SGX-IR: Secure Information Retrieval with Trusted Processors

Fahad Shaon[1] and Murat Kantarcioglu[1]

The University of Texas at Dallas, Richardson TX 75070, USA
{fahad.shaon, muratk}@utdallas.edu

Abstract. To preserve the security and the privacy of the data need for cloud applications, encrypting the data before outsourcing has emerged as an important tool. Furthermore, to enable efficient processing over the encrypted data stored in the cloud, utilizing efficient searchable symmetric encryption (SSE) schemes became popular. Usually, SSE schemes require an encrypted index to be built for efficient query processing. If the data owner has limited power, building this encrypted index before data is outsourced to the cloud could become a computational bottleneck. At the same time, secure outsourcing of encrypted index building using techniques such as homomorphic encryption is too costly for large data. Instead, in this work, we use a trusted processor, e.g, Intel Software Guard eXtension (SGX), to build a secure information retrieval system that provides better security guarantee and performance improvements. Unlike other related works, we focus on securely building the encrypted index in the cloud computing environment using the SGX, and show that the encrypted index could be used for executing keyword queries over text documents and face recognition detection in image documents. Finally, we show the effectiveness of our system via extensive empirical evaluation.

Keywords: Encrypted Index Building, Trusted Processor, Secure Search

1 Introduction

One of the reliable and proven approaches to keep data secure in the cloud environment is to encrypt data before uploading it to the cloud. As a result, searching and selectively retrieving encrypted data efficiently without significant information leakage has attracted a lot of attention recently. The most prominent solution is searchable symmetric encryption (SSE), where users create an encrypted index, outsource the index to cloud service provider and later find a subset of documents using carefully crafted trapdoors for query keywords. However, typical practical SSE systems trade-off some aspects of security for performance. Because providing complete security in the SSE setup has large overhead.

On the other hand, we can build significantly secure encrypted applications for outsourced environments using trusted hardware, which allows a user to execute programs securely. Previously we needed specialized hardware to set up such a system. Now, Intel has included a security module named *Software Guard eXtension (SGX)* in 6^{th} generation and afterward CPU. In short, SGX allows users to create secure isolated compartments (called *enclave*) inside the process. We can perform secure computation in the enclave that can not be altered without detection by the operating systems or hypervisor or other system-level processes. In addition, data stored inside the enclave is not *directly* observable.

Due to omnipresence and practicality, a lot of systems have been built using SGX including accessing the encrypted index. For instance, in Oblix [23], the authors propose few ways to access an inverted index obliviously inside the enclave. In HardIDX [13], authors propose building a secure B+ index, which can later be used to build other applications. In Rearguard [36] authors proposed a system to retrieve the list of documents obliviously from the encrypted index. However, in these works authors assume that the inverted index is already available and only focused on accessing the encrypted index securely. In contrast, we focus on building the index securely using SGX. Because building an inverted index might be trivial but it is very memory consuming computation. An inverted index is traditionally defined as a function that returns the list of documents associated with an input token. So, to build such an index by reading a set of input documents, we need to maintain a hash map (or equivalent data structure) of the token to document lists. For a memory constraint system, such as a smartphone, this computation might be infeasible. So in our design, we push as much computation to the SGX enclave as possible.

Furthermore, the existing works mainly focus on building systems targeting the text index. In our work, we also build a secure index for text and image search. Once we built the inverted index we can utilize existing index accessing systems to efficiently retrieve data.

Securely building an index entirely inside the SGX enclave has its challenges. First, SGX memory access can be observed by host operating system and memory access reviles a lot of information as shown in [17,24]. To protect against such attacks, we need to build data oblivious algorithms, i.e. we do not perform data specific branching so that an attacker that observes memory accesses can not learn any sensitive information. Second, SGX is a very memory-constrained environment one can effectively allocate about $90MB$ of dynamic memory inside the enclave natively. So we need an efficient blocking and caching mechanism to process large datasets. To that end, we adopted SGX-BigMatrix [35] mechanism and we represent our internal data structures in BigMatrix.

We proposed a secure encrypted index building for text and image data. We build text index to support TF-IDF and different variants [9], such as log, augmented, boolean term frequency with cosine normalization. To do that efficiently, we first do document level summarization and create a stream of tuples of token id, document id, and count. Next, we encrypt and send it to the server. In the server, we compute different TF-IDF values. For face recognition, we encrypt the face images and send them to the server. In the server, we scale all the face images to the same size and calculate eigenfaces [37] of the input images. We adopt Jacobi's eigenvector calculation algorithm to compute the eigenfaces.

Our contributions in this work can be summarized as follows:

– We propose data oblivious algorithms for building an encrypted search index for text data that supports TF-IDF based ranked information retrieval.
– We propose data oblivious algorithm for computing eigenvectors for a given matrix and show how to use it for face recognition on encrypted image data.
– We build a prototype of the system and show its practical effectiveness.

2 Background

In this section, we provide background on Intel SGX, Data Obliviousness, and Secure Searchable Encryption to better explain our system.

2.1 Intel SGX

Intel SGX is a set of CPU instructions for executing secure code in Intel processors [5]. Using these instructions a program can create secure compartments in the CPU called an *enclave*. SGX uses systems main memory to store the code and data of enclave in an encrypted format and only decrypts inside the CPU. So, the operating system, hypervisor, and other privileged processes of the system can not *directly* observe computation inside the enclave. To leverage the security benefit we need partition the code into trusted and untrusted components. The trusted code is encrypted and integrity protected, but the untrusted code is directly observable by the operating system. During the program execution, the untrusted component creates an enclave with the trusted code. We can verify that the intended code is loaded and securely provision the code with secret keys using the *attestation* process. The trusted and untrusted components communicate with each other using programmer-defined entry points. Entry points defined in trusted code is called *ECalls*, which can be called by untrusted part once the enclave is loaded. Similarly, entry points defined in untrusted code is called *OCalls*, which can be called by the trusted part. More details about the SGX execution model are described in [10,26].

2.2 Data Oblivious Execution

A data oblivious program executes the same code path for all data inputs. We build our program to be data oblivious because we assume that, an adversary in an SGX environment can observe memory accesses, time to execute, OCalls, and any resource usages from OCalls [20]. However, an adversary in SGX cannot observe the content of the internal CPU registers. So, we remove data specific branching to reduce access pattern leakage, which has been shown to reveal information about data in [17,24].

Data arithmetic assembly instructions, such as `add`, `mult`, etc., are by definition data oblivious because the instruction performs the same task irrespective of any input data. However, conditional branching instructions are *not* data oblivious. For example, all `jcc` (jump if condition is met) family instructions, are not data oblivious because these instructions execute different parts of the code based on input data. To implement programs that require such conditional operations, we first assign values from all possible code paths, to different registers, then set a flag based on the condition that we want to test, swap according to the flag, and finally, return the contents of a fixed register. Such techniques are used in previous works (e.g., [25,32]).

2.3 Searchable Symmetric Encryption

Searchable Symmetric Encryption (SSE) is one of the prominent mechanisms to search encrypted files in a cloud computing environment. To achieve that we built and upload an encrypted index with the encrypted data. Traditionally SSE consists of five algorithms - `Gen`, `Enc`, `Trpdr`, `Search`, and `Dec`. Given a security parameter `Gen` generates a master symmetric key, `Enc` generates the encrypted inverted index and encrypted data sets from the input dataset, `Trpdr` algorithm takes keywords as input and outputs the trapdoor, which is used by the `Search` algorithm to find a list of documents associated with input keywords. Finally, the `Dec` algorithm decrypts the encrypted document given the id and the proper key. We refer the reader to [11] for further discussion of SSE. Traditionally `Gen`, `Enc`, `Trpdr`, and `Dec` are performed in a client device and the `Search` algorithm is performed in a cloud server. In these work we focus on preforming core part of the index building in the cloud server securely using trusted processors.

3 System

In this section, we outline our system details including setup and core building blocks. Our system has two components: client and server. We briefly discuss these components and the threat model below:

Client To organize and properly utilize our server, we need a client program that runs in users device. It is capable of encrypting user data and send it to server for further computation. We also assume that computational capability of our users' systems are significantly limited. Our primary motivation is to off-load the index creation step for encrypted search to cloud. So that users with smaller capability machines can perform very large privacy preserving computation using secure server.

Server Our server has hardware based trusted execution environment (i.e. Intel SGX) and services that manages and monitors secure enclave life-cycle.

Threat model We follow standard threat model of trusted processor base systems. Specifically, we are considering a scenario, where a user has large number of documents on which she wants to build search index in a secure cloud server. Our user do not trust the server completely. User expects that server will follow the given protocol but server will want to infer information form the data. User only trusts the trusted component of the server, e.g. Intel SGX. Apart for the trusted component, all other components of the server, such as, hyper-visor, operating system, main memory, etc., are not trusted by the user. We are assuming that user can verify that server is executing proper code using proper attestation mechanism. In addition, we assume that communication between client and server is done over secure channel (TLS/SSL).

3.1 Storage

We adopted techniques outlined in SGX-BigMatrix [35] to store large dataset in our system. In short we break a large matrix into smaller blocks and load only blocks that are required to perform the intended operation. Once done we remove the block and encrypt the block again with session key and store in disk. We use least recently used (LRU) technique to manage the block caching. In addition, we also keep the IV and MAC of all the blocks into a header file, which is integrity protected.

3.2 Notation

We represent all of our data in two dimensional matrices. In some cases, we also define column names of matrices. We use $A[i]$ to denote i^{th} row of a 2D matrix or table, $A[i].col\_name$ to denote value of the column col_name on i^{th} row.

3.3 Oblivious text indexing in Server

To define the oblivious algorithm we need to avoid conditional branching, instead we perform flag based conditional move. We first define a secure building block *obliviousSelect*, which obliviously selects between two integer values based on comparison variables.

obliviousSelect(a, b, x, y): Let, a and b are two integers to select from, x and y be two comparison variables. We return a if $x == y$, otherwise return b. We show the most important lines of the implementation in the following code listing. We start by copying value of x and y to `eax` and `ebx` registers, then perform `xor`, in line 5. As a result, if x and y are equal then the zero flag is set. Next, we copy value of a and b into `ecx` and `edx`. Now we conditionally move values (based on zero flag) between `ecx` and `edx` registers, in line 9. So, if zero flag is set then `edx` will have the value of `eax` otherwise the value will remain unchanged. Finally, we return the value of `edx` register. In our setup, the adversary can only observe the sequence of operations but can not know exactly which value was selected.

```
oblivousSelect(a, b, x, y):
...
mov %[x],%%eax
mov %[y],%%ebx
xor %%eax, %%ebx
...
mov %[a],%%ecx
mov %[b],%%edx
cmovz %%ecx,%%edx
...
mov %%edx, %[out]
```

Now, we define the oblivious text index building schema. We start by creating token and document pair and encrypting them in the client side. We perform initial tokenization in the client to achieve privacy (i.e., cloud only sees the encrypted data). In addition, we can perform tokenization using traditional algorithms, such as, Porter stemming [29], in one pass over the data. So we tokenize before data encryption. In addition, our client has limited memory so we use hash function to generate token id from lexical token. Let, $\mathcal{D} = \{d_1, d_2, ..., d_n\}$ be a set of input documents, $id(d_i)$ be the document id, Θ_d be set of tokens in document d, $\mathcal{H}$ be a collision resistant deterministic hash function that generates token-id from lexical, $tf_{t,d}$ be the number of times token t occurred in document d. We start by extracting tokens from all the documents. We build matrix I with three columns - *token_id*, *doc_id*, and *frequency*. For all t in Θ_d we add $\langle \mathcal{H}(t), id(d), tf_{t,d} \rangle$ to I. We can perform this step easily on client, because in most of cases, text datasets consists of a lot of small files. Furthermore, if we need to process a large document we can split it into smaller files then process as usual. Next, we encrypt I and send it to server.

In the server, we start by decrypting I inside the enclave. Next, we obliviously sort I in ascending order of token id and assign the result to I', as listed in Algorithm 1 in line 3. We utilize bitonic sorting algorithm for oblivious sorting, more on our sorting implementation in Section 3.4. Next, We define two matrices $\mathcal{U}(token\_id, count, sum)$ to store the number of documents that a token occurred and summation of total occurrences of all tokens. We iterate sequentially over I, calculate token boundary condition $c \leftarrow I'[i].token\_id \neq I'[i-1].token\_id$, in line 8. If c is true, this implies that we are now reading a new token's information otherwise we are reading current token's information. Based on c we obliviously fill *count* and *sum* column of $\mathcal{U}$ with the count and summation of the token or a dummy value, in lines 9 to 13. We sort $\mathcal{U}$ based on *token_id*, to move the dummy values to end, in line 15. At this stage, in $\mathcal{U}$, i^{th} row has the count and sum of $(i-1)^{th}$ token. So, we shift the count and sum one row in lines 15 and 16. To access a token's information obliviously, we need to create an equal length block for all the tokens. One approach can be, find the maximum count of any token and allocate the maximum number of elements per token. However, based on the experimental results, we observe that token frequencies

Algorithm 1 Text index building

```
1: Require: $I = n \times 3$ input matrix
2: Output: $\mathcal{T}$ = Index $p \times 3$ and $DF = p \times 2$ matrices
3: $I' \leftarrow obliviousSort(I, token\_id)$
4: $\# \leftarrow -1$                                   ▷ dummy value
5: $sum \leftarrow 0, count \leftarrow 0$
6: $\mathcal{U}[0] \leftarrow \langle I'[0].token\_id, 0, 0\rangle$
7: for i = 1 to $I'.length$ do
8:     $c \leftarrow I'[i].token\_id \neq I'[i-1].tok\_id$
9:     $\mathcal{U}[i].token\_id \leftarrow obliviousSelect(I[i].tok\_id, \#, 1, c)$
10:    $\mathcal{U}[i].count \leftarrow obliviousSelect(count, \#, 1, c)$
11:    $\mathcal{U}[i].sum \leftarrow obliviousSelect(sum, \#, 1, c)$
12:    $count \leftarrow obliviousSelect(count, 0, 1, c) + 1$
13:    $sum \leftarrow obliviousSelect(sum, 0, 1, c) + I[i].frequency$
14: end for
15: $\mathcal{U}' \leftarrow obliviousSort(\mathcal{U}, token\_id)$
16: $\mathcal{U}'[i].count = \mathcal{U}'[i+1].count$
17: $\mathcal{U}'[i].sum = \mathcal{U}'[i+1].sum$
18: Remove rows with #
19: Generate inverse document frequency from $\mathcal{U}'$
20: $b \leftarrow optimizeBlockSize(\mathcal{U}.count)$
21: $count \leftarrow 0$
22: for i = 0 to $I'.length$ do
23:    $J[i].token\_id \leftarrow \sigma(I'[i].token\_id, \lfloor \frac{count}{b} \rfloor)$
24:    $count \leftarrow obliviousSelect(count + 1, 0, 1, count < b)$
25: end for
26: for i = 0 to $numToken$ do
27:    for j = $b - 1$ to 0 do
28:        $c \leftarrow \mathcal{U}'[i].count\%b < j$
29:        $t \leftarrow \sigma(\mathcal{U}'[i].token\_id, \lfloor \frac{\mathcal{U}'[i].count}{b} \rfloor)$
30:        $X[i * b + j].token\_id \leftarrow obliviousSelect(t, \#, 1, c)$
31:    end for
32: end for
33: $\mathcal{T} \leftarrow mergeAndSort(J, X, doc\_id)$
```

follow Pareto distribution or power law, i.e., a large number of tokens will appear in a relatively small number of documents. So if we block based on the maximum token count, we will have a lot of dummy entries. To reduce the storage overhead, we split large token into smaller blocks. We use optimization strategies outlined in [34] to find the optimal size of b. We adopt this specific technique because it assumes a distribution of frequencies rather than relying on real data. In our scenario, the block size will be revealed to the adversary, so such an approach will help us reduce the information leakage. Next, we define a deterministic collision-resistant hash function σ that returns a new $token\_id$ given $token\_id$ and relative block number. For all the entries in I', we apply σ on token id and generate J matrix with $token\_id'$, doc_id, and $frequency$, in lines 21 to 25. Next, for all the tokens in $\mathcal{U}'$ we add $\mathcal{U}[i]'.count\%b$ dummy entries into X obliviously,

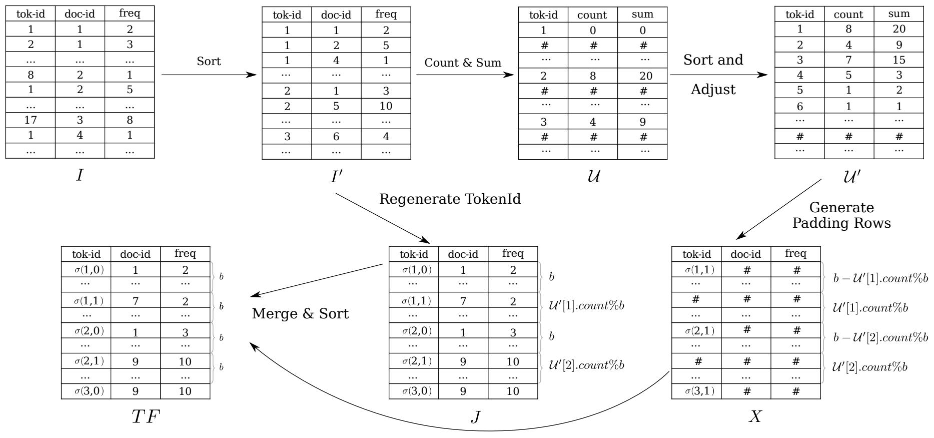


Fig. 1. Text indexing example

in lines 26 to 32. Also, we merge X with J and sort the resulting matrix. Finally, we remove the rows with only dummy entries. So that $\mathcal{T}$ contain m rows per token. Now we push m rows into *any* standard ORAM if we want to access specific token information obliviously in sub-linear time. Otherwise, we can read a token's information obliviously by reading the entire $\mathcal{T}$ matrix once. Figure 1 illustrates an example of secure text indexing.

3.4 Oblivious arbitrary length bitonic sort

We use bitonic sort [6] for sorting in the server, because in bitonic sort we perform exactly the same comparisons irrespective of input data value. Specifically we use the arbitrary length version to save time and space. In our empirical analysis, we have observe that total number to rows in matrices I, J, etc. can go up to 2^{26} or more. So to utilize traditional bitonic sort, we need to pad the matrices with dummy entries to 2^{27} number of rows. In practice, we observe that this adds upto 40% overhead. In addition, we implement an iterative variant of the bitonic sort of arbitrary length. We avoid recursion to save stack space, since SGX is a memory constraint environment. In Algorithm 2 we define our iterative implementation. We utilize existing definition of iterative bitonic sort defined in [8,39] for length 2^k. The core idea behind this implementation is that any number can be expressed as summation of one or more 2^k format numbers, such as, $N = 2^{x_1} + ... + 2^{x_m}$. For given N element array/matrix, we consider it as block of 2^{x_1}, ..., $2^{x_{m-1}}$ rows. We sort first $(m-1)$ blocks in descending order and sort the final block m ascending order using existing definition of non-recursive bitonic sort. Next we merge last two blocks m and $(m-1)$ in ascending order. We iteratively continue to merge the result with previous block. So, finally we get a sorted array/matrix in ascending order. In addition, we also make the exchange step oblivious. So the adversary will not get any additional information from sequence of comparisons and successful swaps.

Algorithm 2 Non-recursive non-trivial bitonic sort for arbitrary length

```
 1: for d = 0 to $\lceil log_2(N) \rceil$ do
 2:     if $((N >> d)\&1) \neq 0$ then
 3:         $start \leftarrow (-1 << (d+1))\&N$
 4:         $size \leftarrow 1 << d$
 5:         $dir \leftarrow (size\&N\&-N) == (N\&-N)$
 6:         $bitonicSort2K(start, size, dir)$
 7:         if !dir then
 8:             $bitonicMerge(start, N - start, 1)$
 9:         end if
10:     end if
11: end for
```

3.5 Image indexing for face recognition

For face recognition we adopt *Eigenface* [37]. It is a very well studied, effective yet simple technique for face recognition using static 2D face image. We first discuss Eigenface technique then outline the oblivious version of it. This face recognition technique consists of three major operations - finding eigenvectors of faces, finding weights of each faces, and finally recognition.

Finding eigenvectors. We start with M face centered upright frontal images that are represented as $N \times N$ square matrices. Let, $\{\Gamma_1, \dots, \Gamma_M\}$ are $N^2 \times 1$ vector representation of these square matrices, $\Psi = \frac{1}{M}\sum_{i=1}^{M}\Gamma_i$ is the average of these vectors, and $\Phi_i = \Gamma_i - \Psi$ is computed by subtracting average Ψ from ith image vector.

Now eigenvectors u_i of co-variance matrix $C = AA^T$, where $A = [\Phi_1\, \Phi_2 \dots \Phi_M]$, can be used to approximate the faces. However, there are N^2 eigenvectors for C. In practice N^2 can be a very large number, thus computing eigenvectors of C can be very difficult. So instead of AA^T matrix we compute eigenvectors of A^TA and take top K vectors for approximating eigenvectors u_i, where $\|u_i\| = 1$.

Finding weights. We can represent Φ_i as a linear combination of these eigenvectors, $\Phi_i = \sum_{j=1}^{K} w_j u_j$ and weights are $w_j = u_j^T \Phi_i$. Each normalized image is represented in this basis as a vector $\Omega_i = \left[w_1\ w_2\ \dots\ w_k\right]^T$ for $i = 1, 2, \dots M$. This is essentially projecting face images into new eigenspace (the collection of eigenvectors).

Recognition. Given a probe image Γ, we first normalize $\Phi = \Gamma - \Psi$ then project into eigenspace such that $\Omega = \left[w_1\ w_2\ \dots\ w_K\right]^T$, where $w_i = u_i^T \Phi$. Now we need to find out nearest faces in this eigenspace by $e_r = min \left\| \Omega - \Omega_i \right\|$. If $e_r <$ a threshold chosen heuristically, then we can say that the probe image is recognized as the image with which it gives the lowest score.

In summary, we consider face images as a point in a high dimensional space (the eigenspace). We compute the eigenspace by finding few significant eigen vectors of the co-variance matrix. Now during recognition phase we project the face into this eigenspace and compute distance between all the training faces. If any training image is bellow the predetermined threshold the we report it as a match.

Oblivious Eigenface To make the entire process data oblivious, we need oblivious eigen vector calculation and oblivious comparison of projected test image. We discuss oblivious eigen vector calculation in a separate subsection. For oblivious distance calculation, we compute the distance function for all input training faces. We create $M \times 2$ matrix $\mathcal{F}$, where first column is face id and second column contains 1 if that face's distance is bellow the threshold and 0 otherwise. Finally, we sort $\mathcal{F}$ based on second column in descending order to get the matching face id.

3.6 Oblivious eigen vector calculation

We adopted Jacobi method [14] of eigen vector computation for oblivious calculation. In Jacobi eigenvalue method, we start with finding maximum value and index of maximum value (k, l) in input symmetric matrix. Next we compute few values based on max, $A_{k,l}$, $A_{l,k}$ Next, we assign zero to $A_{k,l}$ and $A_{l,k}$, and compute $A_{k,k}$ and $A_{l,l}$. Then, we perform rotations on k^{th} column and l^{th} column. However, since this is a symmetric matrix we can perform same computation only on upper triangular matrix. We also perform the same rotation on eigen matrix E, which is initialized with identity matrix We repeat the process until the input matrix becomes diagonal. The values in the main diagonal approximates the eigen values and normalized version of the the eigen matrix E consists of all the eigen vectors of matrix A. In practice, for eigenface, we need top n largest eigen vectors. To extract top n eigen vectors we sort the eigen vectors based on eigen values. In Appendix A we outline an implementation of Jacobi method.

Now, to build an oblivious version, we need to read and write the matrix obliviously. So, we define a few additional oblivious functions, which we use later in the oblivious eigenvector calculation algorithm.

obliviousValueExtract(U, k): Given an array U and an index k, we extract the value of U_x obliviously. We initialize $v \leftarrow U_0$ then we iterate over all the elements in the array and run *obliviousSelect(v,* U_i*, i, k)* and assign the return value to v. As a result, when i is equal to k the return value will be U_k, otherwise it will always return existing value of v. Similarly we define *obliviousValueAssign(U, k, a)*, where we assign value a to k^{th} location of input array U.

obliviousColumnExtract(A, k): Given a 2D matrix A and a column index k, we extract k^{th} column of matrix A. We utilized previously defined oblivious select. For a given row, r, we iterate over all the columns c and assign U_r to output of $obliviousSelect(A_{r,c}, U_r, r, k)$. As a result, when r is equal to k then we get the value of $A_{r,c}$ in U_r, otherwise value of U_r remains the same. Similarly, we define *obliviousColumnAssign()* and *obliviousRowAssign()*, where we assign the input column in a specific column or row of input matrix. In addition, we also define *obliviousConditionalColumnAssign()* and *obliviousConditionalRowAssign()*, with one more boolean parameter, where we perform the assignment if and only if the boolean parameter is true.

obliviousMaxIndex(&m, e, &mR, &mC, eR, eC): Let, $\&m$ be the reference of current maximum value, e be current element value, $\&mR$ and $\&mC$ be the reference of row and column of current maximum element, eR and eC be the row and column of element. We update the value of m with e, mR with eR, and mC with eC, if $m < e$, otherwise we keep the values of $\&m$, $\&mR$, and $\&mC$.

We start by copying the current element and current maximum into floating-point stacks in `st(1)` and `st(0)`. Next, we perform floating-point comparison, as shown in line 3 in the following code listing, which sets zero flag, carry flag, and parity flag accordingly. Next, we perform `fcmovb` conditional move operation that swaps values in the floating-point stack if the carry flag is set, in line 4. As a result, the maximum value will be at the top of the floating-point stack, which we assign back to the maximum variable.

Next, we move row of maximum and current element into two registers, `eax` and `ebx` respectively, in lines 6 and 7. Then we again conditionally swap these two registers and the flag was set during the initial float comparison, in line 8. So, the index of the largest value will be in `eax`, which read back to mR variable. Similarly, we also perform a conditional move for maximum column index in lines 10, 11, and 12.

```
1  obliviousMaxIndex(&m, e, &mR, &mC, eR, eC):
2  ...
3  fucomi %%st(1), %%st
4  fcmovb %%st(1), %%st
5  ...
6  mov %[mR],%%eax
7  mov %[eR],%%ebx
8  cmovb %%ebx, %%eax
9  ...
10 mov %[mC],%%eax
11 mov %[eC],%%ebx
12 cmovb %%ebx, %%eax
13 ...
```

Algorithm 3 Oblivious Eigen vector with Jacobi method

```
1: Require: $A = n \times n$ diagonal matrix
2: Output: $E$ = eigen vectors, $V$ = eigen values
3: $E \leftarrow identity(n)$
4: $\epsilon_1 \leftarrow 10^{-12}$, $\epsilon_2 \leftarrow 10^{-36}$
5: for it $= 0$ to $n^2$ do
6:     $max, k, l \leftarrow obliviousMaxIndex(A)$
7:     $\mathcal{C} \leftarrow max < \epsilon_1$
8:     $U \leftarrow obliviousColumnExtract(A, k)$
9:     $V \leftarrow obliviousColumnExtract(A, l)$
10:    $kk \leftarrow obliviousValueExtract(U, k)$
11:    $ll \leftarrow obliviousValueExtract(V, l)$
12:    $d \leftarrow ll - kk$
13:    $m \leftarrow |max| < \epsilon_2 |d|$
14:    $p \leftarrow \frac{d}{2 \times max}$
15:    $t_1 \leftarrow \frac{max}{d}$, $t_2 \leftarrow |\frac{1}{|p| + \sqrt{p^2+1}}|$
16:    $t \leftarrow obliviousSelect(t_1, t_2, m, 1)$
17:    $c \leftarrow \frac{1}{\sqrt{t^2+1}}$, $s \leftarrow t \times c$, $\tau \leftarrow \frac{s}{1+c}$
18:    $\mathcal{R} \leftarrow s. \begin{bmatrix} -\tau & -1 \\ 1 & -\tau \end{bmatrix}$
19:    $\begin{bmatrix} U \\ V \end{bmatrix} += \mathcal{R} \times \begin{bmatrix} U \\ V \end{bmatrix}$
20:    $kk \leftarrow kk - t \times max$, $ll \leftarrow ll + t \times max$
21:    $obliviousValueAssign(U, k, kk)$
22:    $obliviousValueAssign(V, l, ll)$
23:    $obliviousValueAssign(U, l, 0)$
24:    $obliviousValueAssign(V, k, 0)$
25:    $obliviousConditionalColumnAssign(A, U, k, !\mathcal{C})$
26:    $obliviousConditionalColumnAssign(A, V, l, !\mathcal{C})$
27:    $obliviousConditionalRowAssign(A, U, k, !\mathcal{C})$
28:    $obliviousConditionalRowAssign(A, V, l, !\mathcal{C})$
29:    $U \leftarrow obliviousColumnExtract(E, k)$
30:    $V \leftarrow obliviousColumnExtract(E, l)$
31:    $\begin{bmatrix} U \\ V \end{bmatrix} += \mathcal{R} \times \begin{bmatrix} U \\ V \end{bmatrix}$
32:    $obliviousConditionalColumnAssign(E, U, k, !\mathcal{C})$
33:    $obliviousConditionalColumnAssign(E, V, l, !\mathcal{C})$
34: end for
35: $V_i \leftarrow A_{i,i}, \forall i \in 0$ to $n$
36: $normalize(E)$
37: $sort(E)$ based on $V$
```

Now, we create the data oblivious version of Jacobi's eigenvalue calculation, as listed in Algorithm 3. First, we fix the number of iterations and do not return early based on convergence. As a result, the adversary can not learn information about data based on the iteration count. Next for finding the maximum, *max* and

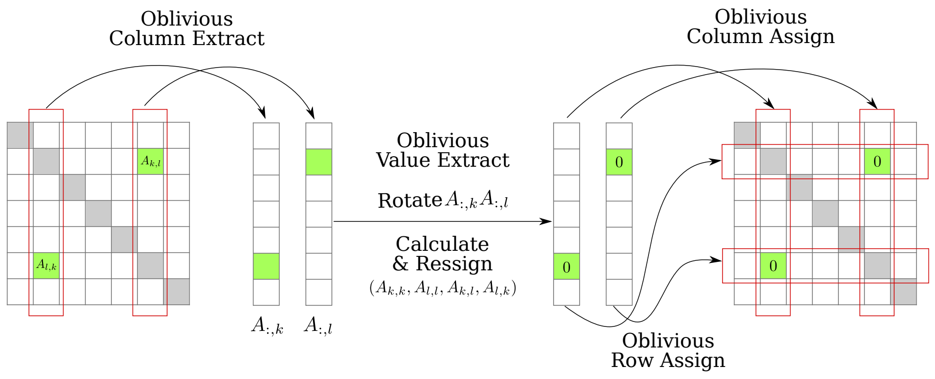


Fig. 2. Oblivious eigen matrix calculation

index of maximum elements (k, l) using $obliviousMaxIndex$ operation in line 6. We extract k^{th} and l^{th} columns, U and V using $obliviousColumnExtract$, in lines 8 and 9. Next, we extract values of kk and ll using $obliviousValueExtract$ operation in lines 10 and 11. Next, we calculate value τ, t, s in lines from 12 to17. For t we first calculate both $\frac{max}{d}$ and $|\frac{1}{|p|+\sqrt{p^2+1}}|$ then we obliviously choose correct version using $obliviousSelect$. Next, we perform the rotation on extracted column and assign k^{th} and l^{th} value of U and V appropriately in lines 18 to 24. Then, we assign the column back to A if the algorithm is not converged, in lines 25 to 28. We also illustrate these operations in Figure 2. We determine convergence condition $\mathcal{C}$ in line 7 by check whether the current maximum is smaller than a predefined small value. We perform similar rotation on matrix E, in lines 29 to 33. We iterate the entire process fix number of times. Finally, we normalize and sort the eigen matrix based on eigenvalues in lines 35 to 37. The normalization process is naturally data oblivious, i.e. we execute same instructions irrespective of input data. For sorting we use our previously defined bitonic sort.

4 Experimental Evaluations

In this section, we discuss the performance of our proposed system. We develop a prototype of the proposed system SGX-IR, using *Intel Software Extensions SDK 2.6* for Linux. We write most of the code in *C++* with the exception of the *oblivious* blocks, which we write in assembly. We also have implemented numerous unit tests to ensure the correctness of the oblivious blocks. Since our building blocks are data oblivious, the final prototype is data oblivious as well. We perform the experiments on a *Intel LR1304SPCFSGX1* server with *Intel® Xeon® CPU E3-1270 @ 3.80GHz* CPU, *64GB* main memory, *128MB* enclave memory, and running *Ubuntu Server 18.04*.

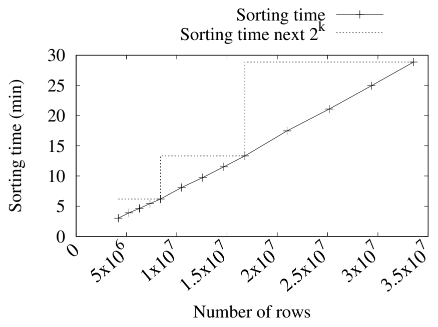


Fig. 3. Bitonic sort time

4.1 Bitonic sort

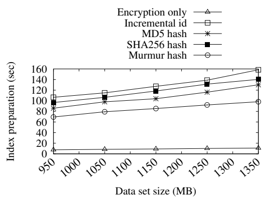
We first briefly discuss the impact of arbitrary length bitonic sort. In Figure 3 we show that sort time of the different size matrices with 3 columns. The solid line represents sorting duration and the dotted line represents sorting duration for the next 2^k element matrix. With the arbitrary length algorithm, the growth of the required time is linear. On the other hand, the required time of the 2^k version is a step function. In extreme cases, it has close to 50% overhead.

4.2 Text Indexing

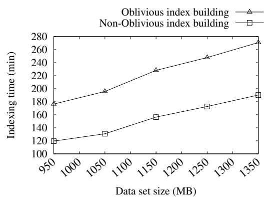
Dataset We use Enron dataset [18] for text indexing experiments. We randomly select sub-set of files from the Enron dataset. Then we parse the data in the client end. We tokenize, stem [29], and build document token pairs. Next, we encrypt and send the data to server. In server we follow the algorithm outlined in Sec 3.3 to sort and generate index.

Performance In Figure 4(a) we show the performance of client-side index preprocessing. We show time to build the input matrix using different types of cryptographic and non-cryptographic hashing functions and keeping an in-memory map for token id generation. We observe that incrementation token id generation is the most expensive and non-cryptographic hash, i.e., MurMur Hash, is the least expensive. In addition, we show the time required for only encrypting the data without performing any tokenization and token id generation, which shows the overhead of read-write and encryption. The gap between encryption only and a hashing token id generation signifies the overhead of our tokenization and matrix generation. Finally, in all theses cases we observe the growth is linear.

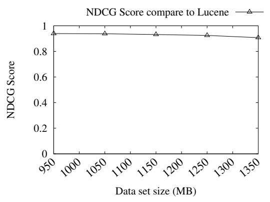
In Figure 4(b) we show server-side index processing cost. We compare our results with a non-oblivious version of a similar index building. For non-oblivious implementation, we sort the input matrix based on token id then build a separate matrix that is equivalent to $\mathcal{T}$ by iterating the sorted matrix. We observe that



(a) Client side processing



(b) Index building time



(c) NDCG Scores compared to Apache Lucene

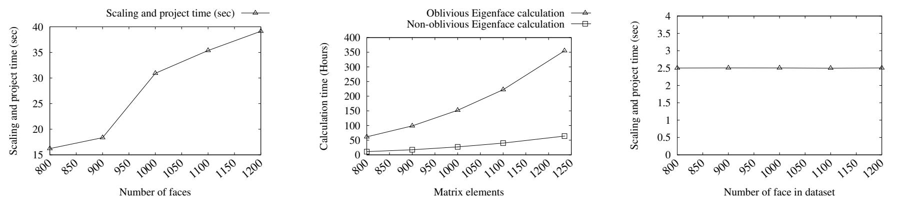
Fig. 4. Enron experiment

the obliviousness implementation is about 1.49 times more expensive. Finally, to show the effectiveness, of our information retrieval system we compare ranked results with Apache Lucene [1] library result. Apache Lucene library is the de-facto standard of information retrieval library and is used in numerous commercial and open-source search engine software, such as Apache Solr [2], Elasticsearch [3]. We adopt normalized discounted cumulative gain (NDCG) score [9] to compare the ranked results of the information retrieval systems. In Figure 4(c) we report the NDCG score of our system compared to Apache Lucene for randomly selected 1,000 tokens. We observe that our scores are about 0.92. In other words, our model works relatively well compared to the industry-standard information retrieval system. In addition, we allow different types of frequency normalizations. So users of the system can tune the normalization functions to improve the results as needed.

4.3 Face Recognition

Dataset We use *Color FERET* [28,27] dataset for testing face recognition. *Color FERET* dataset contains a total of 11338 facial images, which were collected by photographing 994 subjects at various angles. The dataset images contain face images in front of different background often containing other objects. So, wrote a face detection program using OpenCV implementation of haar cascade classifier [38,21] for frontal face. We extract frontal face images with `fa` suffix from the dataset. We found that there are total 1364 such images. Our face detection system successfully detected 1235 images, yielding 90.33% accuracy. Here, most of the failed cases has glass or similar face obstructing additions. We extract the frontal faces and scale to 100×100 faces. Then we randomly selected sub-set of images and build our face recognition dataset.

Performance In Figure 5(a), we show the performance of face image preparation and in Figure 5(c) face finding overhead. Both of these operations are standard matrix operations. So overheads are very minimal under a minute. In Figure 5(b) we show the required time for building the eigenface index, which is dominated



(a) Eigen-face preparation (b) Index building time (c) Eigen-face recognition

Fig. 5. Oblivious eigen-face experiment

by the overhead of in eigenvector calculation. We compare our results with a non-oblivious version of the algorithm. For the non-oblivious version, we implement the Jacobi algorithm without accessing the matrix values obliviously. We observe that both the cases the required times are quite large because of the large number required iteration for the Jacobi algorithm to converge. We observe that, the obliviousness adds around 5 times more overhead. We incur such large overheads because we read the entire matrix to extract and assign the required rows and columns.

5 Related Work

System using Intel SGX. Because of the availability and sound security guarantees, Intel SGX is already used in many studies to build secure systems. For instance, Ohrimenko at al. [25] proposed algorithms to perform machine learning in multi-party settings obliviously using SGX. Here authors also has shown the effectiveness of oblivious primitives to achieve data obliviousness. In [35] authors built oblivious system to analyze large datasets using SGX, which we adopted for our data storage. In [42], the authors proposed a package for Apache Spark SQL named Opaque, which enables SQL query processing in a distributed manner. Opaque offers data encryption and access pattern hiding using Intel SGX. EnclaveDB [30], ObliDB [12], and StealthDB [15] also proposed secure database functionality with Intel SGX. Part of the text indexing processing in Section 3.3 can be expressed with some non-traditional relational algebra, which can be expressed with techniques described in this works. However, for completeness and efficiency reason we describe our own version.

Intel SGX based search index. In Rearguard [36] authors build search indexes for different types of keyword searches. However, authors assumed to have an initial inverted index built in client end. In HardIDX [13] authors proposed building secure B+ index and used it to build encrypted databases and searchable encryption schema. However, proposed algorithms are not made oblivious as a result leaks a lot of information via a side channel. In Oblix [23] authors proposed building different type of oblivious data structures using oblivious ram (ORAM) techniques. In [16], authors used Oblix data structures to build large scale search systems. However, authors build the index in client side. We consider our work as a complement to these works one can build index using our techniques and use these systems later to access the inverted index. Finally, in [4] Intel maintains a somewhat extensive categorized list of academic research publications related to SGX.

Searchable encryption systems. SSE has been studied extensively as shown in an extensive survey of provably secure searchable encryption by Bösch at el. in [7]. One of the relevant SSE schemas is defined by Kuzu at el. in [19], where authors used locality sensitive hashing to convert similarity search into equality search. In [34] authors proposed SSE schema for image data and complex queries, such as, face recognition, image similarity, using simple server storage. In addition, there has been several works to build content based secure image retrieval in [40,22,33,31]. In [41] authors proposed secure image retrieval using SGX as well. However, in their construction authors are building the index in client side and the system is not oblivious.

6 Conclusion

In this work, we propose a secure information retrieval system for text and image data. Unlike other existing works, we focus on building the encrypted index in the cloud securely using trusted processors, such as Intel SGX. We address the information leakage due to memory access pattern issue by proposing data oblivious indexing algorithms. We build a text index to support ranked document retrieval using TF-IDF scoring mechanisms. Also, we build an image index to support the face recognition query. In addition, we also propose a non-recursive version of the bitonic sort algorithm for arbitrary input length.

Acknowledgement

We thank the anonymous reviewers for their insightful comments. The research reported herein was supported in part by NIH award 1R01HG006844, NSF awards CICI-1547324, IIS-1633331, CNS-1837627, OAC-1828467 and ARO award W911NF-17-1-0356.

References

1. Apache lucene. `https://lucene.apache.org/`, accessed 04/20/2020
2. Apache solor. `https://lucene.apache.org/solr/`, accessed 04/20/2020
3. Elasticsearch. `https://www.elastic.co/`, accessed 04/20/2020
4. Intel software guard extension - academic research. `https://software.intel.com/en-us/sgx/documentation/academic-research`, accessed 04/20/2020
5. Anati, I., Gueron, S., Johnson, S., Scarlata, V.: Innovative technology for cpu based attestation and sealing. In: Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy. vol. 13 (2013)

6. Batcher, K.E.: Sorting networks and their applications. In: Proceedings of the April 30–May 2, 1968, spring joint computer conference. pp. 307–314. ACM (1968)
7. Bösch, C., Hartel, P., Jonker, W., Peter, A.: A survey of provably secure searchable encryption. ACM Computing Surveys (CSUR) **47**(2), 18 (2015)
8. Christopher, T.W.: Bitonic sort. `https://www.tools-of-computing.com/tc/CS/Sorts/bitonic_sort.htm`, accessed 04/20/2020
9. Christopher D. Manning, P.R., Schtze, H.: Introduction to Information Retrieval. Cambridge University Press (2008)
10. Costan, V., Devadas, S.: Intel sgx explained. IACR Cryptology ePrint Archive **2016**(086), 1–118 (2016)
11. Curtmola, R., Garay, J., Kamara, S., Ostrovsky, R.: Searchable symmetric encryption: improved definitions and efficient constructions. In: Proceedings of the 13th ACM conference on Computer and communications security. pp. 79–88. ACM (2006)
12. Eskandarian, S., Zaharia, M.: Oblidb: Oblivious query processing using hardware enclaves. arXiv preprint arXiv:1710.00458 (2017)
13. Fuhry, B., Bahmani, R., Brasser, F., Hahn, F., Kerschbaum, F., Sadeghi, A.R.: Hardidx: Practical and secure index with sgx. In: IFIP Annual Conference on Data and Applications Security and Privacy. pp. 386–408. Springer (2017)
14. Golub, G.H., Van der Vorst, H.A.: Eigenvalue computation in the 20th century. In: Numerical analysis: historical developments in the 20th century, pp. 209–239. Elsevier (2001)
15. Gribov, A., Vinayagamurthy, D., Gorbunov, S.: Stealthdb: a scalable encrypted database with full sql query support. arXiv preprint arXiv:1711.02279 (2017)
16. Hoang, T., Ozmen, M.O., Jang, Y., Yavuz, A.A.: Hardware-supported oram in effect: Practical oblivious search and update on very large dataset. Proceedings on Privacy Enhancing Technologies **2019**(1), 172–191 (2019)
17. Islam, M.S., Kuzu, M., Kantarcioglu, M.: Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In: NDSS. vol. 20, p. 12 (2012)
18. Klimt, B., Yang, Y.: The enron corpus: A new dataset for email classification research. In: European Conference on Machine Learning. pp. 217–226. Springer (2004)
19. Kuzu, M., Islam, M.S., Kantarcioglu, M.: Efficient similarity search over encrypted data. In: Data Engineering (ICDE), 2012 IEEE 28th International Conference on. pp. 1156–1167. IEEE (2012)
20. Lee, S., Shih, M.W., Gera, P., Kim, T., Kim, H., Peinado, M.: Inferring fine-grained control flow inside {SGX} enclaves with branch shadowing. In: 26th {USENIX} Security Symposium ({USENIX} Security 17). pp. 557–574 (2017)
21. Lienhart, R., Maydt, J.: An extended set of haar-like features for rapid object detection. In: Image Processing. 2002. Proceedings. 2002 International Conference on. vol. 1, pp. I–900. IEEE (2002)
22. Lu, W., Swaminathan, A., Varna, A.L., Wu, M.: Enabling search over encrypted multimedia databases. In: IS&T/SPIE Electronic Imaging. pp. 725418–725418. International Society for Optics and Photonics (2009)
23. Mishra, P., Poddar, R., Chen, J., Chiesa, A., Popa, R.A.: Oblix: An efficient oblivious search index. In: 2018 IEEE Symposium on Security and Privacy (SP). pp. 279–296. IEEE (2018)
24. Naveed, M., Kamara, S., Wright, C.V.: Inference attacks on property-preserving encrypted databases. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. pp. 644–655. ACM (2015)

25. Ohrimenko, O., Schuster, F., Fournet, C., Mehta, A., Nowozin, S., Vaswani, K., Costa, M.: Oblivious multi-party machine learning on trusted processors. In: 25th USENIX Security Symposium (USENIX Security 16). pp. 619–636. USENIX Association, Austin, TX (2016)
26. Pass, R., Shi, E., Tramer, F.: Formal abstractions for attested execution secure processors. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 260–289. Springer (2017)
27. Phillips, P.J., Moon, H., Rizvi, S., Rauss, P.J., et al.: The feret evaluation methodology for face-recognition algorithms. Pattern Analysis and Machine Intelligence, IEEE Transactions on **22**(10), 1090–1104 (2000)
28. Phillips, P.J., Wechsler, H., Huang, J., Rauss, P.J.: The feret database and evaluation procedure for face-recognition algorithms. Image and vision computing **16**(5), 295–306 (1998)
29. Porter, M.F.: An algorithm for suffix stripping. Program (2006)
30. Priebe, C., Vaswani, K., Costa, M.: Enclavedb: A secure database using sgx. In: EnclaveDB: A Secure Database using SGX. p. 0. IEEE (2018)
31. Qin, Z., Yan, J., Ren, K., Chen, C.W., Wang, C.: Towards efficient privacy-preserving image feature extraction in cloud computing. In: Proceedings of the ACM International Conference on Multimedia. pp. 497–506. ACM (2014)
32. Rane, A., Lin, C., Tiwari, M.: Raccoon: closing digital side-channels through obfuscated execution. In: 24th USENIX Security Symposium (USENIX Security 15). pp. 431–446 (2015)
33. Raval, N., Pillutla, M.R., Bansal, P., Srinathan, K., Jawahar, C.: Efficient content similarity search on encrypted data using hierarchical index structures
34. Shaon, F., Kantarcioglu, M.: A practical framework for executing complex queries over encrypted multimedia data. In: IFIP Annual Conference on Data and Applications Security and Privacy. pp. 179–195. Springer (2016)
35. Shaon, F., Kantarcioglu, M., Lin, Z., Khan, L.: Sgx-bigmatrix: A practical encrypted data analytic framework with trusted processors. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 1211–1228. ACM (2017)
36. Sun, W., Zhang, R., Lou, W., Hou, Y.T.: Rearguard: Secure keyword search using trusted hardware. IEEE INFORM (2018)
37. Turk, M., Pentland, A.: Eigenfaces for recognition. Cognitive Neuroscience, Journal of **3**(1), 71–86 (Jan 1991)
38. Viola, P., Jones, M.: Rapid object detection using a boosted cascade of simple features. In: Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on. vol. 1, pp. I–511. IEEE (2001)
39. Wikipedia contributors: Bitonic sorter — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Bitonic_sorter&oldid=908641033`, accessed 04/20/2020
40. Xia, Z., Zhu, Y., Sun, X., Wang, J.: A similarity search scheme over encrypted cloud images based on secure transformation. International Journal of Future Generation Communication and Networking **6**(6), 71–80 (2013)
41. Yan, H., Chen, Z., Jia, C.: Ssir: Secure similarity image retrieval in iot. Information Sciences **479**, 153–163 (2019)
42. Zheng, W., Dave, A., Beekman, J., Popa, R.A., Gonzalez, J., Stoica, I.: Opaque: A data analytics platform with strong security. In: 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17). USENIX Association, Boston, MA (2017)

A Jacobi method implementation

In this section we provide an implementation of Jacobi algorithm. In this particular implementation we change only upper triangular elements.

Algorithm 4 Eigen vector with Jacobi method

```
1: **Require:** $A = n \times n$ diagonal matrix
2: **Output:** $E$ = eigen vectors, $V$ = eigen values
3: $E \leftarrow identity(n)$
4: $\epsilon_1 \leftarrow 10^{-12}$, $\epsilon_2 \leftarrow 10^{-36}$
5: for it = 0 to $n^2$ do
6:     $max \leftarrow max(A)$ in off-diagonal upper triangle
7:     $(k, l) \leftarrow maxIndex(A)$
8:     if $max < \epsilon_1$ then
9:         $V_i \leftarrow A_{i,i}, \forall i \in 0$ to $n$
10:        $normalize(E)$
11:        return
12:    end if
13:    $d \leftarrow A_{l,l} - A_{k,k}$
14:    if $|A_{k,l}| < \epsilon_2 |d|$ then
15:        $t \leftarrow \frac{A_{k,l}}{d}$
16:    else
17:        $p \leftarrow \frac{d}{2A_{k,l}}$
18:        $t \leftarrow |\frac{1}{|p|+\sqrt{p^2+1}}|$
19:    end if
20:    $c \leftarrow \frac{1}{\sqrt{t^2+1}}$, $s \leftarrow t \times c$, $\tau \leftarrow \frac{s}{1+c}$
21:    $A_{k,k} \leftarrow A_{k,k} - t \times A_{k,l}$, $A_{l,l} \leftarrow A_{l,l} + t \times A_{k,l}$, $A_{k,l} \leftarrow 0$
22:    $\mathcal{R} \leftarrow s. \begin{bmatrix} -\tau & -1 \\ 1 & -\tau \end{bmatrix}$
23:    for $i = 0$ to $k - 1$ do
24:        $\begin{bmatrix} A_{i,k} \\ A_{i,l} \end{bmatrix} + = \mathcal{R} \times \begin{bmatrix} A_{i,k} \\ A_{i,l} \end{bmatrix}$
25:    end for
26:    for $i = k + 1$ to $l - 1$ do
27:        $\begin{bmatrix} A_{k,i} \\ A_{i,l} \end{bmatrix} + = \mathcal{R} \times \begin{bmatrix} A_{k,i} \\ A_{i,l} \end{bmatrix}$
28:    end for
29:    for $i = l + 1$ to $n - 1$ do
30:        $\begin{bmatrix} A_{k,i} \\ A_{l,i} \end{bmatrix} + = \mathcal{R} \times \begin{bmatrix} A_{k,i} \\ A_{l,i} \end{bmatrix}$
31:    end for
32:    for $i = 0$ to $n - 1$ do
33:        $\begin{bmatrix} E_{i,k} \\ E_{i,l} \end{bmatrix} + = \mathcal{R} \times \begin{bmatrix} E_{i,k} \\ E_{i,l} \end{bmatrix}$
34:    end for
35: end for
```