Abstract

Encrypted search systems are on the cutting edge of privacy-preserving technologies, allowing users to both gain insights into their data and allowing cloud storage providers to offer a valuable service, all the while keeping users’ sensitive data private. In the recent past, operating on encrypted data meant a choice between privacy and functionality, but new systems like Coeus and DORY promise to give us both.

This report describes our improved design of a Coeus-like system for privately searching and accessing documents from a public archive that, unlike Coeus, requires two (or more) noncolluding compute parties, where distributed trust allows for several key optimizations which would not be possible in a single-server system that uses homomorphic encryption. We conclude that the federated threat model allows our system to achieve significant performance improvements over the original Coeus system, plus realistic scalability, with costs in the order of a few hundred megabytes per search query and sent between servers—not users—as well as only light computation and bandwidth required on the user’s own device.
1 Objectives

A querying party should be able to send a constant-size search query to a group of independently-run compute servers and get a constant-size response. The query should represent a set of keywords, and the response should represent a list of the most relevant documents containing any of those keywords. The set of possible keywords is known to all parties, and the set of all documents is at least known to the servers.

2 Results and Continued Work

We have implemented an initial proof-of-concept demo in Python and then an optimized version in Rust according to the specification below, with two exceptions: (1) the lattice-based private puncturable pseudorandom function \[^3\] (PPPRF) for keyword encoding is not ready, and (2), while we have confirmed that Checklist’s \[^4\] two-server private document retrieval (two-server PIR) works with our Wikipedia-article-based dataset, we have not yet integrated it into the existing Rust implementation or optimized the bucketing.

We’re building the PPPRF part of our implementation over the new OpenFHE library \[^2\]. It will have uses beyond this search system if it is efficient enough, or otherwise it will show the difficulty of building practical systems that depend on advanced cryptographic primitives. Whatever the status of the PPPRF, encoding keywords as keys or secret shares is still more efficient (in cost and bandwidth) than the homomorphic ciphertexts Coeus requires a client to generate and transmit.

The system scales at worst log-linearly in time with respect to all parameters. The bottleneck is in the modular reductions and bit decomposition, which is due to Paillier and the thousands-of-bits-long Paillier keys being considerably larger than the other values of data in this protocol (such as the 12-bit document relevancy weights). The current document ranking part of the implementation is not practical with full-length Paillier keys, but we have proposed an alternate ranking protocol given in the appendix A, which would eliminate the need for Paillier and modular reduction altogether.

We have arrived at the design outlined in this report and found several variations offering trade-offs between compute cost and bandwidth. The choice of final design depends on the results of future implementation and benchmarks. We aim to submit the finalized system design and optimized implementation to NSDI ‘24.

3 Protocol Specification

This specification will focus on the two-server case, but it is meant to generalize (at a linear computation and communication cost) to arbitrarily many compute servers.

Data flowing through this protocol live in the following plaintext spaces:

- The additive group $\mathbb{Z}_{2^\lambda}$ where $\lambda$ is a computational security parameter.

- The additive group $\mathbb{Z}_{2^\ell}$ for some bit length $\ell$, where $2^\ell$ exceeds the maximum document weight (the maximum of the result of the weights matrix multiplied a vector of all 1s is less than $2^\ell$).

- The additive group $\mathbb{Z}_{p_1 q_1}$ for two random primes $p_1$ and $q_1$ whose product, $n_1$, is large enough such that factoring it is hard (except for Server 1 who picked $p_1$ and $q_1$).

- The additive group $\mathbb{Z}_{p_2 q_2}$ for two random primes $p_2$ and $q_2$ whose product, $n_2$, is large enough such that factoring it is hard (except for Server 2 who picked $p_2$ and $q_2$).
• Bits, in the ring \((\mathbb{Z}_2^*)^\ell\)

Our parameter choices will be something like:

• \(\lambda = 128\) bits
• \(\ell = 12\) bits, and a max. weight of <4,000
• \(N = 27,252\) documents
• \(M = 196,315\) keywords
• \(\log_2 n_i \geq 1024\)-bit Paillier modulus

3.1 Query Generation

The client (querying party) takes a user input which is a string of space-separated keywords and computes a private puncturable pseudorandom function (PPPRF) key \(k'\), from a main key \(k\), such that the vectors \(\langle f_k(i) : i \in [0, 1, ..., M]\rangle\) and \(\langle f_{k'}(i) : i \in [0, 1, ..., M]\rangle\) sum element-wise to the (likely sparse) vector \(\langle 0, 0, 0, ..., 1, 0, 0, ..., 1, ..., 0 \rangle\), or similar. In it, keywords are encoded by 1’s if they are present in the query and 0’s if they are not. The server that gets the main key is instructed to invert its PRF output (for the intention of working with additive shares going forward).

3.2 Computing Document Relevancies

The two servers each expand their PPPRF keys into vectors of secret shares. These are in the group \(\mathbb{Z}_{2^\lambda}\) to begin with but are reduced to shares of the same secrets in \(\mathbb{Z}_{2^\ell}\) by truncating all but the least/last \(\ell\) bits. (This is fine algebraically because \(2^\ell\) divides \(2^\lambda\).) (Presumably \(\ell < \lambda\).)

The servers each take their expanded keyword vector and locally multiply it by the publicly-known weights matrix. (This is fine because it only requires public/constant multiplications on the shares and secret additions, neither of which require communication between the servers, and which cost the same to operate on the secret shares as on the secret values themselves.)

3.3 Servers’ Joint Computation

At this point, each server has shares of the secret vector of documents’ weights, in \(\mathbb{Z}_{2^\ell}^N\). We want to return the indices of the greatest \(k\) elements back to the client.

To do this, the servers

1. obliviously shuffle the weights vector, encrypting the corresponding document metadata (the associated document’s index/ID) each time it is shuffled
2. decompose weights (now modulo \(n_2\) due to shuffling) into bit shares
3. reduce weights’ bits modulo \(n_2\) (if not already done as part of the previous step)
4. reduce weights’ bits modulo \(n_1\)
5. reduce weights’ bits modulo \(2^\ell\) via truncation.
6. run a top-\(k\) algorithm on the weights vector
7. return top-\(k\) values’ shares and indices
3.3.1 Oblivious Shuffle

Each of the shared values of the $N$ secret document relevancy weights are in $\mathbb{Z}_{2^\ell}$. We implicitly lift this into the plaintext space, $\mathbb{Z}_{n_1}$ or $\mathbb{Z}_{n_2}$, of a server’s instantiation of Paillier when we encrypt using that server’s public key. The ciphertext space is twice as large, so when sending an encrypted weight, $\text{enc}_1(w_i)$, it will take up $\lceil \log_2 n_1^2 \rceil$ bits of our bandwidth.

Server 1 sends its shares encrypted with its public key. Server 2 homomorphically adds these encrypted shares to its own. It then encrypts the associated indices with a public key (not necessarily a Paillier public key) given by the client, and refreshes the encrypted shares by homomorphically adding equal random values, per share. The additive inverses of those random values become the new shares for Server 2 to use going forward. The encrypted vector of shares is sent back to Server 1 to decrypt and use going forward. Server 1 is now oblivious to the order of the shuffling. This process is repeated once more so that both servers are now unaware of the original index corresponding to any of the secret-shared weights. (Note: the second server must generate its Paillier key such that the public modulus is larger than the previous one, otherwise the secret values will be corrupted instead of safely lifted during the encryption.)

The cost is 2 Paillier key generations, $2N$ Paillier encryptions, $N$ Paillier decryptions, $2N$ Paillier homomorphic additions, per server. Each of the two rounds takes up $N * \lceil \log_2 n_1^2 \rceil$ bits.

3.3.2 Bit Decomposition

Each of the shared values of the $N$ secret document relevancy weights are now in $\mathbb{Z}_{n_2}$. Using precomputed correlated randomness, “bit-decomposition doubles”, we can have each server subtract their share of a random value $r \in \mathbb{Z}_{n_2}$ from their share of $w$, and reveal that sum. They sum once more get the public value of $w - r$ for a weight $w$. They locally decompose this public value into bits, and then perform a joint bitwise addition to get shares of just the bits of $w$, in $\mathbb{Z}$. (That is, if they XOR/sum these bit shares to reconstruct the bits of $w$, then reconstruct $w$, they would get an integer which is congruent to $w$ modulo $n_2$.) The reason they are able to add $r$ to the bits of $w - r$ is because $r$ was precomputed with a corresponding set of bit shares (in $\mathbb{Z}_{2^\lceil \log_2 n_2^2 \rceil}$).

3.3.3 Modulo and Bit-length Reductions

Because the cost of the rest of the protocol depends on the number of bits of the shares of the weights, it is in our best interest to reduce this as much as possible.

When we have the bits of a document’s weight $w \in \mathbb{Z}$, but we know $((w \mod n_2) \mod n_1)$ mod $2^\ell$ is the actual relevancy value of that document. Because modular reduction, modulo a power of two, for bit values is equivalent to truncation (or bitwise AND), we only have to pay a communication cost for the first two bitwise modular reductions. Note: these moduli are both public, but the MPC algorithm we will use for public reduction is only marginally better: $\log_2 (w) - 1$ AND gates versus $\log_2 (w)$, and all of these gates must be done in sequence, (followed by a one-round if-else).

Because the decomposition leaves us with a sum of $(w - r) + r$, and the values of $w - r$ and $r$ are in the range $[0, n_2)$, their sum must then be in the range $[0, 2n_2 - 1)$. This means computing $w \mod n_2$ (a $\lceil \log_2 (n_2) \rceil$-bit value) from $w$ (a $\lceil \log_2 (n_2 + n_2) \rceil = \lceil \log_2 (n_2) \rceil + 1$ bit-long value) can be done with one subtraction in the most expensive case. (If we compute “borrow, difference := subtract_bits(xs, ys)” then “if_else_bits(borrow, xs, difference)” is the modular

\footnote{Keys can be reused across queries, or even optimized for a particular fast-to-operate-on modulus.}
reduction of $x$s with respect to $y$s.) The communication cost of this step is $\lceil \log_2 (n_1) \rceil + \lceil \log_2 (n_2) \rceil$, per server. Each communication payload takes up $2N$ bits.

### 3.3.4 Top-$k$ Each of the shared values of the $N$ secret document relevancy weights are now in $\mathbb{Z}_2^\ell$. Computing a less-than comparison jointly for shares and secret values in $\mathbb{Z}_2^\ell$ is as simple as a subtraction where you only care about the borrow, a.k.a. the overflow bit. If there is a borrow remains at the end, the first operand is strictly less than the second. All comparisons in each recursive level/frame of the top-$k$ algorithm can be done in parallel. This means the communications between the servers when computing the top $k$ can be batched, to one round per/times the recursive depth.

We don’t have to perform a perfect top-$k$ computation either. If the client wanted the top 10 most relevant results, we can return a list of 30 documents, with the guarantee that the top 10 is a subset of it. Of course, here, the client will then need the actual weight values of these documents, in order to manually sort out the top $k$. This can be fine, but do note that then there we would have to save the weights.

It may be that the top-$k$ algorithm is not stable. If this is the case (it depends on the pivot selection), there is the possibility of leaks that reveal the proportion of identical weight values to unique values. An attacker could use this to narrow down the set of possible search queries. (Example: compare $x$ and $y$ and learn $x < y$ is false, but also $y < x$ is revealed to be false, thus $x$ and $y$ are identical weights.) We can instead decide whether and item should be treated as strictly larger or smaller than the pivot by computing not just $w_i < w_j$ but rather $(w_i < w_j) \oplus ((w_i = w_j) \land (i < j))$, for a pivot $w_j$. In other words, we will consider $w_i$ to be less than $w_j$ if it actually is, but if it happens to be equal in value to $w_j$, we appeal to its index in the original array, and consider it less-than if it sits earlier in the original array. By doing so, we will never compare two elements $w_i = w_j$ with $i < j$ and reveal $w_i \neq w_j$, only ever $w_i \neq w_j$. This solution only works in the semi-honest model — even if we compute the compare the indices under MPC, it only takes one server to “randomize” the shares involved and have 50% chance of having $y < x$ revealed instead of $x < y$. Alternatively we could multiply each weight by the maximum possible document index and add to this scaled weight its document index, which would guarantee no two weights are exactly equal. However, this would mean extending the bit-length parameter $\ell$ to $\ell + \lceil \log_2 N \rceil$.

One might naïvely think, “Why not just cache comparisons and skip those we have already done?” However, while this would prevent revealing $y < x$ after previously revealing $x < y$, it would not prevent a slightly more advanced, transitive attack. (Example: compute $x < y$, $y < z$, $z < x$ and if all are false, then $x = y = z$.) Thus, we still need to check the equality under MPC as described above. So, we may certainly cache old comparisons to be more efficient, but this is no more than that—an optimization—and will not on-its-own solve the equality-leaking issue. This technique does not guarantee stability, but rather something weaker. We don’t require stability, after all if we wanted to protect the order of the documents (which would be deterministic in some ways when top-$k$ is stable) after sorting them, the servers could jointly shuffle the final results given to the client (who is the only one who can individually decipher them). The top documents’ indices (which are encrypted with the client’s public key) are returned to the client who then requests the full contents using Checklist’s two-server PIR protocol.

---

2The pseudocode shows a comparison between relative indices, which usually change between sub-problems. The implementation correctly uses the absolute indices.

3Technically this would be “covert-secure”, because if $x < y$ and $y < x$ are both revealed to be false, both servers will know, and thus realize someone acted maliciously.

4If execution does not reach the third-to-last line of code (see next page), then that run of top-$k$ was stable.
Pseudocode for the top-k function the two servers jointly compute is shown below.

```plaintext
top_k(ws: int[], k: int, d=2, rec_depth=1) -> int[]:
  if k == 0:
    return []
  else:
    m = len(ws)
    if m == 1: return ws
    smaller_than_pivot = []
    greater_than_pivot = []
    pivot = hash(m * k * rec_depth)  // The choice of pivot is pseudorandom.
    pivot = ws[pivot_i]
    comp = [()] * m
    for i in 0..m:  // Do all joint operations for the round in parallel.
      lt, eq = compare_lt_eq_bits(ws[i], pivot)
      comp[i] = lt XOR (eq AND (i < pivot_i))  // feign stability
    if d * k <= count:
      return top_k(greater_than_pivot, k, d, rec_depth+1)
    else if k <= count < d * k:
      return greater_than_pivot
    else if count < k:
      rest = top_k(smaller_than_pivot, k - count, d, rec_depth+1)
      greater_than_pivot += rest  // This can break stability.  
      // This combined length is always between k (inclusive) and dk (exclusive).
      return greater_than_pivot
```

3.4 Cost Estimates

ℓ is the number of bits needed to store a document’s relevancy weight.

λ is the computational security parameter.

N is the number of documents.

M is the number of keywords.

m is the cost of a multiplication in \(\mathbb{Z}_{n^2}^*\) (i.e. in the Paillier ciphertext space).

<table>
<thead>
<tr>
<th>Stage</th>
<th>Rounds/server</th>
<th>Bandwidth/round (bits)</th>
<th>Local computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keyword encoding</td>
<td>1</td>
<td>λ</td>
<td>Θ(1)</td>
</tr>
<tr>
<td>Keyword decoding</td>
<td>0</td>
<td>0</td>
<td>Θ(N · M)</td>
</tr>
<tr>
<td>Matrix-vector multiplication</td>
<td>0</td>
<td>0</td>
<td>Θ(N · M)</td>
</tr>
<tr>
<td>Oblivious shuffle</td>
<td>2</td>
<td>N · [2 log₂ n_i]</td>
<td>m · [6N log₂ n_i + 1]</td>
</tr>
<tr>
<td>Bit decomposition</td>
<td>[log₂ n_2]</td>
<td>2N</td>
<td>Θ(N)</td>
</tr>
<tr>
<td>Modular reduction (n_2)</td>
<td>[log₂ n_2]</td>
<td>2N</td>
<td>Θ(N)</td>
</tr>
<tr>
<td>Modular reduction (n_1)</td>
<td>[log₂ n_1]</td>
<td>2N</td>
<td>Θ(N)</td>
</tr>
<tr>
<td>Modular reduction (ℓ)</td>
<td>0</td>
<td>0</td>
<td>Θ(N)</td>
</tr>
<tr>
<td>Top-k</td>
<td>~ ℓ log N</td>
<td>&lt; 2N</td>
<td>O(N)</td>
</tr>
</tbody>
</table>
If we total these up, letting $l_n$ be the largest Paillier modulus’s bit length, and assuming an average case run-time for the randomized top-$k$, then the total number of messages sent (per compute server) is determined by $3(l_n + 1) + \ell \log_2 N$, and the total bandwidth overall is $\lambda + 8Nl_n + N\ell \log_2 N$.

For reference, with 1024-bit Paillier keys, 12-bit weights, and as many documents as in Simple English Wikipedia, the servers would transmit around $128 + 8 \times 27252 \times 1024 + 27252 + 12 \times \log_2 27252 \approx 28.5 \text{ MB}$ over $3 * (1024 + 1) + 12 \times \log_2 27252 \approx 3,250$ rounds. If the network latency is 5 milliseconds, and the local computation of each round takes a similar time, it would take 33 seconds to answer a query.

For reference, with 1024-bit Paillier keys, 16-bit weights, and as many documents as in the entire English Wikipedia, the servers would transmit around $128 + 8 \times 6573939 \times 1024 + 6573939 \times 16 \times \log_2 6573939 \approx 7.03 \text{ GB}$ over $3 * (1024 + 1) + 16 \times \log_2 6573939 \approx 3,450$ rounds. If the network latency is 5 milliseconds, and the local computation of each round takes a similar time, it would take 35 seconds to answer a query. Note that the number of keywords does not affect communication cost in round or bandwidth, only the time it takes to expand the query and perform a (local, sparse, and very fast) matrix multiplication.

### 3.5 Privacy upon Collusion

If all of the servers collude, an (unordered) list of keywords used by any client, past or present, can be recovered. Everything else needed to run a query (locally, even) is already known to the servers.

### Acknowledgements

A huge thanks to my advisor, Professor Malte Schwarzkopf, as well as Professor Mayank Varia, Kinan Albab, and Junewoo Park for their inspiration, contributions, and feedback on this project.

### References


A Alternate Oblivious Shuffle

Each of the shared values of the $N$ secret document relevancy weights are in $\mathbb{Z}_{2^\ell}$.

\[
W = W_1 + W_2 \in \mathbb{Z}_{2^\ell}^N \\
M = M_1 + M_2 \in \mathbb{Z}_{2^\ell}^N \\
\pi(M_i) = \pi(M)_i \\
R \in \mathbb{Z}_{2^\ell}^N
\]

During a preprocessing phase, the servers compute a set of random masks $M$ that only Server 1 learns, a permutation $\pi$ that only Server 2 learns, and secret shares of the permuted $M$ (that each server receives their half of).

Correctness: $\pi(W - M_2 - R) + \pi(R + M_2) = \pi(W) = \pi(W_1 + W_2)$

Security: Server 1 does not learn $\pi$. Server 2 does not learn $M$. Neither learn $W$. 