Private Set Intersection: Are Garbled Circuits Better than Custom Protocols?

Yan Huang David Evans
University of Virginia

Jonathan Katz
University of Maryland

http://MightBeEvil.org

Abstract

Cryptographic protocols for *Private Set Intersection* (PSI) are the basis for many important privacy-preserving applications. Over the past few years, intensive research has been devoted to designing custom protocols for PSI based on homomorphic encryption and other public-key techniques, apparently due to the belief that solutions using generic approaches would be impractical. This paper explores the validity of that belief. We develop three classes of protocols targeted to different set sizes and domains, all based on Yao's generic garbled-circuit method. We then compare the performance of our protocols to the fastest custom PSI protocols in the literature. Our results show that a careful application of garbled circuits leads to solutions that can run on million-element sets on typical desktops, and that can be competitive with the fastest custom protocols. Moreover, generic protocols like ours can be used directly for performing more complex secure computations, something we demonstrate by adding a simple information-auditing mechanism to our PSI protocols.

1 Introduction

Protocols for private set intersection (PSI) allow two parties holding sets S and S' to compute the intersection $I = S \cap S'$ without revealing to the other party any additional information about their respective sets (except their sizes). Either party, or both, may learn the intersection depending on the application. PSI can be used directly to enable two companies to find their common customers, or to allow a government agency to determine whether anyone on its terrorist watch list is present on a flight manifest. PSI can also be used as a sub-routine of larger privacy-preserving computations. For example, companies can perform data mining only on the customers they have in common (using PSI for pre-processing), or parties might apply some filter, privately specified by the other party, to their input set before computing the intersection (using PSI for post-processing). Many other examples are provided by De Cristofaro et al. [9].

Because of its importance and wide applicability, many protocols for PSI and its variants have been proposed [7–11, 14, 15, 20, 24]. All these works develop *custom* PSI protocols using asymmetric cryptography, usually with the explicit remark that they do so because solving PSI via *generic* techniques (such as garbled-circuit protocols [39]) would be impractical.[1] However, this claim has never been substantiated.

The belief that generic techniques for solving PSI are inefficient may stem from several factors: a general feeling that generic protocols are inherently slow and that garbled circuits do not scale, or perhaps the belief that computing the intersection of two sets of size n requires $\Theta(n^2)$-size circuits since all pairs of elements must be compared. Recent implementations of Yao's (generic) garbled-circuit technique suggest that the first assumption is invalid; the second is simply incorrect (see Section 5).

In this work, we explore using Yao's generic garbled-circuit approach to perform PSI. Our primary goal is to better understand the efficiency of generic protocols as compared to "custom-designed" protocols for specific tasks. As a secondary benefit, we obtain protocols that are substantially more efficient than existing PSI protocols in certain settings. Using generic techniques to generate privacy-preserving protocols has several advantages: by relying on existing software packages for constructing garbled-circuit protocols [17, 32, 33, 36], one need only write down a circuit for the function to be computed rather than having to design and implement a new protocol from scratch. Generic protocols are also inherently more *modular* than custom-tailored protocols. For example, returning to the specific case of PSI, we observe that it is relatively easy to adapt a generic PSI protocol to support additional *private* pre- or post-computation (simply by extending the circuit with the desired pre- or post-

[1] Recently a generic implementation of PSI was considered in the less-accepted *three-party* setting [23]. We discuss this work further in Section 1.3.

Protocol	Number of Non-Free Gates
Bitwise-AND (BWA)	2^{σ}
Pairwise-Comparisons (PWC)	$\left((2n-\hat{n})^2+\hat{n}\right)(\sigma-1)/4$
Sort-Compare-Shuffle-SORT	$2\sigma n\log(2n)+\left((3n-1)\sigma-n\right)+2\sigma n\log^2(2\hat{n})$
Sort-Compare-Shuffle-HE	$2\sigma n\log(2n)+\left((3n-1)\sigma-n\right)+(\sigma+32)n$
Sort-Compare-Shuffle-WN	$2\sigma n\log(2n)+\left((3n-1)\sigma-n\right)+\frac{\sigma(n\log n-n+1)}{3}$

Table 1: Gate counts for our protocols. The size of each set is n, elements are represented using σ bits, and $\hat{n}$ is the size of the intersection.

computation), whereas with custom protocols it may not be possible to modify the protocol to support additional computation while maintaining comparable efficiency.

1.1 Contributions

The goal of our work is to understand how generic protocols for PSI perform relative to custom PSI protocols. We assume the semi-honest setting (see Section 2) which has been the focus of much of the prior work on PSI, and leave consideration of stronger threat models to future work.

We describe three classes of protocols for PSI based on garbled-circuit techniques, each targeted to different set sizes and different universes from which the elements in the sets are drawn. Section 1.2 provides an overview of these protocols, and Sections 3–5 describe each protocol in more depth. In each case, we significantly improve performance by optimizing the underlying circuit design.

We evaluate our protocols on a range of parameters, comparing them both to each other as well as to a recent protocol by De Cristofaro and Tsudik [10] that is the most efficient PSI protocol previously reported in the literature. Somewhat surprisingly, we found that the generic garbled-circuit approach can *outperform* the De Cristofaro-Tsudik protocol in many settings. For example, when the universe is relatively small (e.g., elements can be represented using 32 bits) our most efficient protocol is faster than theirs for all settings of the security parameter, and by several orders of magnitude at high security levels. Even when the universe is large, with elements represented using 160 bits, our protocol remains competitive with their PSI protocol, and is faster than their protocol except at the *ultra-short* security level. Section 7 provides details on our evaluation and results.

We also explore the use of our generic protocols as subroutines within larger privacy-preserving computations. As an illustrative example, we show in Section 6 how a simple auditing mechanism can be incorporated into our protocols so that the intersection is not revealed when its size exceeds some threshold.

1.2 Overview of our Protocols

We consider three classes of protocols for PSI, as summarized in Table 1. All of our protocols are built using Yao's garbled-circuit technique which takes any boolean circuit C and yields a secure protocol for computing C. (Section 2 provides a brief introduction to garbled-circuit protocols.) Thus, we need only describe the different types of circuits we construct and can rely on established proofs of security properties for garbled-circuit protocols in the semi-honest model [29].

Our first protocol (*Bitwise-AND* (BWA)), described in Section 3, uses a circuit based on a bit-vector representation of the parties' sets. The protocol is only practical for small universes; in that case, however, it achieves the best performance.

Section 4 describes the *Pairwise-Compare* (PWC) protocol that uses a circuit performing pairwise comparisons of the elements in the two parties' sets. This protocol has worst-case complexity $\Theta(n^2)$ for computing the intersection of two sets of size n, and is a reasonably good choice — even for large universes — as long as n is small. We present an optimization that improves performance when the size of the intersection is large without sacrificing any privacy. Though relatively simple, this and the previous protocol demonstrate that even straightforward approaches can produce effective PSI solutions using garbled circuits.

Section 5 presents our most involved protocols, which are all based on a *Sort-Compare-Shuffle* design. These protocols have complexity $\Theta(n\log n)$ with small constant factors. The main idea is for each party to sort their set locally, and then (privately) merge their sorted sets into a single sorted list. Then each adjacent pair of elements is compared (obliviously), with the value retained if the elements in the pair are equal, and a dummy value substituted otherwise. Finally, the resulting list of matching/dummy elements is obliviously shuffled before the entire list is revealed. This shuffling step is necessary because otherwise positional information about the matching elements leaks information about the non-matching elements in the parties' sets. We consider three different ways to perform the

final oblivious shuffling: (1) obliviously sorting the entire list of matching/dummy elements using a garbled-circuit approach (SCS-SORT), (2) randomly shuffling the list of matching/dummy elements using a protocol based on homomorphic encryption (SCS-HE), and (3) randomly shuffling the list as before, but using garbled circuits applied to Waksman's oblivious switching network (SCS-WN).

Table 1 gives the costs of our protocols in terms of the number of gates that are garbled and evaluated, as a function of the size n of the input sets, the number of bits σ needed to represent each set element, and the size $\hat{n}$ of the intersection. XOR gates are not counted since these can be implemented "for free" (without performing any cryptographic operations) using the free-XOR optimization [26]. For the BWA and SCS-HE protocols, there are substantial other costs so gate counts alone do not capture the full cost of those protocols.

1.3 Related Work

Most prior work on PSI has focused on developing custom protocols specific to the problem [7–11, 14, 15, 20, 24]. In the semi-honest setting (which is the setting considered here), the protocol by De Cristofaro and Tsudik [10] is the most efficient, and we use this as the baseline comparison with our protocols.

Recently, Jónsson et al. [23] explored a generic approach to the *weighted* set intersection (weighted-PSI) problem. They design a circuit for their problem and then use a (different) generic approach to obtain a secure protocol that they implement within the Sharemind framework [6]. However, their approach only works in a *multi*-party setting with an assumed honest majority. Thus, some of their techniques do not apply to the *two*-party setting we consider, where there is no honest majority. Although the circuit they construct has some high-level similarities to the circuit we construct as part of our SCS-WN protocol, we obtain better efficiency than what would be obtained using their work for several reasons. First, although PSI can be reduced to weighted-PSI, doing so leads to a less efficient circuit than what we propose. The circuits developed by Jónsson et al. require $\Theta(n \log^2 n)$ gates whereas the circuit in our SCS-WN protocol has $\Theta(n \log n)$ gates. In addition, Jónsson et al. construct an *arithmetic* circuit (where wires can take values in a finite field, and gates implement addition or multiplication over that field), whereas we require a *Boolean* circuit in order to use the garbled-circuit technique. Finally, because most prior work on set intersection focuses on the two-party setting, Jónsson et al. [23] are unable to compare their protocol to custom protocols, which is the main motivation of our work.

2 Background

This section reviews our threat model, defines the PSI problem, and provides background on secure computation and the framework we use to implement our protocols.

Threat model. In this work we focus on the standard *semi-honest* (also known as *honest-but-curious*) model where parties are assumed to follow the protocol but may then try to learn additional information from the protocol execution. Goldreich's text [12] provides a formal definition. Semi-honest security is sufficient in scenarios where it is difficult to modify software without detection (say, when parties' represent large institutions or government agencies), or when hardware/software attestation can be used. Many known semi-honest protocols can also provide strong privacy (but not correctness) guarantees when only the party that evaluates the circuit receives the result and the oblivious transfers are done using an OT protocol secure against malicious behavior. Finally, several techniques (e.g., [21, 28, 30, 37]) are available for converting protocols secure in the semi-honest setting to protocols secure under stronger notions of security (although the best known techniques still impose substantial cost). Thus, work on developing efficient semi-honest protocols is a useful step towards efficient protocols that are secure against stronger adversaries.

Private set intersection. Here we have two parties holding sets $S = \{s_1, s_2, \ldots, s_n\}$ and $S' = \{s'_1, s'_2, \ldots, s'_n\}$, respectively, where $s_i, s'_i \in \{0,1\}^\sigma$ and we assume neither S nor S' contains any duplicate elements. We assume each party's set is of (known) size n and all elements are exactly σ bits long (using padding it is easy to handle the case where set sizes are different or even kept hidden, up to a known upper bound, or where elements have different sizes). We also assume both sets change at each invocation (note that computing PSI repetitively with a static input set can cause substantial leakage merely by revealing the outputs). The goal is for the parties to compute the intersection $I = S \cap S'$ without revealing any information other than I. This level of privacy is implied by the standard notion of security in the semi-honest setting [12] that we use in this work. It is easy to modify all protocols so that either one or both of the parties learn the result I.

To the best of our knowledge, the most efficient previously published PSI protocols (in the semi-honest setting) are those of De Cristofaro and Tsudik [10]. Security of their protocols is based on the (non-standard) one-more-discrete-logarithm or one-more-RSA assumptions [4] in the random oracle model.

Garbled circuits. Yao's garbled-circuit approach provides a generic mechanism for constructing a (semi-honest) secure two-party protocol for computing f start-

ing from any boolean circuit for f [29, 39]. The details of the garbled-circuit technique are not necessary for understanding the results of this paper, since we primarily use it as a black box and focus on constructing optimized boolean circuits for various functions. Nevertheless, we provide a brief overview here, and highlight some of the optimizations we use.

In a garbled-circuit protocol, one party (the circuit *generator*) prepares an "encrypted" version of a circuit computing f. The second party (the circuit *evaluator*) then obliviously computes the output of the circuit without learning any intermediate values. Starting with a (known) boolean circuit for f, the circuit generator associates two random cryptographic keys w_i^0, w_i^1 with each wire i of the circuit, where w_i^0 encodes a 0-bit and w_i^1 encodes a 1-bit. Then, for each binary gate g of the circuit with input wires i, j and output wire k, the generator computes ciphertexts

$$\mathrm{Enc}_{w_i^{b_i}}(\mathrm{Enc}_{w_j^{b_j}}(w_k^{g(b_i,b_j)}))$$

for $b_i, b_j \in \{0,1\}$. The resulting four ciphertexts, in random order, constitute a *garbled gate*. The collection of all garbled gates forms the *garbled circuit* that is sent to the evaluator.

Given keys w_i, w_j associated with both input wires i, j of some garbled gate, the evaluator can compute a key for the output wire of that gate by decrypting the appropriate ciphertext. Thus, given one key for each input wire of the circuit, the evaluator can evaluate all gates in the circuit in topological order to compute one key for each output wire of the circuit. These keys can then be mapped to their semantic values using mappings provided by the circuit generator. As described, this requires up to four decryptions per garbled gate. In fact, the construction can be modified using standard techniques so the evaluator uses only a *single* decryption per garbled gate (see [33]).

Several optimizations can be applied to the above, all of which we use in our implementation. The *free-XOR* technique [26] allows XOR gates in the circuit for f to be evaluated "for free," without incurring any communication or cryptographic operations. Pinkas et al. [36] proposed a way to reduce the size of garbled gates from four to three ciphertexts, thus saving 25% of network bandwidth.[2]

It remains only to describe how the evaluator obtains appropriate keys for the input wires, i.e., the keys corresponding to the actual inputs held by the parties. The generator can simply send the keys that correspond to its own input. The parties use *oblivious transfer* (OT) to enable the evaluator to obliviously obtain the input-wire keys corresponding to its own input. An OT protocol allows a *sender*, holding strings w^0 and w^1, to transfer w^b to a receiver holding a selection bit b; the receiver learns nothing about w^{1-b}, and the sender does not learn b.

As we rely on established protocols from the literature, we do not include proofs of security for the protocols developed here. Our protocols can be proven secure based on the decisional Diffie-Hellman assumption (used for our instantiation of OT) in the random oracle model.

Implementation. Fairplay [33] provided the first implementation of Yao's garbled-circuit approach, and several subsequent works (e.g., [31,36]) have explored extensions to the malicious setting. TASTY [16] extended Fairplay to give programmers the flexibility of switching between garbled circuits and approaches using homomorphic encryption. The main drawback of Fairplay (and other tools built on it) is that it requires generating and storing the entire garbled circuit before evaluation can begin. Previous authors (e.g., [22, 35]) have thus inappropriately concluded that the garbled-circuit approach cannot scale to large circuits. Recently, Huang et al. [17] developed a garbled-circuit implementation that uses pipelining to avoid the need to ever store the entire garbled circuit. This allows it to scale to arbitrarily large circuits. In addition, it facilitates circuit-level optimizations since circuits are defined within a high-level programming framework. All of our garbled-circuit protocols are implemented using this framework.

For the oblivious transfer, our implementation uses the Naor-Pinkas protocol [34]. We also use *oblivious transfer extension* [19] which achieves an unlimited number of OTs at the cost of (essentially) k OTs, where k is a (statistical) security parameter. In our experiments, we vary k according to the desired security level (see Table 3).

3 Bitwise-AND Protocol

The BWA protocol is designed for sets whose elements are drawn from a small universe. In this case, a set can be represented by a bit-vector of length 2^σ, and the set intersection can be computed simply by bit-wise ANDing the bit-vectors of the two parties. The output is exactly a bit-vector representation of the intersection.

A circuit for this computation is straightforward, and is obtained by instantiating a binary AND gate 2^σ times. Although the cost of the resulting protocol grows exponentially with σ, the small constant factor involved leads to good performance when σ is small. Indeed, for values of σ up to 16, we found this to be the most efficient protocol in our experiments (see Section 7).

The BWA protocol does not restrict the size of the parties' sets, so a dishonest participant can use a vector of all 1s as its input and thereby learn the other participant's entire set! Hence, it should not be used in a standalone fashion by two mutually distrusting parties. Instead, such a

[2] A second optimization they propose reduces the size of garbled gates by approximately 50%, but cannot be combined with the free-XOR technique.

protocol could be used as either a sub-protocol in a larger private computation where the participants do not control the inputs directly or do not see the outputs explicitly. Alternately, it could be combined with a *self-auditing step* to ensure that the result does not leak too much information or that neither input set is too large. One of the advantages of building our protocols using generic garbled-circuit techniques is that such extensions can easily be added. Section 6 discusses how auditing can be integrated into our protocols.

4 Pairwise Comparisons

The running time of the BWA scheme scales linearly in the size of the universe (2^σ) over which the sets are defined. Thus, as the universe of elements grows, the BWA scheme becomes too inefficient to be useful. For large universes, we can use a Pairwise-Comparisons (PWC) protocol, shown in Algorithm 1. It performs comparisons between each pair of elements from the two parties' sets. The running time of PWC is quadratic in the set size (and linear in σ).

In Algorithm 1, the only part that needs to be implemented by a garbled circuit is the Equal function on line 6 which performs an equality test. An Equal circuit can be implemented by first XOR-ing the two σ-bit inputs to produce a σ-bit intermediate result. The negated-OR of these bits then indicates whether the two inputs match. Thus, an Equal circuit can be implemented using only $\sigma - 1$ non-free gates.

To improve performance, our algorithm reveals each match as soon as it is found. This allows us to avoid performing further comparisons for any elements that have already been matched. (Recall that we assume each party's set contains no duplicates.) This optimization can potentially leak positional information about the elements in the parties' sets since the participants learn the order in which matching elements are found. To avoid this, each party randomly permutes its set before starting the protocol. Then, no information is revealed other than what could already be inferred from the result, namely, the elements in the intersection.

Algorithm 1 *PairwiseComparisons*(S, S′)

```
1: for i ← 1 to S′.size do
2:     matched[i] ← False
3:
4: for i ← 1 to S.size do
5:     for j ← 1 to S′.size do
6:         if ¬matched[j] and (Equal(S[i], S′[j])) then
7:             reveal(S[i])
8:             matched[j] ← True
9:             break
```

A drawback of the above "short-circuiting" optimization is that it substantially increases the round complexity since each **reveal** operation adds an extra round of communication. To benefit from this short-circuiting without the penalty of increased round complexity, we implement the protocol using two threads where the **reveal**s are done asynchronously while the main thread compares every possible pair of elements. Once a match is found by the **reveal** thread, the main thread is notified asynchronously to skip all unnecessary comparisons involving the matched element. Since the notification is asynchronous, it is possible that some Equal circuits are unnecessarily generated. However, our experiments show that the amount of wasted work is an insignificant fraction of the total work except for very small n.

Analysis. To understand the savings of the early reveal optimization, we provide a heuristic estimate for N_{Equal}, the expected number of calls to the Equal function. Let $\hat{n}$ be the size of the intersection. Since the two parties' sets S and S' are randomly shuffled before running Algorithm 1, the $\hat{n}$ elements in the intersection will, on average, be evenly distributed in S. Thus, we expect that on average the elements of S are ordered in such a way that there are $\hat{n}+1$ intervals of $(n-\hat{n})/(\hat{n}+1)$ non-matching elements each, with each interval separated by one of the matching elements. Assuming this to be the case, for each element in the i^{th} interval ($0 \le i \le \hat{n}$) the Equal function will be evaluated exactly $n-i$ times since it will be compared with all the $n-i$ currently-unmatched elements of S'. Each matching element in S is compared, on average, with half the remaining elements in S' before the match is found. Hence,

$$\begin{aligned} N_{\mathsf{Equal}} &\approx \sum_{i=0}^{\hat{n}} \frac{(n-\hat{n})(n-i)}{\hat{n}+1} + \sum_{i=0}^{\hat{n}-1} \frac{n-i}{2} \\ &= \frac{n-\hat{n}}{\hat{n}+1} \cdot \frac{(2n-\hat{n})(\hat{n}+1)}{2} + \frac{1}{2} \cdot \frac{(2n-\hat{n}+1)\hat{n}}{2} \\ &= \frac{(2n-\hat{n})^2+\hat{n}}{4}. \end{aligned}$$

Compared to a naïve implementation where all n^2 comparisons are performed, we see that short-circuiting saves roughly 75% of the comparisons if the two sets are identical, 45% if half the elements are identical, and 30% if $\frac{1}{3}$ of the elements are identical. Our experimental results (Figure 9) are consistent with this analysis.

5 Sort-Compare-Shuffle

Although the pairwise-comparison protocol is intuitive and easy to implement, it requires $\Theta(n^2)$ comparisons and hence circuits with $\Theta(n^2)$ gates. Here we present PSI protocols that require only $\Theta(n \log n)$ element comparisons.

These protocols take advantage of the observation that each participant can locally sort their own input set. We use this extra information to improve efficiency by breaking the task into three sequential sub-tasks as shown in Figure 1.

In each of the protocols of this section, each party begins by locally sorting their set. The parties then implement an oblivious merging network (Section 5.1) to sort the union of their sets, taking advantage of the fact that both input sets are sorted. Next, we use garbled circuits to compare neighboring elements in the sorted sequence to find all the matches (Section 5.2). Directly outputting the matches at this stage would, however, reveal information about elements that are *not* in the intersection. (For example, if the parties learn that the first two elements in the sorted list match, this would reveal to the first party that the second party's set does not contain any elements smaller than the first matched element.) Thus, we obliviously shuffle the list of matched elements so that the positions of the matched elements are not revealed (Section 5.3).

5.1 Sorting

The challenge of doing oblivious sorting using a garbled-circuit approach is that the sorting must be done by a sorting algorithm that uses a *fixed* (i.e., oblivious) sequence of comparisons. Most commonly used sorting algorithms do not lead to a size-optimal circuit. However, sorting networks [3] provide a fast circuit implementation of sorting. We further take advantage of the property that each party's inputs are independently sorted in designing a circuit that merges the two sorted lists to produce the full sorted list.

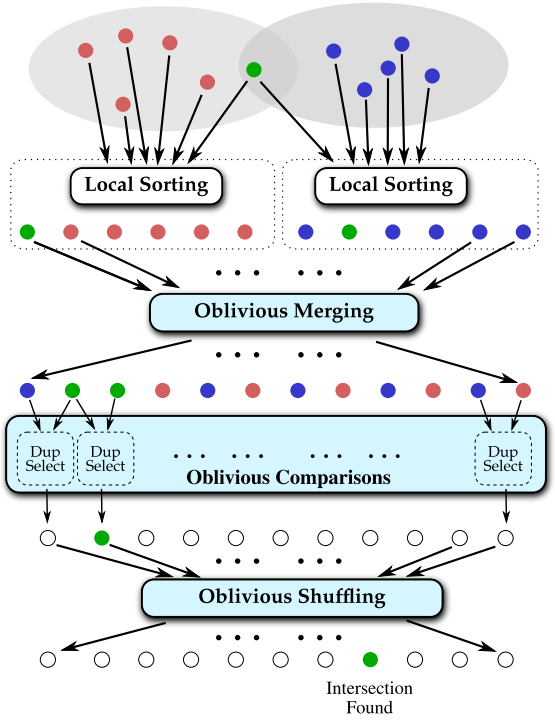


Fig. 1: Sort-Compare-Shuffle Approach (parts requiring cryptographic computation are shaded).

The basic module of a sorting network is a 2-Sorter, which sorts two σ-bit inputs. Figure 2(a) depicts a straightforward implementation of a 2-Sorter circuit. This design uses 4σ non-free binary gates to sort two σ-bit numbers, since the MIN and MAX circuits each use 2σ non-free gates [25].

We observe that the MIN and MAX circuits each contain a GT (greater than) circuit, and they each share the same input. So we can eliminate one GT component to reduce the cost to 3σ non-free binary gates as shown in Figure 2(b). Furthermore, the two MUXs are unnecessary since their outputs are correlated. Based on this insight, we arrive at the final 2-Sorter design shown in Figure 2(c). It uses a conditional-swap circuit CondSwap (Figure 3), where a CondSwap circuit with σ-bit output (Figure 3(a)) is composed of σ parallel CondSwaps with 1-bit output (Figure 3(b)). The latter requires only one non-free gate. Thus, the overall cost of the 2-Sorter circuit is reduced to 2σ non-free binary gates. Kolesnikov and Schneider [26, 27] also designed a conditional-swap circuit (see [26, Fig. 2(b)]). Our CondSwap circuit has an explicit selection input bit, whereas in their case the selection bit is hardwired by the circuit generator.

Since the two input sequences provided by the parties are pre-sorted, we can sort their union using a *bitonic merger* [3] rather than having to use a full-fledged sorting network. A sequence is said to be *bitonic* if there is at most one extremum element and the two subsequences di-

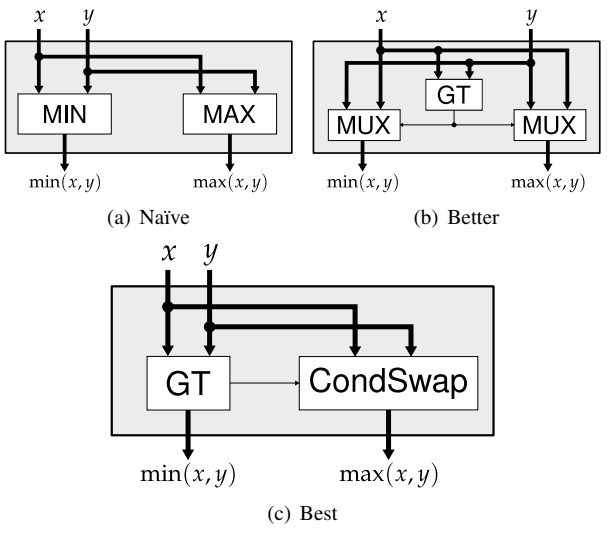
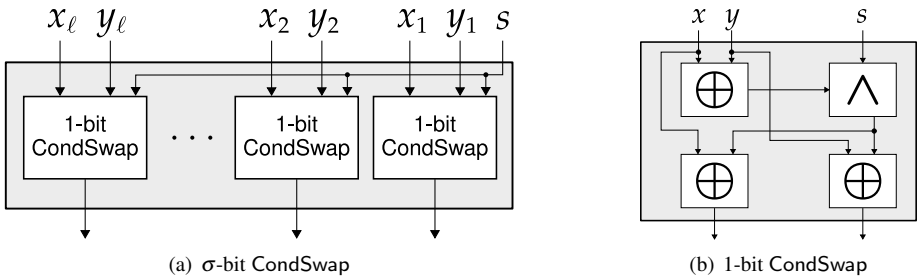


Fig. 2: The design of a 2Sorter.



(a) σ-bit CondSwap (b) 1-bit CondSwap

Fig. 3: CondSwap Circuits.

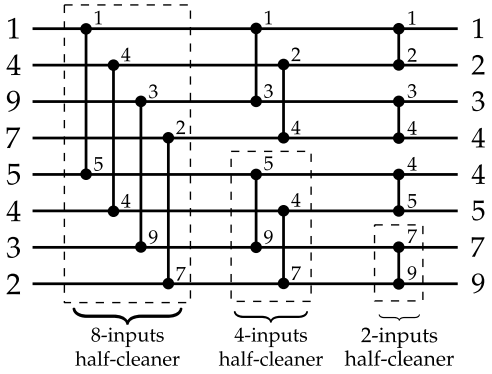


Fig. 4: Example of merging a bitonic sequence.

vided by this extremum element increase monotonically. As a specific example, the sequence that results from concatenating a sequence sorted in increasing order with a sequence sorted in decreasing order is bitonic.

Figure 4 depicts how a bitonic sequence of eight numbers is sorted by a bitonic merger. The basic idea is to apply a *half-cleaner* to recursively divide the sequence into two half-length bitonic sub-sequences where every element of one sub-sequence is larger than every element of the other. The base case, when the length of the subsequences is two, is handled using a 2-element half-cleaner which is just a 2-Sorter. A bitonic merger for $2n$ inputs uses exactly $n\log(2n)$ 2-Sorter circuits (assuming n is a power of 2). Thus, we can construct a circuit that merges two lists of n sorted σ-bit elements into a sorted list of $2n$ elements using $2\sigma n\log(2n)$ non-free binary gates.

5.2 Filtering Matching Elements

After all $2n$ elements are in sorted order, we know that any elements in the intersection must be adjacent. Thus, to find the intersection we can use a duplicate-selection circuit (DupSelect-2) that takes as input two elements, $x_1, x_2 \in \{0,1\}^\sigma$, and outputs x_1 $(= x_2)$ if they are equal and 0^σ otherwise. (This assumes that 0^σ is not a valid element in the input set. If necessary, we can increase σ by one and remap elements to ensure this.)

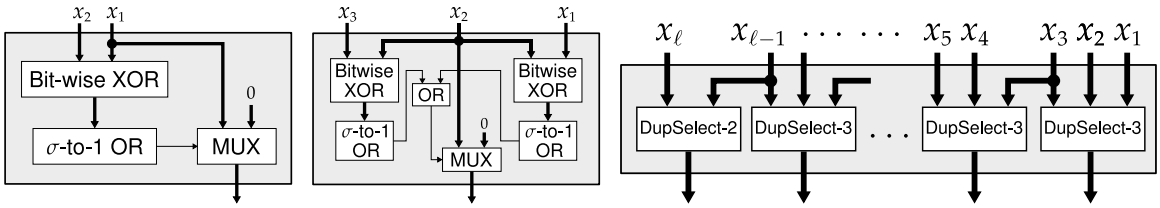
Figure 5(a) shows the design of a DupSelect-2 circuit. Since we have $2n$ elements as input to this stage, $2n-1$ DupSelect-2 circuits are needed to identify all items in the intersection. As each DupSelect-2 circuit requires $2\sigma-1$ non-free binary gates, the total cost of this phase as described is $(2\sigma-1)(2n-1)$.

We next show how to reduce this cost by taking advantage of the property that the initial input sets have no repeated elements. This implies that for every three consecutive elements in the sorted sequence, there can be at most one match. To take advantage of this, we define a 3-input version of the duplicate-selection circuit, called DupSelect-3, as follows:

$$\text{DupSelect-3}\,(a,b,c) = \begin{cases} b & \text{if } a=b \text{ or } b=c \\ 0^\sigma & \text{otherwise} \end{cases}.$$

Figure 5(b) shows the design of a DupSelect-3 circuit using $3\sigma-1$ non-free binary gates. Since we start with $2n$ elements as input to this stage, we need $n-1$ DupSelect-3 circuits and one DupSelect-2 circuit to identify all the elements in the intersection (see Figure 5(c)). This reduces the total number of non-free gates needed for this phase to $(3n-1)\sigma-n$. Another benefit of this design is that it produces only n output elements, rather than the $2n-1$ output elements that would be produced using the DupSelect-2 design. This reduces the size of the circuit needed in the subsequent oblivious shuffling phase (see below) by about 50%.

It is natural to ask whether it is possible to save even more gates by defining "higher-order" DupSelect circuits. We investigated it but found that this is not the case. The reason is that we save gates by cutting a MUX when we go from 2 to 3 inputs by exploiting the fact that within every three consecutive numbers there can be at most one match; this is no longer true once we look at four consecutive numbers. In fact, since there may be up to n items in the intersection, the number of outputs of this stage must clearly be at least n. Therefore, it does not help to combine more duplicate-selection circuits.



(a) DupSelect-2 (b) DupSelect-3 (c) Comparison-based Filtering

Fig. 5: Design and use of DupSelect-2 and DupSelect-3 circuits.

5.3 Shuffling

Following the filtering phase, we (implicitly) have a list of n elements that contains all $\hat{n}$ elements in the intersection, in sorted order, interleaved with an additional $n-\hat{n}$ occurrences of 0^σ. This list of elements cannot yet be revealed to the parties, however, since the positions of the 0-elements and the elements in the intersection may leak information about the parties' initial sets: for example, if the first element in the list is some match $x_1 \neq 0^\sigma$, this reveals that x_1 was the minimum element in both parties' sets. It is therefore necessary to destroy positional information before the elements are revealed.

We explore two general strategies for doing this: sorting the n intermediate values (Section 5.3.1), or randomly permuting them. For implementing the random permutation, we analyze strategies based on homomorphic encryption (Section 5.3.2) and using an oblivious shuffling network (Section 5.3.3).

5.3.1 Sorting

One way to hide positional information is to use an oblivious sorting network to sort the output sequence of the filtering phase (with 0^σ taken, say, to be minimal). This guarantees that no positional information leaks, since the sorted output could be generated from the intersection itself. Jónsson et al. [23] also use this general strategy in their work. As we show in Section 5.3.3, our circuit-based shuffling scheme is substantially more efficient than sorting-based approaches.

Batcher's sorting network provides a way to sort using $\Theta(n\log^2 n)$ gates [3]. Another possibility is to use the randomized Shellsort algorithm of Goodrich [13], which uses $\Theta(n\log n)$ gates but has non-zero error probability (corresponding to a small leak of information). We explored both these possibilities, but found that they are less efficient than the shuffling network presented in Section 5.3.3. In principle, sorting can also be done with $\Theta(n\log n)$ gates using the AKS sorting network [1], but the huge constant factor makes this approach impractical.

One scenario where sorting could be preferable, however, is when the size $\hat{n}$ of the intersection is small relative to the size n of the input sets. In that case sorting can be done using $n/\hat{n}$ calls to a $2\hat{n}$-sorter (that sorts $2\hat{n}$ elements), with total gate count (assuming Batcher's network is used for the $2\hat{n}$-sorter) of $n\log^2(2\hat{n})$. Though generally we cannot assume that $\hat{n}$ is small, it would be inexpensive to compute $\hat{n}$ securely (using a garbled circuit) after the filtering phase, at which point the parties could decide whether to use a sorting-based approach or a shuffling approach for the final phase. We do not explore this further.

5.3.2 Homomorphic Shuffling

Sorting actually does more work than necessary, since it is only necessary to hide positional information about the matches. We can do better by randomly permuting the elements rather than sorting them. In this and the next section, we consider two approaches to obliviously shuffle the results.

Our first shuffling approach uses homomorphic encryption to achieve linear asymptotic complexity. We begin by dividing each output from the end of the filtering phase into two secret shares, with one share given to each party. This can be done within a garbled-circuit computation as follows: Denote the intermediate results at the end of the filtering phase as $m_1,\ldots,m_n$, and recall that at this point neither party knows these values since they are encoded as part of the garbled-circuit computation. One party will provide an additional n random values $r_1,\ldots,r_n$ as input (at the beginning of the garbling stage). The garbled circuit is then extended so as to compute $r'_i = m_i + r_i$, with the other party learning r'_i. Note that r_i, r'_i form two shares of m_i. To ensure security, r_i must be sampled from a sufficiently large domain. Choosing r_i as a random $(\sigma+k)$-bit integer suffices to give statistical security $O(n\cdot 2^{-k})$.

Now one party holds $r_1,\ldots,r_n$ and the other holds $r'_1,\ldots,r'_n$, with $m_i = r'_i - r + i$ for all i. The parties then execute the homomorphic-encryption-based shuffling protocol described in Figure 6. Throughout this protocol only one party's (e.g., Alice's) public key is required, so for simplicity we use $[\![x]\!]$ to denote $[\![x]\!]_{\mathsf{pk}_{Alice}}$, the encryption of x using Alice's public key pk_{Alice}. The key idea of this shuffling protocol is that the shuffler (Bob) cannot decrypt the ciphertexts he shuffles, whereas Alice (who knows the private key) does not know how the other party shuffled

Input to Alice: $r_1, r_2, \cdots, r_n$.
Input to Bob: $r'_1, r'_2, \cdots, r'_n$.
(for all i, we have $r_i + m_i = r'_i$, where m_i is the i-th number output by the filtering phase)

Output of Alice**:** The m_i's in random permuted order.
Output of Bob**:** $\perp$ (or, if desired, the m_i's in sorted order).

Preparation:
Alice chooses a key pair $\langle \mathsf{pk}_{Alice}, \mathsf{sk}_{Alice} \rangle$ and sends pk_{Alice} to Bob.

Execution:
1. Alice encrypts the r_i's ($1 \le i \le n$) with her public key and sends the $[\![r_i]\!]$'s to Bob.
2. Bob computes $[\![m_i]\!] = [\![r'_i - r_i]\!] = [\![r'_i]\!] \cdot [\![r_i]\!]^{-1}$.
3. Bob randomly permutes the $[\![m_i]\!]$'s ($1 \le i \le n$), and sends the resulting shuffled ciphertexts back to Alice.
4. Alice receives and decrypts the ciphertexts to output the m_i's ($1 \le i \le n$).
5. (If desired) Alice sorts the m_i's and sends the result back to Bob.

Fig. 6: Homomorphic-encryption-based shuffling protocol.

the ciphertexts.

Say the r_i's are λ-bit integers. Since each λ-bit full-adder used to perform the additive sharing requires λ binary AND gates, the secret-sharing phase altogether requires λn non-free binary gates. The homomorphic-encryption protocol in Figure 6 uses two rounds of communication, each round of which communicates n ciphertexts, and uses $O(n)$ public-key operations. Although this approach has asymptotic complexity linear in n, the actual cost of the best known homomorphic encryption schemes remains very high (see Section 7), and for the parameters we consider this protocol performs worse than the pure garbled-circuit protocol described in the next section which uses $\Theta(n \log n)$ symmetric-key operations. The other drawback with this approach is that it requires homomorphic encryption which abandons our goal of using only generic secure computation to enable easy integration with other secure computations.

5.3.3 Shuffling Network

Here we explore an alternate approach to random shuffling that uses $\Theta(n \log n)$ symmetric-key operations and remains a pure garbled-circuit protocol. The basic idea is to implement an oblivious random shuffling of the elements using a *switching network*. A switching network can be viewed as a fixed circuit that takes n inputs along with an additional set of "control bits," each of which determines whether some fixed pair of elements is swapped or not. By setting the control bits appropriately, any desired permutation on the n inputs can be realized. In our setting the n inputs will be the n sorted elements from the end of the filtering stage, and one of the parties will choose a random permutation and then set the control bits so as to realize this permutation. The second party will receive as output the n elements, permuted according to the chosen permutation. If the first party should learn the output also, the second party applies another random permutation to the output elements (or simply sorts them) before sending them back. Switching networks can be constructed using $O(n \log n)$ gates [38].

The core component of a switching network is an oblivious swapper (2-Swapper) that takes as input two σ-bit values x and y, and an additional control bit s. If the value of s is 0, the output is x and y in their original order; if $s = 1$, the output is y and x in swapped order.

A switching network is simply a series of 2-Swappers (with independent control bits) applied to predetermined pairs of elements. A 2-Swapper circuit can be realized as a σ-bit CondSwap circuit (see Figure 3(b)). For our application, however, if we let the circuit generator set the control bits, then each AND gate in a CondSwap circuit can be replaced by the circuit generator with a 1-to-1 gate (which is either the identity or the 0-map, depending on the generator's secret s). Importantly, the type of the gate is known only to the circuit generator but is hidden from the circuit evaluator, so no information is leaked by this optimization. Combined with the garbled-row reduction (GRR) technique [36], the garbling of such a gate requires just a single ciphertext, which is one sixth of the cost of a 2-Sorter with GRR optimization. (Following the standard garbled-circuit approach, a unary gate would require two ciphertexts, but using the garbled-row reduction technique we can reduce this to a single ciphertext.)

The Waksman network [38], improving on the Beneš network [5], is a realization of a switching network using exactly $n \log n - n + 1$ 2-Swappers when n is a power of 2. (Constant-factor improvements when n is not a power of two were developed by Inria et al. [18], but we did not use those in our implementation.) Figure 7 illustrates a Waksman network for n inputs, assuming n is a power of 2. Its design is recursive: an n-input Waksman network is built out of two $\frac{n}{2}$-input Waksman networks (denoted by P_0 and P_1 in the figure) and $n-1$ 2-Swappers (denoted by $\mathrm{I}_1, \cdots, \mathrm{I}_{\frac{n}{2}}$ and $\mathrm{O}_2, \cdots, \mathrm{O}_{\frac{n}{2}}$). Using the construc-

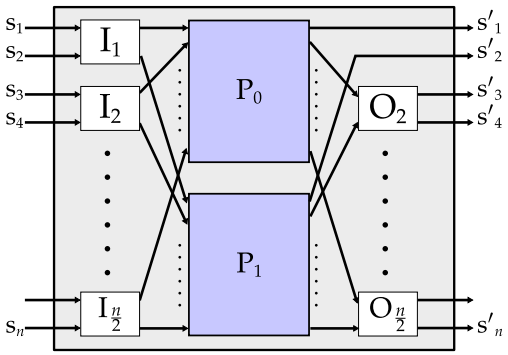


Fig. 7: Waksman Network for n inputs.

tion of a 2-Swapper circuit discussed earlier, the cost of the entire oblivious shuffling stage is only a small fraction (about 15%) of that spent in the oblivious sorting phase of the overall PSI protocol.

Note that choosing the control bits uniformly at random does not induce a random permutation. Instead, an algorithm is used to configure the control bits of a Waksman network to produce any of the $n!$ permutations of the n inputs. To induce a random permutation the circuit generator first chooses a random permutation π on n elements. It then uses the *ConfigureWaksman* function shown in Algorithm 2 to set the control bits, represented by the Boolean arrays I and O (corresponding to the 2-Swapper circuits in Figure 7). This algorithm sets the control bits in a recursive way. It starts from one of the unset swappers near an output port, say O_j, and sets O_j to *non-flip* position (line 4). Then, executing the inner *while* loop (lines 6–10), sets the configuration of $I_{\pi^{-1}(j)/2}$ by inspecting the parity of $\pi^{-1}(j)$. By looking at the permutation image of the other input to $I_{\pi^{-1}(j)/2}$, the algorithm can configure another swapper near the output ports. Therefore, the inner loop iterates over all swappers involved in a single sub-permutation, while the outer loop guarantees that all $n-1$ basic swappers pertaining to this level of the switch are traversed even if π consists of multiple sub-permutations.

The desired permutations of the component switches P_0 and P_1 are also recorded (lines 8, 10) as we set up the I, O swappers (line 7, 9). Thus, at the last two steps (lines 12–13), we only need to invoke *ConfigureWaksman* to deal with the internal swappers inside P_0 and P_1.

The entire *ConfigureWaksman* algorithm is run locally by the circuit generator in our protocol, not within a garbled circuit. It involves no cryptographic operations, so the time it takes to execute is insignificant compared to the rest of the protocol. The configuration algorithms have negligible cost since they are executed as normal (unencrypted) computations and finish in linear time.

Algorithm 2 *ConfigureWaksman*(n, π)

1: **init** Boolean arrays I, O;
2: $\pi_0 \leftarrow \phi, \quad \pi_1 \leftarrow \phi$;
3: **while** $\exists j$ such that $O_j = \perp$ **do**
4: $\quad O_j \leftarrow 0$; $\{O_j$ *defaults to non-flip*$\}$
5: $\quad via \leftarrow 0$;
6: $\quad$ **while** $I_{i/2} \neq \perp$ **do**
7: $\quad\quad [i, via] \leftarrow SetSwapper(I, j, via, \pi^{-1})$;
8: $\quad\quad \pi_0 \leftarrow \pi_0 \cup \{\pi^{-1}(j)/2 \mapsto j/2\}$;
9: $\quad\quad [j, via] \leftarrow SetSwapper(O, i, via, \pi)$;
10: $\quad\quad \pi_1 \leftarrow \pi_1 \cup \{i/2 \mapsto \pi(i)/2\}$;
11:
12: *ConfigureWaksman*$(n/2, \pi_0)$;
13: *ConfigureWaksman*$(n/2, \pi_1)$;

Algorithm 3 *SetSwapper*$(array, \iota, \varphi, \varpi)$

1: $i \leftarrow \varpi(\iota)$;
2: $array_{i/2} \leftarrow (i\%2)$ **xnor** φ;
3: **return** $[i + ((i\,\%\,2 = 1)\ ?\ 1\ :\ -1\ ), 1 - \varphi]$;

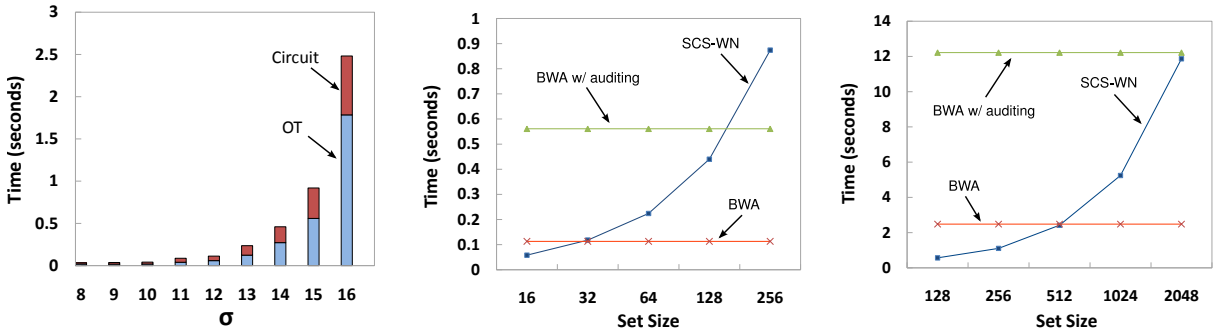
6 Auditing

An advantage of using generic garbled circuits instead of a custom protocol to perform private set intersection is the relative ease with which the generic secure computation can be combined with subsequent computations in a privacy-preserving protocol. As an example, we describe in this section how simple auditing mechanisms can be incorporated into our PSI protocols.

The result of a private set intersection intrinsically leaks a great deal of information about the private input sets, especially when σ is small enough to allow easy probing. For small enough input universes, a dishonest participant can simply set its own input to be the set containing all values in the data universe and learn the other participant's entire set. To mitigate excessive information leaks, auditing logic could be incorporated into any of the pure garbled-circuit protocols we have described.

A simple auditing policy would place a threshold on the maximum size $\hat{n}$ of the intersection that would be revealed. If the size of the result exceeds this threshold, then no output is revealed (and so the participants would only learn that the size of the intersection exceeds the allowed threshold). Such self-auditing logic would be very cheap to implement with garbled circuits, but appears to be difficult to incorporate into custom-designed PSI protocols.

As an initial study, we developed prototypes realizing the threshold-based auditing scheme just discussed. The extra work here is to obliviously calculate $\hat{n}$ (main cost) and then compare this value to a threshold using a comparison circuit (insignificant cost). For the BWA scheme, $\hat{n}$ is calculated by a Counter circuit that sums up the output bits of all the AND gates. For the SCS-* family of



(a) Performance of BWA (b) SCS-WN vs. BWA ($\sigma = 12$) (c) SCS-WN vs. BWA ($\sigma = 16$)

Fig. 8: Set intersection for small element spaces.

schemes described in Section 5, the input signals to the MUX circuits inside the duplicate-selection circuits (cf. Figures 5(a) and 5(b)) are summed using a Counter circuit. As an optimization, our Counter circuit lazily increases the number of bits used to represent its internal state. Constructing it in this way, the Counter circuit uses $n \log n - n$ non-free gates. Note that when the thresholds are known for specific applications, the cost of the Counter circuit can be cut further since there is no reason to represent results that exceed the threshold.

Our experiments show that our size-based auditing circuits incur no measurable performance overhead for the SCS-* protocols. For the BWA scheme, the cost of the auditing is significant because the underlying BWA protocol is so fast. For $12 \leq \sigma \leq 16$, adding size-based auditing increases the overall time by roughly a factor of 5 (e.g., for $\sigma = 16$, it takes 2.48 seconds to run the BWA protocol without auditing but 12.22 seconds with auditing). This is consistent with our analysis: BWA with auditing uses a total of $\sigma 2^\sigma$ non-free gates compared to 2^σ non-free gates without auditing, but the garbled-circuit portion of the basic BWA protocol constitutes only 35–45% of the total running time.

7 Experimental Results

To understand the performance of different PSI protocols, we implemented each protocol and measured its performance on a range of inputs. All experiments were done using two standard desktop computers (Dell Intel® Core™ 2 Duo E8400 3GHz) connected through a 100 Mbps LAN. Except where noted, we ran experiments at the *ultra-short* security level which corresponds to 80-bit security. That is, we use 80-bit wire labels, statistical security parameter $k = 80$ for the OT extension, and public-key parameters for the underlying Naor-Pinkas OT that correspond (according to NIST guidelines) to an 80-bit security level. In all experiments (except those where we fix the size of the intersection), both parties' sets consist of elements chosen at random (without replacement) from some fixed universe. All time measurements are the total time for the OT and garbled circuit execution, but do not include the one-time setup work for circuit object construction (about 1.2 seconds total for the most complex SCS-WN circuit) and OT extension protocol initialization (less than one second for the ultra-short security level). This time is not included in the results since (1) its cost does not depend on the size of the problem instance; and (2) it needs to be done only once for every client-server pair and circuit design.

7.1 Small Sets

We verified through experiments that the BWA protocol is indeed the best choice when the element space is small (up to about $\sigma = 20$). Figure 8 shows the running time of the BWA protocol for various sizes of σ, and compares its performance to that of the Sort-Compare-Shuffle scheme with Waksman-network shuffling (SCS-WN from Section 5.3.3), which we later show is the best protocol for larger element spaces. The BWA protocol is faster when the element space is limited but the set size is relatively large (e.g., $n > 40$ for $\sigma = 12$ and $n > 500$ for $\sigma = 16$). As discussed in the previous section, auditing approximately quintuples the running time of the BWA protocol, but has little impact for the SCS-WN protocol.

Figure 9 compares the running time for the PWC and SCS-* protocols for $\sigma = 32$ and a range of small set sizes. The running time of the BWA protocol grows exponentially in σ and so other PSI protocols, including PWC, become more attractive as σ increases. Since the PWC scheme's performance also depends on the size of the intersection, we include results for different ratios $\hat{n}/n$. In this figure, we only include the SCS-WN variant because it is the fastest of the SCS-* protocols.

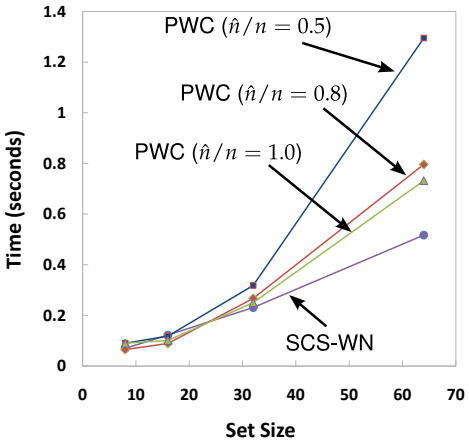


Fig. 9: Small sets, $\sigma = 32$

7.2 Large Sets

Figure 10 shows the running time and bandwidth usage of different PSI protocols running on larger sets ranging from 128 to 8192 elements, with every set element represented by a 32-bit binary string. The only protocol whose expected running time depends on the elements in the parties' sets is the pairwise-comparison-based protocol where the performance improves with the size of the intersection. For this experiment, we fixed the $\hat{n}/n$-ratio to 0.5. Note that both axes are logarithmic scale.

The SCS-WN protocol for this range of parameters is superior to all other protocols by a significant advantage: over 7× faster than SCS with homomorphic-encryption-based shuffling (SCS-HE), and 10–70× faster than PWC. Contrary to expectations, the SCS-HE protocol does not save any bandwidth compared to SCS-WN which uses Waksman-network-based shuffling. In addition, we observe that because of our use of oblivious-transfer extension and our efficient OT implementation the OT step constitutes only about 5% of the total cost. Using SCS-WN, we computed the intersection of two sets each containing more than one million 32-bit numbers ($n = 2^{20}, \sigma = 32$). This required executing a garbled circuit of 1.7 billion non-free gates, which completed in about 6 hours with each participant utilizing a single core of a typical desktop. This shows that our protocol makes large-scale privacy-preserving joint database search feasible for non-real-time applications with minimal hardware cost. When $\sigma = 160$ (effectively, $\sigma = \infty$ by first hashing elements to 160-bit strings using SHA-1), our results (see Figure 11) show that the time and bandwidth costs for SCS-WN and PWC will be about 5 times larger. Performance of the SCS-HE protocol, however, would be much less affected because the cost of HE-based shuffling, which dominates the cost of the protocol, is not affected by increasing σ from 32 to 160.

7.3 Comparison

To compare our protocols to the PSI protocols of De Cristofaro and Tsudik [10], which are the fastest known PSI protocols, we implemented their protocols in Java 1.6 and measured their performance using the same hardware and software settings as we do with our own prototypes.[3]

We ran experiments for various security levels ranging from *ultra-short* (≈80-bit effective security) to *ultra-long* (≈256-bit effective security). Our security parameters are based on NIST's guidelines for key management [2], summarized in Table 3. When we vary the security level, we change accordingly (1) the parameters for the asymmetric operations uses by the Naor-Pinkas OT; (2) the statistical security parameter used for OT extension; (3) the bit length of the wire labels; and (4) the cryptographic hash algorithms (SHA-1 is used in *ultra-short*, *short*, and *medium* term security settings while SHA-256 is used in *long* and *ultra-long* term security settings).

Figure 11 compares the performance of the fastest PSI protocol given by De Cristofaro and Tsudik[4] and our SCS-WN protocol, running on two sets of size 1024 whose elements are represented using either 32 or 160 bits. Because of the asymmetric operations, the running time of the De Cristofaro-Tsudik protocol is independent of σ in this range. As mentioned at the beginning of Section 7, the SCS-WN results do not include the one-time setup time for each client-server pair which would add about two seconds to the first protocol execution for each pair of participants. There is no comparable setup for the De Cristofaro and Tsudik protocols.

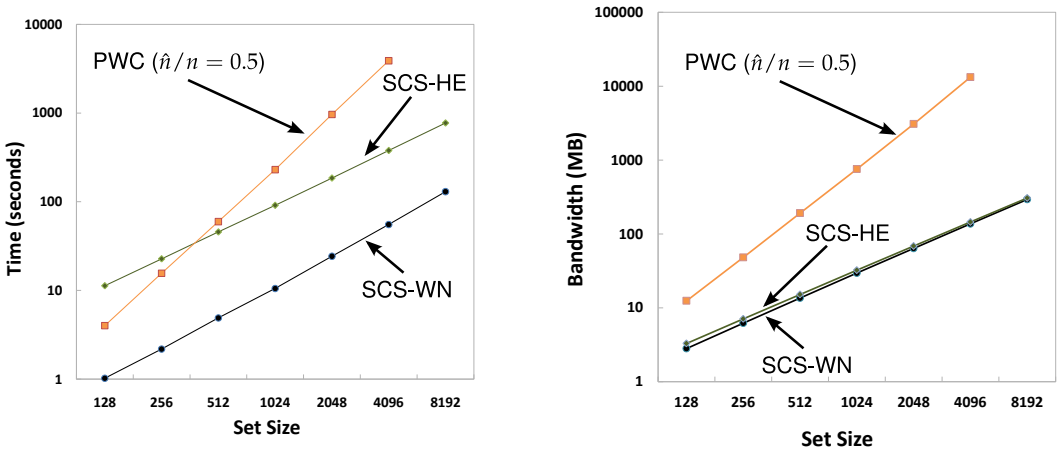
For $\sigma = 32$, our protocol is always comparable to or more efficient than theirs. For $\sigma = 160$, our protocol is competitive with theirs for all security levels, already

[3] De Cristofaro and Tsudik kindly provided us with their C implementation used to obtain experimental results for their protocols. However, they did not implement their designs as client/server protocols; instead, they implemented all steps within a single process. This makes it difficult to make direct comparisons with their results. Hence, we built our own implementation of their designs as a protocol using the same tools as are used in our garbled circuit framework.

[4] Taking total running time into account, their one-more-DL-based protocol runs faster than their one-more-RSA-based protocol. The latter may be faster, however, when some work can be pushed into an offline stage.

	Year *(valid until)*	**Symmetric key length** (bits)	**Dlog Parameters**	
			subgroup size (bits)	field size (bits)
ultra-short	2010	80	160	1024
short	2030	112	224	2048
medium	>2030	128	256	3072
long	$\gg 2030$	192	384	7680
ultra-long	$\ggg 2030$	256	512	15360

Table 3: Key lengths recommended by NIST [2].



(a) Running Time (b) Bandwidth

Fig. 10: Large sets, $\sigma = 32$

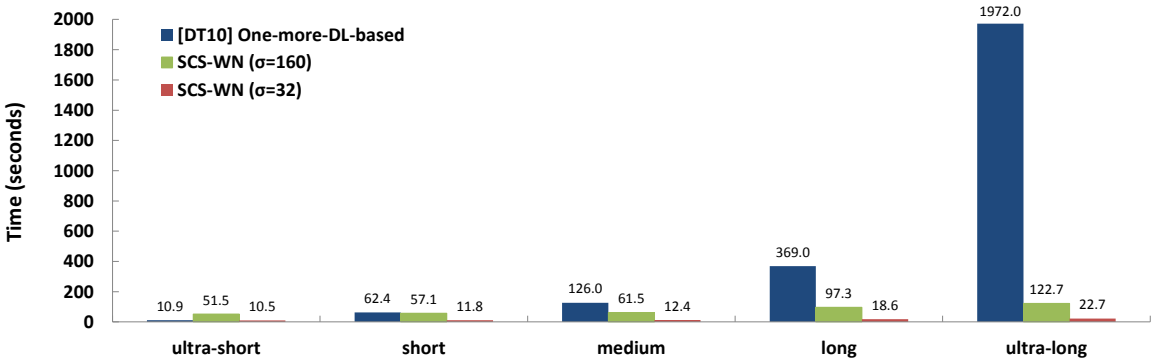


Fig. 11: Comparision of SCS-WN and De Cristofaro-Tsudik protocol [10], $n = 1024$.
SHA-1 is used for ultra-short to medium term security. SHA-256 is used for long and ultra-long term security.

	Exponentiations			**Modular Inverses**	**Modular Mults.**	**SHA**
	short exponents	medium exponents	long exponents			
DT's one-more-DL-based	–	5000	–	2049	4096	2048*
DT's one-more-RSA-based	1024	–	2048	1024	1024	2048*
SCS-WN	–	$3k^{\dagger}$	–	$2k^{\dagger}$	$2k^{\dagger}$	18.34 M‡

Table 2: Number of expensive cryptographic operations, for $n = 1024$ and $\sigma = 160$.
[†] k is the security parameter used in OT extension, which ranges from 80 (for ultra-short) to 256 (for ultra-long).
[∗] The input messages are about p bits where p is the bit length of the asymmetric operations field size.
[‡] The input messages are about 2σ bits.

slightly better at the short security level, and significantly better (by more than a factor of 10) at the highest security level. However, the SCS-WN protocol does consume more bandwidth (147–470 MB, depending on the security level) than their protocols (0.4–2.0 MB).

Since the timing results are sensitive to the implementations of particular cryptographic operations, we also calculate the numbers of expensive cryptographic operations required by each protocol. (Table 1 from the introduction summarizes algebraically the number of gates needed for each of our protocols.) Table 2 summarizes the number of cryptographic operations required for each protocol. The small number of asymmetric operations used in garbled-circuit protocols is due to the operations for setting up the OT extension protocol, which only depend on the security parameters and can be precomputed offline once for

each pair of protocol participants. The relatively high cost of asymmetric operations compared to symmetric encryptions means that even though SCS-WN requires approximately 2000 times the number of operations, the actual running time is lower or comparable for typical implementations.

8 Conclusion

Private set intersection is a useful building block for many privacy-preserving applications. Our results show that protocols based on generic secure computation can offer performance that is competitive with the best known custom protocols, without the need to rely on application-specific techniques. Since our protocols are built using generic garbled circuits they can be easily incorporated into larger secure-computation protocols, or combined with auditing mechanisms. Our work provides evidence that many secure computation problems can be solved without resorting to the design of custom protocols.

Availability

Our framework implementation and all protocol implementations are available under an open source license at http://www.MightBeEvil.com.

Acknowledgments

This work was supported by grants from the National Science Foundation, DARPA, and Air Force Office of Scientific Research. The authors thank Nikita Borisov for his very useful and insightful comments. We thank Emiliano De Cristofaro and Yanbin Lu for sharing their PSI source code and answering our questions about their implementation. We also thank Peter Chapman, Yikan Chen, Dawn Song, David Wagner, and Samee Zahur for valuable discussions about this work.

References

[1] M. Ajtai, J. Komlós, and E. Szemerédi. An $O(n \log n)$ sorting network. In *ACM Symposium on Theory of Computation (STOC)*, 1983.

[2] E. Barker, W. Barker, W. Burr, W. Polk, and M. Smid. NIST Special Publication 800-57: Recommendation for Key Management — Part 1, March 2007.

[3] K. E. Batcher. Sorting networks and their applications. In *Spring Joint Computer Conference*. ACM, 1968.

[4] M. Bellare, C. Namprempre, D. Pointcheval, and M. Semanko. The one-more-RSA-inversion problems and the security of Chaum's blind signature scheme. *Journal of Cryptology*, 16(3):185–215, 2003.

[5] V. E. Beneš. *Mathematical Theory of Connecting Networks and Telephone Traffic*. Academic Press, New York, 1965.

[6] D. Bogdanov, S. Laur, and J. Willemson. Sharemind: A framework for fast privacy-preserving computations. In *European Symposium on Research in Computer Security (ESORICS)*, 2008.

[7] J. Camenisch and G. Zaverucha. Private intersection of certified sets. In *Financial Cryptography*, 2009.

[8] D. Dachman-Soled, T. Malkin, M. Raykova, and M. Yung. Efficient robust private set intersection. In *Applied Cryptography and Network Security (ACNS)*, 2009.

[9] E. De Cristofaro, J. Kim, and G. Tsudik. Linear-complexity private set intersection protocols secure in malicious model. In *ASIACRYPT*, 2010.

[10] E. De Cristofaro and G. Tsudik. Practical private set intersection protocols with linear complexity. In *Financial Cryptography*, 2010.

[11] M. J. Freedman, K. Nissim, and B. Pinkas. Efficient private matching and set intersection. In *EUROCRYPT*, 2004.

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

[13] M. T. Goodrich. Randomized Shellsort: A simple oblivious sorting algorithm. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2010.

[14] C. Hazay and Y. Lindell. Efficient protocols for set intersection and pattern matching with security against malicious and covert adversaries. In *Theory of Cryptography Conference (TCC)*, 2008.

[15] C. Hazay and K. Nissim. Efficient set operations in the presence of malicious adversaries. In *Intl. Workshop on Public-Key Cryptography (PKC)*, 2010.

[16] W. Henecka, S. Kögl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg. TASTY: Tool for automating secure two-party computations. In *ACM Conference on Computer and Communications Security*, 2010.

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

[18] U. Inria, S. Antipolis, B. Beauquier, B. Beauquier, E. Darrot, E. Darrot, and P. Sloop. On arbitrary Waksman networks and their vulnerability. Research Report 3788, INRIA, 1999.

[19] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank. Extending oblivious transfers efficiently. In *CRYPTO*, 2003.

[20] S. Jarecki and X. Liu. Fast Secure Computation of Set Intersection. In *Security and Cryptography for Networks (SCN)*, 2010.

[21] S. Jarecki and V. Shmatikov. Efficient two-party secure computation on committed inputs. In *Eurocrypt*, 2007.

[22] S. Jha, L. Kruger, and V. Shmatikov. Towards practical privacy for genomic computation. In *IEEE Symposium on Security & Privacy*, 2008.

[23] K. V. Jónsson, G. Kreitz, and M. Uddin. Secure multi-party sorting and applications. In *Applied Cryptography and Network Security (ACNS)*, 2011.

[24] L. Kissner and D. Song. Privacy-preserving set operations. In *CRYPTO*, 2005.

[25] V. Kolesnikov, A.-R. Sadeghi, and T. Schneider. Improved garbled circuit building blocks and applications to auctions and computing minima. In *International Conference on Cryptology and Network Security (CANS)*, 2009.

[26] V. Kolesnikov and T. Schneider. Improved garbled circuit: Free XOR gates and applications. In *International Colloquium on Automata, Languages and Programming (ICALP)*, 2008.

[27] V. Kolesnikov and T. Schneider. A practical universal circuit construction and secure evaluation of private functions. In *Financial Cryptography*, 2008.

[28] Y. Lindell and B. Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In *EUROCRYPT*, 2007.

[29] Y. Lindell and B. Pinkas. A proof of security of Yao's protocol for two-party computation. *J. Cryptology*, 22(2):161–188, 2009.

[30] Y. Lindell and B. Pinkas. Secure two-party computation via cut-and-choose oblivious transfer. In *Theory of Cryptography Conference*, 2011.

[31] Y. Lindell, B. Pinkas, and N. P. Smart. Implementing two-party computation efficiently with security against malicious adversaries. In *Security and Cryptography for Networks (SCN)*, 2008.

[32] L. Malka and J. Katz. VMCrypt — modular software architecture for scalable secure computation. Available at `http://eprint.iacr.org/2010/584`.

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

[34] M. Naor and B. Pinkas. Efficient oblivious transfer protocols. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2001.

[35] M. Osadchy, B. Pinkas, A. Jarrous, and B. Moskovich. SCiFI: A system for secure face identification. In *IEEE Symposium on Security & Privacy*, 2010.

[36] B. Pinkas, T. Schneider, N. P. Smart, and S. C. Williams. Secure two-party computation is practical. In *ASIACRYPT*, 2009.

[37] C. Shen and A. Shelat. Two-output secure computation with malicious adversaries. In *EUROCRYPT*, 2011.

[38] A. Waksman. A permutation network. *J. ACM*, 15:159–163, 1968.

[39] A. C. Yao. How to generate and exchange secrets. In *Annual Symposium on Foundations of Computer Science (FOCS)*, 1986.