An Efficient Toolkit for Computing Private Set Operations

Alex Davidson and Carlos Cid

Royal Holloway, University of London
Egham, TW20 0EX, UK
{alex.davidson.2014,carlos.cid}@rhul.ac.uk

Abstract. Private set operation (PSO) protocols provide a natural way of securely performing operations on data sets, such that crucial details of the input sets are not revealed. Such protocols have an ever-increasing number of practical applications, particularly when implementing privacy-preserving data mining schemes. Protocols for computing private set operations have been prevalent in multi-party computation literature over the past decade, and in the case of private set intersection (PSI), have become practically feasible to run in real applications. In contrast, other set operations such as union have received less attention from the research community, and the few existing designs are often limited in their feasibility. In this work we aim to fill this gap, and present a new technique using Bloom filter data structures and additive homomorphic encryption to develop the first private set union protocol with both linear computation and communication complexities. Moreover, we show how to adapt this protocol to give novel ways of computing PSI and private set intersection/union cardinality with only minor changes to the protocol computation. Our work resembles therefore a toolkit for scalable private set computation with linear complexities, and we provide a thorough experimental analysis that shows that the online phase of our designs is practical up to large set sizes.

Keywords: Private set operations, Bloom filters, additively homomorphic encryption, secure computation, data mining.

1 Introduction

The emergence of Big Data has resulted in an increasing need for analytical data mining techniques allowing entities to gain information from the large data sets that they own. Even more so can be learnt by combining internal data sets with private data from external entities. However, in order to safeguard incentives for combining data, participants require privacy-preserving measures to be put into place to stop secret information from being leaked to competitors or untrusted parties. Private set

operation (PSO) protocols provide a natural way of securely performing operations on these combined data sets, such that only the output of the set operation is revealed. Numerous works in research in genetic data computations and information sharing have highlighted the importance of efficient private set operation computation [15,6].

Previous work. Research into private set intersection (PSI) protocols has resulted in several designs that are practically feasible for real-world use. While pioneering work such as [12,20] brought the problem into the attention of the cryptographic research community, more recent research (e.g.[23,10,22,19,24]) has shown that certain techniques and data structures, such as oblivious transfer (OT) and Bloom filters, can be used to design protocols that scale and perform well even for very large data sets. These constructions play a crucial role in developing large-scale data mining applications where data privacy and efficient computation are both important. For example, computations over genetic data, as shown in [15], may require comparing records from databases with millions of elements.

In spite of recent progress in the design of PSI protocols, research into performing other set operations with similar security guarantees has not been as comprehensive. Current designs for computing private set union (PSU) include [13,4,25], while generic designs for computing multiple set operations are given in [1,18,14]. With a much smaller base of research, computational complexities for computing PSU remain super-linear in the size of the sets involved (e.g. $O(n \log \log n)$). Moreover, there has been relatively little work done in computing set cardinality (PSI/PSU-CA) operations where only the size of the output set is revealed. Dedicated techniques for computing these operations are given in [5,11,9] though designs are also given in the generic constructions of [18,1].

Consequently, implementations of PSOs such as union are unlikely to scale well as set sizes increase up to the dimensions being required for current applications. Furthermore, complex data mining can require a conjugation of several set operations. Without a way for computing scalable privacy-preserving protocols for *all* of the main operations it is not possible to carry out these procedures in an efficient manner. It is important that privacy-preserving methods for real-world problems remain almost as efficient as tools with non-cryptographic guarantees in order to motivate the uptake of these new solutions. Furthermore, it would be beneficial to have an efficient 'toolkit' for performing multiple PSO protocols, so that developers would no longer need to implement completely different designs for each set operation to achieve optimal efficiency.

Our contributions. We first address the void in efficient PSU protocols by developing a new two-party construction, secure against semi-honest adversaries where only one participant (the client) learns the output. Our design makes use of similar design structures to previous works such as [10,13,17,9]: the efficient data structure provided by Bloom filter alongside partially homomorphic encryption to allow oblivious computation. However our PSU protocol is the first to demonstrate both linear computation and communication complexities, and as a result it is immediately more scalable than previous designs. Table 1 provides an asymptotic comparison of our design with the previous PSU work; we detail our protocol design in Section 3.

	Communication	Computation	Multi-party?
Kissner et al. [18]	$O(N^2 n \log \lvert E \rvert)$	$O(n^2)$	Y
Brickell et al. [4]	$O((n+m) \log \lvert E \rvert)$	$O((n+m) \log \lvert E \rvert)$	N
Frikken. [13]	$O(n)$	$O(n \log \log n)$	N
Blanton et al. [1]	$O(N^3 n \log(Nn))$	$O(N^3 n \log(Nn))$	Y
Our work	$\boldsymbol{O(n)}$	$\boldsymbol{O(n)}$	N

Table 1. Complexities for previous PSU protocols.

Our protocol for computing PSU is very simple, and we show that minor changes in the computation done by the server (non-output party) can be leveraged to convert the protocol into a PSI or PSI/PSU-CA exchange. These constructions also have linear complexities putting them in line with current practical solutions in the wider research area. We give these adaptations in Section 4. Consequently, our work can be viewed as a toolkit for performing the *main* set operations that are required by conventional applications. The simplicity of the design means that developers only need to consider implementations for an additively homomorphic encryption scheme and a Bloom filter. We focus here on semi-honest adversaries only, but we could ensure security in the malicious setting using a trusted third party, based on similar methods to those of [8,17]; full details are provided in the extended version of this paper.

In Section 5, we demonstrate the concrete practicality of our design by performing a rigorous experimental analysis using an implementation written in `Go`. We show that our designs run with comparable communication overheads and runtimes relative to state-of-the-art PSI protocols. Observe that our construction provides a much more generic functionality than dedicated PSI protocols and so we balance out an expensive offline phase, slightly slower running times and high communication overheads with the ability to perform much more dynamic computations.

The main bottleneck of our design is provided by the encryption scheme that we use (Paillier's [21]). Our protocols are however agnostic to the encryption scheme used and so any improvements that can be made in this phase will directly translate to improvements in our PSO design. The simplicity of our design is highlighted in the small number of lines of code that we require for our implementation; we plan to make our code open-source in the near future.

2 Preliminaries and Notation

2.1 Notation

We will primarily consider two-party protocols with players P_1 and P_2 who own sets S_1 and S_2, respectively. We may commonly refer to P_1 as the 'client' and P_2 as the 'server' in the interaction. The client typically receives output from the computation while the server does not.[1]

We commonly denote the cardinalities of the sets by $n = |S_1|$ and $m = |S_2|$. We denote the domain of elements by E, the security parameter by λ and, when discussing multi-party protocols, the number of players by N, where $c < N$ denotes the number of corrupted players in a protocol instantiation. When discussing the use of homomorphic operations over ciphertexts, we use $+_H$ when invoking additions on underlying plaintext data. Section 2.3 fully describes our notation regarding partially homomorphic encryption (PHE) schemes. For a key pair (pk, sk) for a public-key encryption scheme, we denote generic encryption and decryption by E_{pk} and D_{sk}, respectively.

2.2 Bloom filters

Bloom filters were first introduced by Bloom in [2] as a lightweight data structure that allows for the representation of data sets and checking of inclusion using only hash function evaluations. A Bloom filter is initially represented by a string of B bits that are all initialised to 0. There are k hash functions $h_l : \{0,1\}^\lambda \mapsto \{1, \ldots, B\}$ for $l \in \{1, \ldots, k\}$ published alongside the Bloom filter. We then represent set elements $x \in X$ in the Bloom filter by evaluating $h_1(x), \ldots, h_k(x)$ and changing each index that these hash functions point to from 0 to 1. If a value has already been changed to 1, it is left alone. Any party can use the hash functions to check if an element is stored in the Bloom filter.

[1] It is however possible to enforce bilateral output by running the protocol twice and swapping the roles.

Definition 1. (Represented elements) *We say that an element, e, is* represented *in the Bloom filter, $\boldsymbol{BF}$, if we have that*

$$\boldsymbol{BF}[h_i(e)] = 1, \ \ \forall i \in \{1, \ldots, k\}$$

where $\{h_1, \ldots, h_k\}$ are the hash functions used in conjunction with $\boldsymbol{BF}$. We say that the set S is represented by $\boldsymbol{BF}$ *if every element $e \in S$ is represented in $\boldsymbol{BF}$.*

Optimal Bloom filter parameters. One constraint on Bloom filters is that they can lead to false positives when checking membership: an element $y \notin X$ may appear to be in X after checking all hash outputs if all the values have been set to 1. However, as shown in [10], if $p = 1 - (1 - 1/B)^{kn}$ is the probability that any bit in the Bloom filter is set to 1, then the upper bound of the false-positive probability is given by

$$\epsilon = p^k \times \left(1 + O\left(\frac{k}{p}\sqrt{\frac{ln\, B - k\, ln\, p}{B}}\right)\right),$$

which is negligible in k, the number of hash functions. In practice one will select the values of k and B when building a Bloom filter for a set of size n such that ϵ is capped at a specific low value (e.g. 2^{-50}). In [10] it is claimed that performance optimality is achieved when

$$k = \frac{B}{n} \ln 2, \quad \text{and} \quad B \geq n \log_2 e \cdot \log_2 1/\epsilon, \tag{1}$$

where e is the base of the natural logarithm. By minimising B we get the optimal value of k to be

$$k = \log_2 1/\epsilon. \tag{2}$$

We will assume (as in [10]) that these parameters are always chosen in this way. The proofs that these values are optimal can be found in [3].

Inverting and encrypting Bloom filters. In this work, we use a non-standard representation of a Bloom filter by inverting each entry prior to encryption. Also, rather than treating each entry as a bit, we use 0 and 1 elements from the plaintext space of a given encryption scheme.

Definition 2. (Encrypted Bloom filters) *Let $\boldsymbol{BF}_i$ be the Bloom filter computed for the set S_i (using hash functions $h_1, \ldots, h_k$), with B entries. The corresponding* encrypted Bloom filter *is denoted by $\boldsymbol{EBF}_i$ and has B entries where each entry is defined in the following way:*

$$\boldsymbol{EBF}_i[b] = E_{pk}(\boldsymbol{BF}_i[b])$$

for some public key pk. In the following we define $\boldsymbol{EBF}_i = \{C[1], \ldots, C[B]\}$ *and for* $y_j \in S_i$*, then* $\boldsymbol{EBF}_i[h_u(y_j)] = C_u^{(j)}$ *for* $u = \{1, \ldots, k\}$ *and where* h_u *is the* u^{th} *hash function used in computing the original Bloom filter. In this case* $C_u^{(j)}$ *is the ciphertext obtained by querying the* u^{th} *hash function for* $\boldsymbol{EBF}_i$ *on* y_j*.*

Definition 3. (Inverted Bloom filters) *Let* $\boldsymbol{BF}_i$ *be a Bloom filter. We define the corresponding* inverted Bloom filter *to be* $\boldsymbol{IBF}_i$ *where*

$$\boldsymbol{IBF}_i[j] = \begin{cases} 1 \text{ if } \boldsymbol{BF}_i[j] = 0 \\ 0 \text{ otherwise.} \end{cases}$$

When referring to an encrypted, inverted Bloom filter we will write $\mathbf{EIBF}_i$.

2.3 Partially homomorphic encryption

Let (pk, sk) be a key pair for a public-key encryption scheme, and let $\tilde{x} = E_{pk}(x)$ and $\tilde{y} = E_{pk}(y)$. We say that the encryption scheme is *additively homomorphic* if we have the following properties:

- There is a homomorphic addition operation, $+_H$, over $\tilde{x}$ and $\tilde{y}$ such that $D_{sk}(\tilde{x} +_H \tilde{y}) = x + y$.
- It is possible to compute $\tilde{x} \cdot r$, where r is a scalar and $D_{sk}(\tilde{x} \cdot r) = x \cdot r$ (scalar multiplication)

Paillier's encryption scheme [21] is an example of a semantically secure public key encryption scheme that is additively homomorphic on operations over the ciphertexts.

We further define a final property of such a scheme, known as ReRand, which allows a party with knowledge of the public key to re-randomise ciphertexts. We use this property later in our protocols.

- $\mathsf{ReRand}(\mathsf{pk}, c)$: an algorithm that takes the public key pk and a ciphertext c encrypted under pk as input. The algorithm encrypts the value 0 by computing $E_{\mathsf{pk}}(0) = c_0$ and then outputs $\tilde{c} = c +_H c_0$.

Notice that ReRand does not change the value of the underlying plaintext.

2.4 Security model

Definition 4. (Indistinguishability of distributions) *Let* $X = \{\mathcal{X}_\lambda\}_{\lambda \in S}$ *and* $Y = \{\mathcal{Y}_\lambda\}_{\lambda \in S}$ *be probability ensembles indexed by* S*. We say that these ensembles are computationally indistinguishable for all probabilistic*

polynomial time (PPT) algorithms, $\{\mathcal{D}_n\}_{n\in\mathbb{N}}$*, if there exists a negligible function* $\mathsf{negl} : \mathbb{N} \mapsto [0,1]$ *where*

$$|\Pr[\mathcal{D}_n(\lambda, \mathcal{X}) = 1] - \Pr[\mathcal{D}_n(\lambda, \mathcal{Y}) = 1]| < \mathsf{negl}(n).$$

In this case we write $X \simeq Y$*.*

Let π be a protocol that represents a polynomial-time functionality f. Let S_i be the input set of a participant P_i for $i \in \{1, 2\}$ and let aux_i be a set of auxiliary information that P_i holds. Define the view of the protocol for P_i to be $\mathsf{view}_i^{\pi}(S_1, S_2) = (\mathsf{Inp}_i, r_i, \mathcal{T}_i, \pi(S_1, S_2)_i)$ where $\mathsf{Inp}_i = (S_i, aux_i)$ is the combined input of P_i to π, r_i represents internal coin tosses, $\mathcal{T}_i$ are the messages viewed by P_i and $\pi(S_1, S_2)_i$ is the output for P_i. We use the following to define security against semi-honest adversaries.

Definition 5. (Semi-honest security) *Protocol* π *securely computes the functionality* f *in the presence of static semi-honest adversaries if there exists polynomial-time simulators* Sim_1, Sim_2 *where*

$$\{Sim_1(\mathsf{Inp}_1, f(S_1, S_2))\} \simeq \{\mathsf{view}_1^{\pi}(S_1, S_2)\},$$

$$\{Sim_2(\mathsf{Inp}_2, f(S_1, S_2))\} \simeq \{\mathsf{view}_2^{\pi}(S_1, S_2)\}.$$

Intuitively, this states that each party's view of the protocol can be simulated using only the input they hold and the output that they receive from the protocol. Therefore a corrupted party is unable to learn any extra information that cannot be derived from the input and output explicitly.

3 PSU protocol

In this section, we detail the construction of our PSU protocol using encrypted Bloom filters where encryption is performed via an IND-CPA secure AHE scheme. The homomorphic aspect allows the 'server' to evaluate functions over the ciphertexts without learning anything. Variations of this technique have been used previously for oblivious polynomial evaluation, for example [13,12].

3.1 Overview

Both parties receive the k hash functions which are chosen to evaluate the Bloom filter for elements in the corresponding sets. The elements $y_j \in S_2$ are assumed to be represented by elements in $\mathbb{Z}_N$.

Additionally, we assume that P_1 has a public key pk which is also made available to P_2. P_1 also has a secret key sk that they use for decryption. Both parties also have access to sources of internal randomness that they can use for computing any tasks that require to sample random values. The following is a description of how the protocol operates; we provide a diagrammatic overview of our PSU design in Figure 1.

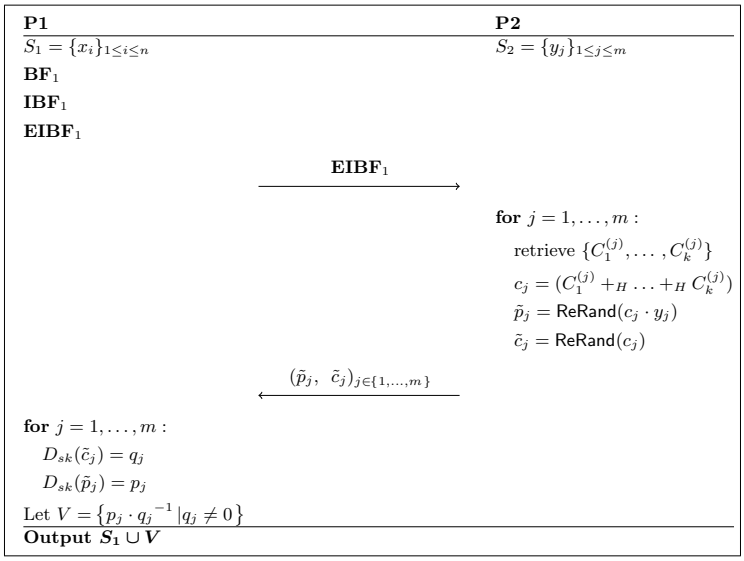

Fig. 1. An overview of our $\pi_{\cup}^{\mathrm{EBF}}$ protocol that uses encrypted, inverted Bloom filters

Protocol steps. *Inputs* - P_1: [(pk, sk), S_1, $|S_2|$], P_2: [pk, S_2, $|S_1|$]

1. P_1 calculates $\mathbf{BF}_1$ representing S_1 using the set of hash functions $h_1, \ldots, h_k$. They then invert each entry in $\mathbf{BF}_1$ to retrieve $\mathbf{IBF}_1$.
2. P_1 separately encrypts each element $\mathbf{IBF}_1[l]$ of the inverted Bloom filter, where $1 \le l \le B$, using pk. P_1 now possesses $\mathbf{EIBF}_1$, denote $\mathbf{EIBF}_1[l] = C[l]$. They send $\mathbf{EIBF}_1$ to P_2.
3. P_2 evaluates each element $y_j \in S_2$ using the k hash functions and retrieves $\{C_1^{(j)}, \ldots, C_k^{(j)}\}$ where $C_d^{(j)} = \mathbf{EIBF}_1[h_d(y_j)]$ for $j \in \{1, \ldots, m\}$.
4. P_2 computes $c_j = (C_1^{(j)} +_H \ldots +_H C_k^{(j)})$ and sends $(\tilde{p}_j, \tilde{c}_j) = (\mathsf{ReRand}(c_j \cdot y_j), \mathsf{ReRand}(c_j))$ to P_1 (in some randomly permuted order).
5. First P_1 checks the value of $\tilde{c}_j$ by computing $D_{sk}(\tilde{c}_j) = q_j$. If $q_j = 0$ then $D_{sk}(\tilde{p}_j) = 0$ so nothing can be learnt. Else $D_{sk}(\tilde{p}_j) = q_j \cdot y_j = p_j$.
6. P_1 computes ${q_j}^{-1}$ for $q_j \neq 0$ and then calculates $p_j \cdot {q_j}^{-1} = y_j$.
7. P_1 adds all y_j to the set V where $q_j \neq 0$ and outputs the set $S_1 \cup V$.

Remark 1. We adopt the notation $c_j \cdot y_j$ for scalar multiplication between a ciphertext c_j and a scalar y_j. This preserves the generality of the protocol relative to the AHE scheme used. However for Paillier encryption this multiplication would usually be invoked via an exponentiation, i.e. $c_j^{y_j}$.

Remark 2. It should be noted that the protocol leaks the size of the intersection cardinality between the players P_1 and P_2. This is similar to the previous PSU designs of [13,14,1], and likewise we don't consider this as a drawback in our design.

Remark 3. Randomisation of the ciphertexts by P_2 prevents y_j being inferred directly from the ciphertext value. Engaging additive homomorphisms is necessarily deterministic by nature, and thus P_1 could use knowledge of $\mathbf{BF}_1$ and $h_1, \ldots, h_k$ to learn values in the intersection.

3.2 Protocol correctness

Since the Bloom filter is inverted before encryption then for any $y_j \in S_1 \cap S_2$ we have that $D_{sk}(c_j) = 0$, therefore any message $c_j \cdot y_j$ that is received for such a y_j also decrypts to 0 and so cannot be learnt. For a value $y_j \notin S_1$ then we have that $D_{sk}(c_j) = 1 < z_j < k$, then decrypting $c_j \cdot y_j$ reveals $z_j \cdot y_j$ and P_1 can add all values y_j to V by multiplying by z_j^{-1}. Since V contains all values $(y_j \in S_2) \wedge (y_j \notin S_1)$ then $S_1 \cup V = S_1 \cup S_2$. Correctness is not perfect due to the possibility of false positives though we can make this negligible in k as discussed in Section 2.2.

3.3 Protocol security

We show that this protocol is secure with respect to the ideal functionality of a PSU computation defined by $\mathcal{F}_\cup$ and the security model defined in Section 2.4. For two parties P_1 and P_2 with sets S_1, S_2 respectively, we define the functionality for the definition to be:

$$\mathcal{F}_\cup(S_1, S_2) = S_1 \cup S_2. \tag{3}$$

As the definition suggests we need to show that it is impossible to derive anything from the execution of the protocol that is not implied by possession of the input and output of the corrupted player in question.

Theorem 1. *Suppose that the protocol, $\pi_\cup^{EBF}$, is instantiated with an IND-CPA secure AHE scheme with re-randomised messages. Then $\pi_\cup^{EBF}$ securely realises $\mathcal{F}_\cup$, as in Equation* (3)*, in the presence of static semi-honest adversaries.*

Proof. We will show that the PSU protocol is secure when P_2 is corrupted first, due to the simplicity of the proof relative to the P_1 corruption case. Recall that the input for player P_1 is $\mathsf{Inp}_1 = (S_1, aux_1 = |S_2| = m)$ and for P_2 it is $\mathsf{Inp}_2 = (S_2, aux_2 = |S_1|)$.

Server corrupted. The simulator receives $\mathsf{Inp}_2 = (S_2, aux_2 = |S_1|)$ and the messages $(\mathcal{T}, \emptyset)$, where $\mathcal{T}$ is the entire message transcript that P_2 witnesses and $\emptyset$ denotes the empty output received. For P_2, $\mathcal{T}$ simply contains an encrypted Bloom filter sent by P_1. Therefore, the simulator is only tasked with constructing an encrypted Bloom filter that is indistinguishable from the one provided in the real execution. From knowledge of $(|S_1|, (h_1, \ldots, h_k))$ the simulator is able to construct an empty Bloom filter using the correct parameters and the same hash functions. The simulator encrypts each entry of the Bloom filter using the IND-CPA encryption scheme. Let $\mathcal{T}'$ denote the simulated transcript; both $\mathcal{T}$ and $\mathcal{T}'$ just contain IND-CPA encrypted Bloom filters. It is trivial to show that any adversary who can distinguish between these transcripts can break the IND-CPA security of the encryption scheme.

Client corrupted. The simulator receives $\mathsf{Inp}_1 = (S_1, aux_1 = |S_2|)$ and the messages $(\mathcal{T}, S_1 \cup S_2)$ where $\mathcal{T} = \{(\tilde{p}_j, \tilde{c}_j)\}_{j \in [1,m]}$. It derives $|S_1 \cap S_2| = I$ from S_1 and $|S_2|$, by calculating $|(S_1 \cup S_2) \setminus S_1| = U$ and subsequently $|S_2| - |(S_2 \setminus S_1)| = I$. It constructs I encryptions, c_g, of 0 and U encryptions $c_j = C_1^{(j)} +_H \ldots +_H C_k^{(j)}$ computed as in the original protocol using the elements $y_j \in (S_1 \cup S_2) \setminus S_1$ constructed via the output and the input set. Finally, it sends $m = I + U$ messages in total where I messages are two encryptions of zero and the remaining U messages are represented by $\{(\tilde{p}_j, \tilde{c}_j)\}$, let $\mathcal{T}'$ be the simulated transcript containing these messages.

It is clear that the adversary learns the same union output in the case of $\mathcal{T}'$ since messages are constructed identically as in the real-setting. Notice that in the real-world execution the ciphertexts $(\tilde{p}_j, \tilde{c}_j)$ are re-randomised after performing homomorphic additions and thus are indistinguishable from brand new encryptions. Since $\mathcal{T}'$ only differs in that each message is a fresh encryption, we can show that any adversary that can distinguish $\mathcal{T}$ with non-negligible advantage must break the security of the encryption scheme after re-randomisation. However, if an adversary is able to do this then they must break its IND-CPA security since re-randomising involves multiplying with a freshly encrypted ciphertext. As a consequence, the simulated transcript must be indistinguishable in its encrypted form from $\mathcal{T}$ by the IND-CPA security of the encryption scheme. Since the correctness of the simulation holds this means that no

adversary that can distinguish between the two real and simulated cases must exist. □

Malicious security. It was shown in previous works [8,17] that it is possible to prove security against malicious adversaries relating to input privacy. Broadly speaking, P_1 presents their set to a trusted certificate authority who verifies that it is honestly generated before creating an encrypted Bloom filter and signing it. When P_2 receives the Bloom filter, they verify the signature before computing the functionality above. This prevents P_1 from creating an adversarially generated Bloom filter that would potentially reveal the entirety of S_2. Since this method requires a trusted third party, this enhanced protocol can be thought of as an authenticated PSU design. This argument also applies for the PSI and PSI/PSU-CA protocol variants. We do not provide the full details here but a discussion will appear in an extended edition of this paper.

3.4 Asymptotic efficiency

Communication complexity. In the first round of our protocol, P_1 sends B ciphertexts to P_2. By Equation (1) we have that $B = nk \log e$. By choosing a constant false-positive probability for ϵ we also render k as a constant and so $O(n)$ total ciphertexts are sent.

In the second round, P_2 sends $2m$ ciphertexts to P_1 and so clearly we have communication $O(m)$ here. If we assume, as in previous works, that $n = m$ then the total communication complexity is given by $O(n)$.[2]

Computational complexity. P_1 computes B encryptions and $2m$ decryptions (in the worst case). P_1 must also compute m inverses of group elements, though techniques for doing this are very efficient. In practice, we can also reduce the number of decryptions by not computing $D_{sk}(\tilde{p}_j)$ if $D_{sk}(\tilde{c}_j) = 0$. On average this will lead to savings that are proportional to the size of the intersection.

P_2 will compute $m(k+1)$ homomorphic additions and so, by the choice of k, the work done by both parties is linear in m. Assuming that $n = m$ we get that computation comprises $O(n)$ operations. The protocols of [18,13,4,1,14] all exhibit computational complexities that are super-linear in n, by comparison.

[2] This can be easily done by padding the smaller of the two sets up to the size of the larger one.

4 Adaptations to PSI and PSI/PSU-CA

An attractive feature of our simple protocol construction is the ease that we can adapt the design to securely compute different set operations. Here we consider the widely used operations PSI and PSI/PSU-CA and how we can adapt our technique for securely computing PSU to compute these functionalities instead. We define the ideal functionalities for PSI ($\mathcal{F}_{\cap}$) and PSI-CA ($\mathcal{F}_{|\cap|}$) as:

$$\mathcal{F}_{\cap}(S_1, S_2) = S_1 \cap S_2, \quad \mathcal{F}_{|\cap|}(S_1, S_2) = |S_1 \cap S_2| \tag{4}$$

(with $\mathcal{F}_{|\cup|}$ defined analogously). We will prove the security of our designs with respect to these functionalities.

4.1 PSI protocol

A PSI protocol can be constructed using the same inverted Bloom filter and AHE scheme that we use for the PSU variant, the only thing that change are the messages that P_2 computes. First, P_2 computes $c_j = C_1^{(j)} +_H \ldots +_H C_k^{(j)}$ as before, for each $y_j \in S_2$ and thus:

$$c_j = \begin{cases} E_{pk}(0) & \text{if } y_j \in S_1 \\ E_{pk}(z_j) & \text{if } y_j \notin S_1 \end{cases}$$

where $1 \leq z_j \leq k$ is the number of encryptions of 1 corresponding to y_j. P_2 then sends the messages $(\mathsf{ReRand}((r_j \cdot c_j) +_H E_{pk}(y_j)), \mathsf{ReRand}(c_j))$ (for randomly sampled r_j) to P_1. Recall that P_1 should only learn those y_j that satisfy $y_j \in S_1$ since the operation is a set intersection. In the case where $y_j \notin S_1$, we have that P_1 receives encryptions of the pair $((r_j \cdot z_j) + y_j, z_j)$. Since r_j is a random mask, intuitively P_1 is unable to learn the value y_j. When $y_j \in S_1$ they receive encryptions of $(y_j, 0)$, where clearly they can learn y_j. Figure 2 gives an overview of this protocol.

Protocol correctness. The correctness of the protocol follows since P_1 outputs those y_j such that c_j is an encryption of 0, since this allows for P_1 to decrypt $\tilde{p}_j$ to retrieve y_j. This only occurs when $y_j \in S_1$ (with respect tot he false-positive probability). Moreover, when $y_j \notin S_1$ they receive a randomly masked decryption and so y_j cannot be learnt.

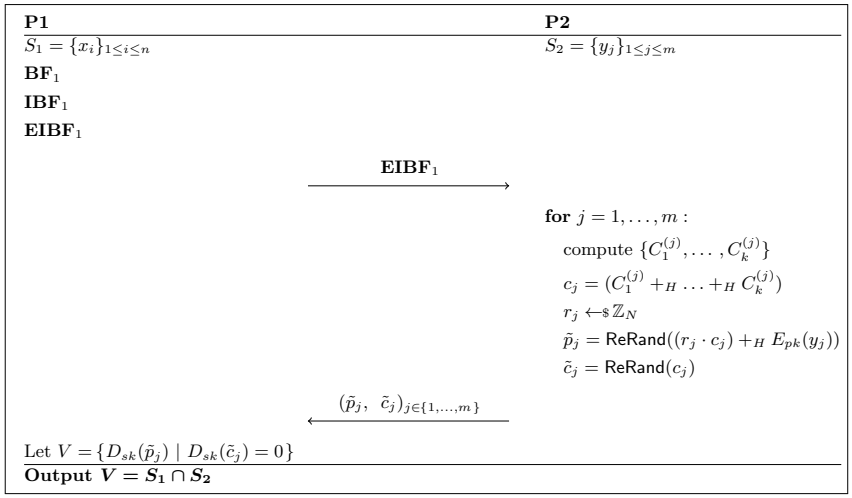

Fig. 2. A protocol that securely realises $\mathcal{F}_\cap$ in a similar way to $\pi_\cup^{\text{EBF}}$.

Protocol security.

Theorem 2. *Suppose that the protocol, $\pi_\cap^{EBF}$, is instantiated with an IND-CPA secure, AHE scheme with re-randomised messages. Then $\pi_\cap^{EBF}$ securely realises $\mathcal{F}_\cap$ in the presence of static semi-honest adversaries.*

Proof. The security argument when P_2 is corrupted is identical to the one shown in Theorem 1 since the encrypted Bloom filter is unchanged. For the corruption of P_1 we note that the security relies now on P_1 not being able to learn elements $y'_j \notin S_1 \cap S_2$ in order to realise $\mathcal{F}_\cap$ securely. The simulator receives the input $\mathsf{Inp}_1 = (S_1, aux_1 = |S_2|)$ and the messages $(\mathcal{T}, S_1 \cap S_2)$. The transcript contains m pairs of encryptions $\{(\tilde{p}_j, \tilde{c}_j)\}_j$ of the form $(r_j \cdot c_j +_H E_{pk}(y_j), \tilde{c}_j)$.

Let $I = |S_1 \cap S_2|$ and $J = |S_2| - I$. The simulator encrypts the I elements in $S_1 \cap S_2$ along with I encryptions of 0 for the messages that the adversary should learn. They then sample J random elements r'_i and random $1 \le z'_i \le k$ for $1 \le i \le J$ and compute their encryptions. They shuffle the order of the entire set of ciphertexts and submit pairs $(\tilde{p}'_j, \tilde{c}'_j)$ for $j \in [1, m]$ to P_1.

By a similar argument to the PSU security proof, the re-randomisation procedure means that P_1 cannot learn anything from the ciphertexts themselves. Therefore, the only situation where the adversary can distinguish is if they can learn a different output. Note that there are I encryptions of $(y_j, 0)$ which correspond exactly to those $y_j \in S_1 \cap S_2$. Therefore, we only have to show that the adversary cannot distinguish

between the decrypted values $((r_j \cdot z_j) + y_j, z_j)$ and (r'_j, z'_j) from the real and simulated worlds respectively.

Since r_j is a random mask, $(r_j \cdot z_j) + y_j$ is also randomly distributed across the domain. Therefore, this is identically distributed to the decrypted value r'_j and thus P_1 cannot distinguish these two values. Furthermore, as long as z'_j is chosen such that it mirrors the probability distribution of values given in $\mathbf{BF}_1$ then this should also be indistinguishable. Finally note that this distribution is entirely public since the simulator can construct the Bloom filter from knowledge of S_1 and $h_1, \ldots, h_k$. □

4.2 PSI/PSU-CA protocol

We can make use of the fact that by calculating one of PSI-CA or PSU-CA then we can calculate the other using the following relation:

$$|X \cap Y| = |X| + |Y| - |X \cup Y| \tag{5}$$

and thus we can concentrate on only computing one of the operations. We can create a secure protocol, $\pi^{\text{EBF}}_{|\cap|}$, for calculating PSI-CA by adapting the protocol $\pi^{\text{EBF}}_{\cup}$ to have P_2 to just send the message $(\tilde{c}_j)$ where c_j is calculated in the same way as the previous protocols and $\tilde{c}_j = \mathsf{ReRand}(c_j \cdot r_j)$. We compute $\tilde{c}_j$ using the ability to compute scalar multiplications on c_j and where r_j is some randomly chosen non-zero integer. We need to mask c_j in this way since only adding an encryption of zero as before would reveal extra information to P_1 on decryption.

The protocol proceeds in the same way except that P_1 only decrypts $\tilde{c}_j$. If $D_{sk}(\tilde{c}_j) = 0$ then they increment a counter c. Once all $\tilde{c}_j$ have been decrypted then P_1 outputs c as the answer. For PSU-CA they compute the count of $\tilde{c}_j$ that do not decrypt to 0 and then output $|S_1| + c$.

Protocol correctness. Correctness is satisfied since $D_{sk}(\tilde{c}_j) = 0$ if and only if $y_j \in S_1$ (and thus $y_j \in S_1 \cap S_2$) with all but the negligible probability of a false positive occurring.

Protocol security.

Theorem 3. *Suppose that the protocol, $\pi^{EBF}_{|\cap|}$, is instantiated with an IND-CPA secure AHE scheme and that ciphertexts are re-randomised. Then $\pi^{EBF}_{|\cap|}$ securely realises $\mathcal{F}_{|\cap|}$ in the presence of static semi-honest adversaries.*

Proof. The proof for security here is encompassed by the previous security arguments, we provide a sketch proof only due to space constraints. The case where P_2 is corrupted is covered as before. The simulator can construct the required number of encrypted values based on knowledge of the output. The adversary cannot distinguish the real and simulated encrypted formats due to the re-randomisation of ciphertexts. The decrypted values reveal nothing apart from the cardinality of the set (which holds by correctness) since the simulator applies an identical random mask to each concealed value. □

4.3 Asymptotic evaluation

It is easily observable that the asymptotic performance of these two adaptations is essentially the same as the PSU variant. The cardinality variant is slightly more efficient since P_2 sends half as many ciphertexts and computes less homomorphic operations. Likewise the PSI variant requires that P_2 compute m fresh encryptions, on for each y_j. Fortunately this cost is absorbed into the $O(n)$ computation cost when taking $n = m$.

In Table 2, we provide a comparison of the asymptotic performance with the most efficient cardinality protocols. We do not provide the same analysis for our PSI protocol due to the relative density of results with similar complexities, though our design is asymptotically competitive with the most practical designs. We also provide a comparison of our toolkit with previous designs by Kissner and Song [18]. Our work improves demonstrably from their designs in both communication and computation. More recent attempts to provide multiple functionalities [1,14] also fall short of realising linear computational complexities and so our toolkit is asymptotically optimal in comparison with these previous works.

	Communication	Computation
[5]	$O(n)$	$O(n)$
[11]	$O(B)$	$O(B)$
$\pi^{\text{EBF}}_{\lvert\cap\rvert}/\pi^{\text{EBF}}_{\lvert\cup\rvert}$	$\boldsymbol{O(n)}$	$\boldsymbol{O(n)}$

		Communication	Computation
[18]	**PSI**	$O(cNn\log\lvert E\rvert)$	$O(n^2)$
	PSU	$O(N^2n\log\lvert E\rvert)$	$O(n^2)$
	PSI/PSU-CA	$O(N^2n\log\lvert E\rvert)$	$O(n^2)$
π^{EBF}	**PSI**	$2n+B$	$O(n)$
	PSU	$2n+B$	$O(n)$
	PSI/PSU-CA	$n+B$	$O(n)$

Table 2. Left: Comparison of our PSI/PSU-CA protocols with [5,11]. **Right:** Comparison of our complexities with the protocols of [18].

5 Experimental evaluation

Parameter choices. To fully evaluate the practicality of our designs we present the results of an implementation of the proposed protocols. The implementations are written in `Go` and all experiments have been run on hardware with 256gb RAM with an Intel(R) Xeon(R) CPU E5-2667 v2 @ 3.30GHz and utilising a maximum of 8 cores (when parallel computation is required). We instantiate the protocol with an open-source implementation of Paillier encryption in `Go`, known as `go-go-gadget-paillier`[3] with optimisations[4] to provide the homomorphic capability over ciphertexts. We provide our own implementation of the encrypted Bloom filter functionality. Our PSO implementation requires only 425 lines of code.

For the experiments, we examine running times for sets sizes ranging from 2^8 to 2^{18} elements; these sizes are used commonly in prior work. We choose a false positive probability of $\epsilon = 2^{-30}$ alongside the choice of optimal parameters for our Bloom filter as described in Section 2.2 – for example $k = 30$ and thus $B = kn \log e$ by Equation (1) for sets of size n. For the Paillier encryption scheme we experiment with moduli N with bit-lengths 1024 and 2048 roughly equivalent to 80 and 116 bit security. We chose the domain of possible elements to be $5n$ where n is the set size and we choose the sets at random from this domain. This choice was made merely to guarantee that the size of the intersection is not too low, ensuring a realistic simulation. During our experimentation we make use of concurrency features in `Go` to make significant savings via parallel execution of operations. Times were $\sim 3\times$ quicker using parallel execution and thus we do not present our single-threaded results.

5.1 Results

In Table 3 we give the full runtimes for our PSO protocols. Table 5 provides the maximum amount of communication data[5] and in Table 4 we provide the time taken for the initial encryption. For reference, in Appendix A, we provide comparisons with efficient PSI designs [10,7,16]. The existing works of [23,22,19] provide even faster designs though these use inherently symmetric primitives which are not comparable with our work. It should be noted however that our designs represent a much more generic functionality since we can compute multiple set operations. These

[3] github.com/roasbeef/go-go-gadget-paillier
[4] github.com/mcornejo/go-go-gadget-paillier
[5] We do not provide estimates for the 2048 bit case since they are derivable by doubling the 1024 bit estimates.

previous designs are only suitable for PSI computation. There are no current implementations of PSU designs for an experimental comparison.

Clearly, there is a large gap in efficiency between our protocols and those of state-of-the-art PSI designs. However, observe that the majority of our running times are spent on encrypting the initial Bloom filter that is sent to P_2. In fact, the homomorphic operations and output computation each take $< 5\%$ of all operating runtime for all set sizes. Subsequently, we can see that that the actual online phase of our protocol could be regarded as practical. As a consequence, the main bottleneck of our design appears to be the encryption phase and thus any optimisation in the underlying encryption scheme would drastically improve the practicality of our construction.

Set size	Timings	**PSU**	**PSI**	**CA**
2^8	Hom. ops	0.49	0.5	0.5
	Out time	0.56	0.54	0.55
	Full time	**11.78**	**11.76**	**11.75**
2^{10}	Hom. ops	1.94	1.96	1.95
	Out time	2.21	2.2	2.22
	Full time	**44.73**	**44.68**	**44.7**
2^{12}	Hom. ops	7.82	7.82	7.87
	Out time	8.61	8.74	8.86
	Full time	**175.7**	**175.79**	**175.96**
2^{14}	Hom. ops	31.37	31.32	31.59
	Out time	35.78	34.9	35.48
	Full time	**702.4**	**702.39**	**703.24**
2^{16}	Hom. ops	126.16	127.43	127.01
	Out time	141.72	138.82	141.76
	Full time	**2836.5**	**2834.68**	**2837.19**
2^{18}	Hom. ops	510.19	503.95	508.53
	Out time	536.48	556.72	556.05
	Full time	**11341.2**	**11327.78**	**11331.67**

Set size	Timings	**PSU**	**PSI**	**CA**
2^8	Hom. ops	3.33	3.36	3.33
	Out time	3.66	3.55	3.58
	Full time	**78.02**	**77.76**	**77.76**
2^{10}	Hom. ops	13.45	13.33	13.44
	Out time	14.77	14.26	14.31
	Full time	**312.44**	**311.61**	**311.76**
2^{12}	Hom. ops	52.97	53.41	53.15
	Out time	55.59	57.98	56.44
	Full time	**1233.59**	**1235.69**	**1233.84**
2^{14}	Hom. ops	212.33	212	212.55
	Out time	228.13	223.31	225.11
	Full time	**4952.94**	**4947.32**	**4949.66**
2^{16}	Hom. ops	856.27	859.67	857.9
	Out time	902.81	906.9	907.27
	Full time	**19881.51**	**19888.79**	**19887.17**
2^{16}	Hom. ops	3411.87	3416.9	3419.2
	Out time	3580.25	3595	3575.94
	Full time	**79272.48**	**79290.82**	**79274.15**

Table 3. Runtimes (secs) for increasing set sizes, **left** = 1024-bit moduli, **right** = 2048-bit. 'Hom. ops' refers to time taken for homomorphic operations; 'Out time' refers to time taken to compute output; 'Full time' includes time for encryption from Table 4.

	2^8	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}
1024 bits	10.7	40.53	159.23	636.17	2568.41	10267.03
2048 bits	70.85	284.02	1124.3	4512	18122	72278.95

Table 4. Encryption times (sec)

Set sizes	2^8	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}
Comms (mb)	2.83	11.32	45.28	181.12	724.49	2897.97

Table 5. Maximum communication costs (mb) for our protocols for 1024 bit security.

5.2 Amortising Bloom filter encryption

Importantly, we can think of the Bloom filter encryption phase as an offline cost. By encrypting with an additively homomorphic scheme, we are able to retain functionality of the Bloom filter even after encryption has took place. Notice that the encrypting party is only required to store new elements, and recall that it is impossible to remove elements even from a standard Bloom filter. After a Bloom filter has been encrypted elements can still be added to the set by adding '1' to any specified ciphertext that currently encrypts '0'.

Using this homomorphic property allows us to amortise the encryption operation over the natural life of a Bloom filter (i.e. until the underlying set has to be recomputed, or the maximum number of elements has been reached). Consequently, it is reasonable to suggest that the encryption phase of our protocol can be thought of as a one-time cost. The encrypted Bloom filter could then be used in multiple PSO instantiations, as long as re-randomisation of ciphertexts takes place. The 'online' phase of our protocol is very efficient to run and so it is an advantageous feature of our design that the main cost can be amortised across several instantiations.

6 Conclusion

In this paper we have devised a new method of computing the main private set operations with linear complexities. Our PSU protocol is the first construction that demonstrates both linear computation and communication. We have also shown that the design is easily adapted to support other private set functionalities. Finally, our experimental work shows the practicality of our protocols in the online phase. Our designs provides therefore an efficient toolkit for generic PSO computations.

Acknowledgements. The authors would like to thank Sumit Debnath, Mikkel Lambaek and Claudio Orlandi for their help in establishing problems with previous versions of this work. This work was supported by the EPSRC and the UK Government as part of the Centre for Doctoral Training in Cyber Security at Royal Holloway, University of London (EP/K035584/1).

References

1. M. Blanton and E. Aguiar. Private and oblivious set and multiset operations. In H. Y. Youm and Y. Won, editors, *ASIACCS 12*, pages 40–41. ACM Press, May 2012.

2. B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.
3. P. Bose, H. Guo, E. Kranakis, A. Maheshwari, P. Morin, J. Morrison, M. H. M. Smid, and Y. Tang. On the false-positive rate of bloom filters. *Inf. Process. Lett.*, 108(4):210–213, 2008.
4. J. Brickell and V. Shmatikov. Privacy-preserving graph algorithms in the semi-honest model. In *ASIACRYPT*, volume 3788 of *Lecture Notes in Computer Science*, pages 236–252. Springer, 2005.
5. E. D. Cristofaro, P. Gasti, and G. Tsudik. Fast and private computation of cardinality of set intersection and union. In J. Pieprzyk, A.-R. Sadeghi, and M. Manulis, editors, *CANS 12*, volume 7712 of *LNCS*, pages 218–231. Springer, Heidelberg, Dec. 2012.
6. A. Davidson, G. Fenn, and C. Cid. A model for secure and mutually beneficial software vulnerability sharing. In *Proceedings of the 2016 ACM on Workshop on Information Sharing and Collaborative Security*, WISCS '16, pages 3–14, New York, NY, USA, 2016. ACM.
7. E. De Cristofaro and G. Tsudik. Practical private set intersection protocols with linear complexity. In R. Sion, editor, *FC 2010*, volume 6052 of *LNCS*, pages 143–159. Springer, Heidelberg, Jan. 2010.
8. S. K. Debnath and R. Dutta. Efficient private set intersection cardinality in the presence of malicious adversaries. In M. H. Au and A. Miyaji, editors, *ProvSec 2015*, volume 9451 of *LNCS*, pages 326–339. Springer, Heidelberg, Nov. 2015.
9. S. K. Debnath and R. Dutta. Secure and efficient private set intersection cardinality using bloom filter. In J. Lopez and C. J. Mitchell, editors, *ISC 2015*, volume 9290 of *LNCS*, pages 209–226. Springer, Heidelberg, Sept. 2015.
10. C. Dong, L. Chen, and Z. Wen. When private set intersection meets big data: an efficient and scalable protocol. In A.-R. Sadeghi, V. D. Gligor, and M. Yung, editors, *ACM CCS 13*, pages 789–800. ACM Press, Nov. 2013.
11. R. Egert, M. Fischlin, D. Gens, S. Jacob, M. Senker, and J. Tillmanns. Privately computing set-union and set-intersection cardinality via bloom filters. In E. Foo and D. Stebila, editors, *ACISP 15*, volume 9144 of *LNCS*, pages 413–430. Springer, Heidelberg, June / July 2015.
12. M. J. Freedman, K. Nissim, and B. Pinkas. Efficient private matching and set intersection. In C. Cachin and J. Camenisch, editors, *EUROCRYPT 2004*, volume 3027 of *LNCS*, pages 1–19. Springer, Heidelberg, May 2004.
13. K. B. Frikken. Privacy-preserving set union. In J. Katz and M. Yung, editors, *ACNS 07*, volume 4521 of *LNCS*, pages 237–252. Springer, Heidelberg, June 2007.
14. C. Hazay and K. Nissim. Efficient set operations in the presence of malicious adversaries. *Journal of Cryptology*, 25(3):383–433, July 2012.
15. F. Hormozdiari, J. W. J. Joo, A. Wadia, F. Guan, R. Ostrovsky, A. Sahai, and E. Eskin. Privacy preserving protocol for detecting genetic relatives using rare variants. *Bioinformatics*, 30(12):204–211, 2014.
16. Y. Huang, D. Evans, and J. Katz. Private set intersection: Are garbled circuits better than custom protocols? In *NDSS 2012*. The Internet Society, Feb. 2012.
17. F. Kerschbaum. Outsourced private set intersection using homomorphic encryption. In H. Y. Youm and Y. Won, editors, *ASIACCS 12*, pages 85–86. ACM Press, May 2012.
18. L. Kissner and D. X. Song. Privacy-preserving set operations. In V. Shoup, editor, *CRYPTO 2005*, volume 3621 of *LNCS*, pages 241–257. Springer, Heidelberg, Aug. 2005.

19. V. Kolesnikov, R. Kumaresan, M. Rosulek, and N. Trieu. Efficient batched oblivious PRF with applications to private set intersection. In E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, editors, *ACM CCS 16*, pages 818–829. ACM Press, Oct. 2016.
20. C. A. Meadows. A more efficient cryptographic matchmaking protocol for use in the absence of a continuously available third party. In *Proceedings of the 1986 IEEE Symposium on Security and Privacy, Oakland, California, USA, April 7-9, 1986*, pages 134–137, 1986.
21. P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In J. Stern, editor, *EUROCRYPT'99*, volume 1592 of *LNCS*, pages 223–238. Springer, Heidelberg, May 1999.
22. B. Pinkas, T. Schneider, G. Segev, and M. Zohner. Phasing: Private set intersection using permutation-based hashing. In *USENIX Security Symposium*, pages 515–530. USENIX Association, 2015.
23. B. Pinkas, T. Schneider, and M. Zohner. Faster private set intersection based on OT extension. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, pages 797–812, 2014.
24. P. Rindal and M. Rosulek. Improved private set intersection against malicious adversaries. Cryptology ePrint Archive, Report 2016/746, 2016. http://eprint.iacr.org/2016/746.
25. J. H. Seo, J. H. Cheon, and J. Katz. Constant-round multi-party private set union using reversed laurent series. In M. Fischlin, J. Buchmann, and M. Manulis, editors, *PKC 2012*, volume 7293 of *LNCS*, pages 398–412. Springer, Heidelberg, May 2012.

A Runtimes and communication from previous work

Security level	**80-bit**					**128-bit**				
Set sizes	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}
De Cristofaro et al. [7]	0.5	2.0	7.9	31.3	124.9	7.7	31.0	124.3	497.2	1982.1
Huang et al.[16]*	1.2	5.1	21.2	100.3	462.7	1.9	7.8	36.5	168.9	762.4
Dong et al.[10]*	0.15	0.5	2.0	8.1	34.3	0.27	1.0	4.1	16.7	67.6

Table 6. Runtimes (seconds) taken from [23]

Security level	**80-bit**					**128-bit**				
Set sizes	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}
De Cristofaro et al. [7]	0.3	1.1	4.3	17.3	69.0	0.8	3.1	12.5	50.0	200.0
Huang et al.[16]*	18.8	90.0	420.0	1920.0	8640.0	30.0	144.0	672.0	3072.0	13824.0
Dong et al.[10]*	1.1	4.5	18.1	72.6	290.4	2.9	11.6	46.2	184.9	739.7

* With optimisations from [23]

Table 7. Communication costs (mb) taken from [23]