\ell = \lambda + \log n_1 + \log n_2$; κ, ρ, φ, λ: security parameters as defined in §2.1; ϵ, k, s, max_b: Hashing parameters as defined in §3.1 and §3.2). Computation gives the number of operations that need to be performed in sequence.

6.2 Empirical Evaluation

We empirically evaluate and compare the performance of the presented semi-honest PSI protocols. We first describe our benchmarking environment and outline our implementations (§6.2.1). We then benchmark the protocols in a LAN and a WAN setting and give their concrete communication (§6.2.2). Finally, we evaluate the performance on the parallelizability of PSI schemes (§6.3) as well as for large-scale PSI (§6.4).

6.2.1 Benchmarking Environment

We ran our experiments in a *LAN* and a *WAN* setting. The LAN setting consists of two PCs (Intel Haswell i7-4770K CPU with 3.5 GHz and 16 GB RAM) that are connected via a Gigabit Ethernet. 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) that are located in North Virginia (US east coast) and Frankfurt (Europe) with an average bandwidth of 98 MBit/s and an average round-trip time of 94 ms.

We evaluate the performance of the PSI protocols in two scenarios. In the first scenario, P_1 and P_2 hold the same number of input elements $n_1, n_2 \in \{2^8, 2^{12}, 2^{16}, 2^{20}, 2^{24}\}$. In the second scenario, P_1 has a larger set than P_2 and we set $n_1 \in \{2^{16}, 2^{20}, 2^{24}\}$ and $n_2 \in \{2^8, 2^{12}\}$. Both parties are not allowed to perform any pre-computation. For the sort-compare-shuffle and pairwise-comparison circuit-based protocols whose complexity depends on the bit-length of elements σ, we fix $\sigma = 32$ (e.g., for PSI on IPv4 addresses). We use the long-term security parameters as described in §2.1. We benchmarked the server-aided PSI protocol of [36] by executing the trusted server on one machine and the two clients that wish to compute the intersection on the second machine.

Implementations The implementation of the blind-RSA-based [15] and garbled Bloom filter [20] protocols were taken from the authors, but we used a hash-table to compute the last step in the blind-RSA protocol that finds the intersection (the original implementation used pairwise comparisons with quadratic run-time overhead) and the OT extension implementation of [3] for the Bloom filter protocol. We use the state-of-the-art Yao's garbled circuits and GMW protocol implementations in the C++ ABY framework [18], which implements point-and-permute [45], half-gates [69], free-XOR [41], fixed-key garbling [7], and OT extension [3]. For Yao's garbled circuits protocol, we evaluated a size-optimized version of the sort-compare-shuffle circuit (comparison circuits of size and depth σ) while for GMW we evaluated a depth-optimized

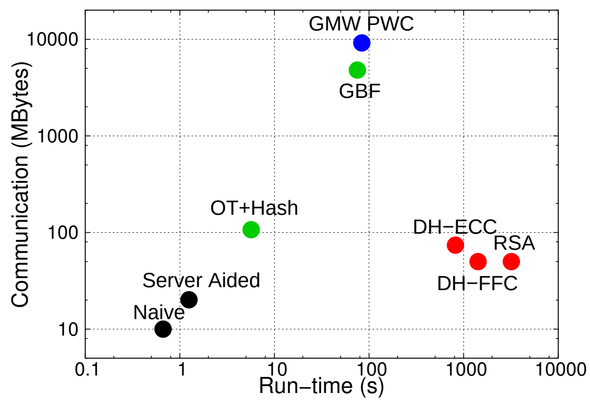

Figure 3: Run-time in s and communication in MBytes of PSI protocols for $n = 2^{20}$ elements and $\kappa = 128$-bit security. Detailed results are given in Tab. 8 and Tab. 9.

version (comparison circuits of size 3σ and depth $\log_2 \sigma$) for σ-bit input values [65]. We instantiated the PRP of the server-aided PSI protocol in [36] and the CRF in the $\binom{2}{1}$-OT extension with AES, and instantiated the RO and the CRF in the $\binom{N}{1}$-OT extension with SHA-256.

We implemented FFC (finite field cryptography) using the GMP library (v. 5.1.2), ECC using the Miracl library (v. 5.6.1), symmetric cryptographic primitives using OpenSSL (v. 1.0.1e), and used the OT extension implementation of [3]. We perform all operations in FFC in a subgroup of order q, where $|q| = 2\kappa$-bits.

We argue that we provide a fair comparison, since all protocols are implemented in the same programming language (C/C++), run on the same hardware, and use the same underlying libraries for cryptographic operations.

For each protocol we measured the time from starting the program until the client outputs the intersecting elements. All runtimes are averaged over 10 executions.

6.2.2 Empirical Comparison

We evaluate the empirical performance of the PSI protocols in the LAN setting and give the concrete communication of the protocols. While the LAN setting does not necessarily represent a real-world setting for PSI, it allows us to benchmark the protocols in an almost ideal network setting and hence focus on the computation complexity of the protocols. We give a classification for $n = 2^{20}$ element sets in Fig. 3 and depict the detailed run-time in Tab. 8 and communication in Tab. 9. We now compare the performance of the different types of PSI protocols and then compare the PSI protocols of the same type.

Comparison between Types From Fig. 3, we can observe that PSI protocols of the same type have a similar run-time and communication with the exception of the OT-based PSI protocols. The insecure naive hashing protocol and server-aided PSI protocol outperform the other PSI protocols by at least an order of magnitude in computation and communication. The public-key-based PSI protocols require only little communication (especially the DH-ECC protocol), but have the highest run-time. The circuit-based PWC protocol has a faster run-time than the public-key-based protocols but requires two orders of magnitude more communication and does not scale well to large sets. Finally, the OT-based PSI protocols differ in performance: the GBF protocol of [20] has a similar run-time and communication as the circuit-based PWC protocol and our OT-based PSI protocol has a faster run-time than the public-key and circuit-based protocols and require at least an order of magnitude less communication compared to the circuit-based protocols. Among all PSI protocols, our novel OT-based PSI protocol is the fastest and requires about the same amount of communication as public-key-based PSI protocols.

Limited Security-Based PSI The naive hashing protocol outperforms the server-aided protocol by factor of 2 in run-time and communication. However, these protocols have weaker security guarantees than the other protocols that we describe.

Public-Key-Based PSI For the public-key-based PSI protocols, we observe that the DH-based protocol of [46] outperforms the RSA-based protocol of [15] when using finite field cryptography (FFC). The elliptic curve cryptography (ECC) instantiation of the DH-based protocol becomes even more efficient and outperforms the FFC instantiation by a factor of 2. The advantage of the ECC-based protocol is its communication complexity, which is lowest among all PSI protocols (cf. Tab. 9). We note that a major advantage of these protocols is their simplicity, which makes them relatively easy to implement.

Circuit-Based PSI Here we compare the sort-compare-shuffle (SCS) circuit of [32], our PWC circuit (§4.3), and the OPRF circuit (§4.4), evaluated using Yao's garbled circuits and GMW. The results can be summarized as follows:

The GMW protocol is around factor 2 faster than Yao's garbled circuits protocol, which is due to the balanced communication. The PWC circuit scales better than the SCS and OPRF circuits with increasing set sizes and is at least 3 times more efficient for sets of 2^{16} elements. Due to its simple functionality, the PWC circuit can scale up to much larger set sizes and can even process two sets of 2^{20} elements sets using GMW.

OT-Based PSI Our OT-based PSI protocol has a higher run-time than the Bloom filter-based protocol for small set sizes since the number of base-OTs (and hence public-key operations) that are required for the $\binom{N}{1}$-OT extension is four times higher. However, this workload is linear in the security parameter and amortizes with increasing set sizes. For larger set sizes of $n \geq 2^{12}$, our OT-based PSI protocol is up to 15 times more efficient in terms of run-time than the garbled Bloom filter protocol and has between factor 20x and 45x less communication.

6.2.3 PSI with Unequal Set Sizes

In many applications of PSI, the set sizes of the parties are not equal. In fact, often a client with a small set of only a few hundred elements wants to perform PSI with a server, which holds a database of millions of records. We perform PSI with unequal set sizes $n_1 \in \{2^{16}, 2^{20}, 2^{24}\}$ and $n_2 \in \{2^8, 2^{16}\}$ using the previously best performing protocols of each category: naive hashing, the server-aided protocol of [36], the DH-ECC protocol of [46], the PWC and OPRF circuits in §4.3 and §4.4, and our OT-based PSI protocol in §5. We evaluate their performance in the LAN and WAN setting and give the resulting run-times in Tab. 11 and concrete communication in Tab. 12. For the circuit PWC protocol and our OT-based PSI protocol, which both use hashing techniques, we used the parameters given in Tab. 10.

The results are similar to the equal set size experiments with one notable exception: the OPRF circuit performs extremely well and achieves a similar run-time as the server-aided protocol and even outperforms

Type	Setting / Set Size n	LAN 2^8	LAN 2^{12}	LAN 2^{16}	LAN 2^{20}	LAN 2^{24}	WAN 2^8	WAN 2^{12}	WAN 2^{16}	WAN 2^{20}
Limited Security	Naive Hashing	**1**	**3**	**38**	**665**	**12,368**	**51**	**119**	**886**	**7,277**
	Server-aided [36]	**1**	5	78	1,250	20,053	124	248	1,987	15,578
Public-Key	DH FFC [46]	386	5,846	88,790	1,418,772	22,681,907	3,577	56,786	880,075	11,557,061
	DH ECC [46]	**231**	**3,238**	**51,380**	**818,318**	**13,065,904**	**1,949**	**28,686**	**466,606**	**5,007,681**
	RSA [15]	779	12,546	203,036	3,193,920	50,713,668	10,508	166,453	1,356,757	21,094,586
Circuit	Yao SCS [32] ($\sigma = 32$)	320	3,593	74,548	-	-	**2,763**	20,826	518,136	-
	GMW SCS [32] ($\sigma = 32$)	361	1,954	40,872	-	-	5,929	14,415	187,750	-
	Yao PWC (§4.3, $\sigma = 32$)	**304**	1,647	19,080	-	-	3,115	12,189	121,198	-
	GMW PWC (§4.3, $\sigma = 32$)	325	**905**	**7,085**	**83,889**	-	2,086	**5,881**	**42,253**	**337,851**
	Yao OPRF (§4.4)	968	12,518	-	-	-	6,001	65,156	-	-
	GMW OPRF (§4.4)	690	6,672	101,231	-	-	6,939	27,660	386,243	-
OT	Bloom Filter [20]	**105**	448	4,179	75,218	-	**1,248**	5,424	31,581	345,484
	OT (§5) + Hashing (§3)	309	**339**	**658**	**5,680**	**83,739**	2,211	**2,809**	**7,857**	**56,738**

Table 8: Run-times in ms for PSI protocols with one thread in the LAN and WAN setting. σ: bit-length of elements. "-" indicates that the execution ran out of memory.

Type	Set Size n	2^8	2^{12}	2^{16}	2^{20}	2^{24}
Limited Security	Naive Hashing	**0.002**	**0.031**	**0.600**	**10.000**	**176.000**
	Server-aided [36]	0.003	0.063	1.133	20.125	354.000
Public-Key	DH-based FFC [46]	0.195	3.125	50.000	800.000	12,800.000
	DH-based ECC [46]	**0.020**	**0.280**	**4.560**	**74.000**	**1,200.000**
	RSA-based [15]	0.195	3.125	50.000	800.000	12,800.000
Circuit	Yao SCS [32] ($\sigma = 32$)	7.522	168.590	3,484.751	-	-
	GMW SCS [32] ($\sigma = 32$)	7.319	162.851	3,348.011	-	-
	Yao PWC (§4.3, $\sigma = 32$)	6.923	93.371	1,220.194	-	-
	GMW PWC (§4.3, $\sigma = 32$)	**4.320**	**57.864**	**749.421**	**9,169.917**	-
	Yao OPRF (§4.4)	44.033	704.210	-	-	-
	GMW OPRF (§4.4)	43.193	690.890	11,054.050	-	-
OT	Garbled Bloom Filter [20]	1.037	17.314	288.560	4,801.639	-
	OT (§5) + Hashing (§3)	**0.055**	**0.424**	**6.500**	**107.000**	**1,757.000**

Table 9: Concrete communication in MB for PSI protocols. σ: bit-length of elements. "-" indicates that the execution ran out of memory.

naive hashing for $n_2 = 2^8$ and $n_1 = 2^{24}$. This good performance of the OPRF circuit can be explained by the asymmetric costs for processing the sets of the client and server. While each element in the set of the client is encrypted by securely evaluating an AES circuit using generic secure computation techniques, the server only needs to encrypt each element in his set using AES with a fixed key and send the resulting ciphertext to the client. Since the set size of the client is small, the overhead for the generic secure computation techniques does not impact the overall run-time significantly.

6.3 Multi-Threaded PSI

We evaluate the parallelizability of the best performing PSI protocol in each category by running up to four threads in parallel and depict the results in Tab. 13. We benchmark the FFC instantiation of the DH-based protocol instead of the ECC instantiation since the Miracl library does not allow for easy parallelization. Of special interest is the last column, which shows the ratio between the runtimes with four threads and a single thread, for an input of 2^{20} elements. The DH-based protocol, which is very simple and is easily

Server Set Size n_1	$\mathbf{2^{16}}$				$\mathbf{2^{20}}$				$\mathbf{2^{24}}$			
Client set size n_2 = 2^8												
Parameter	k	ϵ	s	**max**$_b$	k	ϵ	s	**max**$_b$	k	ϵ	s	**max**$_b$
Circuit PWC (§4.3, $\sigma = 32$)	2	7.8	1	205	2	7.8	1	2,884	2	7.8	1	45,707
OT (§5) + Hashing (§3)	3	1.2	0	0	2	166	0	0	2	166	0	0
Client set size n_2 = 2^{12}												
Parameter	k	ϵ	s	**max**$_b$	k	ϵ	s	**max**$_b$	k	ϵ	s	**max**$_b$
Circuit PWC (§4.3, $\sigma = 32$)	3	1.2	0	136	2	7.8	1	208	2	7.8	1	2,886
OT (§5) + Hashing (§3)	3	1.2	0	0	3	1.2	0	0	2	166	0	0

Table 10: Parameters for circuit PWC and our OT-based protocol used for the unequal set size experiments.

Type	**Setting**	**LAN**						**WAN**			
	Client Set Size n_2	$\mathbf{2^8}$			$\mathbf{2^{12}}$			$\mathbf{2^8}$		$\mathbf{2^{12}}$	
	Server Set Size n_1	$\mathbf{2^{16}}$	$\mathbf{2^{20}}$	$\mathbf{2^{24}}$	$\mathbf{2^{16}}$	$\mathbf{2^{20}}$	$\mathbf{2^{24}}$	$\mathbf{2^{16}}$	$\mathbf{2^{20}}$	$\mathbf{2^{16}}$	$\mathbf{2^{20}}$
Limited Security	Naive Hashing	**30**	**362**	**5,965**	**31**	**362**	**6,126**	**59**	**1,066**	**179**	**1,139**
	Server-aided [36]	63	515	7,267	65	524	7,571	170	1,871	267	1,989
Public-Key	DH ECC [46]	**52,073**	**814,839**	**12,705,815**	**52,057**	**815,715**	**12,712,287**	**156,068**	**2,451,092**	**158,159**	**2,486,141**
Circuit	Yao PWC (§4.3, $\sigma = 32$)	8,776	-	-	11,075	-	-	46,310	-	62,789	-
	GMW PWC (§4.3, $\sigma = 32$)	3,536	46,011	-	6,873	46,249	-	18,206	17,104	21,861	175,789
	Yao OPRF (§4.4)	996	1,194	3,882	11,414	11,764	14,347	6,636	7,947	64,418	67,284
	GMW OPRF (§4.4)	**692**	**821**	**3,425**	**6,283**	**6,394**	**8,975**	**5,730**	**7,545**	**31,653**	**33,593**
OT	OT (§5) + Hashing (§3)	**598**	**2,492**	**31,906**	**630**	**3,138**	**35,054**	**2,208**	**8,601**	**2,485**	**10,701**

Table 11: Run-times in ms for PSI protocols with unequal set sizes $n_1 \gg n_2$ in the LAN and WAN setting. σ: bit length of elments. “-” indicates that the execution ran out of memory.

parallelizable, achieves a speedup of 2.8 as computation is the performance bottleneck. The GMW protocol achieves a speedup of about 1.97 at 2 threads already and, for 3 and 4 threads, does not decrease much due to the communication bottleneck. The OT-based protocols achieve a moderate speedup of about 1.4, also due to communication and hashing to bins being the bottleneck.

6.4 PSI on Billion Element Sets

Finally, we demonstrate the scalability of our OT-based PSI protocol by evaluating it on sets of a billion $\sigma = 128$-bit elements each. For these sizes, the input elements require 15 GB of storage, which exceeds the main memory of our local servers. Instead, the servers store the elements and intermediate values on their respective solid state drive (SSD). We also benchmark the naive hashing protocol as a baseline for performance. We refrained from adding more main memory to process these sets, even though it is the most simple solution, since we are interested in the performance of the protocols if data needs to be stored on the SSD.

To compute the intersection between two sets of a billion elements, naive hashing requires 74 min, of which 19 min (26%) are spent on hashing and transferring data and 55 min (74%) are spent on computing the plaintext intersection. Our OT-based PSI protocol requires 34.2 hours in total, of which 30.0 hours (88%) are spent on simple hashing (Cuckoo hashing runs in parallel and requires 16.3 hours), 3 hours (9%) are spent on computing the OT routine, and 1.2 hours (4%) are spent on computing the plaintext intersection.

Type	Client Set Size n_2	2^8			2^{12}		
	Server Set Size n_1	2^{16}	2^{20}	2^{24}	2^{16}	2^{20}	2^{24}
Limited Security	Naive Hashing	**0.500**	**9.000**	**144.000**	**0.563**	**9.000**	**160.000**
	Server-aided [36]	0.502	9.002	144.002	0.598	9.040	160.040
Public-Key	DH ECC [46]	**2.329**	**30.017**	**592.008**	**2.582**	**37.545**	**592.270**
Circuit	Yao PWC (§4.3, $\sigma = 32$)	533.472	-	-	689.871	-	-
	GMW PWC (§4.3, $\sigma = 32$)	356.271	4,941.319	-	**437.225**	5,134.144	-
	Yao OPRF (§4.4)	**40.965**	**49.402**	**184.402**	646.674	**655.112**	**790.112**
	GMW OPRF (§4.4)	41.454	49.891	187.441	654.564	663.001	798.001
OT	OT (§5) + Hashing (§3)	**1.549**	**20.612**	**290.612**	**2.018**	**27.331**	**361.518**

Table 12: Concrete communication in MB for PSI with unequal set sizes $n_1 \gg n_2$. σ: bit-length of elements. "-" indicates that the execution ran out of memory.

Threads	**1**	**2**	**3**	**4**	**Speedup**
Naive Hashing	0.665	0.494	0.398	0.385	1.73
DH-based FFC [46]	1,418.772	961.123	659.895	509.990	**2.78**
GMW PWC (§4.3, $\sigma = 32$)	83.889	44.831	42.556	42.530	1.97
OT (§5) + Hashing (§3)	5.680	4.599	4.070	3.944	1.44

Table 13: Runtimes in seconds for PSI on sets with sizes $n_1 = n_2 = 2^{20}$ using multiple threads.

References

[1] M. R. Albrecht, C. Rechberger, T. Schneider, T. Tiessen, and M. Zohner. Ciphers for MPC and FHE. In *Advances in Cryptology – EUROCRYPT'15*, volume 9056 of *LNCS*, pages 430–454. Springer, 2015.

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

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

[4] P. Baldi, R. Baronio, E. De Cristofaro, P. Gasti, and G. Tsudik. Countering GATTACA: efficient and secure testing of fully-sequenced human genomes. In *Computer and Communications Security (CCS'11)*, pages 691–702. ACM, 2011.

[5] R. W. Baldwin and W. C. Gramlich. Cryptographic protocol for trustable matchmaking. In *Symposium on Security and Privacy (S&P'85)*, pages 92–100. IEEE, 1985.

[6] D. Beaver. Correlated pseudorandomness and the complexity of private computations. In *Symposium on Theory of Computing (STOC'96)*, pages 479–488. ACM, 1996.

[7] M. Bellare, V. Hoang, S. Keelveedhi, and P. Rogaway. Efficient garbling from a fixed-key blockcipher. In *Symposium on Security and Privacy (S&P'13)*, pages 478–492. IEEE, 2013.

[8] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *Computer and Communications Security (CCS'93)*, pages 62–73. ACM, 1993.

[9] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[10] J. Boyar and R. Peralta. A new combinational logic minimization technique with applications to cryptology. In *Symposium on Experimental Algorithms (SEA'10)*, volume 6049 of *LNCS*, pages 178–189. Springer, 2010.

[11] E. Bursztein, M. Hamburg, J. Lagarenne, and D. Boneh. OpenConflict: Preventing real time map hacks in online games. In *Symposium on Security and Privacy (S&P'11)*, pages 506–520. IEEE, 2011.

[12] N. Büscher and S. Katzenbeisser. Faster secure computation through automatic parallelization. In *USENIX Security Symposium 2015*, pages 531–546. USENIX, 2015.

[13] H. Carter, C. Amrutkar, I. Dacosta, and P. Traynor. For your phone only: Custom protocols for efficient secure function evaluation on mobile devices. *Journal of Security and Communication Networks*, 2013.

[14] E. De Cristofaro, J. Kim, and G. Tsudik. Linear-complexity private set intersection protocols secure in malicious model. In *Advances in Cryptology – ASIACRYPT'10*, volume 6477 of *LNCS*, pages 213–231. Springer, 2010.

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

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

[17] S. K. Debnath and R. Dutta. Secure and efficient private set intersection cardinality using bloom filter. In *Information Security Conference (ISC'15)*, volume 9290 of *LNCS*, pages 209–226. Springer, 2015.

[18] D. Demmler, T. Schneider, and M. Zohner. ABY – a framework for efficient mixed-protocol secure two-party computation. In *Network and Distributed System Security (NDSS'15)*. The Internet Society, 2015. Code: `https://github.com/encryptogroup/ABY`.

[19] M. Dietzfelbinger, A. Goerdt, M. Mitzenmacher, A. Montanari, R. Pagh, and M. Rink. Tight thresholds for cuckoo hashing via XORSAT. In *International Colloquium on Automata, Languages and Programming (ICALP'10)*, volume 6198 of *LNCS*, pages 213–225. Springer, 2010.

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

[21] M. Fischlin, B. Pinkas, A.-R. Sadeghi, T. Schneider, and I. Visconti. Secure set intersection with untrusted hardware tokens. In *Cryptographers' Track at the RSA Conference (CT-RSA'11)*, volume 6558 of *LNCS*, pages 1–16. Springer, 2011.

[22] T. K. Frederiksen and J. B. Nielsen. Fast and maliciously secure two-party computation using the GPU. In *Applied Cryptography and Network Security (ACNS'13)*, volume 7954 of *LNCS*, pages 339–356. Springer, 2013.

[23] M. J. Freedman, C. Hazay, K. Nissim, and B. Pinkas. Efficient set intersection with simulation-based security. *Journal of Cryptology*, 29(1):115–155, 2016.

[24] M. J. Freedman, Y. Ishai, B. Pinkas, and O. Reingold. Keyword search and oblivious pseudorandom functions. In *Theory of Cryptography Conference (TCC'05)*, volume 3378 of *LNCS*, pages 303–324. Springer, 2005.

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

[26] O. Goldreich. *Foundations of Cryptography*, volume 2: Basic Applications. Cambridge University Press, 2004.

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

[28] G. H. Gonnet. Expected length of the longest probe sequence in hash code searching. *Journal of the ACM*, 28(2):289–304, 1981.

[29] S. Gueron, Y. Lindell, A. Nof, and B. Pinkas. Fast garbling of circuits under standard assumptions. In *Computer and Communications Security (CCS'15)*, pages 567–578. ACM, 2015.

[30] C. Hazay and Y. Lindell. Constructions of truly practical secure protocols using standard smartcards. In *Computer and Communications Security (CCS'08)*, pages 491–500. ACM, 2008.

[31] W. Henecka and T. Schneider. Faster secure two-party computation with less memory. In *Symposium on Information, Computer and Communications Security (ASIACCS'13)*, pages 437–446. ACM, 2013.

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

[33] Y. Huang, D. Evans, J. Katz, and L. Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security Symposium 2011*, pages 539–554. USENIX, 2011.

[34] B. A. Huberman, M. Franklin, and T. Hogg. Enhancing privacy and trust in electronic communities. In *ACM Conference on Electronic Commerce (EC'99)*, pages 78–86. ACM, 1999.

[35] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank. Extending oblivious transfers efficiently. In *Advances in Cryptology – CRYPTO'03*, volume 2729 of *LNCS*, pages 145–161. Springer, 2003.

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

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

[38] L. Kissner and D. Song. Privacy-preserving set operations. In *Advances in Cryptology – CRYPTO'05*, volume 3621 of *LNCS*, pages 241–257. Springer, 2005.

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

[40] V. Kolesnikov, R. Kumaresan, M. Rosulek, and N. Trieu. Efficient batched oblivious prf with applications to private set intersection. In *Computer and Communications Security (CCS'16)*. ACM, 2016. To be published. Online at `http://eprint.iacr.org/2016/799`.

[41] V. Kolesnikov and T. Schneider. Improved garbled circuit: Free XOR gates and applications. In *International Colloquium on Automata, Languages and Programming (ICALP'08)*, volume 5126 of *LNCS*, pages 486–498. Springer, 2008.

[42] B. Kreuter, B. Mood, A. Shelat, and K. Butler. Pcf: A portable circuit format for scalable two-party secure computation. In *USENIX Security Symposium 2013*, pages 321–336. USENIX, 2013.

[43] M. Lambæk. Breaking and fixing private set intersection protocols. Cryptology ePrint Archive, Report 2016/665, 2016. `http://eprint.iacr.org/2016/665`.

[44] L. Malka. VMCrypt - modular software architecture for scalable secure computation. In *Computer and Communications Security (CCS'11)*, pages 715–724. ACM, 2011.

[45] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay — a secure two-party computation system. In *USENIX Security Symposium 2004*, pages 287–302. USENIX, 2004.

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

[47] G. Mezzour, A. Perrig, V. D. Gligor, and P. Papadimitratos. Privacy-preserving relationship path discovery in social networks. In *Cryptology and Network Security (CANS'09)*, volume 5888 of *LNCS*, pages 189–208. Springer, 2009.

[48] M. D. Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, 2001.

[49] Robert H Morelos-Zaragoza. *The art of error correcting coding*. John Wiley & Sons, 2006. Code generation tools online at `http://eccpage.com`.

[50] R. Motwani and P. Raghavan. *Randomized algorithms*. Cambridge University Press, New York, NY, USA, 1995.

[51] S. Nagaraja, P. Mittal, C.-Y. Hong, M. Caesar, and N. Borisov. BotGrep: Finding P2P bots with structured graph analysis. In *USENIX Security Symposium 2010*, pages 95–110. USENIX, 2010.

[52] M. Nagy, E. De Cristofaro, A. Dmitrienko, N. Asokan, and A.-R. Sadeghi. Do I know you? – efficient and privacy-preserving common friend-finder protocols and applications. In *Annual Computer Security Applications Conference (ACSAC'13)*, pages 159–168. ACM, 2013.

[53] M. Naor and B. Pinkas. Efficient oblivious transfer protocols. In *SIAM Symposium On Discrete Algorithms (SODA'01)*, pages 448–457. Society for Industrial and Applied Mathematics (SIAM), 2001.

[54] A. Narayanan, N. Thiagarajan, M. Lakhani, M. Hamburg, and D. Boneh. Location privacy via private proximity testing. In *Network and Distributed System Security (NDSS'11)*. The Internet Society, 2011.

[55] J. B. Nielsen, P. S. Nordholt, C. Orlandi, and S. S. Burra. A new approach to practical active-secure two-party computation. In *Advances in Cryptology – CRYPTO'12*, volume 7417 of *LNCS*, pages 681–700. Springer, 2012.

[56] NIST. NIST Special Publication 800-57, Recommendation for Key Management Part 1: General (Rev. 3). Technical report, National Institute of Standards and Technology (NIST), 2012.

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

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

[59] B. Pinkas, T. Schneider, G. Segev, and M. Zohner. Phasing: Private set intersection using permutation-based hashing. In *USENIX Security Symposium 2015*, pages 515–530. USENIX, 2015. Full version: `http://eprint.iacr.org/2015/634`.

[60] B. Pinkas, T. Schneider, N. P. Smart, and S. C. Williams. Secure two-party computation is practical. In *Advances in Cryptology – ASIACRYPT'09*, volume 5912 of *LNCS*, pages 250–267. Springer, 2009.

[61] B. Pinkas, T. Schneider, and M. Zohner. Faster private set intersection based on OT extension. In *USENIX Security Symposium 2014*, pages 797–812, 2014. Full version: `http://eprint.iacr.org/2014/447`.

[62] M. Raab and A. Steger. "balls into bins" - a simple and tight analysis. In *Randomization and Approximation Techniques in Computer Science (RANDOM'98)*, volume 1518 of *LNCS*, pages 159–170. Springer, 1998.

[63] P. Rindal and M. Rosulek. Faster malicious 2-party secure computation with online/offline dual execution. In *USENIX Security Symposium 2016*. USENIX, 2016.

[64] Peter Rindal and Mike Rosulek. Improved private set intersection against malicious adversaries. *IACR Cryptology ePrint Archive*, 2016:746, 2016.

[65] T. Schneider and M. Zohner. GMW vs. Yao? Efficient secure two-party computation with low depth circuits. In *Financial Cryptography and Data Security (FC'13)*, volume 7859 of *LNCS*, pages 275–292. Springer, 2013.

[66] R. Schürer and W. Schmid. *Monte Carlo and Quasi-Monte Carlo Methods 2004*, chapter MinT: A Database for Optimal Net Parameters, pages 457–469. Springer, 2006. Online: `http://mint.sbg.ac.at`.

[67] A. Shamir. On the power of commutativity in cryptography. In *International Colloquium on Automata, Languages and Programming (ICALP'80)*, volume 85 of *LNCS*, pages 582–595. Springer, 1980.

[68] A. C. Yao. How to generate and exchange secrets. In *Foundations of Computer Science (FOCS'86)*, pages 162–167. IEEE, 1986.

[69] S. Zahur, M. Rosulek, and D. Evans. Two halves make a whole: Reducing data transfer in garbled circuits using half gates. In *Advances in Cryptology – EUROCRYPT'15*, volume 9057 of *LNCS*, pages 220–250. Springer, 2015.