FHE Benchmarking: Fetch-by-Cosine-Similarity Workload

August 17, 2025

Abstract

This document describes an FHE implementation of a fetch-by-cosine-similarity workload, where both the dataset and the query are encrypted. Fetch-by-similarity is one of the first workloads to be incorporated into the FHE-benchmarking suite that is developed under the auspices of HomomorphicEncryption.org. The main goal of the procedure that we describe here is to serve as a reference implementation, where clarity is more important than speed. However, we still incorporated a number of simple optimizations into that procedure, so as to be able to run (at least some instances of) this workload in reasonable time.

This workload implements a vector-search functionality: The dataset is a key-value store with unit-length real vectors as keys and arbitrary (short) bit-strings as payload values. A query is another unit-length real vector, and the goal is to fetch the few values corresponding to the keys whose inner-product with the query vector is above some threshold. In particular, the benchmark specification calls for supporting payloads of about 10 bytes, returning upto 32 of them per query. The spec includes three instance sizes: small (50,000 records with dimension-128 keys), medium (1M records with dimension-256 keys), and large (20M records with dimension-512 keys). The implementation that we describe here can process the smallest instance in about 2 minutes, on a contemporary (CPU-only) machine.

List of Corrections

1 The Fetch-by-Cosine-Similarity Workload Specification

The fetch-by-Cosine-similarity workload implements a vector-search functionality, where the dataset is a key-value store with unit-length vectors as keys and arbitrary (short) bit-strings as payload values. The queries are also unit-length vectors, and the goal is to fetch a small number of payload values, corresponding to keys that are close to the query in terms of Cosine similarity.

In more detail, let N be the number of records in the dataset. Each record $i \in [N]$ is of the form $(\vec{k}_i, \mathsf{pload}_i)$, with $\vec{k}_i \in \mathbb{R}^d$ a real vector normalized to ℓ_2-norm of 1, and $\mathsf{pload}_i \in \{0,1\}^m$. A query $\vec{q} \in \mathbb{R}^d$ is likewise normalized to ℓ_2-norm of 1. The workload comes with a "promise" that there exist at most $t = 32$ vectors $\vec{k}_i$ in the dataset with better than 0.8 Cosine-similarity to the query, and all others vectors are much less similar. (See below for how this promise is implemented.) Specifically, there are at most t indexes i such that $\vec{q} \cdot \vec{k}_i > 0.8$, and for all other records $\vec{q} \cdot \vec{k}_i \ll 0.8$ (where '·' denotes the dot product). Below we say that a vector $\vec{k}_i$ is similar to the query $\vec{q}$ if $\vec{q} \cdot \vec{k}_i > 0.8$, and denote

$$\mathsf{Sim}(\vec{q}) = \{i \in [N] : \vec{q} \cdot \vec{k}_i > 0.8\}. \tag{1}$$

The workload includes two interfaces that benchmark submitters must implement:

- Count-matches. On query vector $\vec{q}$, the answer is the number of similar vectors $\vec{q}$. That is, the cardinality of the set $\mathsf{Sim}(\vec{q})$.
- Fetch payloads. On query vector $\vec{q}$, return the payloads pload_i for all $i \in \mathsf{Sim}(\vec{q})$.

The workload also specifies three different instance sizes, namely (N, d, m) tuples. (Submitters need not implement all three, instead each submitter can implement and report the results of any subset.) The specification uses $m = 84$ for the payload-size in all three instance sizes,[1] and the following pairs of number-of-records N and dimension d:

[1] That setting is rather arbitrary, it just happens to be the bitlength supported by our reference code.

- Small: $N = 50,000$ records with keys of dimension $d = 128$.
- Medium: $N = 1,000,000$ records with keys of dimension $d = 256$.
- Large: $N = 20,000,000$ records with keys of dimension $d = 512$.

The benchmarking suite harness contains a script that can be called to run the implementation of submitters, that script accepts command-line arguments to specify which interface of what instance size to run.

Submission to the benchmarking suite must set the implementation parameters so as to achieve security level of at least 128 bits. If they rely on the hardness of LWE variants then their parameters must be set to yield 128 bits of security according to Table 5.2 or Table 5.3 in the HE-security-guidelines document of Bossuat et al. [3].

1.1 Implementing the "Promise"

The fetch-by-similarity workload harness generates a synthetic dataset and queries at random, so that they meet the promise above with overwhelming probability. It begins by choosing a set of $c = \lfloor N/32 \rfloor$ "centers", uniformly at random and independently on the dimension-d unit sphere (so with high probability they are nearly orthogonal to each other, see the analysis below). Eric: For all sizes, $N/32 \gg d$, so how can this be true? Experimentally confirmed: for the small setting, I quickly found two random centers with dot product > 0.8. Shai: By the analysis below, any two centers are more similar than 0.7 with probability about 2^{-63}. Perhaps your experiments had a bug? Denoting the resulting set of centers by $(\vec{c}_1, \ldots, \vec{c}_c)$, each one of the keys $\vec{k}_i$, as well as the query vector $\vec{q}$, is sampled using the Sample-point procedure below.

Eric!

Shai!

Algorithm 1 The sampling procedure, returns either a random point or near one of the centers.

```
procedure SAMPLE-POINT($\vec{c}_1, \ldots, \vec{c}_c$:)    ▷ The $\vec{c}_j$'s were chosen at random on the unit sphere
    $\vec{r} \leftarrow$ a fresh random point on the unit sphere in $\mathbb{R}^d$
    With probability 50%:
        return $\vec{r}$
    Otherwise:
        Choose a random index $j \in \{1, \ldots, c\}$
        Set $\vec{r} := \vec{c}_j + 0.3 \cdot \vec{r}$
        return $\vec{r}/\|\vec{r}\|$                                ▷ Normalize to unit length
```

That procedure outputs an independent point on the unit sphere with probability 50%, and otherwise it samples a point near a randomly-chosen center $\vec{c}_j$. Hence the dataset ends up having c clusters, each with expected size of 16 points, and $N/2$ other keys that are not in any of the clusters.

Similarly, with probability 50% the query is near one of the centers (so is expected to have around 16 matches), and otherwise it is another random point on the unit sphere (and hence expected to have no matches at all).

1.1.1 Analysis of the Sampling Procedure

Below we show that points that are sampled near the same center are likely to have similarity of more than 0.9, while points that are not sampled near the same center are very unlikely to have similarity more than 0.7.

Upper bound for points not near the same center. We start with an upper bound on the similarity between two uniform random points on the unit sphere . It is easy to see that the expected value of the inner product is $O(1/\sqrt{d})$, reflecting the fact that two random unit vectors in high dimension are nearly orthogonal to each other. For a high-probability bound, we use the theorem below (whose proof can be found on StackExchange [7]):

Theorem 1.1. *[7] Fix any y on the sphere $S^{d-1} := \{x \in \mathbb{R}^d : \|x\|_2 = 1\}$. Let z be a random variable, uniformly distributed on S^{d-1}. Then for any $\epsilon \in (0, 1/\sqrt{2})$, $\Pr[|y^T z| > \epsilon] \leq (1 - \epsilon^2)^{d/2}$.*

By symmetry, the probability $\Pr[y^T z > \epsilon]$ (without the absolute value) is half of the expression above. Substituting $\epsilon = 0.7$ and setting $d = 128$ (for the smallest instance), we get

$$\Pr\left[y^T z > 0.7\right] \leq \frac{(1 - 0.7^2)^{64}}{2} < 2^{-63}.$$

In Fig. 1 we plot the bound above against empirical results for $\epsilon \in [0.1, 0.45]$ that we obtained by measuring the inner product of about 4e+9 random points in dimension $d = 128$. Similar calculations for the medium ($d = 256$) and large ($d = 512$) instances yield bounds below 2^{-125} and 2^{-249}, respectively.

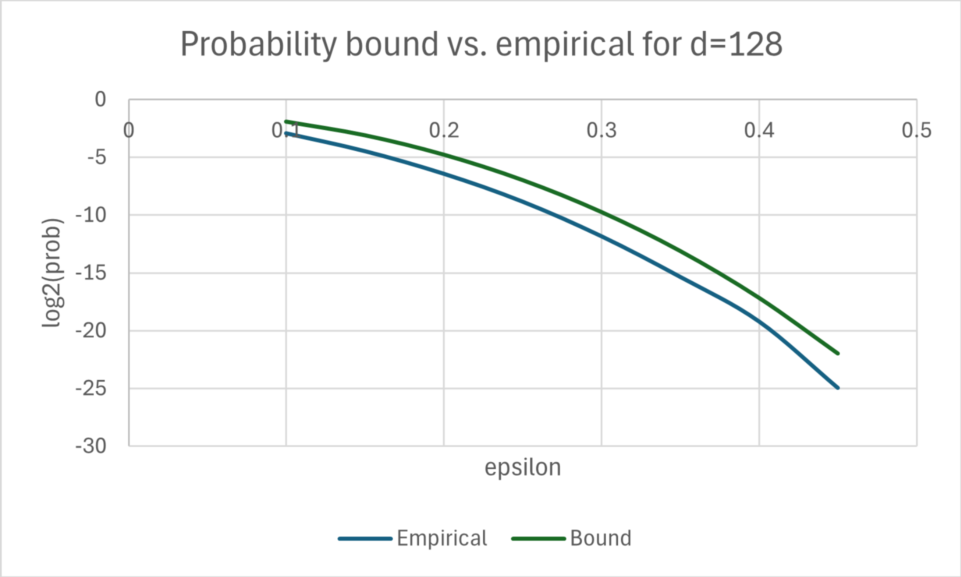


Figure 1: The bounds from Theorem 1.1 vs. empirical results for dimension $d = 128$.

Empirical results for points near the same center. Fixing an arbitrary unit-length center point $\vec{c}$, we recall that points that are chosen near that center point have the form $\vec{v}_i = (\vec{c} + \vec{r}_i)/\ell_i$, where $\vec{r}_i$ is a random point of length 0.3 and $\ell_i = \|\vec{c}+\vec{r}_i\|$. Since $\vec{r}_i$ is a random point on the 0.3-radius sphere, then it is nearly orthogonal to $\vec{c}$, and therefore $\ell_i = \|\vec{c}+\vec{r}_i\| \approx \sqrt{1^2 + 0.3^2} = \sqrt{1.009} \approx 1.044$. For two such points $\vec{v}_1, \vec{v}_2$, the corresponding $\vec{r}_i$'s are also nearly orthogonal to each other, hence

$$\vec{v}_1 \cdot \vec{v}_2 \quad = \quad \frac{\vec{c}+\vec{r}_1}{\ell_1} \cdot \frac{\vec{c}+\vec{r}_2}{\ell_2} \quad = \quad \frac{1}{\ell_1 \ell_2}\big(\|\vec{c}\|^2 + \vec{r}_1 \cdot \vec{c} + \vec{c} \cdot \vec{r}_2 + \vec{r}_1 \cdot \vec{r}_2\big) \quad \approx \quad \frac{1}{1.044^2} \cdot 1 \approx 0.917$$

Indeed, experimental results confirm the above, with 1000 samples we observed an average similarity of 0.9163 and minimum similarity of 0.8945.

2 Our Fetch-by-Cosine-Similarity Implementation

Below we describe our "reference implementation" for the fetch-by-similarity workload as specified in Section 1 above. This implementation could in principle support all three instance sizes, but we only tested it on the small instance size (so far).

2.1 Background: The CKKS Homomorphic Encryption Scheme

Our implementation uses the CKKS cryptosystem [4], as implemented in the OpenFHE software library [1] v1.3. The features of this scheme that are important to our implementations are summarized below.

- CKKS is an approximate-number HE scheme, which means that it works on real/complex numbers and allows some inaccuracy in the computed results. (This is separate from the cryptographic noise that is needed for the underlying LWE problem.)

- The cryptosystem natively supports addition and multiplications of encrypted values (as well as addition and multiplication by non-encrypted constants). Other analytical functions (such as sigmoid or impulse-response) can be approximated by polynomials of sufficient degree.

- CKKS operations are built using polynomial manipulation in an algebraic polynomial ring, modulo a cyclotomic polynomial, and also modulo some integers. For both security and efficiency, the important parameters are the dimension of the ring (i.e., the degree of the cyclotomic polynomial) and the bit-size of the integer moduli.

- CKKS enables packing a vector of plaintext values in a single ciphertext, and performing SIMD operations on the entries of that vector (also called *plaintext slots*). The number of slots in a single ciphertext is half the dimension of the ring in use. The cryptosystem also supports rotation operations on the encrypted vectors (via automorphisms in the underlying cyclotomic ring).

- The integer moduli that are used in CKKS form a *modulus chain*, a set of single-precision moduli $p_0, p_1, \ldots, p_\ell$ (each 40-60 bits long), where at any given time the computation is performed modulo a sub-product of them, $Q_i = \prod_{j=0}^{i} p_i$. We refer to the number of small moduli in the product that was used for an operation as the *level* of that operation. Some operations cause the level to drop, and the number of levels that are consumed by different operation is another important resource.

2.1.1 CKKS Parameters

Our implementation uses CKKS over a ring of dimension $2^{16} = 65536$, which means that each ciphertext can pack upto $2^{15} = 23768$ slots. The entire fetch-by-similarity procedure that we describe below consumes 22 levels (many of which are due to multiply-by-constant operations), so we need at least a 23-level modulus chain. This yields a ciphertext modulus of under 1500 bits, providing more than 128 bits of security. Specifically, the `CCParams` structure of OpenFHE that we use for this procedure is as follows:

```
cParams.SetSecretKeyDist(UNIFORM_TERNARY);
cParams.SetKeySwitchTechnique(HYBRID);
cParams.SetMultiplicativeDepth(23);
cParams.SetSecurityLevel(HEStd_128_classic);
cParams.SetScalingTechnique(FLEXIBLEAUTO);
cParams.SetScalingModSize(42);
cParams.SetFirstModSize(57);
```

With 42-bit scaling and 57-bit first-modulus size, we can get close to $57 - 42 - 1 = 14$-bit numbers in the slots. Specifically, in our implementation the payload values are encoded in numbers in the range [0,512] with precision of 1/16, giving us $9 + 4 = 13$ bits per value. One bit is "wasted" on calibration (see Section 5), so we end up with 12 usable bits per slot.

2.2 The Fetch-by-Similarity Procedure

Recall that we have a dataset with N (key, value) records. The keys are unit-length real vector $\vec{k}_i \in \mathbb{R}^d$ (i.e. $\|\vec{k}_i\| = 1$). Each payload value is an m-bit string ($m = 84$), and below we assume that such a value can be encoded in k plaintext slots (where we will later set $k = 8$). Let $K \in \mathbb{R}^{N\times d}$ denote the matrix with the keys as its rows. The query contains a single secret query (column) vector $\vec{q} \in \mathbb{R}^d$, also of unit length, and we have a promise that no more than $t = 32$ records in the dataset are similar to $\vec{q}$, and the rest are very dissimilar. Namely the set $\mathsf{Sim}(\vec{q})$ from Eq. (1) (or vectors with at least 0.8 similarity to $\vec{q}$) has cardinality at most t, and for every $i \notin \mathsf{Sim}(\vec{q})$ we have $\vec{q} \cdot \vec{k}_i \leq 0.7$.

At a very high level, our implementation consists of the following steps, that will be described separately later in the sequel:

1. **Encoding and encryption.** We encode the matrix K in column-major order (and similarly for the payload), and pack the query in the slot of a single ciphertext. See Section 3.1.

2. **Homomorphic matrix-vector product.** With the matrix encrypted in column-major order, computing the similarity vector $\vec{s} = K \cdot \vec{q}$ only requires multiplying each row by a single entry of the query vector. This is done using a slot-replication procedure, a procedure that outputs ciphertext all of whose slots contain a copy of one entry in the input ciphertext. The replication procedure consumes three multiply-by-contact levels, as described in Section 3.3.

3. **Comparison to threshold.** Once we have the similarity vector, we compute an indicator 0/1 vector $\vec{v}$ by comparing each entry in $\vec{s}$ to the threshold $\tau = 0.8$. Since we are using CKKS, this is an approximate calculation (which means that $\vec{v}$ is an approximate 0/1 vector). This non-linear function consumes six levels, see Section 4.1.

4. **Producing the output.** We now proceed in one of two ways:

   4a. Counting matches. For the interface that only needs to count matches, we simply sum up all the entries in the vector $\vec{v}$ (using the OpenFHE function `EvalSum`) and return the result.

   4b. Fetch payloads. If we need to fetch the payload then we perform the following two steps:

   (i) Extracting each match separately. Using the promise that there are at most $t = 32$ matches, we run t iterations, in each iteration i extracting the i'th match. Namely, in the ith iteration we compute a one-hot vector $\vec{v}_i$, where the only non-zero entry in $\vec{v}_i$ is the i'th non-zero entry of $\vec{v}$. This step consists of two procedures, as explained in Section 4. Between these two procedures, it consume eleven levels.

(ii) Packing the matching payloads. Finally, in each iteration i we extract the payload corresponding to the single 1-entry in $\vec{v}_i$ and move it to a pre-determined set of k slots in the output (e.g., slots $ik \dots (i+1)k - 1$ in the output ciphertext). The result is a single ciphertext that packs all the payloads from the matching records. This step consumes two levels, see Section 5.

2.2.1 A SIMD Optimization

We note that for each query, we expect the vast majority of the records in the dataset to be a non-match, only $t = 32$ matches out of a size-N dataset (with $N \geq 50,000$). We can therefore speed up the computation by splitting the dataset into a number of smaller datasets and applying the same query against all them in parallel. If we split the dataset into p slices, then with high probability no slice will have many more than t/p matches, this enables us to run less than t iterations of Steps 4b(i-ii) above (and it even makes each iteration a little faster).

The downside is that with some small probability, the distribution of matches will be so bad so that one of the slices will have more matches than the number of iterations that we run, resulting in data loss. In our implementation we chose to split the dataset into 512 slices (so the expected number of matches per slice is less that one). The analysis below implies that running Steps 4b(i-ii) for $t = 8$ iterations (rather than thirty-two) is enough, the probability of overflow in any of the slices is bounded below 2^{-46}.

In that analysis, we consider an experiment where a query is chosen first, followed by the dataset. (Of course the probability distribution is exactly the same as in the real procedure that chooses them in the opposite order.) The analysis focuses on a single slice of the dataset (consisting of at most $\lceil N/512 \rceil$ records), and it considers two types of matches between a query and a dataset key: either they were both chosen near the same center, or they were not. We show one bound on the probability of more than one match that was *not* chosen near the same center as the query, and another bound on the probability of more than seven matches matches near the same center. Combining these two (and the union bound over all 512 slices) yields the overall bound.

Not near the same center. Fixing an arbitrary query on the unit sphere, any record key that is not chosen near the same center as the query is just a random unit-sphere point, independent of the query. The analysis from Section 1.1.1 implies that the similarity between the record and query is bounded by 0.7, except with probability 2^{-63} (for the small setting $d = 128$). Since there are at most $\lceil 50000/512 \rceil = 98$ records in each dataset slice, then the probability that two or more of them will be more than 0.7-close to the query is bounded by $\binom{98}{2} \cdot 2^{-126} < 2^{-113}$.

For the medium and large datasets, the equivalent expressions are $\binom{1954}{2} \cdot 2^{-250} < 2^{-229}$ and $\binom{39063}{2} \cdot 2^{-498} < 2^{-460}$, respectively.

Near the same center. Fixing the query fixes one of the centers, and since there are $N/32$ centers and each dataset key is chosen near one of them with probability 1/2, then each key falls near the same center as the query with probability $1/(2 \cdot N/32) = 16/N$. We have 512 dataset slices, each one with $N/512$ records in it, and we want to bound the probability that any one of these slices has more than seven matches.

Denote by M the number of records from one dataset slice that falls near the same center as the query. The random variable M follows the Binomial distribution with $p = 16/N$ and $n = N/512$, so $\mu = E[M] = np = 1/4$. The Chernoff bound for these parameters is quite loose, but a direct calculation using Python's `scipy.stats.binom` yields the bound that we need, about $2^{-55.7}$.

```
$ python
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy as np
>>> from scipy.stats import binom
>>> N=50000
>>> n=np.ceil(N/512)
>>> p=16/N
>>> np.log2(binom.sf(n=n, p=p, k=7))
np.float64(-55.71871915150373)
```

The probability for the medium ($N = 1,000,000$) and large instance ($N = 20,000,000$) was very similar: $2^{-55.355}$ and $2^{-55.34}$, respectively.

Combining these bounds. For any fixed query and each slice of the dataset, the probability of having more than eight matches in that slice is bounded by the sum of the two events from above: either having more than one match which is not near the same center as the query (if there is such a a center at all), or having more than seven matches near the same center as the query. Since there are 512 slices, then we can use the union bound to conclude:

$$\Pr[\text{exists a slice with } M > 8] < 512 \cdot (2^{-55.7} + 2^{-113}) < 2^{-46}.$$

Repeating the same calculation for the medium and large instance sizes yield bounds below 2^{-46} for both.

3 Encrypted Matrix-Vector Product

3.1 Encoding the Input Dataset and Query

The fetch-by-similarity workload features tall and skinny matrices (with many more rows than columns), which need to be multiplied on the right by query vectors: We need to compute $K \cdot \vec{q}$ where $\vec{q} \in \mathbb{R}^d$ and $K \in \mathbb{R}^{N \times d}$ with $N \gg d$. Specifically the benchmark specifies three sizes, $K_{\text{small}} \in \mathbb{R}^{50k \times 128}$, $K_{\text{medium}} \in \mathbb{R}^{1M \times 256}$, and $K_{\text{large}} \in \mathbb{R}^{20M \times 512}$. Moreover, the use-cases for this type of operations typically feature a fairly static matrix, to be multiplied by many query vectors. Hence it is possible to pre-process the matrix before encrypting it, to make the vector-matrix product go faster.

3.1.1 Encrypting the dataset keys

We encrypt the matrix K (whose rows are the keys of the records in our dataset) in a column-major order: Conceptually, the d entries of the key $\vec{k}_j$ for record j will appear in the j'th plaintext slot of d different ciphertexts. In other words, we want the j'th ciphertext to hold in its slots the j'th entries of all the records, namely the j'th column of the matrix K.

In practice, we always have more records in the dataset than plaintext slots in our ciphertexts, so we must hold the j'th column in more than one ciphertext. We will therefore have a total of $n = \lceil N/32768 \rceil$ ciphertexts per column, so a total of $d \times n$ ciphertexts to hold the entire matrix K. Denoting $i_1 = (i \bmod 32768)$ and $i_2 = \lceil i/32768 \rceil$, the entry $K[i, j]$ (for $i \in [N]$ and $j \in [d]$) will be held in the slot of index i_1 in the i_2'nd ciphertext for the j'th column.

We remark that since the key vectors $\vec{k}_j$ are all unit length real vectors, then the numbers that are encrypted in the plaintext slots are all real numbers in the interval $[-1, +1]$, which are well suited for manipulation by the CKKS cryptosystem.

3.1.2 Encrypting the query

On the other hand, the query that the client sends to the server is of small dimension, and so we would like to encrypt it in a single ciphertext. Taking advantage of the fact that the number of slots (32K) is divisible by the dimension of the query (128/256/512), we simply concatenate the query with itself until we have a 32K-dimensional vector, then encrypt that vector and send to the server. (As an aside, repeating the query vector many times to fill the ciphertext makes the replication procedure from Section 3.3 below slightly easier than if we just used zero padding.)

3.1.3 Encrypting the payload

Since our payloads are 84-bit values and our CKKS parameters yield 12 usable bits per plaintext slot, we view each payload as a dimension-7 integer vector over the interval $[0, 4095]$. We divide these number by 16 before encrypting them, thus putting in the plaintext slots number in the range $[0, 256)$ with precision of $1/16$. We can therefore view the payloads of our dataset as a $N \times 7$ real matrix (with entries in $[0, 256)$ and precision of $1/16$), where the payload of record i is stored in the i'th row.

It is important to note that we do not perform complex homomorphic operations on these values during the procedure. We only access them towards the end of the procedure (in Step 4b(ii)), then multiply them by the one-hot vector that we computed in Step 4b(i) and move them to some specified position in the output ciphertext. This entire procedure involves only multiplying by encrypted 0/1 vectors, rotations, and multiplying by 0/1 plaintext masks. Hence we can safely encrypt numbers greater than one in these ciphertext without worrying about the numbers getting too large.

However, we do need to watch for CKKS-processing inaccuracies. Specifically, the one-hot vectors from Step 4b (i) consists of approximate-0's and approximate-1's, which will be multiplied into the payload vectors. While the approximation errors are small, multiplying them by numbers of size upto 256 is a problem. For example, multiplying the approximate 1.01 by the payload value 100 will produce the wrong payload value 101.

To solve this issue, we use the fact that all the payload values for a record will be multiplied by *the same* approximate-1 value: Indeed, in each iteration we compute in Step 4b(i) a single one-hot vector, and multiply this vector entry-wise by each column of the payload matrix. Hence all the entries in a row (corresponding to the payload values for one record) are multiplied by the same entry of the one-hot vector. We therefore add to each record a first "marker" entry containing a known constant, so we use a total of eight slots to encode the payload from each record. (In other words, we insert another column into the encoded payload matrix with the entries all set to the same constant value.)

Denoting the marker value by M, upon decryption we can check if we received some other M' and scale accordingly. Namely, for each record we look up the obtained marker value M', then multiply each of the other entries in that payload record by $16 \times M/M'$ (and round to an integer) to recover the original payload value in $[0, 4095]$.

We note that we also use the same marker value as a work-around for some quirk of our fetch-payload procedure, and set it to the value $M = 512$ which is noticeably larger than all other values in the payload vector. See Section 5 for more details.

3.2 Vector-Matrix Product Using Slot Replication

With the matrix K encoded in a column-major order, we can multiply it on the left by the vector $\vec{q}$, by replicating each entry of $\vec{q}$ in its own ciphertext and then multiplying that ciphertext by the corresponding column of *mat* and sum up all these products. Namely, we use the following simple vector-matrix product procedure:

Algorithm 2 Simple encrypted vector/matrix multiplication.

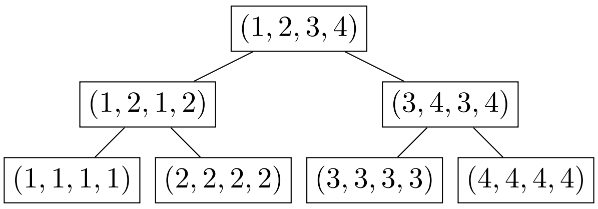
```
procedure EncryptedVMP($\vec{q}, K$)    ▷ $\vec{q}$ is a $d$-vector, $K$ is a $N$-by-$d$ matrix
                                        ▷ $\vec{q}$ is encrypted in the slots of a single ciphertext
                                        ▷ Each column of $K$ is encrypted in the slots of $n$ ciphertexts, $k_{j1} \dots k_{jn}$
    Initialize ciphertexts $a_1 \dots a_n$ to zero    ▷ An encrypted accumulator
    for $j$ in $[d]$ do                               ▷ Proceed one column of $K$ at a time
        Replicate the entry $q_j$ to all the slots of a ciphertext $c_j$
        for $i$ in $[n]$ do                           ▷ go over the ciphertexts of column $j$
            $a_i := a_i \boxplus (c_j \boxtimes k_{ji})$      ▷ Multiply and accumulate
    return $(a_1 \dots a_n)$
```

In Section 3.3 below we describe the slot-replication procedure that we use, generating for each $j \in [d]$ a ciphertext c_j in which all the slots contains the j'th entry of the query vector. We note that the slot replication time depends only on the vector dimension d, not on the dataset size N. Thus, as the dataset gets larger the replication time would become insignificant. In particular, we expect that it will only be a small fraction of the total computation time for the medium and large instances. For the small instance size, however, it still could consume be a large fraction of the total time, and so we would still want to optimize it.

3.3 The Slot Replication Procedure

The slot-replication procedure takes as input a single ciphertext encrypting a dimension-d vector $\vec{q}$, and it output d ciphertexts where the j'th one has the j'th entry q_j in all the slots. Our implementation is closely related to the "full replication" procedure of Halevi-Shoup [5], with a few important differences.

The basic algorithm in [5] is a recursive procedure, where the i'th recursion level takes one ciphertext, encrypting a vector with 2^i distinct elements, and outputs two ciphertext encrypting vectors with 2^{i-1} distinct elements each. This is illustrated below for a 4-element vector:



Each level of this recursion can be implemented with one rotation, two multiplications by plaintext masks, and four additions/subtractions. Hence the overall full-replication procedure has a $\log d$-depth of multiply-by-constant and total complexity of $d - 1$ rotations, $2d - 2$ maskings, and $4d - 4$ additions/subtraction. Halevi-Shoup also described in [5] a shallower version, using trees with higher-degree roots. A recursive level with a degree-δ parent can be implemented using either:

- Brute force replication of each entry with $\delta - 1$ rotations and $O(\delta^2)$ mask/add/subtract,
- Recursive replication of each entry using $\delta - 1$ maskings and $O(\delta \log \delta)$ rotate/add/subtract.

Either way, the multiply-by-constant depth of each level is just one. Halevi-Shoup suggested to use a root of degree $\delta = d/\log d$ (with recursive replication of the entries) and keep all the other nodes of degree two, thus getting depth $\log \log d$ still using only $O(d)$ operations overall.

3.3.1 Our Implementation

We implemented a generalization of the algorithm from [5], and contributed it to the OpenFHE codebase. See `https://github.com/openfheorg/contrib/tree/main/slot-replication`. There are a few differences between our implementation and the algorithm as described in [5]:

- We support trees of arbitrary degrees, possibly a different degree per level, and let the caller determine the tree structure that it wants to use. Our fetch-by-similarity implementation uses a tree of degrees $(8, 4, 4)$ for the small instance size ($d = 128$), degrees $(8, 8, 4)$ for the medium instance size ($d = 256$), and degrees $(16, 8, 4)$ for the large instance size ($d = 512$). [2]
- We allow the caller to start "in the middle of the tree", by providing a partially replicated vector as input. In our implementation for the small instance size, for example, we start from a partially replicated vector of dimension $d = 128$, repeated 256 times to fill all the 32k plaintext slots in a single ciphertext.
- We provide an `init/next-replica` interface, which is geared toward using the replicated outputs one at a time (as needed for our vector-matrix multiplication procedure).
- Perhaps most importantly, our implementation traverses the replication tree in a *depth-first* manner, so it only needs to keep a single root-to-leaf path in memory at any time.
- In nodes with more than two children, we use the brute force replication method rather than the recursive method prescribed in [5]. This has the advantage that all the rotation operations are applied to the same input ciphertext, allowing us to use the "hoisting" optimization of multi-rotation from [6]. That optimization enables computing many rotations of the same input ciphertext "almost for the price of one", and it is implemented via the OpenFHE interface `EvalFastRotation`.

Shai!

Shai: Would be nice to describe some experimental results for just the slot-replication implementation.

4 Computing the Indicator Vectors

As sketched in Section 2.2, after computing the similarity vector $\vec{s} = K \cdot \vec{q}$ we compute a number of 0/1 indicator vectors in order to identify the matches (i.e. the vectors similar to $\vec{q}$) and to fetch the corresponding payloads. We first compute an indicator vector $\vec{v}$ containing all the matches, and then run multiple iterations computing one-hot vectors $\vec{v}_i$ that only have a single one entry in the place of the i'th match $\vec{v}$. As we explained in Section 2.2.1, we split the dataset into $p = 512$ slices and rely on having at most $t = 8$ matches from each slice. Hence the "one-hot vectors" $\vec{v}_i$ that we compute are really one-hot in each slice separately, and we are running $t = 8$ iterations in order to ensure that we get all the matches whp.

For the all-matches vector $\vec{v}$, we compute (approximately) the indicator bit $\chi_>(x) := (x > 0.8)$ in parallel on all the slots in the similarity vector $\vec{s}$. This is described below in Section 4.1. Computing the one-hot vectors is more involved, it consists of the following two steps:

1. **Running-sums.** Applying a running-sum procedure (similar to [5]), we transform the encrypted $\vec{v}$ to an encryption of a running-sum vector $\vec{u}$ with $u_i = \sum_{j \le i} v_j$ for all i. Multiplying entry-wise by the all-matches vector, $\vec{u'} := \vec{u} \boxtimes \vec{v}$, yields an encryption of a vector whose only non-zero entries are the matches (i.e. the 1's in $\vec{v}$), such that the first match has $u'_{i1} = 1$, the second match has $u'_{i2} = 2$, etc.
2. **Equality checks.** Next we run t iterations, the i'th iteration computes the indicator bit $\chi_{=i}(x) := (x == i)$ on all the slots in the vector $\vec{u'}$, yielding the one-hot vector $\vec{v}_i$ that we need. This is described below in Section 4.3.

[2] We run a few experiments to choose those degrees, but nowhere near enough to argue that they are the best possible. Augmenting our slot-rotation implementation with an infrastructure for choosing the best degrees in different environments seems like a useful future-work project.

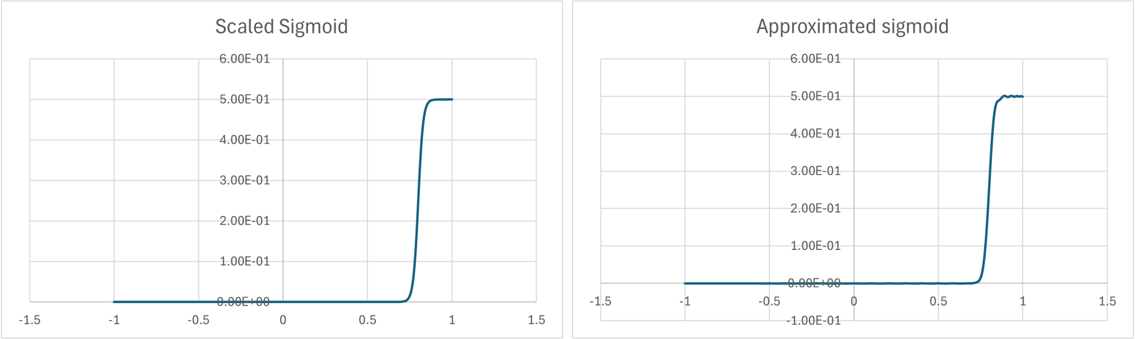


Figure 2: The scaled sigmoid function on the left, and its degree-59 approximation on the right.

Limitation of this method. We stress that our implementation relies on the dataset to have a significant gap between matches and non-matches. Namely, all dataset vectors must either be very similar or very dissimilar to the query, so that approximating $\chi_>(\vec{k}_i \cdot \vec{q})$ is very close to either zero or one.

As we explain below, we approximate the function $\chi_>(\cdot)$ using a polynomial, which is a continuous function: If there exist values $x \in [\pm 1]$ fro which $\chi_>(x) \approx 0$ and others for which $\chi_>(x) \approx 1$, then there must also be values $x \in [\pm 1]$ for which $\chi_>(x) \approx 1/2$. Consider a situation where the first "maybe match" record has such a similarity score, $\chi_>(\vec{k}_i \cdot \vec{q}) \approx 1/2$. Assume further that all the other records have $\chi_>(\vec{k}_j \cdot \vec{q}) \approx 0$ or ≈ 1. When running the two-step process above on such dataset, the running-sums step would yield a vector with non-zero value $\approx 0.5, 1.5, 2.5, \ldots$, so the second step of checking equality to $1, 2, 3, \ldots$ will return no results.

For datasets where such a gap cannot be ensured, using the two-step approach from above seems to require computing the comparison-to-threshold exactly rather than approximately. This could be done either via scheme-switching from CKKS to one of the exact schemes before computing this function, or by using CKKS in a "discrete mode" using techniques similar to Bae et al. [2]. However, to the best of our knowledge none of these options is available in OpenFHE v1.3, which is the FHE library that we use. Hence our implementation would only work for datasets that are "clean enough" to allow the use of the approximate comparison-to-threshold.

4.1 Comparing to a Threshold

Comparison against the threshold is done using Chebyshev-polynomial approximation. But instead of directly trying to approximate the comparison-to-threshold function (which is not smooth or even continuous), we approximate the sigmoid function, which is itself a smooth approximation of the threshold function.

Specifically, we use the OpenFHE interface `EvalChebyshevFunction(func,ct,...)` that gets as input a function, computes a Chebyshev-polynomial approximation of that function (in a given interval), and applies that polynomial to the input ciphertext. The function that we so approximate is a shifted and scaled version of the sigmoid function. Specifically, we use the function

$$f(x) = \frac{0.5}{1 + \exp(-69 \cdot (x - 0.8))}.$$

The shift of 0.8 means that we compare to the similarity threshold 0.8, the constant 69 was determined by some experiments, and the constant 0.5 that scales the output is there to ensure that the result will later fit into out comparison-to-number routine (see discussion later in Section 4.4). We use a polynomial of degree 59 to approximate that function, this takes six levels according to the table at [8]. The function and its approximation in the interval $[\pm 1]$ are depicted in Fig. 2.

4.2 The Running-Sums Procedure

We implemented a generalization of the running-sum procedure from [5]. Specifically, our implementation supports computing efficiently the running sums across multiple ciphertexts, as well as viewing the slots in these ciphertexts as a matrix and computing the running sums across each column separately. (We need the latter extension since we want to compute the running sums for each of the $p = 512$ slices separately, as discussed in Section 2.2.1.) Our implementation also enables a time/depth tradeoff, letting the caller specify a bound on the multiply-by-content depth of the procedure, trading off speed for depth. Ignoring the partition to columns and the depth-bound for now, the basic procedure is described in Algorithm 3.

An important feature of this procedure is that *its running time is almost independent of the number of ciphertexts.* Indeed, the only dependence on the number of ciphertexts n in this procedure is the number of homomorphic additions (specifically $\ell \times n$ additions). The number of shift operations is always $\ell - 1$, regardless of n.

Algorithm 3 Running-sums across multiple ciphertexts.

```
1: procedure RUNNING-SUMS($c_1, \dots, c_n$)        ▷ The $c_i$'s are ciphertexts with $2^\ell$ slots
2:     for $i$ in $[n-1]$ do                          ▷ Compute initial SIMD sums across the ciphertexts
3:         $c_{i+1} := c_{i+1} \boxplus c_i$
                     ▷ For each $c_j$ and slot $i$, we now have $c_j[i] = \sum_{k \le j}$ initial-$c_k[i]$
4:     for $j$ in $[\ell-1]$ do                       ▷ Intra-ciphertext sums
5:         $tmp := c_n \gg 2^j$                        ▷ Zero-fill shift by $2^j$
6:         for $i$ in $[n]$ do                         ▷ Add to all the ciphertexts
7:             $c_i := c_i \boxplus tmp$
8:     return $(c_1, \dots, c_n)$
```

4.2.1 Computing running-sums in multiple columns

It is quite easy to extend the procedure from above to compute separately in multiple columns, when the number of columns divides the number of slots in a ciphertext. Let 2^ℓ be the number of slots and let 2^a be the intended number of columns. We view the slots in the input ciphertexts as a matrix with 2^a columns and $n \cdot 2^{\ell-a}$ rows, as follows: Each column is spread across the slots of a ciphertext in equal strides of size 2^a. That is, the ciphertexts slots holding some column i are exactly those that have index congruent to i modulo 2^a. Then, shifting the ciphertext by $j \cdot 2^a$ has the effect of shifting each column by j, without the different columns interacting with each other. The procedure is almost exactly as above, except that we have j in $[\ell - a - 1]$ on Line 5 and $c_m \gg 2^{j+a}$ on Line 6.

One aspect to watch out for with this procedure is the ordering of rows (i.e. the order of entries within each column), which may not be immediately clear. In order to use a procedure such as above where the number of shifts is independent of the number of ciphertexts, we need to view the matrix as an interleaving of the sub-matrices in the different ciphertexts: The top row in the combined matrix is the top row from the matrix of the 1st ciphertext, the second row is the top row from the matrix of the second ciphertext, and so on up to the n'th row which is the top row of the matrix in the n'th ciphertexts. Then the $n+1$'st row in the combined matrix is the second row from the matrix of the first ciphertext, the $n+2$'nd row is the second row from the matrix of the second ciphertext, etc. This arrangement is illustrated below for three ciphertexts, each with 8 slots, representing a matrix with four columns (and $3 \cdot 8/4 = 6$ rows). The three input ciphertexts with their slots are:

$$\begin{array}{l} ctxt_1 = [a_1 \quad b_1 \quad c_1 \quad d_1 \quad a_4 \quad b_4 \quad c_4 \quad d_4] \\ ctxt_2 = [a_2 \quad b_2 \quad c_2 \quad d_2 \quad a_5 \quad b_5 \quad c_5 \quad d_5] \\ ctxt_3 = [a_3 \quad b_3 \quad c_3 \quad d_3 \quad a_6 \quad b_6 \quad c_6 \quad d_6] \end{array},$$

representing the matrix

$$\begin{bmatrix} a_1 & b_1 & c_1 & d_1 \\ a_2 & b_2 & c_2 & d_2 \\ a_3 & b_3 & c_3 & d_3 \\ a_4 & b_4 & c_4 & d_4 \\ a_5 & b_5 & c_5 & d_5 \\ a_6 & b_6 & c_6 & d_6 \end{bmatrix}.$$

After running the procedure from above (with $a = 2, \ell = 3$, and $n = 3$), we get the ciphertexts

$$\begin{array}{lcccccccc} ctxt_1 = [& a_1 & b_1 & c_1 & d_1 & \sum_{i \le 4} a_i & \sum_{i \le 4} b_i & \sum_{i \le 4} c_i & \sum_{i \le 4} d_i] \\ ctxt_2 = [& \sum_{i \le 2} a_i & \sum_{i \le 2} b_i & \sum_{i \le 2} c_i & \sum_{i \le 2} d_i & \sum_{i \le 5} a_i & \sum_{i \le 5} b_i & \sum_{i \le 5} c_i & \sum_{i \le 5} d_i] \\ ctxt_3 = [& \sum_{i \le 3} a_i & \sum_{i \le 3} b_i & \sum_{i \le 3} c_i & \sum_{i \le 3} d_i & \sum_{i \le 6} a_i & \sum_{i \le 6} b_i & \sum_{i \le 6} c_i & \sum_{i \le 6} d_i] \end{array},$$

which indeed represents the matrix with the running sums in the columns.

4.2.2 Bounding the multiply-by-mask depth

The zero-fill shifts in the procedure above are implementing by a cyclic rotation and multiplication by a 0/1 mask, and so each iteration of Lines 5-7 adds one level of multiply-by-constant depth.

We can trade off depth vs. running-time by collapsing a few iterations into one. For example, consider the two-step process $x_1 := x_0 + (x_0 \gg 2^i)$ and $x_2 := x_1 + (x_1 \gg 2^{i+1})$ that performs two shifts and has multiply-by-constant depth of two. We can achieve the same result with the alternative process $x_2 = x_0 + (x_0 \gg 2^i) + (x_0 \gg 2 \cdot 2^i) + (x_0 \gg 3 \cdot 2^i)$, that has depth of only one but uses three shifts. In general we can collapse v levels into one by increasing the number of shifts that are required from v to $2^v - 1$.

Our implementation let the calling application specify a depth budget b, and it operates within that budget by splitting the procedure into b iterations, each one obtained by collapsing $\lceil (\ell - a)/b \rceil$

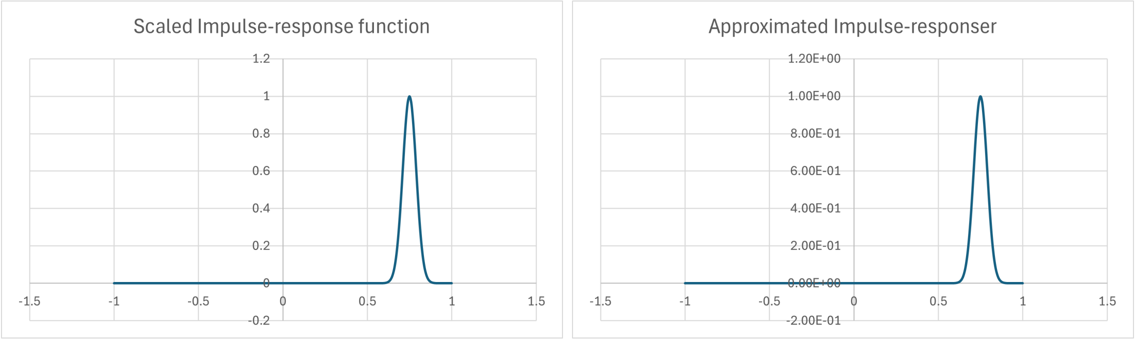


Figure 3: The scaled impulse-response function at 0.75 on the left, and its degree-119 approximation on the right.

levels into one.[3] The full procedure, handling 2^a columns and depth-bound b, is described in Algorithm 4.

Algorithm 4 Running-sums in columns with depth bound.

```
procedure RUNNING-SUMS($c_1, \ldots, c_n$, $a$, $b$)          ▷ The $c_i$'s are ciphertexts with $2^\ell$ slots
                                                  ▷ The slots are viewed as a matrix with $2^a$ columns
                                                  ▷ $b$ is a bound on the multiply-by-constant depth
    for $i$ in $[n-1]$ do                         ▷ Compute initial SIMD sums across the ciphertexts
        $c_{i+1} := c_{i+1} \boxplus c_i$
                                                  ▷ For each $c_j$ and slot $i$, we now have $c_j[i] = \sum_{k \le j}$ initial-$c_k[i]$
    $v := \lceil 2^{\ell-a}/b \rceil$             ▷ How many levels to collapse in each iteration
    $\mathsf{left} := 2^{\ell-a}$                 ▷ How many levels are left
    while $\mathsf{left} > 0$ do                  ▷ Intra-ciphertext sums
        if $v < \mathsf{left}$ then               ▷ Last step may need to collapse fewer levels
            $v := \mathsf{left}$
        $\mathsf{left} := \mathsf{left} - v$
        Initialize ciphertext $acc$ to zero       ▷ An encrypted accumulator
        for $j = 1$ to $2^v - 1$ do
            $acc := acc + (c_m \gg j \cdot 2^{\mathsf{left}})$    ▷ Zero-fill shift by $j \cdot 2^{\mathsf{left}}$
        for $i$ in $[n]$ do                       ▷ Add to all the ciphertexts
            $c_i := c_i \boxplus acc$
    return $(c_1, \ldots, c_n)$
```

4.3 The Equality Check Computation

We again use Chebyshev polynomials and the OpenFHE interface `EvalChebyshevFunction` to implement the comparison-to-number functions. We run eight iterations, the i'th iteration applies to all the slots the non-linear function comparing the slot to the number $x_i = i/4 - 1$. The scaling that we applied to the compare-to-threshold function above ensures that the numbers in all the slots will be in the range $[\pm 1]$, and so it the number x_i. We therefore need polynomials that provide good approximations over the interval $[\pm 1]$ for the functions

$$f_i^*(z) = \begin{cases} 1 & \text{if } |z - x_i| < \epsilon \\ 0 & \textit{otherwise} \end{cases}$$

for a small enough ϵ (say $\epsilon = 0.01$).

As before, instead of directly trying to approximate the non-continuous f_i^*'s, we use a Chebyshev-polynomial approximation for the functions $f_i(z) = f(z - x_i)$ where f is an impulse-response function (with $\sigma = 0.04$)

$$f(z) = \exp(-z^2/2\sigma^2) = \exp(-312.5 \cdot z^2).$$

The constant $\sigma = 0.04$ was determined by some experiments. For these functions, we used Chebyshev polynomials of degree 119, this takes seven levels according to the table at [8]. An example of the function and its approximation in the interval $[\pm 1]$ (for $x_i = 0.75$) are depicted in Fig. 3.

4.4 Our Use of `EvalChebyshevFunction`

As mentioned in Sections 4.1 and 4.3 above, we use Chebyshev polynomials as a good basis for approximating non-linear (smooth) functions over the real interval $[\pm 1]$. OpenFHE provides a utility for using Chebyshev polynomials, by calling

[3] The last iteration may collapse less levels if b does not divide $\ell - a$ exactly.

cyrptoContext→EvalChebyshevFunction(function, ctxt, from, to, degree). Under the hood, this call scales and shifts the ciphertext (via scalar multiplication and addition) to map the interval [from, to] to [±1], then applies to the result a Chebyshev-based approximation in the interval [±1] of the function

$$f(x) := \texttt{function(inverse-shift-and-scale}(x)\texttt{)}.$$

Unfortunately, this implementation means an extra multiply-by-constant operation, consuming one level more than needed, unless the specified interval [from, to] was already [±1]. In our implementation we were therefore careful to ensure that we only ever call that function with [±1] as the input interval.

We use EvalChebyshevFunction twice in our implementation, once to compare the result of the matrix-vector product against the threshold to determine if each entry is a match or not, and the second time after running sums to extract the i'th match in the i'th iteration (cf. the Equality-check step in Section 4).

For the threshold comparison we don't need to scale the input, the dot product between two unit-length vectors is already in the interval [±1]. But we do need to scale the output, since this will later become the input to the equality-check.

Consider scaling the output of the threshold-comparison to (an approximation of) $0/x$ for some number x. After computing the running-sums in the columns, the slots will contain numbers between 0 and $8x$ (since we assume at most right matches per column). This will again be multiplied the by $0/x$ slots of the match vector (to zero-out all the non-matches), so the result will be between 0 and $8x^2$, and we need this to be an interval of size 2 (so we can shift it to [±1]). This means that we need $8x^2 = 2$, so we set $x = 0.5$ as the output scaling factor of the thresholding function.[4] Shai: the footnote looks like a bug to me. Of course, this means that instead of comparing to i in the i'th equality-check iteration, we compare to the scaled version $y_i = i/4 - 1$ (which is between -0.75 and 1).

Shai!

5 Extracting the Payloads

After we compute each "one hot vector" $\vec{v}_i$ for the i'th match, we multiply it by the columns of the payload matrix to extract only the corresponding payload, then move the i'th payload into the i'th block of slots that will hold that payload in the output ciphertext. Since we don't know where the i'th match is located, we first replicate this payload to all the blocks, and then multiply by a mask that only leaves the i'th block and zeros out everything else. Replication is accomplished via a total-sum shift-and-add calculation, namely repeating $x := x + (x \gg 2^i)$ for $i = 0, 1, 2, \ldots$ (where $\gg$ denotes cyclic shift). This will result in each output slot containing the sum of all input slots, and since only one input slot is non-zero then that value will be replicated to all the slots.

However, recall from Section 3 that payloads are not encoded in a single slot, rather each payload is encoded in eight slots (one marker and seven data slots). The simplest way to move all these eight slots to their place is to do it one at a time: We could multiply the one-hot vector by the first payload row and replicate the single non-zero slot in the result to all the slots, and then multiply by a mask that zeros out all but the first slot of the i'th block. Then repeat this process for the 2nd payload row, then the 3rd, etc. until we have all eight slots.

But we can do better: Multiplying the one-hot vector by each of the payload rows we get the eight extracted vectors $x_1, \ldots, x_8$, we can pack them in a single vector

$$y = x_1 + (x_2 \gg 1) + \cdots + (x_8 \gg 7) \quad \text{(where again } \gg \text{ denotes cyclic shift).} \tag{2}$$

We can then move the entire block together using a "column-wise total sum" SIMD procedure. That procedure computes in parallel for each $j = 0, \ldots, 7$ the total sum of all the slots whose index is j modulo 8, and puts that sum in all these slots. Since Eq. (2) ensures only one of these slots has a non-zero value for each j modulo 8, then this value will be replicate to all the relevant slots. Multiplying by a mask that zeros out all but one block, we will end up with the eight payload slots from the i'th mask in eight slots of the target block.[5]

See Algorithm 5 for a pseudocode of the routine that returns the m'th match in each slots ($m \leq 8$). We call this procedure with $m = 1, 2, \ldots, 8$ to extract the (upto) eight matches in each slice, and add all the results to get a single ciphertext that includes all the payloads of all the matches, and will be returned to the client.

[4] In fact, we scale it to $x = 0.504$ since our approximation tends to yield a number slightly smaller than the function that we approximate.

[5] Not only is this method doing only one rather than eight replications, it also reduces the replication time since each slots is replicated to only $n/8$ slots rather than all n of them.

Algorithm 5 The payload extraction procedure.

```
procedure Extract($(c_i : i \in [1,n])$, $(p_{ij} : i \in [1,n], j \in [1,8])$, $a$, $m$)
  – The slots are partitioned to $2^a$ slices
  – The $c_i$'s are approximate 0/1 ciphertexts with $2^\ell$ slots, with at most a single 1 in each slice
  – The $p_{ij}$'s are payload ciphertexts with $2^\ell$ slots: slot $k$ of $p_{ij}$ is the $j$'th payload value for
    record $k + 2^\ell(j-1)$
  – Extracting payload of the $m$'th match into slots $8(m-1), \ldots, 8m-1$ in each slice

    Initialize ciphertext $y$ to zero                          ▷ To hold the payload values
    for $j \in [0,7]$ do                                       ▷ Extract the $j+1$'st payload value in each slice
        Initialize ciphertext $x$ to zero
        for $i \in [1,n]$ do                                   ▷ Go over all the ciphertexts
            $x := x \boxplus (p_{ij} \boxtimes c_i)$           ▷ In each slice, all but (at most) one of the $c_i$'s are zero
        $y := y \boxplus (x \gg j \cdot 2^a)$                  ▷ Rotate each slice by $j-1$ and add
    for $m \in [3, \ell - a - 1]$ do                           ▷ Sum 8-slot windows in each slice via shift and add
        $y := y \boxplus (y \gg 2^{m+a})$
    mask := 1 in slot indexes $i$ s.t. $\lfloor i/2^a \rfloor \in [8(m-1), \ldots, 8m-1]$, 0 otherwise
    return $y \boxtimes$ mask
```

A slot-ordering issue. One quirk of this SIMD procedure is that the eight payload slots may appear shifted inside the target output block. To see that, consider a one-hot vector where the index of the single one-entry is (say) congruent to 3 modulo 8. Multiplying this one-hot vector by all the eight payload rows we get the vectors $x_1, \ldots, x_8$ where all the non-zero payload slots are at the same index which is 3 modulo 8. Computing the row y as in Eq. (2), we would then have the 1st payload slot at index 3 modulo 8, the second at index 4 modulo 8, etc. After replicating and multiplying by the mask, the eight payload slots will appear in the right block, but their order will be shifted, namely (7,8,1,2,3,4,5,6) (with the j'th payload entry appearing in position $j+2$ modulo 8).

Working around this issue, we made sure that the first entry in each payload vector is a marker whose values is at least twice as large as all the other values. Upon decryption, the client can just look for the largest value and know that it corresponds to the marker, thereby making it possible to shift everything back to its right place. (We also recall again that this entire replication-and-extraction procedure is run in parallel in all the slices of the output ciphertext.)

References

[1] A. A. Badawi, A. Alexandru, J. Bates, F. Bergamaschi, D. B. Cousins, S. Erabelli, N. Genise, S. Halevi, H. Hunt, A. Kim, Y. Lee, Z. Liu, D. Micciancio, C. Pascoe, Y. Polyakov, I. Quah, S. R.V., K. Rohloff, J. Saylor, D. Suponitsky, M. Triplett, V. Vaikuntanathan, and V. Zucca. OpenFHE: Open-source fully homomorphic encryption library. Cryptology ePrint Archive, Paper 2022/915, 2022.

[2] Y. Bae, J. Kim, D. Stehlé, and E. Suvanto. Bootstrapping small integers with CKKS. In K. Chung and Y. Sasaki, editors, *Advances in Cryptology - ASIACRYPT 2024 - 30th International Conference on the Theory and Application of Cryptology and Information Security, Kolkata, India, December 9-13, 2024, Proceedings, Part I*, volume 15484 of *Lecture Notes in Computer Science*, pages 330–360. Springer, 2024.

[3] J. Bossuat, R. Cammarota, I. Chillotti, B. R. Curtis, W. Dai, H. Gong, E. Hales, D. Kim, B. Kumara, C. Lee, X. Lu, C. Maple, A. Pedrouzo-Ulloa, R. Player, Y. Polyakov, L. A. R. Lopez, Y. Song, and D. Yhee. Security guidelines for implementing homomorphic encryption. *IACR Commun. Cryptol.*, 1(4):26, 2024.

[4] J. H. Cheon, A. Kim, M. Kim, and Y. S. Song. Homomorphic encryption for arithmetic of approximate numbers. In T. Takagi and T. Peyrin, editors, *Advances in Cryptology - ASIACRYPT 2017, Part I*, volume 10624 of *Lecture Notes in Computer Science*, pages 409–437. Springer, 2017.

[5] S. Halevi and V. Shoup. Algorithms in HElib. In *Advances in Cryptology - CRYPTO 2014, Part I*, volume 8616 of *Lecture Notes in Computer Science*, pages 554–571. Springer, 2014. `https://eprint.iacr.org/2014/106`.

[6] S. Halevi and V. Shoup. Faster homomorphic linear transformations in HElib. In *Advances in Cryptology - CRYPTO 2018, Part I*, volume 10991 of *Lecture Notes in Computer Science*, pages 93–120. Springer, 2018. `https://eprint.iacr.org/2018/244`.

[7] C. Leuridan. Concentration of measure on sphere: Bounding the probability of a large angle. `https://math.stackexchange.com/questions/4560617/` (version 2022-10-27). Based on an exercise in chapter 3 of High-Dimensional Statistics: A Non-Asymptotic Viewpoint by Martin J. Wainwright, Cambridge University Press, 2019.

[8] OpenFHE lattice cryptography library - arbitrary smooth function evaluation. `https://github.com/openfheorg/openfhe-development/blob/main/src/pke/examples/FUNCTION_EVALUATION.md`. Accessed Aug 6, 2025.