Secure Stable Matching at Scale

Jack Doerner
University of Virginia
jhd3pa@virginia.edu

David Evans
University of Virginia
evans@virginia.edu

abhi shelat
Northeastern University
abhi@neu.edu

ABSTRACT

When a group of individuals and organizations wish to compute a *stable matching*—for example, when medical students are matched to medical residency programs—they often outsource the computation to a trusted arbiter in order to preserve the privacy of participants' preferences. Secure multi-party computation offers the possibility of private matching processes that do not rely on any common trusted third party. However, stable matching algorithms have previously been considered infeasible for execution in a secure multi-party context on non-trivial inputs because they are computationally intensive and involve complex data-dependent memory access patterns.

We adapt the classic Gale-Shapley algorithm for use in such a context, and show experimentally that our modifications yield a lower asymptotic complexity and more than an order of magnitude in practical cost improvement over previous techniques. Our main improvements stem from designing new oblivious data structures that exploit the properties of the matching algorithms. We apply a similar strategy to scale the Roth-Peranson instability chaining algorithm, currently in use by the National Resident Matching Program. The resulting protocol is efficient enough to be useful at the scale required for matching medical residents nationwide, taking just over 18 hours to complete an execution simulating the 2016 national resident match with more than 35,000 participants and 30,000 residency slots.

1. INTRODUCTION

In 1962, David Gale and Lloyd Shapley proved that for any two sets of n members, each of whom provides a ranking of the members of the opposing set, there exists a bijection of the two sets such that no pair of two members from opposite sets would prefer to be matched to each other rather than to their assigned partners [15]. A set of pairings that satisfies this property is known as a *stable matching*; it can be computed using an algorithm that Gale and Shapley developed.

Fifty years later, the development of a theory of stable matching and the exploration of its practical applications would win Shapley and Alvin Roth the Nobel Prize in Economics [53]. Today, stable matching algorithms are used to match medical residents to residency programs [41], students to schools [1, 54], candidates to sororities [37], to run special types of auctions [3], and to manage supply chains [42].

In practice, stable matching processes are often outsourced to trusted arbiters in order to keep the participants' reported preferences private. We consider how to run instances of stable matching using secure multi-party computation, so that participants' privacy and confidence in the results can be maintained without relying on a single common trusted party. We express the problem as a two-party secure computation in which all members of the pairing sets trust two representatives to execute on their behalf without colluding. The participants can XOR-share their preference lists between the two representatives so that even the trusted representatives learn nothing about the preferences of any participant.

Executing an algorithm as complex and data-dependent as the Gale-Shapley stable matching algorithm as a secure computation has been a longstanding goal. Secure computation requires that all data-dependent memory accesses be hidden in order to maintain privacy; this has traditionally been a significant contributor to the inefficiency of secure computation relative to its insecure counterpart. For example, the protocols of Golle [19] and Franklin *et al.* [13] required roughly $O(n^5)$ and $O(n^4)$ public-key operations respectively and were too complicated to implement.

Recent advances in ORAM design [56, 63] have reduced costs significantly, but have not yielded solutions scalable enough for interesting matching problems. Using a state-of-the-art ORAM, the best previous implementation of Gale-Shapley still required over 33 hours to match 512×512 participants [63]. We overcome this barrier by combining general-purpose ORAMs with special-purpose constructs to create efficient oblivious data structures that leverage data partitions and memory access patterns inherent to the problem to restrict the ways in which the data can be accessed without leaking any data-dependent access information.

Contributions. The primary contributions of this paper are the development of strategies for RAM-based secure computation for algorithms that predominantly access memory in a data-dependent, but "read-once" fashion. In particular, we present the design of an oblivious linked list structure that can be used when the order in which data is accessed must be hidden, but it is known that each element is accessed at most once; we further refine this structure to support multiple lists in order to enable more complex access patterns (Section 3.1). We also introduce a modification to the ORAM access protocol that enables efficient function application within an ORAM access (Section 4.2).

These techniques are developed and evaluated in the context of two secure stable matching algorithms, but we believe they have wider applicability to constructing secure variants of many algo-

CCS'16, October 24 - 28, 2016, Vienna, Austria

ISBN 978-1-4503-4139-4/16/10...$15.00

DOI: http://dx.doi.org/10.1145/2976749.2978373

rithms that involve data-dependent memory access. Our secure Gale-Shapley implementation exhibits the best asymptotic performance of any yet developed, and it is over 40 times faster in practice than the best previous design. We also develop the first ever secure version of the instability chaining algorithm used in most practical stable-matching applications, including the national residency match. We evaluate our protocol by simulating the 2016 US residency match and find that the total execution cost using commodity cloud resources is less than $16.

2. BACKGROUND

Our secure stable matching protocols build on extensive prior work in secure multi-party computation and RAM-based secure computation, which we briefly introduce in this section.

Multi-Party Computation. Secure multi-party computation [17, 59] enables two or more parties to collaboratively evaluate a function that depends on private inputs from all parties, while revealing nothing aside from the result of the function. Generic approaches to multi-party computation (MPC) can compute any function that can be represented as a Boolean-circuit. Our experiments use Yao's garbled circuit protocol [33, 58], although our general design is compatible with any Boolean-circuit based MPC protocol.

Garbled Circuits. Garbled circuits protocols involve two parties known as the *generator* and *evaluator*. Given a publicly known function f, the generator creates a garbled circuit corresponding to f and the evaluator evaluates that circuit to produce an output that can be decoded to the semantic output. Although garbled circuits were once thought to be of only theoretical interest, recent works have shown that such protocols can be practical [23, 24, 25, 30, 36, 43], even in settings where full active security is required [2, 7, 14, 22, 26, 30, 32, 34, 35]. Current implementations [8, 11, 38, 62] can execute approximately three million gates per second over a fast network (using a single core for each party).

RAM-based Secure Computation. In traditional MPC, general input-dependent array access incurs a linear-time overhead since all elements in the array need to be read to hide the position of interest. RAM-based secure computation combines circuit-based MPC with oblivious random-access memory (ORAM) to enable secure random memory accesses in sublinear time [18, 20]. An ORAM scheme consists of an *initialization protocol* that accepts an array of elements and initializes a new oblivious structure with those elements, and an *access protocol* that performs each logical ORAM access using a sequence of physical memory accesses. To be secure, an ORAM must ensure that for any two input arrays of the same length, the physical access patterns of the initialization protocol are indistinguishable, and that for any two sequences of semantic accesses of the same length, the physical access patterns produced by the access protocol are indistinguishable.

To use ORAM in secure computation, the parties run a secure-computation protocol to store *shares* of the state of the underlying ORAM protocol, and then use circuit-based secure computation to execute the ORAM algorithms [20]. For each memory access, the circuit obliviously translates a secret logical location into a set of physical locations that must be accessed. The ORAM's security properties ensure that these physical locations can be revealed to the two parties without leaking any private information, and the data stored at those locations can be passed back into the circuit for use in the oblivious computation.

Several ORAM designs for secure computation have been proposed [12, 16, 20, 27, 57] which offer various trade-offs in initialization cost, per-access cost, and scalability. The ORAM with the best asymptotic per-access cost to date is Circuit ORAM [56]; the most efficient in practice over a wide range of parameters is Square-Root ORAM [63]. We evaluate both experimentally in Section 5.

3. SECURE GALE-SHAPLEY

We first consider the structure of the standard Gale-Shapley algorithm, typically presented via a process in which proposers (members of set A) present pairings to reviewers (members of set B), who may accept or reject them. The inputs are the lists of preferences for each participant. For the secure two-party version, these lists are divided among two parties either by partitioning the lists or XOR-sharing the entries.

The algorithm steps through each proposer's preference list from most to least-preferred, swapping between proposers as they become matched or invalidated by other matches. This algorithm requires that the sizes of the proposer and reviewer sets are equal, ensuring that everyone ends up part of some pair. We use n to denote the size of these sets. The algorithm iterates over at most n^2 potential pairings, but, critically, it cannot determine in advance which proposer's preferences will be evaluated, nor how far along that proposer's preference list it will have advanced at any point.

As any iteration could require access to any pairing, a straightforward approach is to store the preferences in an ORAM. Such an implementation would require n^2 accesses to an ORAM of length n^2. This would dominate the overall cost, since all other ORAMs and queues required by the textbook algorithm are of length n. Thus, our design focuses on reducing the costs of reading the preferences.

Notation. We use $\langle x \rangle$ to indicate a variable which is secret-shared between multiple parties. We refer to this state interchangeably as "oblivious", "private", and "garbled". The garbled variable $\langle x \rangle$ is distinct from the variable x, which is public. Arrays have a public length and are accessed via public indices; we use $\langle \mathsf{Array} \rangle$ to denote an array of oblivious data, $\langle \mathsf{Array} \rangle_i$ to specify element i within that array, and $\langle \mathsf{Array} \rangle_{i:j}$ to indiciate an array slice containing elements i through j of $\langle \mathsf{Array} \rangle$, inclusive. We indicate multidimensional array access with multiple indices delimited by commas. Conditionals on secret values are indicated using $\langle$**if**$\rangle$ and $\langle$**else**$\rangle$. The instructions within oblivious conditionals are always executed, but have no effect if the condition is false.

3.1 Oblivious Linked Multi-lists

We observe that in the Gale-Shapley algorithm, each proposer's *individual* preference list is accessed strictly in order, and each element is accessed only once. Furthermore, a secure implementation of Gale-Shapley does not involve any accesses that depend on oblivious conditions (the algorithm must obliviously select *which* preference list is accessed on each iteration, but exactly one preference list is always accessed). Instead of using a generic ORAM, we design a new data structure to satisfy these requirements more efficiently, which we call an *oblivious linked multi-list*.

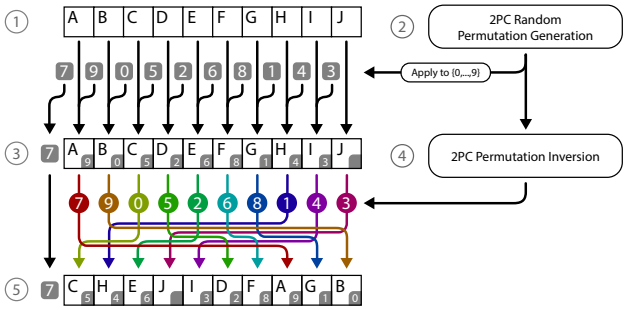
The oblivious linked multi-list is designed to be able to iterate independently through n separate arrays, each containing an arbitrary (and not necessarily uniform) number of elements, while hiding which of its component arrays is currently being iterated, and the iteration progress of all component arrays. It is defined by two algorithms: InitializeMultilist and TraverseMultilist, shown in pseudocode in Figure 1 and illustrated in Figure 2.

The InitializeMultilist algorithm takes as input a single array of garbled data, comprising a concatenation of the n component lists. In addition, it takes an array of public entry pointers (i.e. the indices of the first elements of each of the component lists in the input array). It returns a $\langle$multilist$\rangle$ data object.

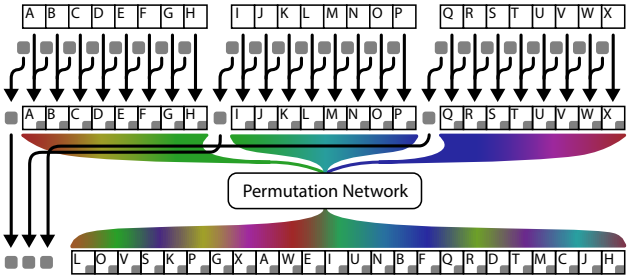
```
define InitializeMultilist(⟨data⟩, entryIndices):
    ⟨π⟩ ← random permutation on |⟨data⟩| elements.
    ⟨π^{-1}⟩ ← InvertPermutation(⟨π⟩)
    ⟨multilist⟩ ← ∅
    ⟨entryPointers⟩ ← ∅
    for i from 0 to |⟨data⟩| − 1:
        if i ∈ entryIndices:
            ⟨entryPointers⟩ ← ⟨entryPointers⟩ ∪ {⟨π^{-1}⟩_i}
        ⟨multilist⟩_i ← {⟨data⟩_i, ⟨π^{-1}⟩_{i+1}}
    ⟨multilist⟩ ← Permute(⟨multilist⟩, ⟨π⟩)
    return {⟨multilist⟩, ⟨entryPointers⟩}

define TraverseMultilist(⟨multilist⟩, ⟨p⟩):
    p ← Reveal(⟨p⟩)
    return ⟨multilist⟩_p
```

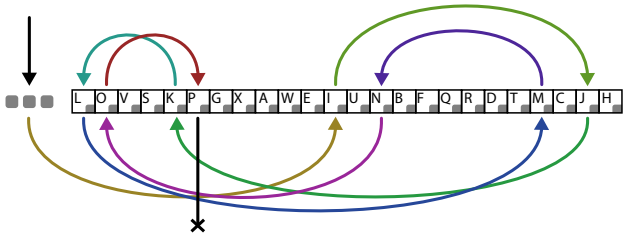
Figure 1: **Oblivious Linked Multi-List.** Pseudocode for initialization and traversal.



(a) **Initialization.** The input ① is combined with a random permutation ② which has been shifted left by one place. The result ③ is then permuted according to the inverse of the first permutation ④, resulting in a shuffled linked list ⑤. The leftmost element of the first random permutation ② is retained outside the structure and represents the entry point.



(b) **Interleaving multiple arrays to form an oblivious linked multi-list.** Multiple arrays can be concatenated and permuted together, becoming indistinguishable from one another. Individual entry points allow them to be independently traversable.



(c) **Traversal of one of the interleaved lists.** Each element contains a reference to the next element in the sequence. After the last element in the original sequence, traversal cannot continue.

Figure 2: **Illustrations of the Oblivious Linked List structure.**

To explain the initialization procedure, we first consider an *oblivious linked list* that can iterate over only a single component array. To construct an oblivious linked list, we generate a random oblivious permutation and its inverse using the method of Zahur *et al.* [63]. The forward permutation comprises one set of Waksman control bits from each party, and the inverse permutation is stored as an array mapping one set of indices to another. To each element i of the data array, we append element $i+1$ of the inverse permutation, which corresponds to the physical index of element $i+1$ of the permuted data array. We then apply the permutation to the data array using a Waksman Network [55], and store the first element of the inverse permutation (the *entry pointer*) in a variable. Both the permuted data array and the entry pointer are returned. This process is illustrated in Figure 2a.

We can now extend our oblivious linked list into an oblivious linked multi-list by permuting multiple input arrays together, and storing the garbled entry pointers for each in a separate data structure. This is illustrated in Figure 2b.

The TraverseMultilist algorithm takes as input a $\langle$multilist$\rangle$ data object and a garbled pointer, $\langle p \rangle$. Its operation is simple: it reveals the contents of $\langle p \rangle$ and selects the data element at the physical index indicated thereby. This physical element will contain the requested semantic element, as well as a garbled pointer, $\langle p' \rangle$, to its successor, both of which are returned to the caller. The complete traversal of one component list in a linked multi-list is illustrated in Figure 2c.

Performance. Initializing a linked multi-list requires executing a Waksman network, at a cost of $\Theta(n \log n)$. Iteration can be performed in constant time; therefore, the amortized cost is $\Theta(\log n)$ per element. However, pointers to the current positions in the component lists must be stored in some structure external to the multi-list itself, and in many cases this will incur additional costs. We will use an ORAM and an oblivious queue serve this purpose, as described in Section 3.2.

3.2 Applying Our Construction

In Secure Gale-Shapley, we use our multi-list to hold the joint preferences list for the proposers and reviewers, subdivided by proposer ID into n lists of n elements ordered by proposer preference. We use an ORAM of length n to store current matches by reviewer ID, and an oblivious queue [60] to store unmatched proposers. Garbled pointers into the multi-list representing the iteration progress for each proposer are stored in the match status ORAM for matched proposers, and in the unmatched proposer queue otherwise. Complete pseudocode is given in Figure 3.

Initialization. As input to InitializeMultilist we must provide a master preferences list, containing all potential pairings ordered first by proposer ID and then by proposer rank. Proposer rank, however, is secret, and therefore we cannot expect to be able to collect preferences ordered in that fashion. On the other hand, the Gale-Shapley algorithm requires that participants must express preferences over all members of the opposite set, so it is reasonable for preferences to be submitted pre-sorted first by proposer ID and then by reviewer ID. With preference data submitted pre-sorted in this fashion, it is trivial to merge it into a single master preferences list,

Figure 3: **Secure Gale-Shapley Algorithm.** SecureGaleShapley expects to ingest preferences ordered first by proposer index, then by reviewer index. It returns an array of proposer indices, ordered by the reviewer indices to which the proposers have been paired.

whereupon we can apply n Batcher sorts [4], one for each proposer, to re-order it as necessary for InitializeMultilist.

Early Termination. In practice, executions of the algorithm rarely require the worst case n^2 iterations. As a heuristic, one can execute fewer rounds and test whether a matching has been computed. If a matching is not found, information about the preferences has been leaked, though it is difficult to understand exactly what information has been leaked about the participants. We leave the analysis of these heuristics to future work, and presume for the purposes of our algorithm that exactly n^2 iterations must always be performed.

Even after a stable matching is found, each iteration must reveal exactly one uniform unused physical index from the multi-list containing the preferences. This is a problem, as there is no easy way to select a uniform unvisited element from the entire multi-list. Our solution is to pad the preference list array with $n^2 - n$ dummy blocks, linked as a single chain and intermingled with the rest of the preferences array during the permutation phase. We store only a garbled copy of the physical index of the first padding block. The algorithm takes between n and n^2 iterations to find a stable matching, after which it follows the dummy chain.

3.3 Security

The security of our scheme depends on established properties of ORAM constructions [63], oblivious queues [60], and the underlying garbled circuits protocol [33]. We do not modify these in any way that alters their security properties, which provide the privacy and correctness guarantees desired for our protocol. The variants we use are secure only in an honest-but-curious setting, where "semi-honest" computation parties follow the algorithm correctly but wish to learn any sensitive information they can from its execution, and eavesdroppers can witness the entire protocol, but cannot affect it. The honest-but-curious setting is admittedly a very weak adversarial model, but there has been substantial work showing that semi-honest protocols can be adapted to resist active adversaries. We leave to future work the challenge of hardening our protocols to resist such adversaries. Many of the scenarios where secure stable matching might be used do involve professional organizations and government agencies as participants, who may be sufficiently trusted to be considered semi-honest.

The only element used by our protocol which has not been evaluated elsewhere is the oblivious linked multi-list used to hold participant preferences. Our structure reveals the index of one uniform untouched element from the preferences array on each access. Intuitively, the index revealed is not correlated with the contents of the target element, and no element is accessed (nor is any index revealed) more than once, and thus our modification only leaks that the preferences list is a permutation, which is already known. It does not leak anything about the preferences themselves or about the current state of the algorithm. The permutation generation and inversion processes we use are secure against semi-honest adversaries [63]. By replacing these and using malicious-secure ORAM and MPC protocols, our design could be adapted to achieve security against malicious adversaries.

3.4 Complexity Analysis

The textbook Gale-Shapley algorithm performs $\Theta(n^2)$ operations upon an $\Theta(n^2)$-length memory that holds the matrix of participant preferences. A naïve Secure Gale Shapley implementation based

upon Linear Scan incurs a total complexity of $\Theta(n^4)$. Square-Root ORAM has an asymptotic access complexity in $\Theta(\sqrt{n \log^3 n})$ [63], so a straightforward implementation based upon that construction has a complexity in $\Theta(n^3 \log^{1.5} n)$.

In contrast, our Secure Gale-Shapley algorithm performs $\Theta(n^2)$ operations upon a $\Theta(n)$-length memory. This results in a total complexity in $\Theta(n^{2.5} \log^{1.5} n)$ when using a Square-Root ORAM. The asymptotic complexity can be further reduced to $\Theta(n^2 \log^3 n)$ by using Circuit ORAM [56], which has an asymptotic access complexity in $\Theta(\log^3 n)$. For any stable matching problem that can feasibly be solved today, however, Square-Root ORAM is more efficient in practice because of its lower concrete costs. This is confirmed by our experimental results in Section 5.

4. SECURE INSTABILITY CHAINING

In the 1940s, the job market for medical residents in the United States underwent a crisis [45]. Demand for residents was rapidly outstripping supply, leading to excessive competition among hospitals and fostering applicant-hostile practices such as extremely time-limited employment offers. In reaction to applicant protests, the medical community formed a central clearinghouse, now known as the *National Resident Matching Program* (NRMP) to allocate graduates to residencies. As the supply of residents grew to exceed the number of available positions, it became apparent that the original matching algorithm, which had been designed to produce results favorable to hospitals, was unfair to aspiring residents. In response, the NRMP commissioned the design of a new algorithm for resident-optimal matching, based upon the theory of Stable Matching. Roth proposed an algorithm [46] based upon his earlier inductive proof of stability with John H. Vande Vate [49].

Roth's algorithm, shown in Figure 4, follows a process he called *instability chaining*: the algorithm finds stable matchings among subsets of proposers, starting from the empty set and adding new proposers one at a time, resolving any new instabilities as they are introduced. For problem instances that fit the requirements of the Gale-Shapley algorithm, Roth's algorithm produces the same result. However, it also supports one-to-many matchings, and cases in which the sizes of the sets are unequal and not all participants are ranked. Roth and Peranson [48] described the algorithm and evaluated it experimentally using data from past NRMP matches.

4.1 Secure Roth-Peranson

Unlike Gale-Shapley, the Roth-Peranson algorithm does not mandate that each participant rank all participants from the opposing set. Indeed, it is expected that the number of counterparties ranked by most participants will be small relative to the number of counterparties available. Consequently, we establish a public bound on the number of rankings that each participant can input, indicated by q for the proposers, and r for the reviewers. In addition, we establish a public bound on the number of positions each reviewer has to fill, which we indicate as s. In many real-world instances the individual position quotas are public knowledge, but we do not require this. We use n and m to represent the numbers of proposers and reviewers, respectively.

The adaptations required to create a secure version of the Roth-Peranson matching algorithm are similar to those described for Secure Gale-Shapley in Section 3. To track the tentative matches for each reviewer, we need an ORAM with m elements, each element being a list of the corresponding reviewer's matches. As this list is stored *within* the ORAM, we have implemented it as an ordinary garbled array. Consequently, the reviewer-status ORAM has m elements, each of size s. Pseudocode for our Secure Roth-Peranson algorithm is found in Figure 5.

```
define RothPeranson(ProposerPrefs, ReviewerPrefs, RPosCounts, n, m):
    ProposerPrefsPosition ← ∅
    RMatches ← ∅
    for i from 0 to n − 1:
        ProposerPrefsPosition_i ← 0
    for i from 0 to m − 1:
        RMatches_i ← ∅
    for si from 0 to n − 1:
        while true:
            sr ← ProposerPrefsPosition_si
            ProposerPrefsPosition_si ← ProposerPrefsPosition_si + 1
            if sr ∉ ProposerPrefs_si:
                break
            ri ← ProposerPrefs_si,sr
            if si ∉ ReviewerPrefs_ri:
                break
            rr ← ReviewerPrefs_ri,si
            if |RMatches_ri| < RPosCounts_ri:
                RMatches_ri ← RMatches_ri ∪ {si}
                break
            wi ← 0
            w ← RMatches_ri,0
            for j from 1 to RPosCounts_ri − 1:
                if ReviewerPrefs_ri,RMatches_ri,j > ReviewerPrefs_ri,w:
                    wi ← j
                    w ← RMatches_ri,j
            if rr < ReviewerPrefs_ri,w:
                RMatches_ri,wi ← si
                si ← w
    return RMatches
```

Figure 4: **Standard Roth-Peranson Algorithm.** RothPeranson expects to ingest proposer preferences as a dense multidimensional array ordered first by proposer index, then by proposer rank, and reviewer preferences as a sparse multidimensional array ordered first by reviewer index, then by proposer index. It returns an array of sets of proposer indices ordered by the reviewer indices to which the proposers have been paired.

Initialization. Unlike Gale-Shapley, the Roth-Peranson algorithm expects participants to express preferences over only a subset of their counterparties. While this permits the combined master preference list to be much smaller than otherwise, it also prevents us from constructing it by simply concatenating and interleaving individual preference lists as we could in the case of Gale-Shapley. Instead, we specify that the algorithm takes participant preferences inputs in the form of two master lists: one each for the proposers (of size nq) and reviewers (of size mr). Both lists are sorted first by proposer index, then reviewer index, and only ranked pairings are included. Each element will contain as garbled data both the proposer and reviewer indices, a rank, and a bit indicating whether the preference belongs to a proposer or reviewer. We combine the two master preference lists using a Batcher merge [4]. We then iterate over the combined list and check each sequential pair: if a pair shares proposer and reviewer indices, we push their combined data into a queue. In this way, unrequited preferences are omitted. We flatten the queue into an array containing q elements for each of the n proposers by conditionally popping elements or inserting dummies as appropriate. We then sort each group of q elements according to proposer rank, yielding the final preference array which is used to initialize an oblivious multi-list.

XOR-Sharing. If preferences are to be split among the computation parties by XOR-sharing, there is an additional problem that must be solved. It is reasonable to expect each participant to submit their preferences sorted by counterparty ID. This means that

```
define CmpSortOnIndices(⟨a⟩, ⟨b⟩):
    ⟨result⟩ ← Cmp(⟨a⟩.⟨si⟩, ⟨b⟩.⟨si⟩)
    ⟨if⟩ ⟨result⟩ = 0: ⟨result⟩ ← Cmp(⟨a⟩.⟨ri⟩, ⟨b⟩.⟨ri⟩)
    ⟨if⟩ ⟨result⟩ = 0: ⟨result⟩ ← Cmp(⟨a⟩.⟨is_reviewer⟩, ⟨b⟩.⟨is_reviewer⟩)
    return ⟨result⟩

define SecureRothPeranson(⟨ProposerPrefs⟩, ⟨ReviewerPrefs⟩, ⟨RPositionBounds⟩, n, m, q, r, s):
    ⟨CollationQueue⟩ ← new oblivious queue of n∗q elements
    ⟨MergedPrefs⟩ ← BatcherMerge(⟨ProposerPrefs⟩, ⟨ReviewerPrefs⟩, CmpSortOnIndices)
    for i from 0 to n∗q+m∗r−2:
        ⟨if⟩ ⟨MergedPrefs⟩_i.⟨si⟩ = ⟨MergedPrefs⟩_{i+1}.⟨si⟩ ∧ ⟨MergedPrefs⟩_i.⟨ri⟩ = ⟨MergedPrefs⟩_{i+1}.⟨ri⟩:
            ⟨CombinedPref⟩ ← { ⟨si⟩ ← ⟨MergedPrefs⟩_i.⟨si⟩, ⟨ri⟩ ← ⟨MergedPrefs⟩_i.⟨ri⟩,
                               ⟨sr⟩ ← ⟨MergedPrefs⟩_i.⟨rank⟩, ⟨rr⟩ ← ⟨MergedPrefs⟩_{i+1}.⟨rank⟩ }
            ⟨CollationQueue⟩ ← QueuePush(⟨CollationQueue⟩, ⟨CombinedPref⟩)
    ⟨Prefs⟩ ← ∅
    for i from 0 to n−1:
        for j from 0 to q−1:
            ⟨if⟩ QueuePeek(⟨CollationQueue⟩).⟨si⟩ = i:
                ⟨Prefs⟩_{i∗q+j} ← QueuePop(⟨CollationQueue⟩)
            ⟨else⟩:
                ⟨Prefs⟩_{i∗q+j} ← {⟨si⟩ ← i, ⟨ri⟩ ← ∅, ⟨sr⟩ ← ∞, ⟨rr⟩ ← ∅}
        ⟨Prefs⟩_{i∗q:(i+1)∗q−1} ← BatcherSort(⟨Prefs⟩_{i∗q:(i+1)∗q−1}, CmpSortOnRanks)
    {⟨multilist⟩, ⟨entryPointers⟩} ← InitializeMultilist(⟨Prefs⟩, {0, q, 2q, ..., n∗q})
    UnmatchedProposers ← new oblivious queue of n elements
    for i from 0 to n−1:
        UnmatchedProposers ← QueuePush(UnmatchedProposers, ⟨entryPointers⟩_i)
    ⟨dummy⟩ ← ⟨entryPointers⟩_n
    ⟨done⟩ ← false
    RMatches ← new ORAM of m elements
    for i from 0 to m−1:
        RMatches ← OramWrite(RMatches, {⟨s⟩ ← ⟨RPositionBounds⟩_i, ⟨matches⟩ ← ∅}, i)
    ⟨p⟩ ← QueuePop(UnmatchedProposers)
    for i from 0 to n∗q−1:
        {⟨ProposedPair⟩, ⟨p′⟩} ← TraverseMultilist(⟨multilist⟩, ⟨p⟩)
        ⟨if⟩ ⟨done⟩ = true:
            ⟨p⟩ ← ⟨p′⟩
        ⟨else⟩:
            ⟨if⟩ ⟨ProposedPair⟩.⟨ri⟩ ≠ ∅:
                ⟨ProposedReviewer⟩ ← OramRead(RMatches, ⟨ProposedPair⟩.⟨ri⟩)
                for j from 0 to s−1:
                    ⟨if⟩ j ≤ ⟨ProposedReviewer⟩.⟨s⟩:
                        {⟨tentativeMatch⟩, ⟨p″⟩} ← ⟨ProposedReviewer⟩.⟨matches⟩_j
                        ⟨if⟩ ⟨tentativeMatch⟩ = ∅ ∨ ⟨tentativeMatch⟩.⟨rr⟩ > ⟨ProposedPair⟩.⟨rr⟩:
                            ⟨ProposedReviewer⟩.⟨matches⟩_j ← {⟨ProposedPair⟩, ⟨p′⟩}
                            {⟨ProposedPair⟩, ⟨p′⟩} ← {⟨tentativeMatch⟩, ⟨p″⟩}
            ⟨if⟩ ⟨ProposedPair⟩.⟨ri⟩ = ∅:
                ⟨if⟩ QueueEmpty(UnmatchedProposers):
                    ⟨p⟩ ← ⟨dummy⟩
                    ⟨done⟩ ← true
                ⟨else⟩:
                    ⟨p⟩ ← QueuePop(UnmatchedProposers)
            ⟨else⟩:
                ⟨p⟩ ← ⟨p′⟩
    ⟨Result⟩ ← ∅
    for i from 0 to n−1:
        ⟨Result⟩_i ← OramRead(RMatches, i).⟨matches⟩
    return ⟨Result⟩
```

Figure 5: **Secure Roth-Peranson Algorithm.** SecureRothPeranson expects to ingest preferences ordered first by proposer index, then by reviewer index. It returns an array of sets of proposer indices, ordered by the reviewer indices to which the proposers have been paired. Highlighting indicates each of the phases of the main algorithm as laid out in Section 4.4: setup, permutation, and proposal/rejection.

the proposers will submit preference lists sorted by reviewer ID, and the master proposer preference list can be created by concatenation. The reviewer master preference list, however, must also be sorted first by proposer ID, then reviewer ID. Because individual reviewer preference lists will be sparse, this ordering cannot be achieved by blind interleaving, and because the counterparty IDs will be hidden, it cannot be achieved outside of the protocol. Therefore, we must create the master reviewer preference list inside of the protocol by way of repeated Batcher merges: we merge pairs of individual preference lists, yielding half as many lists, each of twice the original length. We repeat the process until a single, correctly ordered master reviewer preference list remains. The cost for this process is $\Theta(\sum_{i=1}^{\log m} mr \log 2^i r)$, which reduces to $\Theta(mr \log^2 m + mr \log m \log r)$. This is better than the $\Theta(mr \log^2 mr)$ cost that would be incurred by re-sorting all of the elements.

4.2 Improving ORAM Access

Although most ORAM schemes are compatible with our construction, we use Square-Root ORAM [63], and take advantage of function application to reduce the number of ORAM accesses required. An ordinary ORAM access will perform some number of conditional oblivious copies between its data and an external lo-

cation, after which the desired element will have been retrieved. To store the element back after modification, another sequence of copies must be performed. Instead, we apply a conditional oblivious function to each element that would have been copied, obviating the second set of copies. This works well when the function to be applied is simple, but the design of Zahur *et al.* [63] requires $\Theta(T)$ copies per access, and therefore $\Theta(T)$ function applications, where T is the ORAM refresh period. For a function such as the one we use, which has a complexity in $\Theta(s)$ (incurred by linearly scanning the tentative matches stored within each ORAM element), the number of extra gates is significant. To avoid this inefficiency, we modify the ORAM access protocol to allow function application with only a single execution of the function circuit.

ORAM Background and Notation. Square-Root ORAM stores its data in $\mathsf{Oram}.\langle\mathsf{Shuffle}\rangle$, shuffled according to some secret permutation. Each data element retains a copy of its logical index; the logical index of the element with physical index i can be accessed via $\mathsf{Oram}.\langle\mathsf{Shuffle}\rangle_i.\langle index\rangle$. The ORAM uses a recursive position map structure, $\mathsf{Oram}.\mathsf{Posmap}$, to relate physical indices in $\mathsf{Oram}.\langle\mathsf{Shuffle}\rangle$ to logical indices, so that elements can be accessed without scanning. As each element is accessed, the ORAM moves it from $\mathsf{Oram}.\langle\mathsf{Shuffle}\rangle$ to $\mathsf{Oram}.\langle\mathsf{Stash}\rangle$, where it will be linearly scanned on subsequent accesses. The ORAM tracks which physical indices in $\mathsf{Oram}.\langle\mathsf{Shuffle}\rangle$ have been accessed using a set of public Booleans, $\mathsf{Oram}.\mathsf{Used}$. After $\mathsf{Oram}.T$ accesses, the ORAM is refreshed and the process starts again from the beginning; progress toward the refresh period is tracked via $\mathsf{Oram}.t$. Φ indicates the function to be applied.

Construction. We designate $\mathsf{Oram}.\langle\mathsf{Stash}\rangle_0$ to be the active element location: whichever ORAM element will be accessed must be moved into this slot, and the function Φ is applied to it at the end. The last active element remains in this slot between accesses. On the next access it must be mixed back into the $\langle\mathsf{Stash}\rangle$. This arrangement has the additional advantage, unused by our algorithm, of allowing the most-recently accessed block to be accessed repeatedly at no additional expense (so long as it can be publicly revealed that accesses are repeated).

An access proceeds as follows. If $\mathsf{Oram}.t$ is zero, then we know that the element we need cannot be in $\mathsf{Oram}.\langle\mathsf{Stash}\rangle$. Otherwise, we scan the stash and use a conditional oblivious swap circuit [29] to exchange each element with the element in $\mathsf{Oram}.\langle\mathsf{Stash}\rangle_0$, conditioned on the currently-scanned element having the target logical index. If the target element was not found during the stash scan, it will be retrieved from $\mathsf{Oram}.\langle\mathsf{Shuffle}\rangle$, but before that can happen we must provide a blank space for it by moving the element in $\mathsf{Oram}.\langle\mathsf{Stash}\rangle_0$ to an empty slot at the end of the $\langle\mathsf{Stash}\rangle$.

Next, regardless of whether the target element has been found thus far, we query $\mathsf{Oram}.\mathsf{Posmap}$ for its position in $\mathsf{Oram}.\langle\mathsf{Shuffle}\rangle$. If the target element has already been found, the position map will return the physical index of a random unvisited element, which is moved to an empty slot at the end of $\mathsf{Oram}.\langle\mathsf{Stash}\rangle$. If the target element has *not* been found so far, then the index returned from the position map will locate it, and we can move it to $\mathsf{Oram}.\langle\mathsf{Stash}\rangle_0$. Finally, we apply Φ to the element located in $\mathsf{Oram}.\langle\mathsf{Stash}\rangle_0$, which will be the target element. Pseudocode for our access function is shown in Figure 7, with Zahur et al.'s original access function in Figure 6 for comparison.

4.3 Security

With the exception of the modified ORAM access method described in Section 4.2, our Secure Roth-Peranson protocol uses the same oblivious data structures and underlying protocols as our Secure Gale-Shapley protocol. The assumptions and security argument from Section 3.3 apply to these elements.

The security property an ORAM access method must establish is that any two same-length access sequences exhibit observable memory patterns that are indistinguishable. This property holds for our new access method, as it does for the original. In the first stage of an access, the original algorithm scans $\mathsf{Oram}.\langle\mathsf{Stash}\rangle$ and applies a function to any element matching the desired index. Our access method performs a similar process, applying a swap circuit with $\mathsf{Oram}.\langle\mathsf{Stash}\rangle_0$ as its second input in place of an arbitrary function. After the stash scan, it either moves an element from $\mathsf{Oram}.\langle\mathsf{Stash}\rangle_0$ to $\mathsf{Oram}.\langle\mathsf{Stash}\rangle_t$ and copies an element from $\mathsf{Oram}.\langle\mathsf{Shuffle}\rangle_p$ to $\mathsf{Oram}.\langle\mathsf{Stash}\rangle_0$, or copies an element from $\mathsf{Oram}.\langle\mathsf{Stash}\rangle_p$ to $\mathsf{Oram}.\langle\mathsf{Stash}\rangle_t$. These operations are performed within an oblivious conditional, so both code paths appear to execute regardless of which takes effect. Finally, the function Φ is applied to a single block at a fixed physical index. The observable memory behavior of this algorithm depends only

```
define Access(Oram, ⟨i⟩, Φ)
    ⟨found⟩ ← false
    for j from 0 to Oram.t:
        ⟨if⟩ Oram.⟨Stash⟩_j.⟨index⟩ = ⟨i⟩:
            ⟨found⟩ ← true
            Φ(Oram.⟨Stash⟩_j)
    p ← GetPos(Oram.Posmap, ⟨i⟩, ⟨found⟩)
    ⟨if⟩ not ⟨found⟩:
        Φ(Oram.⟨Shuffle⟩_p)
    Oram.⟨Stash⟩_t ← Oram.⟨Shuffle⟩_p
    Oram.Used ← Oram.Used ∪ {p}
    Oram.t ← Oram.t + 1
    if Oram.t = Oram.T:
        for j from 0 to Oram.T − 1:
            p′ ← Oram.Used_j
            Oram.⟨Shuffle⟩_p′ ← Oram.⟨Stash⟩_j
        Oram ← Initialize(Oram.⟨Shuffle⟩)
```

Figure 6: **Zahur et al.'s ORAM access method [63].**

```
define Access(Oram, ⟨i⟩, Φ)
    ⟨found⟩ ← false
    if Oram.t > 0:
        ⟨if⟩ Oram.⟨Stash⟩_0.⟨index⟩ = ⟨i⟩:
            ⟨found⟩ ← true
        for j from 1 to Oram.t:
            ⟨if⟩ Oram.⟨Stash⟩_j.⟨index⟩ = ⟨i⟩:
                ⟨found⟩ ← true
                Swap(Oram.⟨Stash⟩_j, Oram.⟨Stash⟩_0)
        ⟨if⟩ not ⟨found⟩:
            Oram.⟨Stash⟩_t ← Oram.⟨Stash⟩_0
    p ← GetPos(Oram.Posmap, ⟨i⟩, ⟨found⟩)
    ⟨if⟩ not ⟨found⟩:
        Oram.⟨Stash⟩_0 ← Oram.⟨Shuffle⟩_p
    ⟨else⟩:
        Oram.⟨Stash⟩_t ← Oram.⟨Shuffle⟩_p
    Oram.Used ← Oram.Used ∪ {p}
    Oram.t ← Oram.t + 1
    Φ(Oram.⟨Stash⟩_0)
    if Oram.t = Oram.T:
        for j from 0 to Oram.T − 1:
            p′ ← Oram.Used_j
            Oram.⟨Shuffle⟩_p′ ← Oram.⟨Stash⟩_j
        Oram ← Initialize(Oram.⟨Shuffle⟩)
```

Figure 7: **Our improved ORAM access method.**

on public values (i.e., Oram.t); thus it retains the necessary trace indistinguishability ORAM property.

4.4 Complexity Analysis

Unlike Secure Gale-Shapley, the execution time of Secure Roth-Peranson is not obviously dominated by a single stage of the algorithm. Instead, there are multiple phases, and the cost incurred by each depends on the bounds of the input:

1. **Sharing.** This stage is necessary only if reviewer preferences are XOR-shared between the two computation parties. The preference lists for the individual reviewers are combined into a master preference list by repeated Batcher merging. The cost of this process is $\Theta(mr\log^2 m + mr\log m\log r)$.

2. **Setup.** The master preference lists for the proposers and reviewers are combined into a single array using a Batcher merge and an oblivious queue, such that pairings that are not ranked by both a proposer and a reviewer are omitted. The asymptotic cost of this process is $\Theta((nq+mr)\log(nq+mr))$. The combined master preference array is then sorted according to the proposers' indices and rankings using n Batcher sorts over lists of length q, at a total cost of $\Theta(nq\log^2 q)$.

3. **Permutation.** The preference array is shuffled using a Waksman network, incurring a cost of $\Theta(nq\log nq)$.

4. **Proposal/rejection.** The algorithm adds proposers one by one and iterates through the proposers' preference lists in a manner similar to Gale-Shapley. It must iterate exactly as many times as there are potential proposer-rankings (i.e. nq). For each iteration, the algorithm performs one access to an ORAM containing the reviewers' tentative matches (m elements of size s). Using Square-Root ORAM, the cost of the proposal-rejection phase is in $\Theta(nqs\sqrt{m\log^3 m})$.

Thus, the total cost of our Secure Roth-Peranson algorithm is $\Theta((nq+mr)\log(nq+mr)+nq\log^2 q+nq\log nq+nqs\sqrt{m\log^3 m})$. XOR-sharing incurs an added cost of $\Theta(mr\log^2 m+mr\log m\log r)$.

Reducing bounds by distributing positions. For many applications, including the medical residency match, the number of available positions is not constant among reviewers. In such cases, the cost of the proposal/rejection phase can be reduced by setting the parameter s to be smaller than the maximum, and distributing the positions of reviewers who exceed the bound among sub-reviewers. A potentially significant decrease in s may lead to only a small increase in m and q. The optimum balance depends upon the input parameters and implementation details, but this splitting should reduce overall cost significantly in some cases.

In order for both parties to split the reviewers in an identical way, it must be publicly known which reviewers are to be split, and how many sub-reviewers they are to be split into, which leaks information about the number of positions offered by each reviewer. This is acceptable in many applications (such as resident-hospital matching), as the position quotas are already public knowledge. If such a leak is unacceptable, the bound s cannot be lowered.

All that remains is to specify that the splitting of reviewers be done in such a way that the result of the algorithm is unchanged. This will be the case if we require that all sub-reviewers share identical preference lists, that proposers rank all sub-reviewers for each reviewer they would have originally ranked, and that those sub-reviewers be ranked contiguously. These properties will ensure that any proposer who is rejected by one sub-reviewer will immediately propose to and be considered by the next sub-reviewer. At any iteration, all tentative matches should be equivalent to those that would have been made without a reviewer split.

As we assume an honest-but-curious model, we can trust that the split will be performed correctly. For any implementation where XOR-sharing is used to hide the preferences from the computation parties, reviewer splitting must be performed by the members of the matching sets before they submit their preferences. In cases where the data is partitioned among the computation parties rather than being XOR-shared, we suggest that the computation parties be responsible for performing the split.

5. RESULTS

We implemented and benchmarked our secure stable matching protocols using the Obliv-C [61] multi-party computation framework, which executes Yao's Garbled Circuits protocol [59] with various optimizations [5, 25, 62]. Our code was compiled using `gcc` version 4.8.4 under Amazon's distribution of Ubuntu 14.04 (64 bit), with the `-O3` flag enabled.

We ran each benchmark on a pair of Amazon EC2 `C4.2xlarge` nodes, located within the same datacenter. These nodes are provisioned with 15GiB of DDR4 memory and four physical cores partitioned from an Intel Xeon E5-2666 v3 running at 2.9GHz, each core being capable of executing two simultaneous threads. The inter-node bandwidth was measured to be 2.58 Gbps, and inter-node network latency to be roughly 150 µs.

5.1 Gale-Shapley

In addition to our oblivious linked multi-list, we used other specialized oblivious data structures in our secure Gale-Shapley implementation where doing so provides us with the best performance. We used the fastest available implementations of Square-Root and Circuit ORAM, from Zahur *et al.*. We also used Zahur *et al.*'s oblivious queue construction [60], modified to avoid dynamic allocation of new layers by including a constant, public size bound.

As a point of comparison, we implemented and benchmarked a "textbook" version of Secure Gale Shapley, which omitted our oblivious linked multi-list construction in favor of storing the preferences array in a single ORAM of size $\Theta(n^2)$. The textbook version still uses the other oblivious data structures including the oblivious queue. It is equivalent to the version of Secure Gale-Shapley described by Zahur *et al.* [63], which is the best previously-published secure stable matching result. For both the textbook and improved versions of Secure Gale-Shapley, we benchmarked variants using Square-Root ORAM, Circuit ORAM, and Linear Scan.

Figure 8 and Table 1 present our findings, which are consistent with our analytical results and confirm that Square-Root ORAM outperforms both Circuit ORAM and Linear Scan for all tested parameters. At 512×512 members, we achieve more than $40\times$ improvement relative to the previous best technique, completing the benchmark in under 48 minutes, compared to over 33 hours. In addition to the results presented in our figures, we tested our improved algorithm with Square-Root ORAM at 1024×1024 members, and found that it required 228 minutes to complete.

5.2 Roth-Peranson

We implemented Secure Roth-Peranson using the constructions described in Sections 3.1 and 4.2, and tested it on synthesized data across a range of parameters, as well as data chosen to simulate the full national medical residency match.

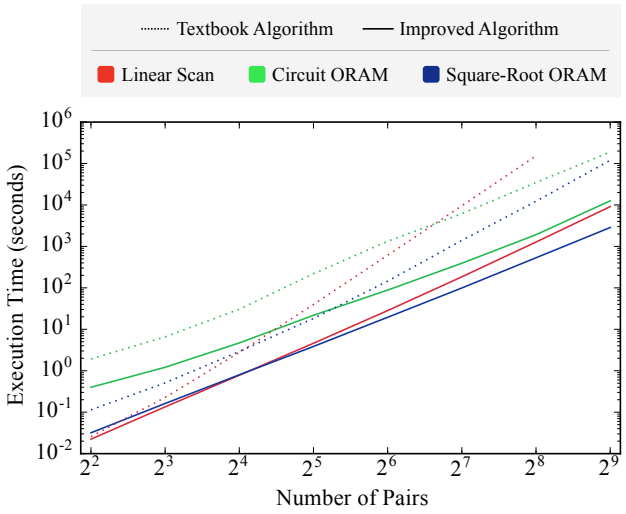


Figure 8: **Secure Gale-Shapley Execution Time vs Pair Count.** Values are mean wall-clock times in seconds for full protocol execution including initialization. For benchmarks of 4–64 pairs, we collected 30 samples; for 128–256 pairs we collected three samples; and for 512 pairs we collected one sample.

Pairs	Linear Scan		Circuit ORAM		Square-Root	
	Textbook	Improved	Textbook	Improved	Textbook	Improved
64	3.05	0.12	5.97	0.39	0.49	0.06
128	48.21	0.80	27.82	1.72	5.00	0.33
256	771.69	5.62	157.49	8.43	44.84	1.73
512	–	41.23	858.36	55.65	440.31	9.41
1024	–	207.65	–	240.54	–	42.33

Table 1: **Secure Gale-Shapley Gate Count vs Pair Count.** Values represent billions of non-free gates required for full protocol execution including initialization.

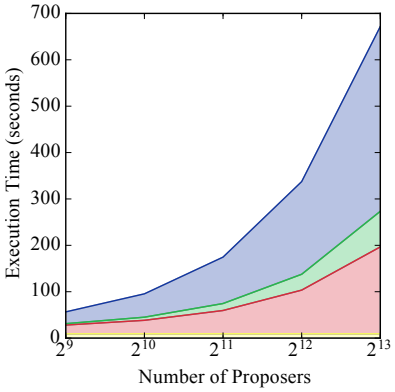
5.2.1 *Parametric benchmarks*

We benchmarked our implementation using synthetic data and varying each of the bounds (n,m,q,r,s) independently in order to demonstrate their effect on the execution time. We recorded statistics individually for each of the phases described in Section 4.4. The results of this experiment are summarized in Figure 9. Execution cost increases linearly with all five parameters, consistent with our analytical results. We also collected the total number of non-free gates executed for each sample, observing a consistent execution speed of around 3.7M gates/second across the experiments.

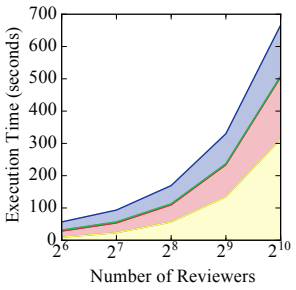
5.2.2 *National Medical Residency Match*

To assess the performance of our Secure Roth-Peranson algorithm in a realistic context, we used it to compute matches for a dataset designed to model the 2016 national medical residency match. The NRMP does not release raw preference data, even in de-identified form [31]. They do, however, release comprehensive statistical information about each year's match [41]. We used this to construct a synthetic dataset with similar properties.

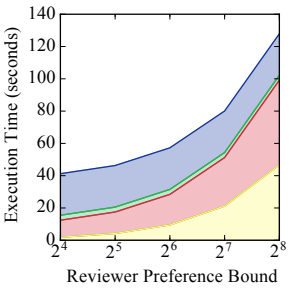
The primary NRMP match for 2016 involved 4,836 residency programs having a total of 30,750 available positions, and 35,476 aspiring residents who collectively submitted 406,173 rankings. A subset of the participants were subject to the match variations described at the end of this section; however, as our algorithm does not handle variations, we consider all of them to be unique individuals participating in accordance with the basic scheme. Thus, for our benchmark, $n = 35476$ and $m = 4836$. The average number of positions per program was 6.35; we chose $s = 12$. The average number of ranked applicants per position varied according to program category. We chose to limit programs to 10 ranked candidates



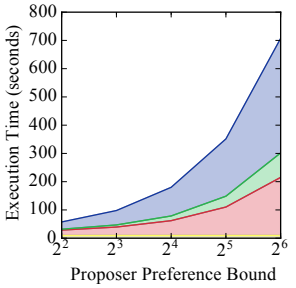
(a) **Proposer Count.** For this benchmark we varied n between 2^9 and 2^{13}, and set $m = 64$, $q = 4$, $r = 64$, $s = 16$



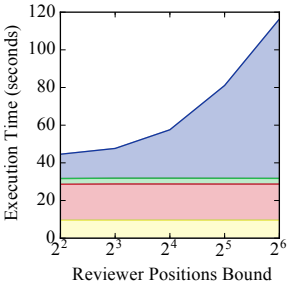
(b) **Reviewer Count.** For this benchmark we varied m between 2^6 and 2^{10}, and set $n = 2^9$, $q = 4$, $r = 64$, and $s = 16$



(d) **Reviewer Preference Bound.** For this benchmark we varied r between 2^4 and 2^8, and set $n = 2^9$, $m = 64$, $q = 4$, and $s = 16$



(c) **Proposer Preference Bound.** For this benchmark we varied q between 4 and 64, and set $n = 2^9, m = 64, r = 64, s = 16$



(e) **Reviewer Positions Bound.** For this benchmark we varied s between 4 and 64, and set $n = 2^9$, $m = 64$, $q = 4$, and $r = 64$

Figure 9: **Secure Roth-Peranson Parametric Benchmark Results.** We show the impact of the five major parameters (n, m, q, r, s). Times spent during the sharing, setup, permutation, and proposal/rejection phases are recorded individually. Y-axis values represent average wall-clock times from 30 samples.

Algorithm Phase	Time (hours)	Billions of Non-Free Gates
Sharing	1.07	18.14
Setup	1.60	29.65
Permutation	0.56	6.56
Proposal/Rejection	15.01	172.52
Total	18.22	226.87

Table 2: **Secure Roth-Peranson NRMP Benchmark Results.** For this benchmark we set $n = 35476$, $m = 4836$, $q = 15$, $r = 120$, and $s = 12$. These parameters are intended to be representative of the match performed by the National Residency Matching Program.

per position, giving us $r = 120$ (programs with fewer than 12 positions are still permitted to rank up to 120 candidates). It should be noted that no program category exceeds an average of 8.4 ranked applicants per position except for anaesthesiology PGY-2, which is a significant outlier with an average of 19.4. However, anaesthesiology PGY-2 programs have 6.24 positions each on average, so an average-sized anaesthesiology PGY-2 program may still rank 19 candidates per position. Finally, the average aspiring resident ranked 11.45 programs. We chose $q = 15$.

We believe these parameters to be accommodating to the vast majority of participants in the NRMP match, but recognize that a few outliers must accept limitations. Programs with an unusually large number of positions can be accommodated by splitting as described in Section 4.4. However, we lack data to determine how many programs would be required to split; as such we have omitted this step. Therefore, our results should be considered primarily a demonstration of the feasibility of calculating an NRMP-scale match securely, rather than a report of the precise cost of doing so.

The preferences of each resident (proposer) are chosen randomly from the available programs (reviewers), and vice versa. This is unrealistic, but cannot impact performance results, since our algorithm is data-oblivious by nature.

We collected only three samples for this benchmark due to its long execution time. Gate count and average execution time are reported in Table 2. It required just over eighteen hours (or 225 billion gates) to complete. This seems efficient enough to be of practical use in cases such as the NRMP, where the computational cost is insignificant (less than $16 total at current AWS prices) compared to the administrative costs already incurred by existing methods.

Complexities of the Actual NRMP Match. Roth and Peranson designed several extensions to their basic algorithm to accommodate properties of the NRMP match, including couples matching and contingent programs, which cannot be handled by our version.

Couples matching allows residents with romantic partners to synchronize their rankings such that their proposals are accepted or rejected together, and breaking a tentative match containing one member of a couple causes the other member's tentative match to break as well. *Contingent programs* require residents to also match with prerequisite programs. The process for matching such programs is effectively identical to couples' matching, except that one proposer submits two linked ranking lists and proposes to multiple reviewers simultaneously. Contingent programs can combine with couples' matching to create four-way dependency structures.

Roth and Peranson's match variation extensions function by allowing those proposers and reviewers who were displaced by couples or contingent matches which were themselves subsequently displaced to rewind their preferences and propose again from the beginning. The instability chaining algorithm is naturally amenable to this process, and it is performed at the end of each round, before new proposers are added. Roth and Peranson also specify that a loop detector is necessary. These match variations remove the guarantee that a stable matching exists, and they make the problem of finding a stable match (if one exists) NP-complete [44].

Unfortunately, our linked multi-list construction is fundamentally incompatible with these extensions, due to the fact that it permits each potential pairing to be accessed only once. Before each rewinding, it would be necessarily to completely reshuffle or regenerate the preferences array. Reshuffling after each iteration would add a term of $\Omega(n^3 q \log nq)$ to our asymptotic complexity, causing it to become impractical for large inputs. Moreover, Roth and Peranson's extensions do not guarantee that the algorithm completes in a fixed number of rounds; thus any straightforward secure implementation would leak the number of rounds required. Although our method does not support the additional extensions used in the NMRP match, we note that many other important matchings (such as public school assignments) do not require these extensions.

6. PRIOR WORK

Gale-Shapley is the first problem presented in Kleinberg and Tardos' introductory algorithms textbook [28], and there is a vast literature on stable matching. Gusfield and Irving provide a book-length technical survey [21] and Alvin Roth published a general-audience book [47]. Here, we focus only on related work on privacy-preserving stable matching.

Golle [19] developed a privacy-preserving version of the classic Gale-Shapley algorithm in a setting where the matching protocol is performed by a group of matching authorities. Privacy and correctness are guaranteed when a majority of the matching authorities are honest. Golle argued that generic multi-party computation protocols were too impractical to implement an algorithm as complex as Gale-Shapley, and developed a protocol using threshold Pallier encryption and re-encryption mixnets. Golle's protocol requires $O(n^5)$ asymmetric cryptographic operations. Although he claimed it was "practical", it has never been implemented.

Franklin *et al.* [13] identified cases where Golle's protocol would not work correctly, and developed two new protocols using a similar approach. Their first protocol was based on an XOR secret sharing scheme and used private information retrieval to process bids. It required running an encryption mixnet on $O(n)$ ciphertexts for each of n^2 rounds, requiring in total $O(n^4)$ public key operations and $\tilde{O}(n^2)$ communication rounds. Their second protocol was not based on Golle's, but instead used garbled circuits in combination with Naor-Nissim's protocol for secure function evaluation [39]. This resulted in a two-party protocol with $O(n^4)$ computation complexity and $\tilde{O}(n^2)$ communication rounds. As with Golle's, it does not appear to be practical and has never been implemented.

Teruya and Sakuma presented a secure stable matching protocol, also building on Golle's protocol, but using additive homomorphic encryption to simplify the bidding process [52]. This reduced the number of communication rounds needed to $O(n^2)$ and resulted in a protocol practical enough to implement. They implemented their protocol as a client-server system, using mobile devices running on a LAN. The largest benchmark they report is for $n = 4$, which took over 8 minutes to complete.

Terner [51] built garbled-circuit implementations of variants of the Gale-Shapley algorithm, reporting execution times of over 12 hours for experiments with 100×100 participants.

Keller and Scholl [27] were the first to consider using RAM-based secure computation to implement stable matching. They designed a secure version of Gale-Shapley using an ORAM, and implemented their protocol using Path ORAM [50] and the SPDZ MPC protocol [10]. They report an experiment that matched $128 \times$

128 participants in roughly 2.5 hours, but it also required an estimated 1000 processor-days of offline compute time (i.e., work independent of the input) which they did not include. In all cases, the algorithm and secure computation techniques together limit the applicability of the entire scheme to toy instances.

The best previous results reported for implementing secure stable matching are Zahur *et al.*'s results using Square-Root ORAM [63], which are the baseline comparison we use in Section 5.1. They implemented a textbook version of Gale-Shapley, and reported completing a match involving 512×512 participants in just over 33 hours (over 40 times longer than our approach takes for the same benchmark running on an identical testbed).

Blanton *et al.* [6] made an observation about read-once data access patterns in the structure of Breadth First Search, and proposed a $\Theta(V^2)$ secure version based upon permuting the rows and columns of an adjacency matrix. Though their observation is similar to our own, the underlying differences between Gale-Shapley and BFS preclude adapting their solution. In particular, BFS permits the algorithm to iterate over an entire column at once, whereas both Gale-Shapley and Roth-Peranson must shift between proposers as they become matched and unmatched, and resume iteration when revisiting. This necessitates a far more complex construction.

7. CONCLUSION

Our results confirm that with appropriately adapted algorithms and data structures it is now possible to execute complex algorithms with data-dependent memory accesses as scalable secure two-party computations. The NRMP matching pool is one of the largest of its type in the world. Similar or identical algorithms are used for many other problems including matching residents to residency programs in other countries [9]; placing applicants for pharmacy, optometry, psychology, dentistry and other residencies [40]; matching rushees to sororities [37]; and assigning students to public schools in Boston and New York City [54]. Most of these are significantly smaller than the scale demonstrated by our simulated NRMP match, and we judge the cost of executing an NRMP-scale match as an MPC to be well within reasonable bounds for such use cases. We are optimistic that private stable matching protocols can be applied to important matching processes in practice.

Availability

All of our code is available under the BSD 2-Clause Open Source license from https://www.oblivc.org/matching.

Acknowledgments

The authors thank Samee Zahur for insightful conversations about this work and assistance with Obliv-C and ORAM, and Elaine Shi for constructive comments and advice. This work was partially supported by grants from the National Science Foundation SaTC program (NSF Award CNS-1111781 and TWC-1664445), the Air Force Office of Scientific Research, and Google.

8. REFERENCES

[1] Atila Abdulkadiroğlu, Parag A Pathak, and Alvin E Roth. The New York City High School Match. *American Economic Review*, 2005.

[2] abhi shelat and Chih-Hao Shen. Fast Two-Party Secure Computation with Minimal Assumptions. In *ACM CCS*, 2013.

[3] Chaitanya Bandela, Yu Chen, Andrew B. Kahng, Ion I. Măndoiu, and Alexander Zelikovsky. Multiple-object XOR Auctions with Buyer Preferences and Seller Priorities. In *Competitive Bidding and Auctions*. 2003.

[4] K. E. Batcher. Sorting Networks and Their Applications. In *Spring Joint Computer Conference*, 1968.

[5] Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Phillip Rogaway. Efficient Garbling from a Fixed-Key Blockcipher. In *IEEE Symposium on Security and Privacy*, 2013.

[6] Marina Blanton, Aaron Steele, and Mehrdad Alisagari. Data-oblivious Graph Algorithms for Secure Computation and Outsourcing. In *ACM Asia CCS*, 2013.

[7] Luís T. A. N. Brandão. Secure Two-Party Computation with Reusable Bit-Commitments, via a Cut-and-Choose with Forge-and-Lose Technique. In *ASIACRYPT*, 2013.

[8] Niklas Buescher and Stefan Katzenbeisser. Faster Secure Computation through Automatic Parallelization. In *USENIX Security Symposium*, 2015.

[9] Canadian Resident Matching Service. The Match Algorithm. http://www.carms.ca/en/residency/match-algorithm/, 2016.

[10] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty Computation from Somewhat Homomorphic Encryption. In *CRYPTO*. 2012.

[11] Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY —- a Framework for Efficient Mixed-Protocol Secure Two-Party Computation. In *Network and Distributed Systems Symposium*, 2015.

[12] Sky Faber, Stanislaw Jarecki, Sotirios Kentros, and Boyang Wei. Three-Party ORAM for Secure Computation. In *ASIACRYPT*, 2015.

[13] Matthew Franklin, Mark Gondree, and Payman Mohassel. Improved Efficiency for Private Stable Matching. In *Topics in Cryptology – CT-RSA*, 2007.

[14] Tore Kasper Frederiksen, Thomas P Jakobsen, and Jesper Buus Nielsen. Faster Maliciously Secure Two-Party Computation Using the GPU. In *Security and Cryptography for Networks*. 2014.

[15] David Gale and Lloyd S. Shapley. College Admissions and the Stability of Marriage. *The American Mathematical Monthly*, 69(1), 1962.

[16] Craig Gentry, Kenny A Goldman, Shai Halevi, Charanjit Julta, Mariana Raykova, and Daniel Wichs. Optimizing ORAM and Using it Efficiently for Secure Computation. In *Privacy Enhancing Technologies*, 2013.

[17] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In *ACM Symposium on the Theory of Computing*, 1987.

[18] Oded Goldreich and Rafail Ostrovsky. Software Protection and Simulation on Oblivious RAMs. *Journal of the ACM*, 43(3), 1996.

[19] Phillippe Golle. A Private Stable Matching Algorithm. In *Financial Cryptography and Data Security*, 2006.

[20] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure Two-Party Computation in Sublinear (amortized) Time. In *ACM CCS*, 2012.

[21] Dan Gusfield and Robert W Irving. *The Stable Marriage Problem: Structure and Algorithms*. MIT Press, 2003.

[22] Chih hao Shen and abhi shelat. Two-output secure computation with malicious adversaries. In *Eurocrypt 2011*, 2011.

[23] Wilko Henecka, Stefan Kögl, Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg. TASTY: Tool for Automating Secure Two-partY computations. In *ACM CCS*, 2010.

[24] Andreas Holzer, Martin Franz, Stefan Katzenbeisser, and Helmut Veith. Secure Two-Party Computations in ANSI C. In *ACM CCS*, 2012.

[25] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. Faster Secure Two-Party Computation using Garbled Circuits. In *USENIX Security Symposium*, 2011.

[26] Yan Huang, Jonathan Katz, and David Evans. Efficient Secure Two-Party Computation Using Symmetric Cut-and-Choose. In *CRYPTO*, 2013.

[27] Marcel Keller and Peter Scholl. Efficient, Oblivious Data Structures for MPC. In *ASIACRYPT*, 2014.

[28] Jon Kleinberg and Éva Tardos. *Algorithm Design*. 2005.

[29] Vladimir Kolesnikov and Thomas Schneider. Improved Garbled Circuit: Free XOR Gates and Applications. *Automata, Languages and Programming*, 2008.

[30] Benjamin Kreuter, abhi shelat, and Chih-Hao Shen. Billion-Gate Secure Computation with Malicious Adversaries. In *21st USENIX Security Symposium*, 2012.

[31] Mei Liang. Director of Research, National Resident Matching Program. Personal communication, May 2016.

[32] Yehuda Lindell. Fast Cut-and-Choose Based Protocols for Malicious and Covert Adversaries. In *CRYPTO*, 2013.

[33] Yehuda Lindell and Benny Pinkas. A Proof of Security of Yao's Protocol for Two-Party Computation. *Journal of Cryptology*, 22(2), 2009.

[34] Yehuda Lindell and Benny Pinkas. Secure Two-Party Computation via Cut-and-Choose Oblivious Transfer. In *Theory of Cryptography Conference*, 2011.

[35] Yehuda Lindell and Benny Pinkas. An Efficient Protocol for Secure Two-Party Computation in the Presence of Malicious Adversaries. *Journal of Cryptology*, 28(2), 2015.

[36] Yehuda Lindell, Benny Pinkas, and Nigel P. Smart. Implementing Two-Party Computation Efficiently with Security Against Malicious Adversaries. In *Sixth Conference on Security and Cryptography for Networks*, 2008.

[37] Susan Mongell and Alvin E. Roth. Sorority Rush as a Two-Sided Matching Mechanism. *American Economic Review*, 81(3), 1991.

[38] Benjamin Mood, Debayan Gupta, Henry Carter, Kevin Butler, and Patrick Traynor. Frigate: A Validated, Extensible, and Efficient Compiler and Interpreter for Secure Computation. In *IEEE European Symposium on Security and Privacy*, 2016.

[39] Moni Naor and Kobbi Nissim. Communication Preserving Protocols for Secure Function Evaluation. In *STOC*, 2001.

[40] National Matching Service. Current Clients. https://natmatch.com/clients.html, 2016.

[41] National Resident Matching Program. 2016 Main Residency Match. http://www.nrmp.org/wp-content/uploads/2016/04/Main-Match-Results-and-Data-2016.pdf, 2016.

[42] Michael Ostrovsky. Stability in Supply Chain Networks. *American Economic Review*, 98(3):897–923, June 2008.

[43] Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Stephen C. Williams. Secure Two-Party Computation Is Practical. In *ASIACRYPT*, 2009.

[44] Eytan Ronn. NP-Complete Stable Matching Problems. *Journal of Algorithms*, 11(2), 1990.

[45] Alvin E. Roth. The Evolution of the Labor Market for Medical Interns and Residents: A Case Study in Game Theory. *Journal of Political Economy*, 92:991–1016, 1984.

[46] Alvin E. Roth. Interim Report #1: Evaluation of the Current NRMP Algorithm, and Preliminary Design of an Applicant-Processing Algorithm. https://web.stanford.edu/~alroth/interim1.html, 1996.

[47] Alvin E. Roth. *Who Gets What—–and Why: The New Economics of Matchmaking and Market Design*. Houghton Mifflin Harcourt, 2015.

[48] Alvin E. Roth and Elliott Peranson. The Redesign of the Matching Market for American Physicians: Some Engineering Aspects of Economic Design. *American Economic Review*, 1999.

[49] Alvin E. Roth and John H. Vande Vate. Random Paths to Stability in Two-Sided Matching. *Econometrica*, 58(6):1475–1480, 1990.

[50] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an Extremely Simple Oblivious RAM Protocol. In *ACM CCS*, 2013.

[51] Ben Terner. Stable Matching with PCF Version 2, an Eťude in Secure Computation. Master's thesis, U of Virginia, 2015.

[52] Tadanori Teruya and Jun Sakuma. Round-Efficient Private Stable Matching from Additive Homomorphic Encryption. In *Conference on Information Security*, 2013.

[53] The Royal Swedish Academy of Sciences. Stable Matching: Theory, Evidence, and Practical Design. http://www.nobelprize.org/nobel_prizes/economic-sciences/laureates/2012/popular-economicsciences2012.pdf, 2012.

[54] Tracy Tullis. How Game Theory Helped Improve New York City's High School Application Process. *The New York Times*, 5 December, 2014.

[55] Abraham Waksman. A Permutation Network. *Journal of the ACM*, 15(1), January 1968.

[56] Xiao Wang, Hubert Chan, and Elaine Shi. Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound. In *ACM CCS*, 2015.

[57] Xiao Shaun Wang, Yan Huang, T.-H. Hubert Chan, Abhi Shelat, and Elaine Shi. SCORAM: Oblivious RAM for Secure Computation. In *ACM CCS*, 2014.

[58] Andrew C. Yao. Protocols for Secure Computations. In *Symposium on Foundations of Computer Science*, 1982.

[59] Andrew Chi-Chih Yao. How to Generate and Exchange Secrets (Extended Abstract). In *IEEE Symposium on Foundations of Computer Science*, 1986.

[60] Samee Zahur and David Evans. Circuit Structures for Improving Efficiency of Security and Privacy Tools. In *IEEE Symposium on Security and Privacy*, 2013.

[61] Samee Zahur and David Evans. Obliv-C: A Lightweight Compiler for Data-Oblivious Computation. Cryptology ePrint Archive, Report 2015/1153. http://oblivc.org, 2015.

[62] Samee Zahur, Mike Rosulek, and David Evans. Two Halves Make a Whole: Reducing Data Transfer in Garbled Circuits Using Half Gates. In *EUROCRYPT*, 2015.

[63] Samee Zahur, Xiao Wang, Mariana Raykova, Adrià Gascón, Jack Doerner, David Evans, and Jonathan Katz. Revisiting Square Root ORAM: Efficient Random Access in Multi-Party Computation. In *IEEE Symposium on Security and Privacy*, 2016.