Faster Private Set Intersection based on OT Extension

(Full Version)*

Benny Pinkas
Bar-Ilan University, Israel

Thomas Schneider
TU Darmstadt, Germany

Michael Zohner
TU Darmstadt, Germany

June 10, 2014

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 all implemented and compared in the same setting.

In this work, we give an overview on existing PSI protocols that are secure against semi-honest adversaries. We take advantage of the most recent efficiency improvements in OT extension to propose significant optimizations to previous PSI protocols and to suggest a new PSI protocol whose runtime 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, and give recommendations on which protocol to use in a particular setting.

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 is used for privacy-preserving computation of functionalities such as relationship path discovery in social networks [41], botnet detection [45], testing of fully-sequenced human genomes [3], proximity testing [48], or cheater detection in online games [11]. Another use case is measurement of the performance of web ad campaigns, by comparing purchases by users who were shown a specific ad to purchases of users who were not shown the ad. This is essentially a variant of PSI where the input of the web advertising party is the identities of the users who were shown the ad, and the input of the merchant, or of an agency that operates on its behalf, is the identities of the buyers. It was published that Facebook and Datalogix, a consumer data collection company, perform this type of measurements.[1] (They used the insecure hash-based solution described in §1.1, but can instead use a properly secure PSI protocol while still being reasonably efficient.)

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 runtime while neglecting other relevant factors such as communication.

In this paper, we give an overview on existing efficient PSI protocols, optimize the recently proposed PSI protocols of [30] and [19], based on garbled circuits and Bloom filters, respectively, and describe a new PSI protocol based on recent results in the area of efficient OT extensions [1, 38]. We compare both the theoretical and empirical performance of all protocols on the same platform and conclude with remarks on the protocols and their suitability for different scenarios.

1.1 Classification of PSI Protocols

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

*Please cite the conference version of this work published at USENIX Security'14 [53].

[1] https://www.eff.org/deeplinks/2012/09/deep-dive-facebook-and-datalogix-whats-actually-getting-shared-and-how-you-can-opt

this protocol is very efficient, it is insecure if the input domain is not large 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 a high entropy, a protocol that compares hashes of the inputs can be used [46].)

PSI is one of the best studied problems in secure computation. Since its introduction, several techniques have been used to realize PSI protocols. While the first PSI protocols were special-purpose solutions based on public-key primitives, other solutions were based on circuit-based generic techniques of secure-computation, that are mostly based on symmetric cryptography. A 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. Finally, we describe PSI protocols that utilize a third party to achieve even better efficiency.

Public-Key-Based PSI A PSI protocol based on the Diffie-Hellmann (DH) key agreement scheme was presented in [32] (related ideas were presented earlier in [40]). This 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. [24] 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 [22], 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 §6. A similar approach that uses oblivious pseudo-random functions to perform PSI was presented in [23]. A protocol that uses polynomial interpolation and differentiation for finding intersections between multi-sets was presented in [37].

Another PSI protocol that uses public-key cryptography (more specifically, blind-RSA operations) and scales linearly in the number of elements was presented in [16] and efficiently implemented and benchmarked in [17].

A PSI protocol based on additively homomorphic encryption was described in [12], but is excluded from this evaluation since it scales quadratically in the number of elements and is hence slower than related solutions.

Circuit-Based PSI Generic secure computation protocols have been subject to huge 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 [30] and evaluated using the Yao's garbled circuits framework of [31]. The authors showed that their Java implementation scales very well with increasing security parameter and outperforms the blind-RSA protocol of [16] for larger security parameter.[2] We reflect on and present new optimizations for circuit-based PSI in §3.

OT-Based PSI A recent PSI protocol of [19] uses Bloom filters [10] and OT extension [33] to obtain very efficient PSI protocols with security against semi-honest and malicious adversaries. We describe this protocol and our optimization using random OT extension [1] in §4.

Third Party-Based PSI Several PSI protocols have been proposed that utilize additional parties, e.g., [4]. In [28], a trusted hardware token is used to evaluate an oblivious pseudo-random function. This approach was extended to multiple untrusted hardware tokens in [20]. Several efficient server-aided protocol for PSI were presented and benchmarked in [35]. For their PSI protocol with a semi-honest server, the authors report a runtime of 1.7 s for server-aided PSI on one million elements using 20 threads between cloud instances in the US east - and west coast and 10 MB of communicated data. In comparison, our fastest PSI protocol without a server requires 4.9 s for 2^{18} elements using four threads and sends 78 MB (cf. Tab. 1 and Tab. 8). Note that this comparison is sketchy and is only meant to demonstrate that using a third party can increase performance. In our work we focus on PSI protocols without a third party.

1.2 Our Contributions

We describe in detail the PSI protocols based on generic secure computation and on Bloom filters, and suggest how to improve their performance using carefully analyzed features of OT extension. We then introduce a new OT-based PSI protocol, and perform a detailed experimental comparison of all the PSI protocols that we described. In the following, we detail our contributions.

Optimizations of Existing Protocols We improve the circuit- and Bloom-filter-based PSI protocols using recent optimizations for OT extension [1]. In particular, in §3 we evaluate the circuit-based solution of [30] 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). In §4.3 we redesign the Bloom filter-based protocol of [19] to benefit from using random OT and to support multi-core environments.

[2]Subsequent work of [17] claimed that the blind-RSA protocol of [16] runs faster than the circuit-based protocol of [30] even for larger security parameter. Their implementation is in C++ instead of Java.

A Novel OT-Based PSI Protocol We present a new PSI protocol that is directly based on OT (§5) and directly benefits from recent improvements in efficient OT extensions [1, 38]. The basic version of the protocol can efficiently compare one element with many elements, but for PSI on n elements it requires $O(n^2 \log n)$ communication. In §6 we use carefully analyzed hashing techniques in order to achieve $O(n \log n)$ communication. 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.

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 the same 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 §7. Certain experimental results were unexpected. We give a partial summary of our results in Tab. 1: the values in parenthesis give the overhead of the original protocols and highlight the gains achieved by our optimizations.

PSI Protocol	**DH** ECC [32]	**Circuit [30]** optimized GMW §3.2 (original GMW [1])	**Bloom Filter** optimized §4.3 (original [19])	**OT** §5+§6
Runtime (s)	416	762 (1,304)	68 (154)	14
Comm. (MB)	24	14,040 (23,400)	740 (1,393)	78

Table 1: Runtime and transferred data for private set intersection protocols on sets with 2^{18} 32-bit elements and 128-bit security with a single thread over Gigabit LAN.

We highlight here the conclusions of our results:

- The Diffie-Hellman-based protocol [32], 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 [30] 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 [30] that circuit-based PSI protocols are faster than the blind-RSA-based PSI protocol of [16] for larger security parameters and given sufficient bandwidth.

- While for larger security parameter previously proposed circuit- and OT-based protocols can be faster than the public-key-based protocols on a Gigabit LAN, the DH-based protocol of [32] outperforms *all* of them in an Internet network setting. Our new OT-based protocol (§5+§6) is the only protocol that maintains its performance advantage in this setting and even outperforms public-key-based PSI protocols for a mobile network setting.

2 Preliminaries

We give our notation and security definitions in §2.1 and review recent relevant work on oblivious transfer in §2.2.

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 $|Y| = n_2$. When the two input sets are of equal size, we use $n = n_1 = n_2$. We refer to elements from X as x and elements from Y as y. All elements in X and Y have bit-length σ (cf. §A 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 refer to a correlation resistant one-way function as CRF, and to a pseudo-random generator as PRG.

We write $\binom{N}{1}$-OT^m_ℓ for m parallel 1-out-of-N oblivious transfers on ℓ-bit strings, and write OT^m_ℓ for $\binom{2}{1}$-OT^m_ℓ.

Security parameters We denote the symmetric security parameter as κ, the asymmetric security parameter as ρ, the statistical security parameter as λ, and use the recommended key sizes of the NIST guideline [50], summarized in Tab. 2. We denote the bit size of elliptic curve points with φ, i.e., $\varphi = 284$ for Koblitz curve K-283 when using point compression.

Security	**SYM** (κ)	**FFC** and **IFC** (ρ)	**ECC** (φ)	**Hash**
80-bit	80	1,024	K-163	SHA-1
128-bit	128	3,072	K-283	SHA-256

Table 2: NIST recommended key sizes for symmetric cryptography (SYM), finite field cryptography (FFC), integer factorization cryptography (IFC), elliptic curve cryptography (ECC) and hash functions.

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 software attestation is enforced or where an untrusted third party is able 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 steps.

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., [15, 18, 22, 24, 29, 34]).

The random oracle model As most previous works on efficient PSI, we use the random oracle model to achieve more efficient implementations [9]. We provide details and argue about the use of random oracles in §B.

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_{ℓ}^{m}), 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 [47], which uses public-key operations and has amortized complexity of $3m$ public-key operations when performing m OTs.

OT extension [7, 33] 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 are orders of magnitude faster. The security parameter κ is essentially independent of the number of OTs m, and can be as small as 80 or 128. Thereby, the computational complexity for performing OT is reduced to such an extent, that the network bandwidth becomes the main bottleneck [1, 21].

Recently, the efficiency of OT extension protocols has gained a lot of attention. In [38], an efficient 1-out-of-N OT extension protocol was shown, that has sub-linear communication in κ for short messages. Another protocol improvement is outlined in [1, 38], which decreases the communication from R to S by half. Additionally, several works [1, 49] 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 elaborate on these OT extension protocols in §C.

3 Circuit-Based PSI

Unlike special purpose private set intersection protocols, the protocols that we describe in this section are based on a *generic* secure computation protocol that can be used for computing arbitrary functionalities. State-of-the-art for computing the PSI functionality is the sort-compare-shuffle (SCS) circuit of [30], which has size $\mathcal{O}(n \log n)$ (cf. §D.1 for details.) We discuss these protocols by reflecting on the generic secure computation protocol of Goldreich-Micali-Wigderson (GMW) [26] (§3.1) and outlining major optimizations for evaluating the SCS circuit for PSI using GMW (§3.2).

The usage of generic protocols holds 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 protocol to compute the size of the intersection, or a function that outputs 1 iff 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.

3.1 The GMW Protocol

We focus on the GMW protocol [26] for generic secure computation, which was implemented in the semi-honest model for multiple parties in [13], optimized for two parties in [55], and extended to the malicious model in [49].

The GMW protocol represents the function to be computed as a Boolean circuit and uses an XOR-based secret-sharing and OT to evaluate the circuit. A circuit with input bit u from P_1 and v from P_2 is evaluated as follows. First, P_1 and P_2 secret-share their input bit $u = u_1 \oplus u_2$ and $v = v_1 \oplus v_2$ and P_i obtains the shares labeled with i. The parties then evaluate the Boolean circuit gate-by-gate, as detailed next. To evaluate an XOR gate with input wires u and v and output wire w, P_1 locally computes $w_1 = u_1 \oplus v_1$ while P_2 locally computes $w_2 = u_2 \oplus v_2$.

Evaluating AND gates using multiplication triples An AND gate with input wire u and v and output wire w requires an interaction between both parties using a *multiplication triple* [6]. A multiplication triple is a set of shares $\alpha_1, \alpha_2, \beta_1, \beta_2, \gamma_1, \gamma_2 \in \{0,1\}$ with $(\alpha_1 \oplus \alpha_2) \wedge$

$(\beta_1 \oplus \beta_2) = \gamma_1 \oplus \gamma_2$. Given a multiplication triple, to evaluate an AND gate implementing u AND v, the parties compute $d_i = \alpha_i \oplus u_i$ and $e_i = \beta_i \oplus v_i$, exchange d_i, e_i, reconstruct $d = d_1 \oplus d_2$ and $e = e_1 \oplus e_2$, and compute the shares of the gate output wire as $w_1 = (d \wedge e) \oplus (d \wedge \beta_1) \oplus (e \wedge \alpha_1) \oplus \gamma_1$ and $w_2 = (d \wedge \beta_2) \oplus (e \wedge \alpha_2) \oplus \gamma_2$. These are all extremely efficient operations and therefore the efficiency of the evaluation depends on the efficiency of generating multiplication triples.

As described in [1], multiplication triples can be generated using two random OTs on one-bit strings as follows: both parties choose $\alpha_i \in_R \{0,1\}$ and run two random OTs, where in the first OT P_1 acts as sender and P_2 as receiver with choice bit α_2, and in the second OT P_2 acts as sender and P_1 as receiver with choice bit α_1. From each OT, the sender obtains (x_0^i, x_1^i) and sets $\beta_i = x_0^i \oplus x_1^i$ and the receiver obtains $x_{\alpha_i}^i$. To compute valid γ_0, γ_1 values for the triple, note that $(\alpha_1 \oplus \alpha_2) \wedge (\beta_1 \oplus \beta_2) = (\alpha_1 \wedge \beta_1) \oplus (\alpha_1 \wedge \beta_2) \oplus (\alpha_2 \wedge \beta_1) \oplus (\alpha_2 \wedge \beta_2) = \gamma_0 \oplus \gamma_1$. P_i locally computes $\alpha_i \wedge \beta_i$. Values $\alpha_1 \wedge \beta_2$ and $\alpha_2 \wedge \beta_1$ are computed using the output of the random OT as $\alpha_1 \wedge \beta_2 = x_{\alpha_1}^2 \oplus x_0^2$ and $\alpha_2 \wedge \beta_1 = x_{\alpha_2}^1 \oplus x_0^1$. [3] Finally, P_1 sets $\gamma_1 = (\alpha_1 \wedge \beta_1) \oplus x_0^1 \oplus x_{a_1}^2$ and P_2 sets $\gamma_2 = (\alpha_2 \wedge \beta_2) \oplus x_0^2 \oplus x_{\alpha_2}^1$. These computations can be done in a preprocessing step before the input is known, are independent of circuit's structure, and highly parallelizable.

3.2 Optimized Circuit-Based PSI

We describe in this section an optimization which greatly reduces the overhead of circuit based PSI for GMW (as is detailed in Tab. 5 in §7, the reduction in the runtime for inputs of size 2^{18} is about 40%). The optimization is based on a protocol proposed in [43].

As outlined in §D.1, the size of the SCS circuit is dominated by the multiplexer gates. In each multiplexer operation with σ-bit inputs x and y and a choice bit s, we compute $z[j] = s \wedge (x[j] \oplus y[j]) \oplus x[j]$ for each $1 \leq j \leq \sigma$ using σ AND gates in total. The evaluation of this multiplexer circuit in the GMW protocol requires random $\text{OT}_1^{2\sigma}$, namely 2σ random OTs of single-bit inputs. We observe that the same wire s is input to multiple AND gates which allows for the following optimization.

Consider an input wire u that is the input to multiple AND gates of the form $w[1] = (u \text{ AND } v[1]), \ldots, w[\sigma] = (u \text{ AND } v[\sigma])$. Similar to the evaluation of a single AND gate described in §3.1, these gates can be evaluated using a multiplication triple generalized to vectors, which we call a *vector multiplication triple*.

A vector multiplication triple has the following form: $\alpha_1, \alpha_2 \in \{0,1\}$; $\beta_1, \beta_2, \gamma_1, \gamma_2 \in \{0,1\}^\sigma$, where P_i holds the shares labeled with i that satisfy the condition $(\alpha_1 \oplus \alpha_2) \wedge (\beta_1[j] \oplus \beta_2[j]) = \gamma_1[j] \oplus \gamma_2[j]$. To evaluate the AND gates, both parties compute $d_i = \alpha_i \oplus u_i$ and $e_i[j] = \beta_i[j] \oplus v_i[j]$, exchange $d_i, e_i[j]$, set $d = d_1 \oplus d_2$, $e[j] = e_1[j] \oplus e_2[j]$, and $w_i[j] = (d \wedge e[j]) \oplus (d \wedge \beta_i[j]) \oplus (e[j] \wedge \alpha_i) \oplus \gamma_i[j]$. The vector multiplication triple can be pre-computed analogously to the regular multiplication triples described in §3.1, but using random OT_σ^2, namely only two random OTs applied to σ-bit strings: The parties each choose $\alpha_1, \alpha_2 \in_R \{0,1\}$ and perform a random OT_σ^1 with P_1 acting as sender and P_2 acting as receiver with choice bit α_2, and a second random OT_σ^1 with P_2 acting as sender and P_1 acting as receiver with choice bit α_1. From these random OTs, P_i obtains $\beta_i \in \{0,1\}^\sigma = x_0^i \oplus x_1^i$ and, analogously to the regular multiplication triple generation, a valid $\gamma_i \in \{0,1\}^\sigma$.

Efficiency Overall, evaluating σ AND gates with a vector multiplication triple requires to send $2\sigma + 2$ bits (instead of 4σ bits with σ regular multiplication triples). Generating a vector multiplication triple requires 2 random OTs on σ-bit strings (instead of 2σ random OTs with σ regular multiplication triples); as the communication of random OT is independent of the input length, this improves communication by factor σ.

In the SCS circuit we have $2n\log_2 n + n + 1$ multiplexers, each of which can be evaluated using a single vector multiplication triple. This reduces the number of random OTs from $2\sigma(2n\log_2 n + n + 1)$ to $2(2n\log_2 n + n + 1)$.

Further applications of vector multiplication triples As a side note, we comment that our vector multiplication triples can be used in every circuit where wires are used as input in two or more AND gates. As such, another beneficial application is multiplication. When computing a multiplication between two σ-bit numbers x and y using the school method multiplication circuit [55], each bit x_i is multiplied with every bit of y: $\forall_{1 \leq i \leq \sigma} \forall_{1 \leq j \leq \sigma} (x_i \wedge y_j)$. Here, using vector multiplication triples allows to reduce the total number of random OTs by a factor two, from $4\sigma^2 - 2\sigma$ OTs to $2\sigma^2$.

4 Bloom Filter-Based PSI

The recent PSI protocol of [19] uses Bloom Filters (BF) and OT to compute set intersection. We summarize Bloom filters in §4.1 and the PSI protocol of [19] in §4.2. We then present a redesigned optimized version of the protocol in §4.3. This optimization reduces the runtime for inputs of size 2^{18} by 55% – 60% (cf. §7, Tab. 5).

[3] To verify the correctness of the equations, note that we can rewrite $\alpha_1 \wedge \beta_2 = \alpha_1 \wedge (x_0^2 \oplus x_1^2) = (\alpha_1 \wedge (x_0^2 \oplus x_1^2) \oplus x_0^2) \oplus x_0^2 = x_{\alpha_1}^2 \oplus x_0^2$ and $\alpha_2 \wedge \beta_1$ as $x_{\alpha_2}^1 \oplus x_0^1$, analogously.

4.1 The Bloom Filter

A BF that represents a set of n elements consists of an m-bit string F and k independent uniform hash functions $h_1,...,h_k$ with $h_i : \{0,1\}^* \mapsto [1,m]$, for $1 \le i \le k$. Initially, all bits in F are set to zero. An element x is inserted into the BF by setting $F[h_i(x)] = 1$ for all i. To query if the BF contains an item y, one checks all bits $F[h_i(y)]$. If there is at least one j such that $BF[h_j(y)] = 0$, then y is not in the BF. If, on the other hand, all bits $BF[h_i(y)]$ are set to one, then y is in the BF except for a false positive probability ε. An upper bound on ε can be computed as $\varepsilon = p^k(1+O(\frac{k}{p}\sqrt{\frac{\ln m - k\ln p}{m}}))$, where $p = 1-(1-\frac{1}{m})^{kn}$. The authors of [19] propose to choose the number of hash functions as $k = 1/\varepsilon$ and the size of the BF as $m = kn/\ln 2 \approx 1.44kn$. In their experiments, they set $\varepsilon = 2^{-\kappa}$, resulting in $k = \kappa$ and a filter of size $m \approx 1.44\kappa n$. [4]

The intersection between two BFs F_X and F_Y, representing sets X and Y, respectively, can be computed as $F_{(X\wedge Y)} = F_X \wedge F_Y$. However, as described in [19], $F_{(X\wedge Y)}$ has more bits set to one than a BF $F_{(X\cap Y)}$ that was generated from the intersection $X \cap Y$. For example, assume that there are two sets $X = \{x\}$ and $Y = \{y\}$ with $x \neq y$. If there exist i, j such that $h_i(x) = h_j(y)$, we have $F_{(X\wedge Y)}[h_i(x)] = 1$. However, the intersection $X \cap Y = \varnothing$, results in $F_{(X\cap Y)}[h_i(x)] = 0$. Thus, learning $F_{(X\wedge Y)}$ reveals more information about the set of the other party than is revealed by only obtaining the result, so a different approach is needed, as described next.

4.2 Garbled Bloom Filter-Based PSI

To avoid unintentional information leakage when using Bloom filters for PSI, the authors of [19] introduced a variant of the BF, called Garbled Bloom Filter (GBF). Like a BF, a GBF G uses κ hash functions $h_1,...,h_\kappa$, but instead of single bits, it holds shares of length ℓ at each position $G[i]$, for $1 \le i \le m$. These shares are chosen uniformly at random, subject to the constraint that for every element x contained in the filter G it holds that $\bigoplus_{j=1}^{\kappa} G[h_j(x)] = x$.

To represent a set X using a GBF G, all positions of G are initially marked as unoccupied. Each element $x \in X$ is then inserted as follows. First, the insertion algorithm tries to find a hash function $t \in [1...\kappa]$ such that $G[h_t(x)]$ is unoccupied (the probability of not finding such a function is equal to the probability of a false positive in the BF, which is negligible due to the choice of parameters). All other unoccupied positions $G[h_j(x)]$ are set to random ℓ-bit shares. Finally, $G[h_t(x)]$ is set to $G[h_t(x)] = x \oplus \left(\bigoplus_{j=1,j\neq t}^{\kappa} G[h_j(x)]\right)$ to obtain a valid sharing of x. We emphasize that because existing shares need to be re-used, the generation of the GBF cannot be fully parallelized. (We describe below in §4.3 how the protocol can be modified to enable a parallel execution.)

In the semi-honest secure PSI protocol of [19], P_1 generates a m-bit GBF G_X from its set X and P_2 generates a m-bit BF F_Y from its set Y. P_1 and P_2 then perform OT_ℓ^m, where for the i-th OT P_1 acts as a sender with input $(0, G_X[i])$ and P_2 acts as a receiver with choice bit $F_Y[i]$. Thereby, P_2 obtains an intersection GBF $G_{(X\wedge Y)}$, for which $G_{(X\wedge Y)}[i] = 0$ if $F_Y[i] = 0$ and $G_{(X\wedge Y)}[i] = G_X[i]$ if $F_Y[i] = 1$. P_2 can check whether an element y is in the intersection by checking whether $\bigoplus_{i=1}^{k} G_{(X\wedge Y)}[h_i(y)] \stackrel{?}{=} y$. (Note that P_2 cannot perform this check for any value which is not in its input set, since the probability that it learns all GBF locations associated with that value is equal to the probability of a false positive, which is negligible due to the choice of parameters.) The bit-length of the shares in the GBF can be set to $\ell = \lambda$.

Optimization Since one input of the sender in the OT is fixed to zero, the OT extension protocol can be optimized such that only one value needs to be transferred from sender to receiver, similarly to the correlated OT extension protocol of [1] which reduces both the computation and the communication complexity. Additionally, the parties only need to evaluate the hash function when the bit in the Bloom filter is set to 1, which reduces the computation complexity for both by half.

4.3 Random GBF-Based PSI

We introduce a further optimization of the GBF-based PSI protocol of [19], which we call the *random Garbled Bloom Filter* protocol. The core idea is to have parties collaboratively generate a *random* GBF. This is in contrast to the original protocol where the GBF had to be of a specific structure (i.e., have the XOR of the entries of $x \in X$ be x). The modified protocol can be based on random OT extension (in fact, on a version of the protocol which is even more efficient than the original random OT extension). For each position in the filter, each party learns a random value if the corresponding bit in its BF is 1. P_1 then sends to P_2 the XOR of the GBF values corresponding to each of its inputs, and P_2 compares these values to the XOR of the GBF values of its own inputs.

The primitive that enables this solution is a variant of random OT extension, which we denote as an oblivious pseudo-random generator (OPRG). It takes as inputs bits b_1, b_2 from each party, respectively, generates a random

[4] In our application it is insufficient to set ε to be equal to the *statistical* security parameter, since in the PSI protocol one of the parties might mount a brute force attack where it attempts to find items that are mapped to "1" locations in the Bloom filter. The parameters must ensure that the success probability of this attack is negligible.

string s, and outputs to P_t s if $b_t = 1$ and nothing otherwise, for $t \in \{0,1\}$. Additionally, we require that the parties remain oblivious to whether the other party obtained s. A protocol for computing this functionality is obtained by modifying the existing random OT extension protocol of [1] as follows.

Recall that in random OT extension, S has no input in the i-th OT and outputs two values (x_0^i, x_1^i) as $x_0^i = H(q_i)$, $x_1^i = H(q_i \oplus s)$, while R inputs a choice bit vector b and outputs $x_{b[i]}^i = H(t_i)$ (cf. §2.2 and §C). The new functionality is obtained by having S ignore the x_0^i output that it receives, and ignore also the x_1^i output if $b_1 = 0$. Similarly, R ignores its output if $b_2 = 0$. The random OT extension protocol thus becomes more efficient, since the parties can ignore the parts of the computation in the original protocol that are required for computing the values that they now ignore.

Our resulting Bloom filter-based protocol works as follows. First, P_1 and P_2 each generate a BF, F_X and F_Y respectively. They evaluate the OPRG with P_1 being the sender and P_2 being the receiver, using the bits of F_X and F_Y as inputs, to obtain random GBFs G_X and G_Y with entries in $\{0,1\}^\ell$. For each element x_j in its set X, P_1 then computes $m_{P_1}[j] = \bigoplus_{i=1}^{\kappa} G_X[h_i(x_j)]$, with $1 \le j \le n_1$. Finally, P_1 sends all m_{P_1} values in random order to P_2, which identifies whether an element y in its set is in the intersection by checking whether a j exists such that $m_{P_1}[j] = \bigoplus_{i=1}^{\kappa} G_Y[h_i(y)]$.

Correctness For each item in the intersection, P_2 gets from P_1 the same XOR value that it computed from its own GBF, and therefore identifies that the item is in the intersection. For any item which is not in the intersection, it holds with overwhelming probability that the XOR value computed by P_2 is independent of the n_1 values received from P_1. The probability of a false positive identification for that value is therefore $n_1 \cdot 2^{-\ell}$. The probability of a false positive identification for any of the values is $n_1 n_2 \cdot 2^{-\ell}$. To achieve correctness with probability 1-$2^{-\lambda}$, we therefore set $\ell = \lambda + \log_2 n_1 + \log_2 n_2$.

Security The security of each party can be easily proved using a simulation argument. P_2's security is obvious, since the only information that P_1 learns are the random outputs of the random OT protocol, which are independent of P_2's input and can be easily simulated by P_1. P_1's security is apparent from observing that the information that P_2 receives from P_1 is composed of

- The XOR values that P_2 computed for each item in the intersection.
- The XOR values that P_1 computed for its $n_1 - |X \cap Y|$ items that are not in the intersection. These values are independent of P_2's BF unless one of these items is a false-positive identification in the filter, which happens with negligible probability ε.

Therefore, the information received from P_1 can be easily simulated by P_2 given its legitimate output, i.e., $X \cap Y$.

Efficiency As shown in Tab. 3, our resulting random GBF-based PSI protocol has less computation and communication complexity than the original GBF protocol in [19] (even with the optimizations described in §4.2 that are based on the OT extension protocol of [1]). In terms of communication, in our new protocol, P_1 has to send the $n_1\ell$-bit vector m_{P_1} and P_2 has to send $m\kappa$ bits in the random OTs. (This is compared to $2m\lambda$ bits and $2m\kappa$ bits sent in the original protocol. Later in our experiments in §7 we show that the communication is reduced by a factor between 1.9 and 3, cf. Tab. 6.)

The computation complexity of our protocol is $\mathrm{HW}(F_X)$ hash function evaluations for P_1 and $\mathrm{HW}(F_Y)$ hash function evaluations for P_2, where $\mathrm{HW}(\cdot)$ denotes the Hamming weight. When the number of hash functions k and the size of the BF m are chosen optimally, we can approximate the average Hamming weight in a BF using the probability that a single bit is set to 1, which is $1-(1-(\frac{1}{m}))^{kn} \approx \frac{1}{2}$. Thus, $\mathrm{HW}(F) \approx \frac{m}{2}$.

A main advantage of our protocol is that it allows to parallelize *all* operations: BFs can be generated in parallel (bits in the BF are changed only from 0 to 1) and, most importantly, the random GBF can also be constructed in parallel, in contrast to the original GBF-based protocol.

Optimization	Party	# Bits Sent	# calls to H
Original GBF-based PSI [19]	P_1	$2m\lambda$	$2m$
	P_2	$2m\kappa$	m
[19] with OT of [1]	P_1	$m\lambda$	m
	P_2	$m\kappa$	$m/2$
Random GBF-based PSI (§4.3)	P_1	$n_1\ell$	$m/2$
	P_2	$m\kappa$	$m/2$

Table 3: Communication and computation complexities for Bloom-filter-based PSI of [19] and our optimization. (λ: statistical security parameter, κ: symmetric security parameter, n_i: number of elements of party P_i, $m \approx 1.44\kappa \max(n_1, n_2)$, $\ell = \lambda + \log_2 n_1 + \log_2 n_2$).

5 Private Set Intersection via OT

We propose a new private set intersection protocol that is based on the most efficient OT extension techniques, in particular the random OT functionality [1, 49] and the efficient 1-out-of-N OT of [38]. This PSI protocol scales very efficiently with an increasing set size.

We first describe the protocol for a private equality test (PEQT) between two elements x and y (§5.1) and then describe how to efficiently extend it for comparing y to a set $X = \{x_1, ..., x_n\}$ (§5.2). The resulting protocol can then be simply extended to perform PSI between sets X and Y by applying the parallel comparison protocol for each element $y \in Y$ (§5.3). The overhead of the protocol can be greatly improved using hashing (§6).

5.1 The Basic PEQT Protocol

In the most basic private equality test (PEQT) protocol, P_1 and P_2 check whether their σ-bit elements x and y are equal by engaging in random $\binom{2}{1}$ $\mathrm{OT}^{\sigma}_{\ell}$, where P_2 uses the bits of y as its choice vector. From each random OT, P_1 obtains two uniformly distributed and random ℓ-bit strings (s^i_0, s^i_1), and P_2 obtains $s^i_{y[i]}$. P_1 then computes $m_{P_1} = \bigoplus_{i=1}^{\sigma} s^i_{x[i]}$ (the XOR of the strings corresponding to the binary representation of x) and sends it to P_2. P_2 compares this value to $m_{P_2} = \bigoplus_{i=1}^{\sigma} s^i_{y[i]}$ and decides that $x = y$ iff $m_{P_1} = m_{P_2}$.

The basic private equality test can be improved by using a base-N representation of the inputs and a $\binom{N}{1}$ OT in the protocol. Specifically, let $N = 2^{\eta}$. P_1 and P_2 check whether their σ-bit elements x and y are equal by representing them using $t = \sigma/\eta$ letters from an alphabet of size N, and then engaging in random $\binom{N}{1}$-OT^{t}_{ℓ}.

For this, P_2 cuts its σ-bit element y into t blocks $y[i]$ of bitlength η each: $y = y[1]||\dots||y[t]$; similarly, P_1 interprets $x = x[1]||\dots||x[t]$. In the i-th random $\binom{N}{1}$-OT, P_2 inputs $y[i]$ as choice bits and P_1 obtains N random and uniformly distributed ℓ-bit strings $(s^i_0, ..., s^i_{N-1})$; P_2 obtains $s^i_{y[i]}$. P_1 sends $m_{P_1} = \bigoplus_{i=1}^{t} s^i_{x[i]}$ to P_2 who compares it to $m_{P_2} = \bigoplus_{i=1}^{t} s^i_{y[i]}$ and decides that $x = y$ iff $m_{P_1} = m_{P_2}$.

Correctness If $x = y$ then the choices that both parties make for their sums are equal, i.e., $m_{P_1} = \bigoplus_{i=1}^{t} s^i_{x[i]} = \bigoplus_{i=1}^{t} s^i_{y[i]} = m_{P_2}$, and P_2 successfully identifies equality.

If $x \neq y$ then the probability that $m_{P_1} = m_{P_2}$ is $2^{-\ell}$. To see that this is true, assume w.l.o.g. that the inputs differ in their last sub-string, i.e., $x[t] \neq y[t]$. Equality only holds if the last element received by P_2, namely $s^t_{y[t]}$, is equal to $\bigoplus_{i=1}^{t} s^i_{x[i]} \oplus \bigoplus_{i=1}^{t-1} s^i_{y[i]}$. The value of $s^t_{y[t]}$ is independent of the other values, and therefore this equality happens with probability $2^{-\ell}$ and thus we can set ℓ to be equal to the statistical security parameter λ.

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 t outputs in the random $\binom{N}{1}$-OT protocols, and of the value m_{P_1} sent by P_1. If $x = y$ then m_{P_1} is equal to the XOR of the first t values. Otherwise, all $t + 1$ values are uniformly distributed. 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 [25].

Efficiency Since in the i-th random OT P_1 needs only the output $s^i_{x[i]}$, it suffices to evaluate one hash function per random OT. When using the random $\binom{2}{1}$-OT extension protocol[5] of [1] and $\ell = \lambda$, the parties perform random $\mathrm{OT}^{\sigma}_{\lambda}$, send $\sigma\kappa + \lambda$ bits, and do σ hash function evaluations each. In comparison, when using the random $\binom{N}{1}$-OT extension protocol of [38], the parties perform only σ/η OTs and send 2κ bits per OT (cf. §C.3); in total, they have to send $2\sigma\kappa/\eta + \lambda$ bits and do σ/η hash function evaluations each. The analysis in §C.3 shows that setting $\eta = 8$ results in optimal performance for our PSI protocols.

5.2 Private Set Inclusion Protocol

In a private set inclusion protocol, P_1 and P_2 check whether y equals any of the values in $X = \{x_1, ..., x_{n_1}\}$. The set inclusion protocol is similar to the basic PEQT protocol, but in order to perform multiple comparisons in parallel, the OTs are computed over longer strings, essentially transferring (in parallel) a random string for each element in the set X.

In more detail, both parties run a random $\binom{N}{1}$-$\mathrm{OT}^{t}_{n_1\ell}$, where P_2 uses the bits of y as choice bits. Each received string is of length $n_1\ell$ bits. That is, in the i-th random OT, P_1 obtains N random strings $(s^i_0, ..., s^i_{N-1}) \in \{0,1\}^{n_1\ell}$, and P_2 obtains one random string $s^i_{y[i]}$. The strings are parsed as a list of n_1 sub-strings of length ℓ bits each. We refer to the j-th sub-string in these lists as $s^i_w[j]$, for $1 \leq j \leq n_1$ and $0 \leq w < N$. Using these sub-strings, P_1 and P_2 can then compute the XOR of the strings corresponding to their respective inputs, compare the results and decide on equality, as was described in the basic PEQT protocol in §5.1. In more detail, P_1 computes $m_{P_1}[j] = \bigoplus_{i=1}^{t} s^i_{x_j[i]}[j]$ and sends the $n_1\ell$-bit string m_{P_1} to P_2. P_2 decides whether y matches any of the elements in X by computing $m_{P_2} = \bigoplus_{i=1}^{t} s^i_{y[i]}$ and checking whether a j exists with $m_{P_1}[j] = m_{P_2}$.

Correctness and security follow from the properties of the protocol of §5.1. However, now we require that the value m_{P_2} and *all* the n_1 values $m_{P_1}[j]$ are distinct, which happens with probability $n_1 2^{-\ell}$. Thus, to achieve correctness with probability 1-$2^{-\lambda}$, we must increase the

[5]Note that for $\sigma < \kappa$ we can perform σ base-OTs instead of using OT extension. However, here we analyze the costs when using OT extension for simplicity and consistency reasons.

bit-length of the OTs to $\ell = \lambda + \log_2 n_1$. Also, note that P_2 learns the position j at which the match is found, which can be avoided by randomly permuting the inputs.

Efficiency The set inclusion protocol that compares y to *many* values has the same number of random OTs as the basic comparison protocol comparing y to a *single* value, but it requires the transferred strings to be of length $n_1(\lambda + \log_2(n_1))$ bits instead of λ bits. Note, however, that since we use random OTs there is no need to send these strings in the OT protocol. Instead, all strings corresponding to the same value of the same input bit can be generated from a single seed using a pseudo-random generator. Therefore, the amount of data transferred in the OTs is the same as for the single comparison PEQT protocol.

The only additional data that is sent is the $n_1(\lambda + \log_2 n_1)$-bit string m_{P_1}, which P_1 sends to P_2. Hence, the total amount of communication is $2\sigma\kappa/\eta + n_1(\lambda + \log_2 n_1)$ bits.

In addition, the PRG which is used to generate the output string from the OT must be evaluated multiple times to generate the $n_1(\lambda + \log_2 n_1)$ bits. Therefore, the set inclusion protocol, which compares y to n_1 elements, is less efficient than a single run of the PEQT protocol, but is definitely more efficient than n_1 invocations of the PEQT protocol.

5.3 The OT-Based PSI Protocol

To obtain the final PSI protocol that computes $X \cap Y$, P_2 simply invokes the private set inclusion protocol of §5.2 for each $y \in Y$. Correctness and security follow from the properties of the private set inclusion protocol.

Efficiency Overall, to compute the intersection between sets X and Y of σ-bit elements, the protocol requires $n_2\sigma/\eta$ random $\binom{N}{1}$-OTs of $n_1(\lambda + \log_2 n_1)$ bit-strings and additionally $n_1 n_2(\lambda + \log_2 n_1)$ bits to be sent. Using the random $\binom{N}{1}$-OT of [38], the total amount of communication is $2n_2\sigma\kappa/\eta + n_1 n_2(\lambda + \log_2 n_1)$ bits. For large n_1 and n_2, this amount of communication grows too large for an efficient solution. In order to cope with large sets, one can use a hashing scheme, as shown in §6.

6 Hashing Schemes and PSI

Several private set intersection protocols are based on running many invocations of pairwise private equality tests (PEQT). These protocols include [12, 24, 30] or our set inclusion protocol in §5. A straightforward implementation of these protocols requires n^2 invocations of PEQT for sets of size n, and therefore does not scale well. In [22, 24] it was proposed to use hashing schemes to reduce the number of comparisons that have to be computed. The idea is to have each party use a publicly known random hashing scheme to map its input elements to a set of bins. If an input element is in the intersection, both parties map it to the same bin. Therefore, the protocol can check for intersections only between items that were mapped to the same bin by both parties.

Naively, 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 the total overhead is $O(n)$. However, privacy requires that the parties hide from each other how many of their inputs were mapped to each bin.[6] 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 protocol, since the number of comparisons per bin now depends on the size of the most populated bin rather than on the actual number of items in the bin. However, while the parties need to pretend externally that all their items are real, they do not need to apply all their internal computations to their dummy items (since they know that these items are not in the intersection). A careful implementation of this observation, which takes into account timing attacks, can further optimize the computation complexity of the underlying protocols.

The work of [22, 24] gave asymptotic values for the bin sizes that are used with this technique, and of the resulting overhead. They left the task of setting appropriate parameters for the hashing schemes to future work. We revisit the hashing schemes that were outlined in [22,24], namely, simple hashing, balanced allocations, and Cuckoo hashing (§6.1). We evaluate the performance when using hashing schemes for PSI (§6.2), and describe an analysis of the involved parameters (§6.3). We conclude that Cuckoo hashing yields the best performance (for parameters which we find to be most reasonable).

6.1 Hashing Schemes

Simple Hashing In the simplest hashing scheme the hash table consists of b bins $B_1 \ldots B_b$. 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 al-

[6] 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.

ready stored in that bin. Estimating the maximum number of elements that are mapped to any bin, denoted max_b, is a non-trivial problem and has been subject to extensive research [27, 42, 54]. When hashing m elements to $b = m$ bins, [27] showed that $max_b = \frac{\ln m}{\ln\ln m}(1 + o(1))$ 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 $O(\frac{\ln m}{\ln\ln m})$, respectively. When decreasing the number of bins to a value b satisfying $c \cdot b \ln b = m$ for some constant c, it was shown in [54] that $max_b = (d_c - 1 + \alpha)\ln b$, where d_c is the largest solution to $f(x) = 1 + x(\ln c - \ln x + 1) - c = 0$, and α is a parameter for adjusting the conservativeness of the approximation, and should be set to be slightly larger than 1. In this case the expected and maximum number of elements mapped to a bin are of the same order $O(\ln b) \approx O(\ln m)$. This is preferable for our purposes, since even though privacy requires that we set each bin to be as large as the most populated bin, this size is of the same order as the expected size of a bin when no privacy is needed.

Balanced Allocations The balanced allocations hashing scheme [2] uses two uniformly random hash functions $h_1, h_2 : \{0,1\}^\sigma \mapsto [1, m]$. An element e is mapped by checking which of the two bins $B_{h_1(e)}$ and $B_{h_2(e)}$ is less occupied, and mapping the element to that bin. A lookup for an element q is then performed by checking both bins, $B_{h_1(q)}$ and $B_{h_2(q)}$, and comparing the elements in these bins to q. The advantage of this scheme, shown in [2], is that when hashing m elements into $b = m$ bins, max_b is only $\frac{\ln\ln m}{\ln 2}(1 + o(1))$, i.e., exponentially smaller than in simple hashing.

Cuckoo Hashing Similar to balanced allocations hashing, Cuckoo hashing [51] uses two hash functions $h_1, h_2 : \{0,1\}^\sigma \mapsto [1, b]$ to map m elements to $b = 2(1 + \varepsilon)m$ 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 $i \in \{1, 2\}$. The procedure is repeated until no more evictions are necessary, or until a threshold number of relocations been performed. In the latter case, the last element is put in a special stash s. It was shown that for a stash of size $s \leq \ln m$, insertion of m elements fails with probability m^{-s} [36]. A lookup in this scheme is very efficient as it only compares e to the two items in $B_{h_1(e)}$ and $B_{h_2(e)}$ and to the s items in the stash. In exchange for the improved lookup overhead, the size of the hash table is increased to about $2m$ bins.

6.2 Evaluation of Hashing-Based PSI

We evaluate the asymptotic overhead of applying the OT-based PSI protocol that was introduced in §5.3 while using any of the hashing scheme that we described. Also note that P_1 can save communication since instead of sending all masks for each bin (including masks for both dummy and real values), it can send only the masks of its real values (in permuted order, so that P_2 does not know which value was in each bin). P_2 can then simply check every mask received from P_1 against every computed mask. However, in this case the bit-length ℓ of the masks has to be increased to $\ell' = \lambda + \log_2 n_1 + \log_2 n_2$, since P_2 has to perform a total of $n_1 n_2$ comparisons and the overall error probability must be at most $2^{-\lambda}$. In the following, we address the mask length for checking one item against a set of n_1 items as $\ell_1 = \lambda + \log_2 n_1$ and the mask length for checking a set of n_2 items against n_1 items as $\ell_2 = \lambda + \log_2 n_1 + \log_2 n_2$.

PSI based on simple hashing A protocol based on simple hashing allocates the n inputs of P_2 to b bins, such that $n = O(b \ln b)$ and b is approximately $O(n/\ln n)$. Each bin is padded with dummy items to contain the maximum number of items that is expected in a bin, which is $O(\ln b) = O(\ln n)$. For each bin, the parties need to compute the intersection between sets of $O(\ln n)$ items. Each item can be represented using $O(\ln\ln n)$ bits.[7] The protocol requires $O(\ln n \ln\ln n)$ random OTs for each bin. The total number of OTs is therefore $O(n \ln\ln n)$. The length of the values transferred in the OTs (the masks) is $\ell_2 \ln n$ bits.

PSI based on balanced allocations A major problem occurs when using balanced allocations hashing for PSI: every item can be mapped to one of two bins, and therefore it is unclear with which of P_1's bin should P_2 compare its own input elements e. Furthermore, the protocol must hide from each party the choice of bins made by the other party to store e, since that choice depends on other input elements and might reveal information about them. The solution to this is to use balanced allocations by P_2 alone, whereas P_1 maps each of its input elements to *two* bins using simple hashing with both hash functions h_1 and h_2. When using $b = n$ bins, P_2 has $O(\ln\ln n)$ items in each bin, whereas P_1 has $O(\ln n/\ln\ln n)$ items in every bin (actually, it has twice as many items as with simple hashing, since it maps each item twice). The items can be represented using strings of $O(\ln\ln n)$ bits. The protocol continues as before. P_2 learns the output, but since P_1 does not use balanced allocations, P_1 does not learn

[7]This holds since the items in a bin can be hashed to a shorter representation, as long as no collisions occurs. The length of the hashed value should be about $\lambda + \log((\ln n)^2) = O(\ln\ln n)$.

P_2's choices in that hashing scheme. The number of OTs is linear in the number of items stored by P_2 multiplied by the representation length, e.g., $O(n \cdot (\ln\ln n)^2)$ OTs on $\ell_2 \ln n / \ln\ln n$ bit strings. This overhead is larger than that of the simple hashing-based scheme.

PSI based on Cuckoo hashing Designing PSI based on Cuckoo hashing encounters the same privacy problem as when using balanced allocations hashing, and therefore the same solution is used. P_2 uses Cuckoo hashing whereas P_1 maps each of its elements using simple hashing with each of the two hash functions. P_2 maps a single item to each of the $2n$ bins, whereas P_1's bins contain $O(\ln n)$ items. In addition, P_2 has a stash of $s \leq \ln n$ elements. Each of these elements must be compared with each of P_1's n elements. An item in a bin can again be represented using $O(\ln\ln n)$ bits, whereas an item in the stash can be represented using $O(\ln n)$ bits. Furthermore, when checking items in the stash, we check one item against n_1, allowing us to reduce the bit-size of the masks in the OTs to ℓ_1 instead of ℓ_2. The protocol therefore performs $O(n \ln\ln n)$ OTs on inputs of length $O(\ell_2 \ln\ln n)$ bits (for the items in the bins), and in addition $O((\ln n)^2)$ OTs of inputs of length $O(\ell_1 \ln n)$ bits (for the items in the stash, which are each compared to all items of P_1's input). Overall, the protocol has the same asymptotic overhead as the protocol that uses simple hashing.

6.3 Maximum Bin Size and Overhead

When using hashing schemes for private set intersection, the number of bins b and the corresponding maximum bin size max_b must be set to values that balance efficiency and security. If max_b is chosen too small, the probability of a party failing to perform the mapping, denoted P_{fail}, increases. As a result, the output might be inaccurate (since not all items can be mapped to bins), or one of the parties 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 increases with b and max_b. An asymptotic analysis of the maximum bin size was presented in [22, 24], but leaves the exact choice of b and max_b and the resulting P_{fail} to further work. In the following, we analyze the complexity of the hashing schemes when used in combination with our set inclusion protocol, described in §5. To compare the performance of the hashing schemes on a unified base, we depict in Tab. 4 the overall communication, divided into the number of OTs (where we run t OTs per element) and the number of bits sent from P_1 to P_2.

In §E we detail the analysis of setting the optimal parameters for usage of the different hashing schemes in our PSI protocol, and of the resulting number of OTs and communication overhead. The results are depicted in Tab. 4 and show that Cuckoo hashing has the lowest communication. In addition, this scheme has a stronger guarantee on the upper bound of P_{fail}, since we achieve rehash probability of n^{-s}. We therefore use this scheme in our implementation and experiments.

Parameter	**Total # OTs**	**Comm. [bits]** P_1 **to** P_2	**Comm. [MB] total,** $n = 2^{18}$
No hashing	nt	$n^2\ell_1$	622,720 / 622,624
Simple Hashing	$3.7nt$	$n\ell_2$	476 / 121
Balanced Alloc.	$2.9nt(\ln\ln n)$	$2n\ell_2$	942 / 239
Cuckoo Hashing	$(2(1+\varepsilon)n+s)t$	$sn\ell_1 + 2n\ell_2$	319 / 89

Table 4: Number of OTs and communication for the different hashing-based protocols. The total communication given in the last column is calculated for $\ell_1 = \lambda + \log_2 n$, $\ell_2 = \lambda + 2\log_2 n$, $\kappa = 128$, $\lambda = 40$, $\varepsilon = 0.2$, $s = 4$. The first value in this column is for $t = \sigma = 32$ $\binom{2}{1}$-OTs per element and the second value is for $t = 32/8$ $\binom{N}{1}$-OTs, $N = 2^8$. It is composed of the total number of OTs in the 2nd column times the communication per OT plus the communication from P_1 to P_2 in the 3rd column.

A note on approximations When using a hashing scheme with fixed bin sizes it is possible that the number of items mapped to a certain bin, say by P_1, is larger than the capacity of the bin. (This event happens with probability P_{fail}.) In such a case it is possible for P_1 to ask to use a new hash function. This request reveals some information about P_1's input. Another option is for P_1 to ignore the missed item, and therefore essentially compute an approximation to the intersection. This choice, too, might reveal information about P_1's input, albeit in a more subtle way through multiple invocations of the functionality. Similarly, in the Bloom filter-based protocol, the occurrence of a false positive might leak information. The best solution to this issue is to make sure that the probability of these events happening is negligible, so that it is almost certain that these events will not occur in practice. This is the approach that we take in our comparisons. (Another approach would be to allow the computation of an approximation of the original intersection function, while analyzing the privacy leakage effects of this computation, and deciding whether to tolerate them. The result might be a more liberal choice of parameters which will result in a more efficient implementation of the original protocol.)

7 Experimental Evaluation

In the following we experimentally evaluate the PSI protocols described before. We describe our benchmarking environment in §7.1 and then detail the comparison between the protocols in §7.2. Tab. 5 compares the single-threaded runtimes of all protocols over Gigabit LAN, Tab. 6 compares the communication complexities, and Tab. 7 compares the single-threaded runtimes on different networks. In the tables we highlight the protocol with lowest runtime and communication for each type.

7.1 Benchmarking Environment

We ran our experiments on two Intel Core2Quad desktop PCs (**without** AES-NI extension) with 4 GB RAM, connected via Gigabit LAN. In each experiment, P_1 and P_2 held the same number of input elements n and were not allowed to perform any pre-computation. We set n as in [19], i.e., $n \in \{2^{10}, 2^{12}, 2^{14}, 2^{16}, 2^{18}\}$, but omitted $n = 2^{20}$, since many implementations exceeded the available main memory. We use $\sigma = 32$ as the bit length of the elements.[8] We use a statistical security parameter $\lambda = 40$ and a symmetric security parameter $\kappa \in \{80, 128\}$ (other security parameters are chosen according to Tab. 2). For our set-inclusion protocol we set $\eta = 8$, i.e., use 1-out-of-2^8 OT extensions (cf. §C.3).

In our tables, the asymptotic performance is given for the party with the majority of the workload, and are divided to public-key operations (asym) and symmetric cryptographic operations (sym).

Implementations The implementation of the blind-RSA-based [16] and garbled Bloom-Filter [19] 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 runtime overhead). We implemented a state-of-the-art Yao's garbled circuits protocol (using garbled-row-reduction, point-and-permute, free-XOR, and pipelining, cf [31]) by building on the C++ implementation of [13] and using the fixed-key garbling scheme of [8][9]. 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 version (comparison circuits of size 3σ and depth $\log_2 \sigma$) for σ-bit input values [55].

We implemented FFC (finite field cryptography) and IFC (integer factorization 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 [1] which requires about 3 symmetric cryptographic operations per OT for the asymptotic performance analysis.

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.

7.2 Performance Comparison

We divide the performance comparison into three categories, depending on whether the protocol is based on public-key operations, circuits, or OT. Afterwards, we provide experiments for different networks and give a comparison between the best protocols in each category.

Public-Key-Based PSI For the public-key-based PSI protocols, we observe that the DH-based protocol of [32] outperforms the RSA-based protocol of [16] when using finite field cryptography (FFC). Similarly to [1], we also obtain the somewhat surprising result that for 80-bit security elliptic curve cryptography (ECC) using the Miracl library is slower than FFC using the GMP library. For larger security parameters, however, ECC becomes more efficient and outperforms FFC by a factor of 3 for 128-bit security for the DH-based protocol. (The reason for this phenomenon might be better implementation optimizations in the GMP library.) The advantage of the ECC-based protocol is its communication complexity, which is lowest among all PSI protocols, cf. Tab. 6. We note that a major advantage of these protocols is their simplicity, which makes them comparably easy to implement.

Circuit-Based PSI Here we tested Yao- and GMW-based implementations, as well as an implementation of our optimized vector multiplication-triple-based GMW protocol (§3.2). Following is a summary of the results:

- Both the computation complexity and the communication complexity of the circuit-based PSI protocols are the highest among all protocols that we tested.
- The basic GMW protocol has the highest overall runtime and communication complexity.
- Our vector multiplication triple optimization reduces the runtime and communication of GMW by approximately 40%. For security parameter $\kappa = 80$, this implementation is slightly faster than Yao's protocol, but it is slightly slower for $\kappa = 128$. Communication-wise, the vector multiplication triple GMW is more efficient than Yao's protocol.

[8]For protocols whose complexity depends on σ, elements from a large domain can be hashed to short representations as described in §A.

[9]The security of the fixed-key garbling scheme is somewhat controversial but we included it for performance reasons.

Type	**Symm. Security Parameter** κ	**80-bit**					**128-bit**					**Asymptotic**
	Set Size n	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}	
Public-Key	**DH-based FFC [32]**	**0.4**	**1.6**	**6.2**	**24.7**	**98.8**	4.8	19.1	76.5	306.0	1,224.1	$2n$ asym
	DH-based ECC [32]	0.7	2.8	11.0	44.1	177.5	**1.6**	**6.5**	**26.1**	**104.2**	**416.2**	$2n$ asym
	RSA-based [16]	0.5	2.0	7.9	31.3	124.9	7.7	31.0	124.3	497.2	1,982.1	$2n$ asym
Circuit [30]	**Yao [8,31]**	1.2	5.7	27.7	128.2	-	**1.6**	**6.3**	**28.4**	**129.1**	-	$12n\sigma\log_2 n$ sym
	GMW [1]	1.9	8.6	35.2	161.9	806.5	2.6	12.8	58.9	276.4	1,304.2	$30n\sigma\log_2 n$ sym
	Vector-MT GMW §3.2	**1.2**	**5.1**	**21.2**	**100.3**	**462.7**	1.9	7.8	36.5	168.9	762.4	$18n\sigma\log_2 n$ sym
OT	Garbled Bloom Filter [19]	0.3	0.9	3.9	16.1	71.9	0.6	2.0	8.5	37.1	154.4	$4.32n\kappa$ sym
	Random Garbled Bloom Filter §4.3	0.15	0.5	2.0	8.1	34.3	0.27	1.0	4.1	16.7	67.6	$3.6n\kappa$ sym
	Set Inclusion §5 + Hashing §6	**0.13**	**0.2**	**0.8**	**3.3**	**13.5**	**0.26**	**0.3**	**0.9**	**3.7**	**13.8**	$0.75n\sigma$ sym

Table 5: Runtimes in seconds for PSI protocols with one thread over Gigabit LAN ($\sigma = 32$: bit size of set elements, asym: public-key operations, sym: symmetric cryptographic operations).

Type	**Symm. Security Parameter** κ	**80-bit**					**128-bit**					**Asymptotic**
	Set Size n	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}	
Public-Key	DH-based FFC [32]	0.4	1.5	6.0	24.0	96.0	1.1	4.5	18.0	72.0	288.0	$3n\rho$
	DH-based ECC [32]	**0.1**	**0.2**	**1.0**	**3.8**	**15.0**	**0.1**	**0.4**	**1.5**	**6.0**	**24.0**	$3n\varphi$
	RSA-based [16]	0.3	1.1	4.3	17.3	69.0	0.8	3.1	12.5	50.0	200.0	$2n\rho + 2n\kappa$
Circuit [30]	Yao [8,31]	28.1	135.0	630.0	2,880.0	12,960.0	45.0	216.0	1,008.0	4,608.0	20,736.0	$9n\kappa\sigma\log_2 n$
	GMW [1]	31.3	150.0	700.0	3,200.0	14,400.0	50.0	240.0	1,120.0	5,120.0	23,040.0	$10n\kappa\sigma\log_2 n$
	Vector-MT GMW §3.2	**18.8**	**90.0**	**420.0**	**1,920.0**	**8,640.0**	**30.0**	**144.0**	**672.0**	**3,072.0**	**13,824.0**	$6n\kappa\sigma\log_2 n$
OT	Garbled Bloom Filter [19]	3.4	13.5	54.0	216.0	864.0	7.6	30.2	121.0	483.8	1,935.4	$2.88n\kappa(\kappa+\lambda)$
	Random GBF §4.3	1.1	4.5	18.1	72.6	290.4	2.9	11.6	46.2	184.9	739.7	$1.44n\kappa^2 + n(\lambda + 2\log_2 n)$
	Set Inclusion §5 + Hashing §6	**0.2**	**0.8**	**3.3**	**13.4**	**54.3**	**0.3**	**1.2**	**4.8**	**19.4**	**78.3**	$0.5n\kappa\sigma + 6n(\lambda + 2\log_2 n)$

Table 6: Communication complexity in MB for PSI protocols. ($\sigma = 32$: bit size of set elements, security parameters $\kappa, \lambda, \rho, \varphi$ as defined in §2.1). Numbers are computed from the asymptotic complexity given in the last column.

- The runtime of Yao's protocol hardly increases with the security parameter, since we use AES-128 for both versions. Note, however, that our implementation of Yao's protocol exceeded the main memory when processing 2^{18} elements.
- Our Yao implementation does not use the AES-NI hardware support. Using AES-NI is likely to improve the runtime of the Yao implementation.

We give a more detailed performance comparison for GMW and Yao's protocol in Appendix §D.2.

OT-Based PSI The random garbled Bloom filter protocol of §4.3 improves the original garbled Bloom filter protocol of [19] by more than a factor of two in runtime and by factor of 2-3 in communication.

We also implemented our protocol of §5, where we used Cuckoo hashing with parameters $\varepsilon = 0.2$ and $s = 4$, cf. §6. This protocol had the best runtime, and was about 5 times faster than the random garbled Bloom filter protocol for $\kappa = 128$. In terms of communication, our set inclusion protocol uses less than 20% of the communication of the random garbled Bloom filter protocol for $\kappa = 80$ and less than 10% communication for $\kappa = 128$.

The main difference between the set inclusion protocol and the random garbled Bloom filter protocol is the dependency of the performance on the symmetric security parameter κ. In the random garbled Bloom filter protocol, the number of OTs is independent of the bit-length σ but scales linearly with κ. On the other hand, the number of OTs for the set inclusion protocol is independent of κ but linear in σ. As a result, the runtime of the Bloom filter protocol (but not of the set inclusion protocol) is greatly affected when κ is increased.

Experiments for Different Networks For each protocol type (public-key-based, circuit-based, and OT-based), we benchmark the best performing PSI protocol in different network scenarios: Gigabit LAN, 802.11g WiFi, intra-country WAN, inter-country WAN, and mobile Internet (HSDPA) and depict our results in Tab. 7. We characterize each network scenario by its bandwidth and latency. By latency we mean one-way latency, i.e., the time from source to sink, and we used the same bandwidth for up- and downlink. We simulated these network types using the Linux command `tc` and ran the protocols on $n = 2^{16}$ elements for $\kappa = 128$ and with one thread.

The only protocol that is nearly unaffected by the change in network environment and for which the network has not become the bottleneck is the DH-based ECC protocol. In this protocol computation is the bottleneck which can be improved by using multiple threads.

For the other protocols we observe how the main bottleneck transitions from computation to communication:

Type	**Network** **(Bandwidth (Mbit/s) / Latency (ms))**	**Gigabit LAN** (1,000 / 0.2)	**802.11g WiFi** (54 / 0.2)	**Intra-country WAN** (25 / 10)	**Inter-country WAN** (10 / 50)	**HSDPA** (3.6 / 500)
Public-Key	DH-based ECC [32]	104.2	104.8	107.6	111.8	115.9
Circuit [30]	Yao [8,31]	129.1	779.5	1,735.5	4,631.8	11,658.6
	Vector-MT GMW §3.2	168.9 (11.3)	370.5 (18.1)	770.4 (27.5)	1,936.5 (67.2)	5,310.9 (170.2)
OT	Random Garbled Bloom Filter §4.3	16.6	37.2	70.8	164.9	445.0
	Set Inclusion §5 + Hashing §6	**3.7**	**5.0**	**8.8**	**22.8**	**77.5**

Table 7: Runtimes in seconds for PSI protocols with one thread in different network scenarios for $n = 2^{16}$, $\sigma = 32$, and $\kappa = 128$; online time for Vector-MT GMW in ().

For Yao's protocol this transition happens very early, already when changing from Gigabit LAN to WiFi (factor 6 in runtime).[10] Our vector-MT GMW protocol and our random garbled Bloom filter protocol suffer less drastically from the decreased bandwidth (factor 2.3 in runtime). However, from the WiFi connection on, the performance of all three protocol decreases approximately linear in the bandwidth. Note that, although our vector-MT GMW protocol has only 66% of the communication complexity of Yao's protocol, it is more than two times faster in slower networks. This can be explained by the direction of the communication. In Yao's protocol, the large garbled circuit is sent in one direction, whereas the communication in GMW can be evenly distributed in both directions s.t. it uses both up- and downlink.

For our set inclusion protocol, the network saturation happens when using intra-country WAN. From this point on, the performance also decreases linearly with the bandwidth. Still, this protocol is the fastest of all protocols in all network settings.

Experiments with Multiple Threads Tab. 8 shows the runtimes with four threads. Of special interest is the last column, which shows the ratio between the runtimes with four threads and a single thread for $n = 2^{18}$ elements and security parameter $\kappa = 128$. The DH-based protocol, which is very simple and easily parallelizable, achieves almost the optimal speedup of 4x as computation is the performance bottleneck. The GMW protocol achieves only a speedup of about 2x, possibly due to the gate-by-gate evaluation of the circuit resulting in multiple rounds of communication as the bottleneck. The OT-based protocols achieve a very good speedup of about 3x.

Comparison From the results we observe that OT-based protocols have the lowest runtime on a fast network. The public-key-based protocols require costly public-key operations, which scale very poorly with increasing security parameter, but need less communication than the OT- or circuit-based protocols. The circuit-based protocols have a smaller runtime than the public-key-based protocols using FFC or RSA for $\kappa = 128$, but by far the highest communication complexity.

Our set inclusion protocol achieves both the most efficient runtime and a very low communication overhead. Compared to the second fastest protocol, namely our optimized random garbled Bloom-filter protocol, the set inclusion protocol is at least 5 times faster and uses 10 times less communication (for 128 bit security). Moreover, this protocol has the second best communication overhead, requiring only 3 times the communication of the DH-ECC-based protocol of [32], but running faster in all network environments that we tested.

We stress that the choice of the preferable PSI protocol depends on the application scenario. For instance,

- If communication is the bottleneck and a great amount of computational resource is available, then the DH-based PSI protocol using ECC is the most favorable. That protocol is also the simplest protocol to implement.

- The circuit-based protocols are unique in that they are based on generic secure computation techniques and can therefore be easily modified to compute more complex variants of PSI.

- While our set inclusion protocol performs very efficiently for $\sigma = 32$, it would require twice the runtime for $\sigma = 64$, while the random garbled Bloom filter protocol would have approximately the same runtime (which would still be greater).

Acknowledgements We thank the anonymous reviewers of USENIX Security 2014 for their helpful comments on our paper. This work was supported by the European Union's 7th Framework Program (FP7/2007-2013) under grant agreement n. 609611 (PRACTICE), by the German Federal Ministry of Education and Research (BMBF) within EC SPRIDE, by the Hessian LOEWE excellence initiative within CASED, and by a grant from the Israel Ministry of Science and Technology (grant 3-9094).

[10]The performance advantage of using fixed-key AES garbling instead of SHA-1/SHA-256 already diminished in the WiFi setting.

Type	Symm. Security Parameter κ	80-bit					128-bit					Speedup
	Set Size n	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}	
Public-Key	DH-based FFC [32]	0.1	0.5	1.8	6.7	26.9	1.3	5.2	20.8	80.1	320.1	**3.82x**
Circuit [30]	Vector-MT GMW §3.2	0.8	3.6	15.9	71.3	288.1	1.1	5.6	23.4	96.1	400.9	1.90x
OT	Random Garbled Bloom Filter §4.3	0.08	0.2	0.8	3.2	13.1	0.14	0.5	1.7	6.4	25.9	2.61x
	Set Inclusion §5 + Hashing §6	**0.04**	**0.16**	**0.4**	**1.2**	**4.7**	**0.04**	**0.2**	**0.5**	**1.4**	**4.9**	2.81x

Table 8: Runtimes in seconds for PSI protocols with four threads and $\sigma = 32$; speedup for $n = 2^{18}$ and $\kappa = 128$.

References

[1] 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.

[2] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal. Balanced allocations. *SIAM Journal of Computing*, 29(1):180–200, 1999.

[3] 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.

[4] R. W. Baldwin and W. C. Gramlich. Cryptographic protocol for trustable matchmaking. In *IEEE S&P'85*, pages 92–100. IEEE, 1985.

[5] K. E. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Computing Conference*, volume 32 of *AFIPS Conference Proceedings*, pages 307–314. Thomson Book Company, Washington D.C., 1968.

[6] D. Beaver. Efficient multiparty protocols using circuit randomization. In *Advances in Cryptology – CRYPTO'91*, volume 576 of *LNCS*, pages 420–432. Springer, 1991.

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

[8] 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.

[9] 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.

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

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

[12] 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 (SCN)*, 2013.

[13] S. G. Choi, K.-W. Hwang, J. Katz, T. Malkin, and D. Rubenstein. Secure multi-party computation of Boolean circuits with applications to privacy in online marketplaces. In *Cryptographers' Track at the RSA Conference (CT-RSA'12)*, volume 7178 of *LNCS*, pages 416–432. Springer, 2012.

[14] S. G. Choi, J. Katz, R. Kumaresan, and H.-S. Zhou. On the security of the "free-xor" technique. In *Theory of Cryptography Conference (TCC'12)*, volume 7194 of *LNCS*, pages 39–53. Springer, 2012.

[15] 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.

[16] 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.

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

[18] D. Dachman-Soled, T. Malkin, M. Raykova, and M. Yung. Efficient robust private set intersection. In *Applied Cryptography and Network Security (ACNS'09)*, volume 5536 of *LNCS*, pages 125–142. Springer, 2009.

[19] 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.

[20] 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.

[21] 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.

[22] M. J. Freedman, C. Hazay, K. Nissim, and B. Pinkas. Efficient set-intersection with simulation-based security. In *Journal of Cryptology*, 2013. To appear.

[23] 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.

[24] 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.

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

[26] 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.

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

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

[29] C. Hazay and K. Nissim. Efficient set operations in the presence of malicious adversaries. In *Public Key Cryptography (PKC'10)*, volume 6056 of *LNCS*, pages 312–331. Springer, 2010.

[30] 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.

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

[32] 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.

[33] 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.

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

[35] 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)*, LNCS. Springer, 2014.

[36] Adam Kirsch, Michael Mitzenmacher, and Udi Wieder. More robust hashing: Cuckoo hashing with a stash. *SIAM J. Comput.*, 39(4):1543–1561, 2009.

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

[38] 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.

[39] 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.

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

[41] 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.

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

[43] P. Mohassel and S. S. Sadeghian. How to hide circuits in MPC an efficient framework for private function evaluation. In *Advances in Cryptology – EUROCRYPT'13*, volume 7881 of *LNCS*, pages 557–574. Springer, 2013.

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

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

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

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

[52] 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.

[53] B. Pinkas, T. Schneider, and M. Zohner. Faster private set intersection based on OT extension. In *23rd USENIX Security Symposium (USENIX Security'14)*. USENIX, 2014.

[54] 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.

[55] 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.

[56] A. Waksman. A permutation network. *Journal of the ACM*, 15(1):159–163, 1968.

A 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.

In some settings, inputs come from a small and densely populated domain, such as national identity numbers (e.g., social security numbers in the US, which can be represented by 30 bits) or credit card numbers (54 bits). In our experiments in §7 we use 32-bit representations matching such a densely populated domain.

In other settings, the input representation is sparse, for instance when a person is identified by an ascii string containing his or her name, or when the number of inputs is very small, say 100, and therefore the original input representation is obviously sparse. (This is especially true in protocols where inputs are randomly mapped into bins, and only a small number of inputs resides in each bin, as in the PSI protocols we describe in §6.)

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 [44], p. 45). We ran extensive experiments that empirically verified this analysis.

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

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

For example, for an input of $n = 90$ items and a collision probability of 2^{-20}, the representation can be $d = 32$ bits long. If we need the collision probability to be 2^{-40} then we should set $d = 52$. Similarly, for $n = 10^6$, and error probability $p = 2^{-20}$ the representation can be $d = 59$ bits long, and can fit in a single long word. For an error probability of 2^{-40} we must use $d = 79$.

B Random Oracle Model

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 [9]. 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 proved 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 protocol that we describe that is in the standard model is the protocol based on oblivious polynomial evaluation, 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.

C Oblivious Transfer Extension

In this section we summarize OT extension and review the recent development in OT extension protocols. We refer to a random oracle as RO.

C.1 1-out-of-2 OT Extension

An initial 1-out-of-2 OT extension protocol was suggested in [7]. A more practical OT extension protocol was designed in [33], extending OT_κ^κ (κ OTs of κ bits) to OT_ℓ^m (m OTs of ℓ bits) in the following way.

Assume that S and R wish to perform OT_ℓ^m, where S holds $(x_0^i, x_1^i) \in \{0,1\}^{2\ell}$, for $1 \le i \le m$, and R uses $b \in \{0,1\}^m$ as choice vector. S and R first engage in a OT_κ^κ in reverse roles, where R acts as sender with seeds $(k_0^j, k_1^j) \in_R \{0,1\}^{2\kappa}$ and S acts as receiver with a choice vector $s \in_R \{0,1\}^\kappa$ and obtains $k_{s[j]}^j$, for $1 \le j \le \kappa$. R then chooses a random $m \times \kappa$ bit-matrix T, where $t^j \in \{0,1\}^m$ denotes the j-th column of T and computes $v_0^j = t^j \oplus G(k_0^j)$ and $v_1^j = t^j \oplus G(k_1^j) \oplus b$, where $G : \{0,1\}^\kappa \mapsto \{0,1\}^m$ is a PRG, and sends (v_0^j, v_1^j) to S. S generates a $m \times \kappa$ bit-matrix Q as $q^j = v_{s[j]}^j \oplus G(k_{s[j]}^j)$, computes $y_0^i = x_0^i \oplus H(q_i)$ and $y_1^i = x_1^i \oplus H(q_i \oplus s)$, where q_i denotes the i-th row of Q and $H : \{0,1\}^\kappa \mapsto \{0,1\}^\ell$ is a CRF, and sends (y_0^i, y_1^i) to R. Finally, R obtains $x_{b[i]}^i = y_{b[i]}^i \oplus H(t_i)$.

Overall, when using OT extension, the sender in OT_ℓ^m 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 $2m\kappa$ bits. (In addition, there is a preprocessing cost of OT_κ^κ, which is negligible compared to the main protocol if $\kappa \ll m$.)

C.2 Random OT Extension

In [1, 38], a more efficient variant of the OT extension protocol was outlined. It reduces the message that R has to send by half from $2m\kappa$ to $m\kappa$ bits. This is done by having R generate the T matrix as $t^j = G(k_0^j)$ and only sending a single $v^j = t^j \oplus G(k_1^j)$. S can then compute his $m \times \kappa$ bit-matrix Q as $q^j = s_j v^j \oplus G(k_0^j)$. The rest of the OT extension protocol remains unchanged. In addition, several works [1, 49] use a special purpose OT functionality, called random OT, where (x_0^i, x_1^i) are chosen uniformly and randomly during the OT and are output to S. To obtain a random OT extension protocol, they propose to leave out the last message, containing the (y_0^i, y_1^i). Instead, S outputs $(x_0^i = H(q_i), x_1^i = H(q_i \oplus c))$ and R outputs $x_{b[i]}^i = H(t_i)$. This random OT extension protocol reduces the bits that S 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.

Experiments in [1] demonstrate that the performance bottleneck of random OT extension is essentially the network bandwidth, even over a fast Gigabit Ethernet, whereas the computation workload scales well with an increasing number of threads. Using four threads, their cache optimized implementation achieves a throughput of almost 4 million random OTs per second for 80-bit security. In comparison, the public-key-based OT protocol of [47] implemented over a finite field achieves only a throughput of 3,000 random OTs per second.

C.3 1-out-of-N OT Extension

In [38], an efficient 1-out-of-N OT extension protocol was introduced which allows to transfer short messages with sublinear communication in the security parameter. The protocol builds on the original OT extension protocol of [33] and encodes the choices of R using a Walsh-Hadamard code $C^N_{WH} = c_0,...,c_{N-1}$, which encodes $\lceil \log_2 N \rceil$-bit words with N-bit codewords that have at least $N/2$ Hamming distance from each other. More detailed, in the i-th 1-out-of-N OT, S inputs $x^i_0,...,x^i_{N-1}$ and R inputs $b_i \in [0...N-1]$. The parties perform κ' base-OTs such that S holds $s \in_R \{0,1\}^{\kappa'}$ and $k^j_{s[j]}$ and R holds k^j_0 and k^j_1 (κ' is a security parameter, see below). R then computes $m \times \kappa'$ 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 $m \times \kappa'$ 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 1-out-of-N 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 κ as in the original 1-out-of-2 OT extension protocol of [33], the parties have to increase the number of base-OTs to $\kappa' \approx 2\kappa$ (cf. [38]). The reason for the increase in base-OTs is that the Hamming distance between the codewords has to be at least κ. Using the Walsh-Hadamard codes and $N = 2^\eta$, this means that for η with $2^{\eta-1} < \kappa$, we have to repeat the codewords in order to achieve the required Hamming distance. Additionally, to minimize the transmitted data for our PSI protocol, we depict the communication complexity for different η in Tab. 9. From this table we observe that the communication is minimal for $\eta = 8$ which we choose for our experiments.

D Generic Secure Computation

In this section we reflect on the sort-compare-shuffle circuit for PSI (§D.1) and discuss the comparison between Yao's protocol and GMW (§D.2).

D.1 Sort-Compare-Shuffle Circuit for PSI

The straightforward way of using a circuit for PSI is to compare each input item of P_1 to each input of P_2. However, this approach results in a circuit of size $O(n^2)$. A more efficient approach is the *sort-compare-shuffle (SCS)* circuit described in [30] that has a size of $O(n \log n)$. (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 in order to hide any information that could be obtained from the resulting order.

Sort To sort both sets into a single sorted list, both parties locally pre-sort their sets and merge them using a bitonic merging circuit [5]. In contrast to a sorting network, a bitonic merging circuit takes advantage of the fact that the inputs are already sorted and allows the parties to obtain a globally sorted list of $2n$ input elements using $n \log_2(2n)$ sorter circuits. A sorter circuit takes as input two elements x and y, swapping them if $x > y$ and preserving the order if $x \leq y$. Each sort gate consists of a comparison and a conditional swap sub-circuit.

Compare All elements in the sorted list are then compared to their neighbors to determine if a duplicate exists. Since each party's input consists of different values, duplicates only occur for items in the intersection of the two inputs. A duplicate item is passed on, whereas if no duplicate is found then the item is replaced by a special bottom symbol.

Shuffle Finally, all elements are shuffled using a Waksman permutation network [56]. An n input Waksman circuit consists of $n \log_2(n) - n + 1$ conditional swap gates, which either forward their two input elements or swap their order depending on the required randomly chosen output permutation.

Overall The overall size of the SCS circuit for inputs words of length σ is $\sigma(3n \log_2 n + 4n) - n$ 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, where $n = \frac{n_1+n_2}{2}$. It is important to note that approximately 2/3 of the AND gates in the circuit are due to multiplexers. We show in §3.2 how these gates can be optimized in GMW.

D.2 A comparison between GMW and Yao

The complexity of a secure computation using GMW is measured by the circuit's *size*, i.e., the number of AND

η	1	2	3	4	5	6	7	8	9
Comm.	$2.0n\kappa\sigma$	$1.0n\kappa\sigma$	$0.67n\kappa\sigma$	$0.50n\kappa\sigma$	$0.40n\kappa\sigma$	$0.33n\kappa\sigma$	$0.29n\kappa\sigma$	$0.25n\kappa\sigma$	$0.44n\kappa\sigma$

Table 9: Communication complexity of our PSI protocol for 1-out-of-2^η OT to achieve at least κ-bit security.

gates, and its *depth*, i.e., the highest number of AND gates on a path from any input to any output. Overall, the GMW protocol requires less communication than Yao's protocol: 2κ bits per AND gate vs. 3κ bits in Yao's protocol using garbled row reduction [52]. The computational workload in GMW is dominated by 3 evaluations of a CRF and 3 evaluations of a PRG for each party per AND gate. These OTs 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 while the evaluator performs one symmetric key operation, cf. [8]. To pre-compute the garbled circuit, the circuit garbler has to know the specific function and the size of the inputs in advance.

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 depth of the circuit. In [55] 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 party that evaluates the garbled circuit has to do one symmetric cryptographic operation per gate in the online phase.

Consequently, GMW is suited for use in the preprocessing model or where the bandwidth is limited but incurs a higher computational overhead than Yao's garbled circuits and suffers drastically from higher latency.

For our experiments in §7, 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. Consequently, the size of the SCS circuit was increased from $3n\sigma\log_2(n)$ to $5n\sigma\log_2(n)$, but its depth was decreased from $\sigma\log_2(n)$ to $\log_2(n)\log_2(\sigma)$.

Finally, note that we used fixed-key AES garbling [8] in Yao's protocol, which is somewhat controversial among researchers. The fixed-key AES garbling scheme encrypts a non-linear gate with input keys A and B and output key O as $O = \mathrm{AES}_C(K) \oplus K \oplus T$, where $K = 2A \oplus 4B \oplus T$, T is a tweak for the block-cipher and C is a constant key. Yao's protocol with the free-XOR optimization of [39] requires a circular 2-correlation robustness assumption [14], which is a variant of correlation robustness [33]. However, correlation robustness is also required in OT extension, where we instantiate the CRF with SHA-1 for $\kappa = 80$ and SHA-256 for $\kappa = 128$ instead of AES-128. Thus, an open question remains whether the fixed-key garbling scheme is sufficient for correlation robustness and whether we can use the same construction as CRF in OT extension.

E An Analysis of the Usage of the Different Hashing Schemes

Simple Hashing As described in §6.1, a formula in [54] shows that when throwing $n = cb\ln b$ balls into b bins using simple hashing yields $max_b = (d_c - 1 + \alpha)\ln b$ w.h.p. Here, d_c solves $f(x) = 1 + x(\ln c - \ln x + 1) - c = 0$ and $\alpha > 1$ is a parameter affecting P_{fail}. It is also shown in [54] that P_{fail} decreases exponentially with α.

[54] do not provide an exact formula for the dependency of P_{fail} on α. We ran experiments showing that this error probability decreases sharply even for a small value of $\alpha = 1 + \varepsilon$, such as $\alpha = 1.1, 1.2, \ldots$. An empirical calculation of the exact probability is hard since it needs to identify failure events that happen with very small probabilities (e.g., 2^{-40}). Therefore in the rest of this analysis we set $\alpha = 2$ as a safe bet, which yields $max_b = (d_c + 1)\ln b$. The number of OTs is $b \cdot max_b \cdot \sigma = b \cdot (d_c + 1) \cdot \ln b \cdot \sigma = \frac{d_c+1}{c} \cdot n \cdot \sigma$. Our overall goal is to find a c that provides reasonable P_{fail} and yields a low number of OTs. Note that the number of OTs is inversely correlated with c , while P_{fail} is directly correlated with c. We compute the overall number of OTs by fixing c and solving d_c for different values of c. Since related work only approximates P_{fail}, we empirically evaluate P_{fail} for each c by throwing n balls into $n = c \cdot b\ln b$ bins for 10^5 repetitions and using $n = 2^{18}$. We then compute max_b given d_c, and set P_{fail} as the ratio between the throws where the number of balls in a bin exceeded max_b and the number of repetitions. The results are depicted in Tab. 10. As we can observe, P_{fail} increases with c, while the overhead decreases with c. The first value for which we obtain $P_{\text{fail}} = 0$ is $c = 1$, which requires 3.7 OTs per bit and element. If a higher P_{fail} can be tolerated, then larger values of c can be tolerated. (This uncertainty about the value of P_{fail} was one of the reasons that led us not to use simple hashing in our experiments.) Although the number of OTs when using simple hashing is increased by a factor 3.7 compared to using no hashing, the amount of communication that P_2 sends back decreases from $n^2\lambda$ to $n\lambda$ as no masks are sent for dummy elements (cf. Tab. 4).

Balanced Allocations The maximum bin size when throwing n elements into $b = n$ bins was estimated in [2]

c	**0.25**	**0.5**	**1**	**2**	**4**	**8**
d_c	2.2	2.8	3.7	5.3	8.1	13.3
#OTs	$8.8nt$	$5.6nt$	$3.7nt$	$2.7nt$	$2.0nt$	$1.7nt$
P_{fail}	0	0	0	0.4%	2.1%	100%

Table 10: Number of OTs and empirical error probabilities for different c using simple hashing (10^5 repetitions).

as $max_b = \frac{\ln\ln b}{\ln 2}(1 + o(1))$. An empirical analysis of max_b demonstrated that replacing $o(1)$ with 1 results in $P_{\text{fail}} = 0$ for all set sizes we use in our experiments. We thereby obtain $max_b = 2\frac{\ln\ln b}{\ln 2} \approx 2.9\ln\ln b$. Note that since max_b depends on n, the number of OTs increases super-linearly in n. Since two hash functions are used, the number of elements that P_1 has to map into bins doubles compared to simple hashing. Thereby, the number of bits that are sent back also doubles to $2n\lambda$ bits. We depict the resulting overhead in Tab. 4.

Cuckoo Hashing To achieve a lower failure probability P_{fail} in Cuckoo hashing, we choose the number of bins $b = 2(1+\varepsilon)n$ and introduce a stash of size $s \leq \ln n$. We thereby achieve an overhead of $2(1+\varepsilon)$ OTs as well as st additional OTs for the stash. Similar to balanced allocations, P_1 has to send $2n\lambda$ as well as $sn\lambda$ bits for the stash. For our experiments in §7 we use the parameters suggested in [36], $\varepsilon = 0.2$ and $s = 4$, which were shown experimentally to require no rehashing even for sets as small as $n = 1{,}000$. Overall, we obtain a rehash probability of n^{-4} which decreases with the set size in our experiments and is at most 2^{-40} for sets of size 2^{10}.