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 to preserve the privacy of participants' preference rankings. *Secure multi-party computation* presents an alternative that offers the possibility of private matching processes that do not rely on any common trusted third party. However, stable matching algorithms are computationally intensive and involve complex data-dependent memory access patterns, so they have previously been considered infeasible for execution in a secure multi-party context on non-trivial inputs.

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 insights are to design new oblivious data structures that exploit the properties of the matching algorithms. We then apply our secure computation techniques to the instability chaining algorithm of Roth and Peranson, currently in use by the National Resident Matching Program. The resulting algorithm is efficient enough to be useful at the scale required for matching medical residents nationwide, taking just over 17 hours to complete an execution simulating the 2016 NRMP 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 than to their assigned partners [13]. A set of pairings that satisfies this property is known as a *stable matching*; it can be found 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 [51]. Stable matching algorithms are used in many interesting applications, including matching medical residents to residency programs [39], students to schools [1, 52], and candidates to sororities [35], as well as in special types of auctions [2] and in managing supply chains [40].

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 in terms of a two-party secure computation in which all members of the pairing sets trust two representatives to execute on their behalf without colluding. Distribution of preference inputs to these nodes can be achieved in one of two ways: either all participants XOR-share their preference lists between the two nodes, and the nodes learn nothing, or the members of the first set trust one representative to collect their preferences and keep them private, and the members of the second set trust the other.

Executing an algorithm as complex and data-dependent as the Gale-Shapley stable matching algorithm as a secure computation has been a longstanding goal for researchers. Secure computation requires (among other things) that all data-dependent memory accesses be hidden in order to maintain security. This requirement has traditionally been a significant contributor to the inefficiency of secure computation relative to its insecure counterpart. For example, the protocols of Golle [17] and Franklin *et al.* [11] required roughly $O(n^5)$ and $O(n^4)$ public-key operations respectively and were too complicated to implement.

Recent advances in ORAM design [54, 61] 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 [61]. We overcome this barrier by adapting stable matching algorithms to use efficient oblivious data structures that take advantage of known properties of the memory access patterns, combining general-purpose ORAMs with special-purpose constructs. These structures leverage data partitions and patterns inherent to the problem to restrict the ways in which the data can be accessed without leaking any data-dependent access information. We also customize an ORAM design to enable efficient function application as part of the access protocol.

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 access must be hidden, but it is known that each element must be accessed exactly once; we further refine this structure to support multiple lists in order to enable more complex access patterns (Section 3.2). We also introduce a modification to the ORAM access method that enables efficient function application within an ORAM access (Section 4.2).

These techniques are developed and evaluated in the context of secure stable matching algorithms, but we believe they have wider applicability to constructing secure variants of many algorithms that involve data-dependent memory access. We construct secure versions of two important stable matching algorithms in particular. Our Secure Gale-Shapley implementation exhibits the best asymp-

totic 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 residency match, and find that it can match 35,476 aspiring residents to 30,750 positions at 4,836 residency programs.

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 [15, 57] 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) have been developed that can derive protocols for computing *any* function starting from a Boolean-circuit representation. Our experiments use Yao's garbled circuit protocol [30, 56], 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 generates a garbled circuit corresponding to f and the evaluator evaluates that circuit to produce an output that can be decoded to the semantic output. The generator and evaluator execute an oblivious transfer protocol to enable the evaluator to learn the input keys corresponding to its input without revealing that input to the generator. After obtaining its input keys, the evaluator can obliviously evaluate the garbled circuit to obtain the output keys which are decoded in the final step. Although garbled circuits were once thought to be of only theoretical interest, recent works have shown that generic multi-party protocols can be practical [20, 21, 22, 27, 33, 41], even in settings where full active security is required [5, 12, 23, 29, 31, 32]. Current implementations [6, 9, 36, 60] 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 need to be touched in order 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 [16, 18]. An ORAM scheme consists of an *initialization protocol* that accepts an array of elements and initializes a new oblivious structure with that data, 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 [18]. 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 [10, 14, 18, 24, 55] 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 [54]; the most efficient in practice over a wide range of parameters is Square-Root ORAM [61]. 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. In the case of the secure version, these lists are divided among two parties either by partitioning the lists or XOR-sharing the entries.

The algorithm proceeds through each proposer's preference list from most-preferred to least, swapping between proposers as they become matched or invalidated by other matches. We assume the size of the proposer and reviewer sets are equal (which ensures everyone ends up part of some pair), and 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 on any particular iteration, nor how far along that proposer's preference list it will have advanced.

As any iteration could require access to any pairing, a straightforward approach requires that the preferences be stored within an ORAM. Consequently, the algorithm must perform n^2 accesses to an ORAM of length n^2. All other ORAMs and queues required by the textbook algorithm are of length n; thus, the cost of the ORAM accesses for reading the pairings dominates the cost of a naïve implementation, and it is critical to reduce this cost.

3.1 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 are always of public length and accessed via public indices; we use $\langle \mathsf{Array} \rangle$ to denote an array of oblivious data, and $\langle \mathsf{Array} \rangle_i$ to specify element i within that array. We indicate the array slicing operation using a colon: $\langle \mathsf{Array} \rangle_{i:j}$ represents elements i through j of $\langle \mathsf{Array} \rangle$, inclusive. We indicate multidimensional array access with multiple indices delimited by commas. Multidimensional arrays are considered to be equivalent to arrays containing other arrays as elements, and interior arrays need not have the same length. Conditionals on secret values are indicated using $\langle$**if**$\rangle$ and $\langle$**else**$\rangle$. The instructions within such oblivious conditionals are always executed, but have no effect if the condition is false.

3.2 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 require any accesses to be dependent on oblivious conditions (the algorithm must obliviously select *which* preference list is accessed, but exactly one preference list is always accessed). Thus, we need a data structure that iterates over n elements, in order, while hiding the progress of that iteration.

Instead of using a generic ORAM, we design a new data structure to satisfy these requirements more efficiently, which we call an *oblivious linked list*. This construction works similarly to an ordinary linked list in that it consists of a series of elements, each

having, in addition to its data, a reference to the next element in the series. However, the list is jointly randomly permuted and the forward references are stored in garbled form.

Figure 1a illustrates the process of initializing an oblivious linked list from an input array. To construct an oblivious linked list, we first generate an oblivious permutation and its inverse using the method of Zahur *et al.* [61]. The randomly-selected forward permutation comprises one set of Waksman control bits from each party, sequentially applied to the input data. 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 [53], and store the first element of the inverse permutation in a variable.

Unlike an ORAM or oblivious queue, our oblivious linked list can be traversed in constant time. As shown in Figure 1c, each successive access is performed by revealing the physical index of the current element to both parties, who will find its garbled data and the physical index of the next element. This permits us to iterate through a single preference list. As illustrated in Figure 1b, we can extend it to iterate through multiple preference lists by permuting multiple lists together in a single array, and storing their metadata (i.e., the garbled form of the next physical index in each list) in another data structure.

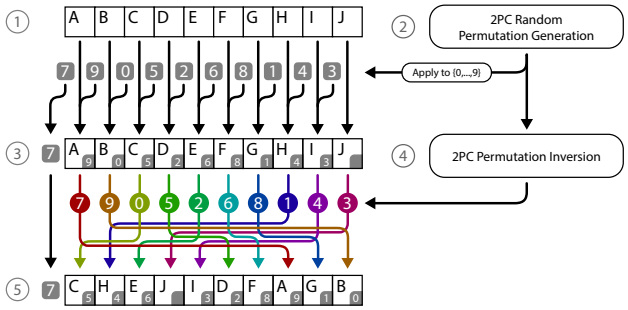
While our solution still requires a general ORAM of n elements in order to maintain the current matches for each of the reviewers, removing the n^2 element ORAM makes a significant improvement in practice and a slight improvement in theory. Furthermore, our new data structure can be initialized by sorting, which dramatically reduces the initialization time compared to previous techniques. We present complete pseudocode for both the oblivious linked list and a version of Secure Gale-Shapley based upon it in Figure 2.

Performance. As mentioned above, reading the next element of an oblivious linked multi-list requires constant time. Initializing the data structure requires an application of a Waksman network. Thus, the amortized overhead for each of the n^2 elements is $\Theta(\log n)$. In contrast, using an ORAM to handle these items would require polylogarithmic overhead.

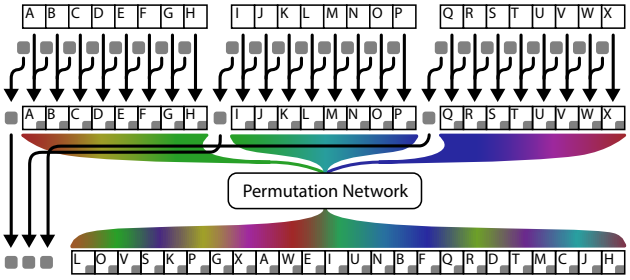
Initialization. Our secure Gale-Shapley algorithm can ingest preference lists only in the order of public indices, but the preferences must eventually be sorted first by proposers' indices, then by proposers' preferences. As the preference lists are not to be stored in an ORAM, this reordering cannot be achieved by insertion. Instead, we specify that both parties supply their data (however it may be partitioned) pre-sorted first according to proposers' indices, then according the reviewers' indices. Once the data is merged into a single matrix, we apply n Batcher sorts [3], one for each proposer's preference list.

Early Termination. In practice, executions of the algorithm rarely require the worse 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.

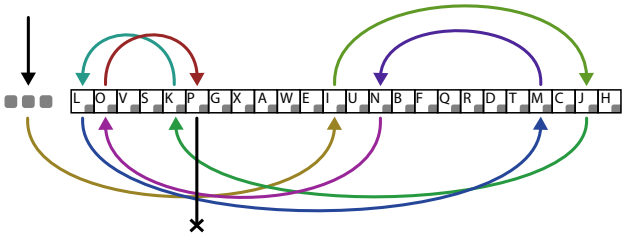
Each iteration must reveal exactly one uniform unused index from the multi-list containing the preferences. This is a problem, as there is no easy way to select a uniform unvisited element from the entire multi-list without requiring significant additional bookkeeping. Our solution is to pad the preference 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 will take between n and n^2 iterations to find a stable matching, after which it will follow this dummy chain in order to maintain data obliviousness.



(a) **The initialization process in detail.** 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) **The interleaving of 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 are retained, allowing them to be independently traversable.



(c) **The traversal of one of three 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 1: **Illustrations of the Oblivious Linked List structure.**

3.3 Security

The security of our scheme depends on established properties of ORAM constructions [61] and oblivious queues [58], and the underlying garbled circuits protocol [30]. We do not modify these

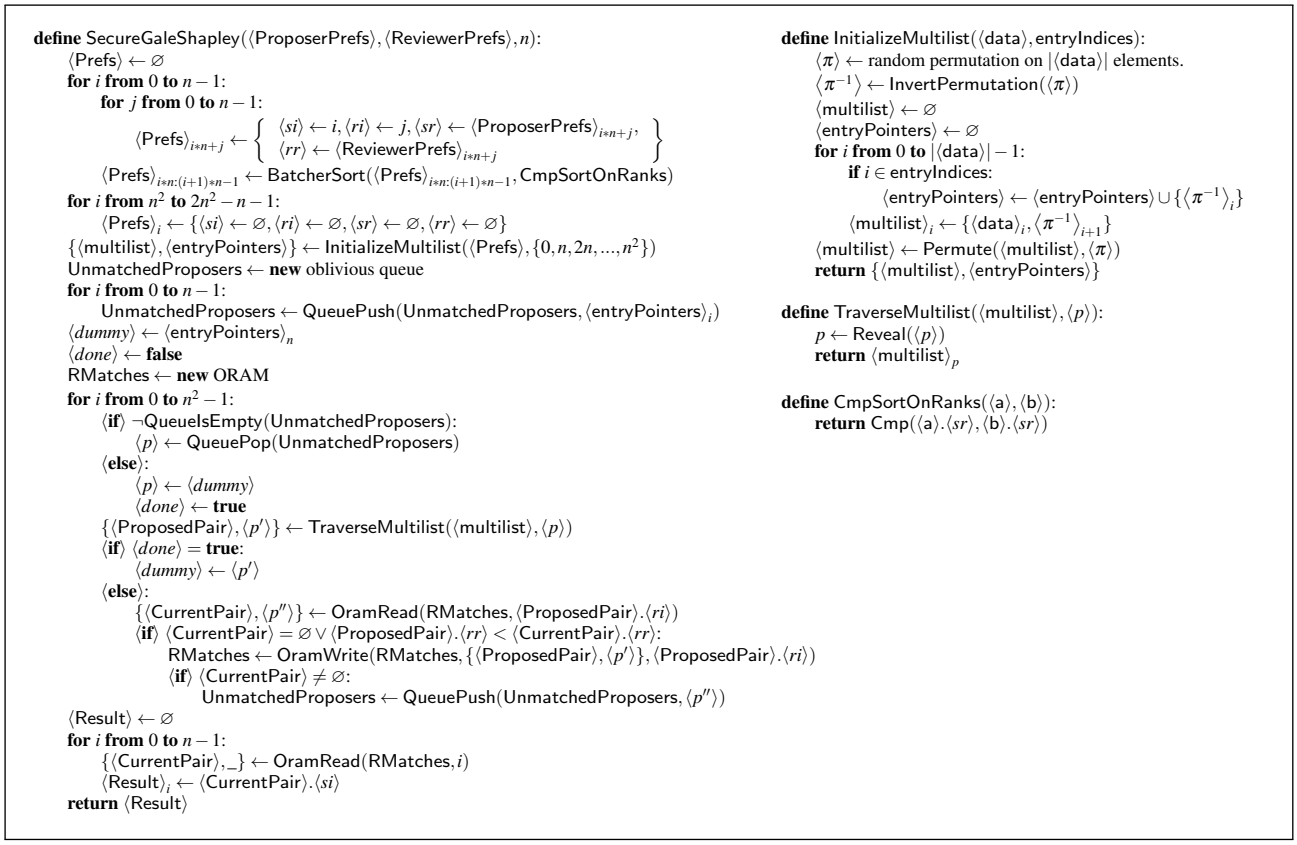


Figure 2: **Secure Gale-Shapley Algorithm.** SecureGaleShapley expects to ingest preferences ordered first by proposer index, then by reviewer index. It returns an array of proposer indices, ordered by the reviewer indices to which the proposers have been paired.

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 a very weak adversarial model, but there has been substantial work showing that semi-honest protocols can be adapted to resist active adversaries with little overhead by exploiting parallelism. We leave to future work the task of applying those methods to our techniques. Many of the scenarios where secure stable matching might be used do involve professional organizations and government agencies as participants, who may be trusted enough 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 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 [61]. By replacing these and using a malicious-secure ORAM and MPC protocol, 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 $\Theta(n^2)$-size 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 $\Theta(\sqrt{n \log^3 n})$ [61], 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 [54], 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 market for medical residents in the United States underwent a crisis [43]. At that point, residencies were allocated in a decentralized way, but demand 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 instituted a central clearinghouse to allocate graduates to residencies. This would later become known as the National Resident Matching Program, or NRMP.

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 for resident-optimal matching, based upon the theory of Stable Matching. Roth proposed an algorithm [44] based upon his earlier inductive proof of stability with John H. Vande Vate [47]. Roth's algorithm, shown in Figure 3, follows a process he called *instability chaining*: proposers (in this case, the aspiring residents) are added one at a time, resolving all conflicts, and continuing through all the proposers.

For instances of the matching problem in which the number of proposers is equal to the number of reviewers, each participant provides a full preference ranking of all participants in the opposite set, and each participant seeks to be paired with only a single member of the opposite set, Roth's algorithm produces the same result as Gale-Shapley. However, it also supports cases in which the sizes of the sets are unequal and not all participants are ranked, as well as other extensions which will be discussed in Section 5.2.2. Roth and Peranson [46] described the final construction and evaluated it experimentally using data from past NRMP matches.

```
define RothPeranson(ProposerPrefs, ReviewerPrefs, RPosCounts, n, m):
    UnmatchedProposers ← new queue
    ProposerPrefsPosition ← ∅
    RMatches ← ∅
    for i from 0 to n − 1:
        ProposerPrefsPosition_i ← 0
        UnmatchedProposers ← QueuePush(UnmatchedProposers, i)
    for i from 0 to m − 1:
        RMatches_i ← ∅
    while ¬QueueIsEmpty(UnmatchedProposers):
        si ← QueuePop(UnmatchedProposers)
        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 3: **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 a list of sets of proposer indices ordered by the reviewer indices to which the proposers have been paired.

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 relatively small, and likely independent of the total number of counterparties available. As a result, we establish a public bound on the number of rankings that each participant can input: this is indicated by q in the case of the proposers, and r in the case of the reviewers. In addition, we establish a public bound on the number of positions each reviewer has to be filled, which we indicate as s (in many real-world instances the individual position counts are public knowledge, but we do not require this to be the case). We use n to represent the number of proposers, and m to represent the number of reviewers.

The adaptations required to create a secure version of the Roth-Peranson matching algorithm are quite similar to those specified for Secure Gale-Shapley in Section 3. In addition, we use loop-flattening [34] to reduce the two nested loops shown in Figure 3 to one loop of exactly nq iterations. To track the matches for each reviewer, we need an ORAM with m elements, each of which is a list of the corresponding reviewer's tentative matches. As this list is stored *within* the ORAM, it needs to be a simple structure and is implemented as a plain array to be linearly scanned. Consequently, the reviewer-status ORAM will have m elements, each of size s. Complete pseudocode for our formulation of Secure Roth-Peranson may be found in Figure 4.

Initialization. We must devise a way to efficiently collate the inputs of the reviewers and proposers, in spite of the fact that both are stored in sparse arrays and it is likely that there are many unrequited preferences that must be omitted. We achieve this using a batcher merge and an oblivious queue.

First, we require that preferences be ingested in the form of two master lists (one containing the preferences of the proposers, and the other of the reviewers), the elements of both lists being ordered according first to proposer index, and then reviewer index. Each element contains as garbled data both the proposer and reviewer indices for the pairing it represents, a rank, and a bit indicating whether the preference belongs to a proposer or a reviewer. Unranked pairs may be omitted, and the lists may be padded by adding dummy elements. These two master lists are combined using a batcher merge. Once we have the combined master preference list, we iterate over it, comparing each pair of contiguous elements and pushing a single element containing both rankings into an oblivious queue if they share proposer and reviewer indices.

Finally, 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 can be linked and shuffled as described in Section 3.2.

4.2 Improving ORAM Access

Unlike the Gale-Shapley algorithm, Secure Roth-Peranson uses large ORAM elements whose sizes are determined by the bound on the number of positions available for any one reviewer. Although our construction is compatible with most ORAM schemes, we assume the use of Square-Root ORAM [61].

We use function application to avoid accessing the ORAM more than once per iteration. That is, rather than retrieving an element, operating on it, and then storing the element back, we pause the ORAM access process when the relevant element is found, apply a function to it, and then resume the access process where it left off. When the function to be applied is simple, this works well. However, Zahur *et al.* [61] specify that the function is to be applied individually to all data elements touched during the access process, meaning that function application incurs the cost of $\Theta(T)$ extra gates, where T is the ORAM refresh period. For a complex function such as the one we need, which linearly scans through a

```
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 4: **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. We use highlighting to indicate each of the four main phases of the algorithm as laid out in Section 4.4: collation is red, sorting is yellow, permutation is green, and proposal/rejection is blue.

number of elements equal to the positions bound s, the number of extra gates is significant. To avoid these inefficiencies, we modify the ORAM access function to allow function application without executing the function multiple times.

ORAM Background and Notation. Square-Root ORAM stores its data in Oram.⟨Shuffle⟩, 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 Oram.⟨Shuffle⟩$_i$.⟨*index*⟩. The ORAM uses a recursive position map structure, Oram.Posmap, to relate physical indices in Oram.⟨Shuffle⟩ to logical indices, so that elements can be accessed without scanning. As each element is accessed, the ORAM moves it from Oram.⟨Shuffle⟩ to Oram.⟨Stash⟩, where it will be linearly scanned on subsequent accesses. The ORAM tracks which physical indices in Oram.⟨Shuffle⟩ have been accessed using a set of public Booleans, Oram.Used. After Oram.T accesses, the ORAM is refreshed and the process starts again from the beginning; progress

```
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 5: **Zahur et al.'s ORAM access method [61].**

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 we will be accessing must be moved into this slot during the course of the access; the function Φ will be applied to $\mathsf{Oram}.\langle\mathsf{Stash}\rangle_0$ at the end. The last active element remains this slot in the interim 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).

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 [26] 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 complete access function is shown in Figure 6, and pseudocode for Zahur et al.'s original access function is shown in Figure 5, with differences marked for ease of 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 of our new ORAM access method is dependent upon the property that for any two accesses, the memory patterns it exhibits are indistinguishable. As with the original access method,

```
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 6: **Our improved ORAM access method.**

this is true. In the first stage of the access, the original algorithm scanned $\mathsf{Oram}.\langle\mathsf{Stash}\rangle$ and applied 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 the second input in place of an arbitrary function. After the stash scan, our algorithm 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 it 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 one will take effect. Finally, the function Φ is applied to a single block with a constant physical index. The observable memory behavior of this algorithm depends only on public values (i.e., the number of elements currently in $\mathsf{Oram}.\langle\mathsf{Stash}\rangle$); therefore it retains the necessary ORAM property of data-obliviousness.

4.4 Complexity Analysis

Unlike 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. They are as follows:

1. **Collation.** Here the algorithm combines the raw preference inputs from the two parties 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 $\Theta(nm\log q + nm\log r)$.

2. **Sorting.** Having collated the potential pairings, the algorithm sorts them according to the proposers' indices and preferences. This requires n Batcher sorts over lists of length q, with a cost of $\Theta(nq\log^2 q)$.

3. **Permutation.** The preference array is shuffled using a Waksman network. This incurs 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. The algorithm must propose exactly as many times as there are potential proposer-rankings — that is, the number of proposers multiplied by the bound on the number of reviewers each proposer can rank. For each iteration, the algorithm performs one access to an ORAM containing the reviewers' current tentative matches; this ORAM contains m elements of size s. During the ORAM access, the algorithm must scan the full contents of the ORAM element, adding an additional factor of s. We indicate the cost of accessing an element in a generic ORAM scheme as $C(x,B)$, where x is the number of elements and B the size of the elements. In the case Square-Root ORAM, $C(x,B) = \Theta(B\sqrt{x\log^3 x})$, and the total 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 $\Theta(nm\log q + nm\log r + nq\log^2 q + nq\log nq + nqC(m,s) + nqs)$ with a generic ORAM, or $\Theta(nm\log q + nm\log r + nq\log^2 q + nq\log nq + nqs\sqrt{m\log^3 m})$ with Square-Root ORAM.

Reducing bounds by distributing positions. In order to reduce the cost of the proposal/rejection phase in practice, we propose that the parameter s be set smaller than the true bound on the number of positions available for each reviewer, and that reviewers exceeding the bound distribute their positions among sub-reviewers.

It is likely for many applications that the number of available positions follows a distribution, rather than being constant among reviewers. In such cases, a bound could be chosen which excludes only those who lie a certain distance above the mean. Thus, a potentially significant decrease in s may lead to only a small increase in m. The optimum balance between these two variable will depend upon the input parameters and implementation details, but it is reasonable to expect that this technique will be very effective in some scenarios.

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 implies that some information about the number of positions offered by each reviewer will be leaked. In many applications, this is acceptable, as the available position counts are public knowledge anyway. For instance, in the case of resident-hospital matching, the number of positions available at any given program is public, and could be estimated from the number of current residents, even were it not. In scenarios where 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 particular iteration, all tentative matches made by all reviewers should be equivalent to those that would have been made without a large-reviewer split. It should be noted that requiring proposers to rank all sub-reviewers implies that lowering s may require an increase in q (in addition to m). As above, the optimum balance will be situation-dependent.

As we assume an honest-but-curious model, we can trust the parties will perform the split correctly. For any implementation wherein 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.

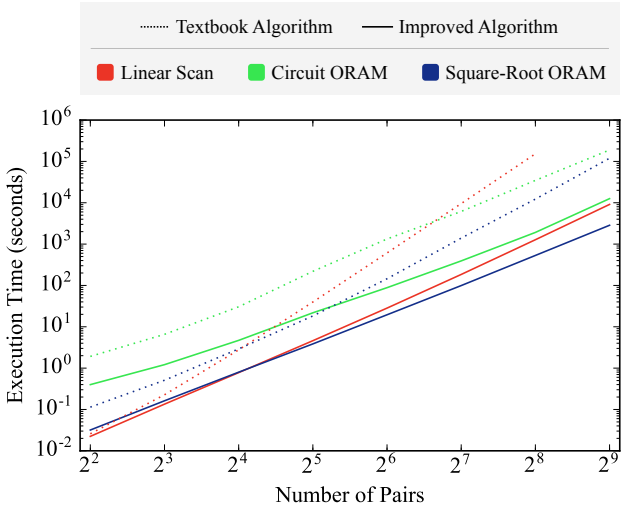


Figure 7: **Execution Time vs Pair Count.** Values are mean wall-clock times in seconds for full protocol execution including initialization, for implementations using Linear Scan, Circuit ORAM, and Square-Root ORAM. For benchmarks of 4–64 pairs, we collected 30 samples; for 128–256 pairs we collected 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 Counts.** Values represent billions of non-free gates required for full protocol execution including initialization.

5. RESULTS

We implemented and benchmarked our secure stable matching protocols using the Obliv-C [59] multi-party computation framework, which executes Yao's Garbled Circuits protocol [57] with various optimizations [4, 22, 60]. 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 Gale-Shapley secure sta-

ble matching 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.*, and used function application to reduce the number of necessary ORAM accesses wherever possible. We also used Zahur *et al.*'s oblivious queue construction [58], 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 naïve 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.* [61], which is the best previously-published secure stable matching result. For both the naïve and improved versions of Secure Gale-Shapley, we benchmarked versions using Square-Root, Circuit ORAMs, and Linear Scan.

Figure 7 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 $40x$ 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

To validate our the analytical understanding of the algorithm and determine whether it is efficient enough to be useful in practice, we implemented Secure Roth-Peranson using the constructions described in Sections 3.2 and 4.2, and the same toolchain and experimental setup as used for Gale-Shapley. We tested it on synthesized data across a range of parameters, as well as data chosen to simulate the full national medical residency match.

5.2.1 Parametric benchmarks

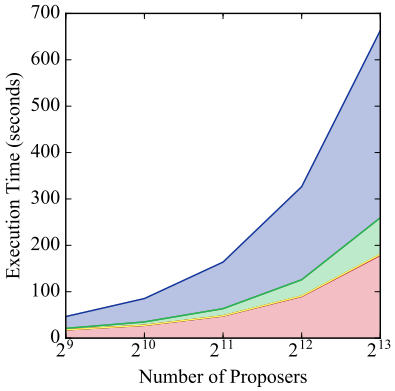
We benchmarked our implementation using synthetic data and varying each of the five bounds (n,m,q,r,s) independently in order to demonstrate their practical effect on the execution time. We recorded statistics individually for each of the four major phases described in Section 4.4.

The results of this experiment are laid out in Figure 8. We find them to be highly consistent with our analytical results - that is, execution cost increases linearly with all five parameters. In addition to wall-clock time, we collected the total number of non-free gates executed for each sample. We found the gate counts to be strongly correlated with execution time, the relationship being roughly 3.7M gates/second.

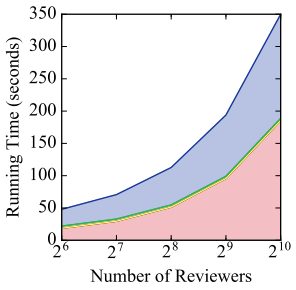
5.2.2 National Medical Residency Match

To assess the performance of our Secure Roth-Peranson algorithm in a realistic context, we used it to compute matches for a dataset with similar properties to the NRMP residency match for 2016. The NRMP does not release raw preference data, even in de-identified form [28]. They do, however, release comprehensive statistical information about each year's match [39]; we have used this to construct a synthetic dataset with similar properties.

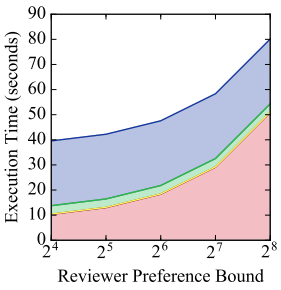
The primary NRMP match for 2016 included 4,836 residency programs having a total of 30,750 available positions, and 35,476 aspiring residents who collectively submitted 406,173 rankings. A subset of the participants were subject to the match variations described at the end of this section; however, as 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



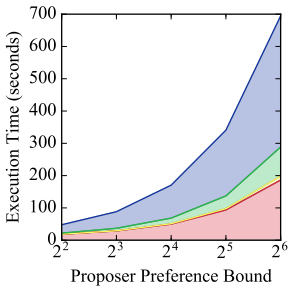
(a) **Proposer Count.** For this benchmark we varied n between 2^9 and 2^{13}, and set $m = 64$, $q = 4$, $r = 64$, $s = 16$



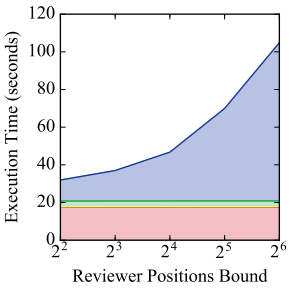
(b) **Reviewer Count.** For this benchmark we varied m between 2^6 and 2^{10}, and set $n = 2^9$, $q = 4$, $r = 64$, and $s = 16$



(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 8: **Parametric Benchmark Results.** We show the impact of each of the five major parameters (n, m, q, r, s) upon the cost of executing a secure stable matching. Times spent during the collation phase (represented in red), the sorting phase (yellow), the permutation phase (green), and the proposal/rejection phase (blue) are recorded individually. Y-axis values represent average wall-clock times from 30 samples.

Algorithm Phase	Time (hours)	Billions of Non-Free Gates
Collation	1.49	27.12
Sorting	0.02	0.40
Permutation	0.56	6.56
Proposal/Rejection	15.01	172.52
Total	17.08	206.61

Table 2: **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. The gate execution rate is 5M gates/second for the first two phases, but lower for the last two phases due to additional computation and communication costs.

number of ranked applicants per position varied according to program category. We chose to limit programs to 10 ranked candidates 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, we suggest that our results be considered primarily a demonstration of the feasibility (or lack thereof) of calculating an NRMP-scale match securely, rather than a report of the precise expense required to do so.

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

Due to its long execution time, we executed this benchmark only once and recorded both execution time and gate count. Our results are reported in Table 2. It required about seventeen hours (or two hundred million gates) to complete. We judge this to be efficient enough to be of practical use in cases such as that of the NRMP, where the computational cost is insignificant 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 handle various desired properties for the NRMP match, including couples matching and contingent programs, which are not handled by our secure 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 has 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 [42].

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 alone would add a term of $\Theta(n^2 q \log nq)$ to our asymptotic complexity, causing it to become more expensive than the naïve ORAM method, and 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 [25], and there is a vast literature on stable matching. Gusfield and Irving provide a book-length technical survey [19] and Alvin Roth published a general-audience book [45]. Here, we focus only on related work on privacy-preserving stable matching.

Golle [17] 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 it was claimed to be "practical", it has never been implemented.

Franklin *et al.* [11] 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 [37]. 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 [50]. 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 [49] reports on a garbled-circuit implementation of vari-

ants of the Gale-Shapley algorithm. As noted, the memory access pattern for the classic algorithm is data-dependent, and thus his implementation suffers from inefficiency in the translation for a program into a boolean circuit. He reports on experiments for 100×100 participants that require more than 12 hours.

Keller and Scholl [24] 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 [48] and the SPDZ MPC protocol [8]. They report an experiment that matched 128×128 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 [61], which is 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 take for the same benchmark running on an identical testbed).

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 [7]; placing applicants for pharmacy, optometry, psychology, dentistry and other residencies [38]; matching rushees to sororities [35]; and assigning students to public schools in Boston and New York City [52]. Most of these are significantly smaller than the scale demonstrated in 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 http://www.oblivc.org.

Acknowledgments

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

References

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

[2] 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.

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

[4] 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.

[5] 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.

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

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

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

[9] 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.

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

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

[12] 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.

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

[14] 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.

[15] 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.

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

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

[18] 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 Conference on Computer and Communications Security*, 2012.

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

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

[21] Andreas Holzer, Martin Franz, Stefan Katzenbeisser, and Helmut Veith. Secure Two-Party Computations in ANSI C. In *ACM Conference on Computer and Communications Security*, 2012.

[22] Yan Huang, David Evans, Jonathan Katz, and Lior Malka.

Faster Secure Two-Party Computation using Garbled Circuits. In *USENIX Security Symposium*, 2011.

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

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

[25] Jon Kleinberg and Éva Tardos. *Algorithm Design*. Pearson, 2005.

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

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

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

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

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

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

[32] 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.

[33] 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.

[34] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. ObliVM: A Programming Framework for Secure Computation. In *IEEE Symposium on Security and Privacy*, 2015.

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

[36] 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.

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

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

[39] 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.

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

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

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

[43] 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.

[44] 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.

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

[46] 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.

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

[48] 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 Conference on Computer and Communications Security*, 2013.

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

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

[51] 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.

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

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

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

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

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

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

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

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

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

[61] 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.