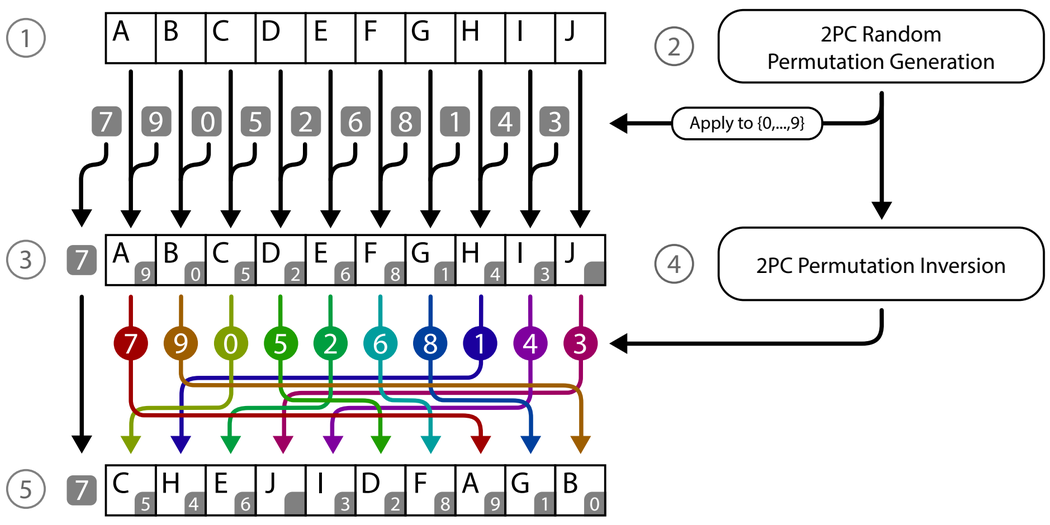
# Secure Stable Matching
## An Efficient Solution Using Multiparty Computation

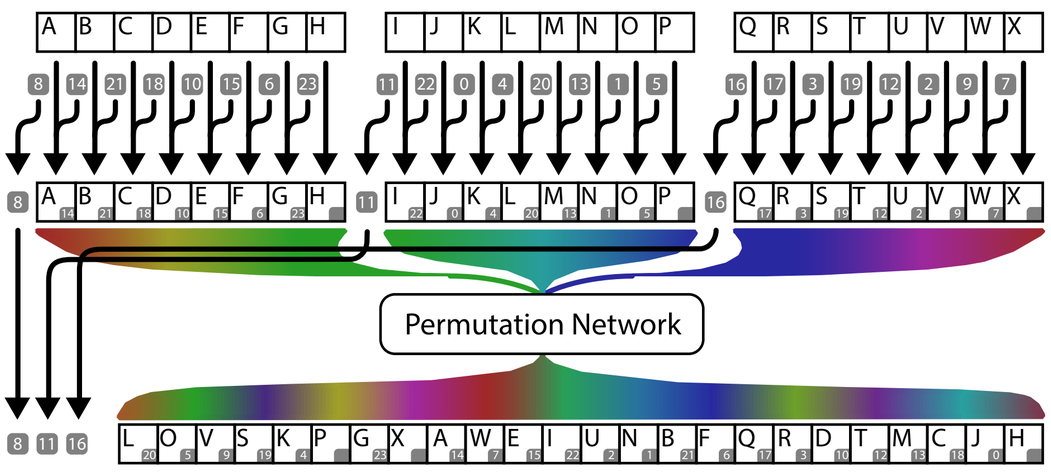

Jack Doerner
jhd3pa@virginia.edu

David Evans
evans@virginia.edu

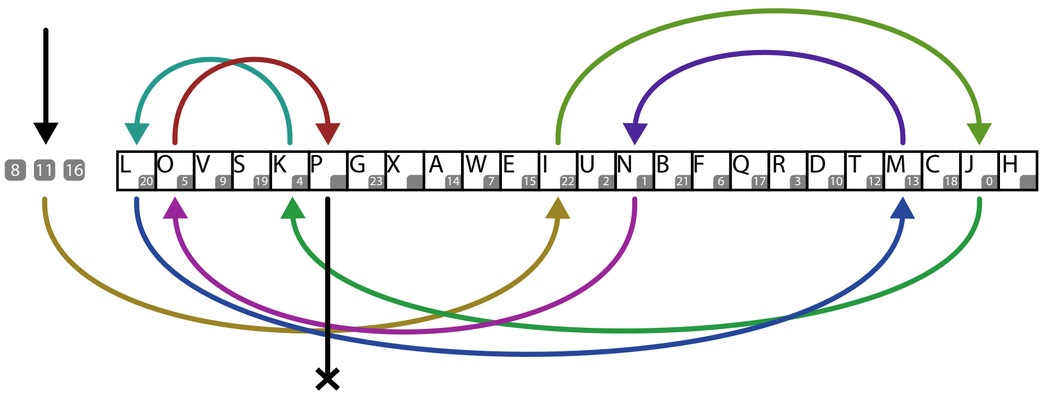
abhi shelat
abhi@virginia.edu


**The initialization of an Oblivious Linked List**




**The interleaving of three input arrays to form an Oblivious Linked Multi-List**




**The traversal of one of three interleaved lists.**

```
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

define SecureGaleShapley(⟨SuitorPreferences⟩, ⟨ReviewerPreferences⟩, n):
    ⟨Preferences⟩ ← ∅
    for i from 0 to n − 1:
        for j from 0 to n − 1:
            ⟨Preferences⟩_{i∗n+j} ← {⟨si⟩ ← i, ⟨ri⟩ ← j, ⟨rr⟩ ← ⟨ReviewerPreferences⟩_{i∗n+j}}
        ⟨Preferences⟩_{i∗n:(i+1)∗n−1} ← BatcherSort(⟨Preferences⟩_{i∗n:(i+1)∗n−1}, ⟨SuitorPreferences⟩_{i∗n:(i+1)∗n−1})
    for i from n² to 2n² − n − 1:
        ⟨Preferences⟩_i ← {⟨si⟩ ← ∅, ⟨ri⟩ ← ∅, ⟨rr⟩ ← ∅}
    {⟨multilist⟩, ⟨entryPointers⟩} ← InitializeMultilist(⟨Preferences⟩, {0, n, 2n, ..., n²})
    UnmatchedSuitors ← new oblivious queue
    for i from 0 to n − 1:
        UnmatchedSuitors ← QueuePush(UnmatchedSuitors, {i, ⟨entryPointers⟩_i})
    ⟨dummy⟩ ← ⟨entryPointers⟩_n
    ⟨done⟩ ← false
    ReviewerMatches ← new ORAM
    for i from 0 to n² − 1:
        ⟨if⟩ ¬QueueIsEmpty(UnmatchedSuitors):
            {⟨nextSuitor⟩, ⟨p⟩} ← QueuePop(UnmatchedSuitors)
        ⟨else⟩:
            ⟨p⟩ ← ⟨dummy⟩
            ⟨done⟩ ← true
        {⟨ProposedPair⟩, ⟨p′⟩} ← TraverseMultilist(⟨multilist⟩, ⟨p⟩)
        ⟨if⟩ ⟨done⟩ = true:
            ⟨dummy⟩ ← ⟨p′⟩
        ⟨else⟩:
            {⟨CurrentPair⟩, ⟨p″⟩} ← OramRead(ReviewerMatches, ⟨ProposedPair⟩.⟨ri⟩)
            ⟨if⟩ ⟨CurrentPair⟩ = ∅ ∨ ⟨ProposedPair⟩.⟨rr⟩ < ⟨CurrentPair⟩.⟨rr⟩:
                ReviewerMatches ← OramWrite(ReviewerMatches, {⟨ProposedPair⟩, ⟨p′⟩}, ⟨ProposedPair⟩.⟨ri⟩)
                ⟨if⟩ ⟨CurrentPair⟩ ≠ ∅:
                    UnmatchedSuitors ← QueuePush(UnmatchedSuitors, {⟨CurrentPair⟩.⟨si⟩, ⟨p″⟩})
    ⟨Result⟩ ← ∅
    for i from 0 to n − 1:
        {⟨CurrentPair⟩, _} ← OramRead(ReviewerMatches, i)
        ⟨Result⟩_i ← ⟨CurrentPair⟩.⟨si⟩
    return ⟨Result⟩
```

A Stable matching is a bijection between two sets of participants (the Suitors and the Reviewers) such that there is no potential Suitor-Reviewer pair who would rather be matched to each other than to whomever they have been assigned.

Stable matching algorithms are used in many interesting applications, including matching medical residents to residency programs, students to schools, and candidates to sororities, as well as in special types of auctions and in managing supply chains. In practice, stable matching processes are often outsourced to a trusted arbiter in order to hide the participants' reported preferences from their counterparties. We develop a method to run instances of stable matching using secure computation, in order to obviate the need for a trusted third party, while preserving the participants' privacy.

We observe that in the classic Gale-Shapley algorithm, each suitor'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* 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.

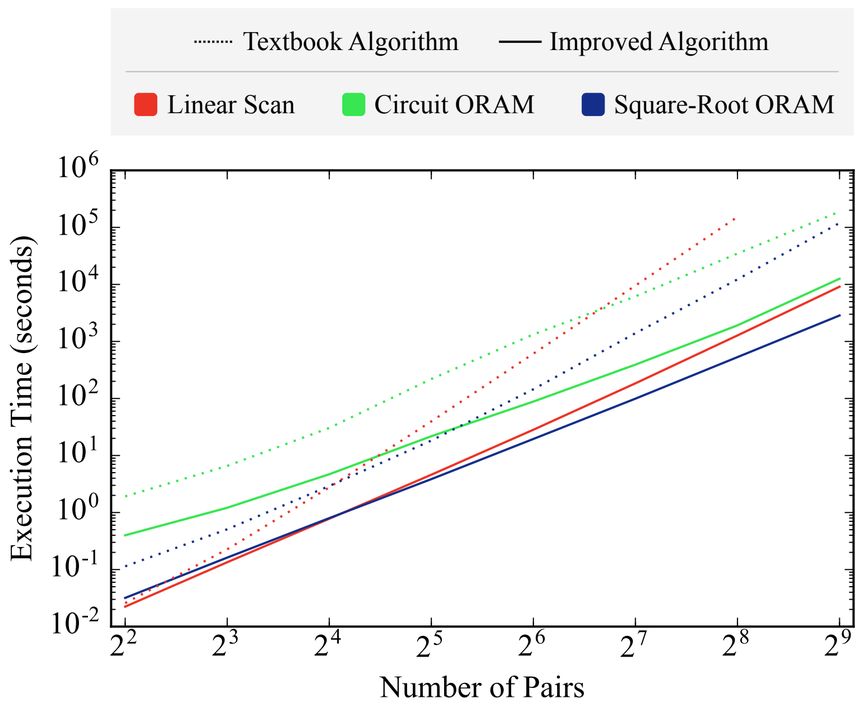
Instead of using a generic ORAM, we design a new data structure to satisfy these requirements more efficiently, which we call an *oblivious linked list*. This construction works similarly to an ordinary linked list in that it consists of a series of elements, each having, in addition to its data, a reference to the next element in the series. However, the list is jointly permuted by both parties according according to some shared randomness, and the forward references are stored in garbled form.

To construct an oblivious linked list, we first generate an oblivious permutation and its inverse. 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, and store the first element of the inverse permutation in a variable.

Unlike an ORAM or an oblivious queue, our oblivious linked list can be traversed in constant time. Each successive access is performed by revealing the physical index of the current element to both parties, who will find the physical index of the next element stored in garbled form along with the element's data.

This construction permits us to iterate through a single preference list. 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 an ORAM of $n$ elements in order to maintain the current matches for each of the reviewers, removing the otherwise necessary $n^2$ 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.




**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 64 pairs or fewer, we collected 30 samples; for benchmarks of 128 and 256 pairs we collected three samples; and for benchmarks of 512 pairs we collected one sample.