Fully Homomorphic Encryption for Data Privacy in Entity Resolution

Presented to the S. Daniel Abraham Honors Program

In Partial Fulfillment of the

Requirements for Completion of the Program

Stern College for Women

Yeshiva University

May 2, 2024

Judith Wechter

Mentor: Professor Alan Broder, Computer Science

Table of Contents

Introduction - Entity Resolution

The world we live in is full of data. Hospital records, financial records, government databases, Facebook users, and Amazon purchases are all examples of existing data collections. If harnessed properly, this data can be used to improve business practices and life in general. To do that, we must be able to properly understand all of this data. First, it is important to realize that each data point in each database represents a real-world entity. This can be a person, a boat, a product, a property, or anything really. For example, a credit card company has a customer named John Smith, who lives at 123 Appledale Rd. with phone number 123-456-7890. In the credit card company's database, every record has three subfields to represent each customer's information: name, address, and phone number. For this particular customer, those fields are populated with "John Smith", "123 Appledale Rd.", and "123-456-7890" respectively. This information represents a real person named John Smith, and this is how the credit card company keeps track of him. If we want to cross-reference entities between different databases, we just compare the records from the different databases, subfield by subfield. Any records that have matching subfields can be assumed to refer to the same real-world entity. Writing a computer program to do this comparison is not difficult. We just have to read all the records from both databases, one at a time, and compare each record from one database to every record from the other, subfield by subfield. This process would be successful in finding all matching records across two distinct databases, but it is exceptionally slow, with time complexity O(N^2). When dealing with databases like Facebook users that have more than three billion records [1], such an algorithm is just not good enough.

A second issue with such a model is that its success assumes that each separate database has the same subfields for each record, which is certainly not the case in the real world. One

database might decide to identify records by name, address, and phone number, while another chooses the subfields name, email, and birthdate. Consider John Smith from Appledale Rd. Credit Card Company A has a record for him in their database with subfield name as "John Smith", address as "123 Appledale Rd.", and phone number as "123-456-7890". Hospital B, on the other hand, has a record in their database with subfields name as "John Smith", email as "john.smith@gmail.com", and birth date as "01/01/2001". When comparing these records from the two databases, there is no way to know if these "John Smiths" refer to the same real-world entity. How can a computer program even attempt to compare records from these two databases? Since each database has different subfields, there is no clear way to compare records to each other, and no way to know if the two records match up.

Even without the above two issues, a straight subfield-by-subfield comparison of records from different databases might still not provide useful results. The format and conventions each database uses to keep track of data of the same subfield can vary between databases. Let us consider John Smith again. Credit Card Company A has a record in their database for "John Smith", "123 Appledale Rd.", and "123-456-7890". Hospital B's database with the same subfields of name, address, and phone number, has a record of "John H. Smith", "12-3 Appledale Road", and "(123) 456 7890". A human can easily see that although these records are slightly different, they almost certainly refer to the same real-world John Smith. A computer program, on the other hand, would be unable to identify that these records refer to the same entity. The only equality a computer knows is when the subfields of two records are exactly the same, character for character. Where people can see just a difference in convention (i.e. the difference between "Rd." and "Road"), computers see only differences.

There can be more than just conventional differences between databases. Sometimes there are slight spelling differences that might also be slight enough for a human to deem insignificant. Most people would determine that “John H. Smith” and “Jon H. Smith” are most likely the same person, especially if the rest of the subfields matched. A computer has no such nuance. Unless the records being compared have exactly the same subfields, populated with text that follows the exact same conventions, and have no spelling errors or other typos, a computer can't determine that two records refer to the same entity.

This is a problem. There are so many data sources that exist and it can be crucial to combine sources and to compare entities that might appear in more than one source. Companies can make sure they have accurate data by comparing what they have to other databases. The process of trying to identify records from disparate sources that refer to the same real-world entity is known as entity resolution. Much research has been done on this topic and many methods have been developed to code programs that can identify with degrees of accuracy records that refer to the same entity. One such method is a two-step process known as “blocking and scoring”. This method uses a technique known as “fuzzy matching”, as opposed to “exact matching”. In the latter, only exact character-by-character string matches are considered the same. With “fuzzy matching”, spelling errors and differences in convention are also taken into account.

The first step, blocking, is a means of separating the records we want to compare into sections or blocks that are likely to refer to the same entity [2]. This can be accomplished by transforming records from lists of subfields into a single field meant to be representative of the record in its entirety, called a blocking key. There are many different methods and algorithms that can be employed to accomplish this. A simple example can be demonstrated by considering

John Smith yet again. The raw record from the database contains two subfields: name, and phone number. We can simplify this into a single field that contains the first name concatenated with the last name and phone number, removing all capitalization and punctuation. This is obviously a trivial example of a blocking algorithm that does not take into account any spelling or conventional discrepancies. But for example's sake, the record containing “John Smith” and “123-456-7890” becomes the blocking key “johnsmith1234567890”. Once every record has been transformed, we block together all records that have the same blocking key. We establish that these records might refer to the same real-world entity. An optimal blocking algorithm blocks enough records together so as not to miss any potential matches, but leaves enough records unblocked to ensure that blocks are not too large and similar to the entire pool of available records [2].

The next step is scoring. Considering again the initial raw data for each record, we score the similarity of all records contained within the same block to determine how likely it is that they do refer to the same real-world entity [2]. Many algorithms have different approaches to determining how similar two strings are to each other. For fuller results, the same data can be blocked multiple times using different algorithms and then scored again each time. If records have a high similarity score for multiple blocking algorithms, there is a high chance that they refer to the same entity.

It can be very slow to score every single record from one database against every single record from another. When dealing with databases that contain millions or even billions of records, this would take much too long. That is the purpose of blocking. By determining which records are most likely to be similar and only scoring those records against each other, we cut down significantly on the amount of similarity scores that need to be calculated [2]. The purpose

of the scoring step is that by scoring records, we allow for different records - as long as they are similar enough - to be considered a match even if they are not exactly the same. In this way, we approximate what a person does when he sees records that are similar but not exactly the same.

Besides comparing and combining records between different databases, there is another application of entity resolution, one which is slightly less obvious but much more useful. Say we have two records, both with the name John Smith. One is John Smith with the phone number 123456789, and the other is John Smith from 123 Appledale Road. These John Smiths may or may not be the same person. Without more information, we can't be certain either way. But if we find a third record of a John Smith who has the phone number 123456789 *and* the address 123 Appledale Road, we now can combine all three of these records into a single entity. A hospital might have the first two records, but not enough information to combine them into a single entity. But if the hospital has access to a government database that contains the third record, it can use that record as the bridge to connect the original two records into a single entity. This can be incredibly useful. A credit card company might not want to do entity resolution with a hospital to combine their respective records into entities, but it might be useful for the credit card company to connect its own records into entities using links from the hospital records.

The Problem

In order to combine records into entities based on data from more than one database, we must have access to the records from all databases that we want to compare. If we can't see the records from any one of the databases, we can't use the blocking and scoring algorithms for entity resolution. This might seem obvious, but it poses a serious issue in real-world databases where data is often private. Health, financial, personal, and government data is almost always confidential. A credit card company can't just utilize sensitive government information to

determine if any of its own records refer to the same real-world entity. To use private data, we must find a way to use the algorithms necessary for entity resolution on data that is encrypted. Encrypted data is data written in mathematical code so that only those with the decryption "key" can read it. If the government data was encrypted when the credit card company used it for entity resolution then it can be used to link credit card company records together with no security breaches. But how can the credit card company use government data for entity resolution while it's encrypted? Encrypted data can't be read. How can it be blocked and then scored? To keep the data safe, it must stay encrypted, but how can it be used for blocking and scoring while it is encrypted?

Fully Homomorphic Encryption

There is an emerging field of study that focuses on running computation on data in such a way that it stays encrypted throughout the process which can potentially solve this problem. This is known as Fully Homomorphic Encryption (FHE). This would allow for the data to stay encrypted throughout, while still allowing entity resolution to take place.

Homomorphic encryption is a form of encryption that allows for computation to be run on data while it remains encrypted. There are different levels of homomorphic encryption. The most basic is known as partial homomorphic encryption. In this type of homomorphic encryption, the data remains encrypted throughout, but there is a limit to the types of operations that can be performed [3]. Usually, these systems are limited to addition and multiplication [3]. FHE, in contrast, is the most robust form of homomorphic encryption, in that it allows for unlimited operations [3]. In this paper, we will focus on this form of homomorphic encryption.

A very common form of data encryption is known as RSA public key cryptography, developed by Ron Rivest, Adi Shamir, and Leonard Adleman in 1977 [4]. The idea is that there

are two keys, one public and one private [4]. The public key is used for encrypting data and is known by everyone. Anyone can have access to it and use it to encrypt a message [4]. The private key, in contrast, is used to decrypt coded messages back into plaintext and is meant to be known only by the intended recipient of the encrypted messages [4]. This system relies on something known as a "trapdoor function". Just like it is easy to fall through a trapdoor, but almost impossible to climb back up through it, so too it is very easy to encrypt a message using the public key, but it is almost computationally impossible (when big enough numbers are used) to use the knowledge of the public key to decrypt a message [5]. In this way, we ensure that only the intended recipient can read the message[1].

To generate these public and private keys for RSA, first, we choose two random, very large prime numbers, p and q [6]. We then multiply p and q together to get another large number, n, and then multiply (p-1) and (q-1) together to get yet another large number, Φ(n) [6]. We then choose a number E to be used for encryption. E must be relatively prime to Φ(n) and a positive integer less than Φ(n) [6]. Next, we choose a D for decryption such that E * D modulo Φ(n) = 1 [6]. Together, E and n are used as the public key and D is kept as the private key used for decryption [6].

To encrypt plaintext message M using RSA, we use the formula C = M^E modulo n where C is the ciphertext or coded message [5]. It is almost impossible to decrypt this message without knowing D because n and E were generated based on two very large prime numbers [5]. Without knowing what those numbers were, it is almost impossible to work backward to figure out what the original message said [5]. Multiplying two prime numbers together is simple, but given the product, there is no easy way to turn it into its two prime factors. For large enough

[1] RSA can also be used in the opposite direction- where encryption is done with the private key and decryption with the public one. Instead of verifying the accuracy of the message, this system would confirm the identity of the sender [4].

numbers, it would take so long to try every possibility that it's not even worth trying. The intended recipient who has the private key, D, however, can very easily decrypt C by applying M = C^D modulo n [5].

FHE builds off of standard RSA encryption. In 1978, Ronald L. Rivest, Len Adelman, and Michael L. Dertouzos were the first to discuss the idea of direct computation on encrypted data [7]. They discovered that when two numbers were encrypted using RSA, and then multiplied together, the encrypted result was the same as multiplying those same numbers in their unencrypted form and encrypting the product afterward [7]. This is known as a mathematical homomorphism. The term they coined for this phenomenon is "privacy homomorphisms" [7] for it would prove important in data privacy. They recognized that this property would allow for computation to be performed on data when it is still encrypted and still get the correct result [7].

More than thirty years later, in 2009, Craig Gentry developed the first usable FHE scheme out of logic gates [7]. His method, called "bootstrapping", reduced the amount of ciphertext noise [8]. This noise is an additional number added to encrypted data to further obscure things and make it harder to break the encryption. However, if multiple operations are performed sequentially, the noise continues to build up which can lead to computation errors and the inability to decrypt the data [7]. Bootstrapping reduces this noise to keep FHE accurate and secure. It works by decrypting and re-encrypting the data every so often to reset the noise. It uses a recursive, lattice-like method to keep the data secure while the noise is being reset [7]. Essentially, encrypted data is encrypted again while it is still encrypted with a new public, private key pair [7]. Then this second encryption is decrypted using the new private key, which leaves the message encrypted as it was originally, but with a reset noise level [7]. This scheme

allows computation to be performed on encrypted data and provides results that are encrypted in the same way. It also reduces the noise that built up as a result of sequential computation so that unlimited computation can be done.

But Gentry's model of FHE comes with a downside - it is incredibly slow. It takes about thirty minutes for a single logic gate to run on standard x86 hardware [9]. Considering the number of logic gates needed to run even the most basic algorithm, FHE performance must be accelerated significantly to be viable in production. Over time, there has been much research on speeding up this computation and much progress has been made. However, it is estimated that FHE performance must be sped up by a factor of one million times to operate at speeds viable for commercial use [7]. The development of quantum computing is thought to be the key to solving this hardware problem.

FHE works by building on public key encryption like RSA. It has the same encryption, decryption, and key generation functions as regular public key encryption, with an added evaluate function that applies some function to the data while it is still encrypted and an added homomorphism requirement [7]. This requires that the decrypted result of applying an encrypted function to encrypted data is the same as the result of applying the same decrypted function to the same decrypted data [7].

FHE works by encrypting individual bits into polynomials [7]. To encrypt a single bit b, we choose a very large odd number p [7]. For each encryption, we choose a random large multiple of p, $q_i p$ [7]. We then sum b with $q_i p$ and a noise factor [7]. This noise is defined by doubling a small random number r_i for every encryption [7]. Finally, we get that our ciphertext c is equal to $q_i p + 2r_i + b$ [7]. The public key for this encryption is $q_i p + 2r_i$ [7].

To decrypt c back into b, we need the private key and to neutralize the noise [7]. The private key is p, our original large odd number [7]. By applying b = c modulo p modulo 2, we remove the noise and return to our original bit b [7]. By applying modulo p, we decrypt the message and the additional modulo 2 counteracts the noise.

Assuming the research continues and the hardware is improved enough to accelerate performance to commercial speeds, FHE can be a great way to use entity resolution across databases. Different companies can cross-reference records and discover which of their own records can be combined into distinct entities without compromising protected confidential data from other sources. Credit Card Company A can now block their records “John Smith”, “123456789” with “John Smith”, “123 Appledale Rd.” based on the third record “John Smith”, “123 Appledale Rd.”, “1234567890” from Hospital B without ever actually seeing that record. Credit Card Company A only knows that there exists a record somewhere in Hospital B’s database that somehow allowed them to block their John Smiths together, but it will never know exactly what that record is. No confidentiality has been violated, and because of FHE’s homomorphic qualities, we can be sure that the results are just as accurate as they would have been if derived from plain text, decrypted data.

For this reason, some have already begun to integrate FHE with entity resolution. Two papers [10-11] discuss this very subject. Both of these groups of researchers chose the same method of entity resolution in their integration. In this paper, we have discussed the blocking and scoring method. These other papers refer to another approach to entity resolution known as Position Prefix Join (PP Join).

PP Join uses the Jaccard similarity between records to determine their similarity [10]. Given two sets, the Jaccard similarity between them is the size of their intersection divided by

the size of their union [10]. Because the input for Jaccard similarity is sets and not strings, to use this metric, we must first tokenize the strings that we want to compare [10]. This means that we must break down longer strings into sets of individual words. Various algorithms can be used to accomplish this goal, such as the n-grams algorithm which is preferred by both papers [10-11].

Based on the Jaccard similarity between two records, we decide whether or not they are considered similar enough to be the same entity. If the Jaccard similarity meets a certain predetermined threshold, then we join these records together into a single entity. Otherwise, we consider them too dissimilar.

The PP Join method of entity resolution is similar in certain ways to the blocking and scoring system we discussed earlier. One notable difference is that PP Join uses *only* the Jaccard similarity algorithm to compare strings, as opposed to the blocking and scoring method, which can use other similarity algorithms as well. So while it is true that some researchers have already begun researching integrating PP Join with FHE, there has not been adequate research designated to integrate *other* simple string manipulation and comparison algorithms with FHE. It is the goal of this paper to bolster this research.

The Transpiler

Toward this end, we have taken a few common string algorithms that can be used in blocking and scoring and experimented with integrating them with FHE. To accomplish this, we used the Google Fully Homomorphic Encryption Transpiler [12]. The transpiler is software that allows us to perform transformations on data without decrypting it. It takes care of all of the behind-the-scenes FHE so that we don't have to. It encrypts the inputs and the code, applies the encrypted code to the encrypted inputs to get an encrypted result, and then decrypts that result and displays it to the user. The system runs on a Bazel build system which takes all of the

dependency files necessary to build a particular program and puts them together in the correct order. The transpiler comes with a directory of example code that the developers wrote to demonstrate how it works. Each example is its own subdirectory that contains multiple files, including the actual code or algorithm, test code, header files, and a Bazel build file. Everything is set up so that to run one of these examples we only have to type a Bazel run command and the transpiler takes care of the rest.

For this project, we decided to use this transpiler because some of the existing examples in the directory are basic string manipulation algorithms. This transpiler was the only one we found that had built-in examples that transformed strings. Because this project is all about string manipulation, this was the system we needed. For purposes of this project, we decided to edit the existing example string code to fit the algorithms we are interested in instead of building up our own example directories with all of the necessary build and test files.

The transpiler runs with C code, so we implemented the string algorithms in C code as well. However, because of how the transpiler works behind the scenes, there are a few complications that arise when converting the string algorithms to be compatible with the FHE compiler. The transpiler works by turning lines of code into logic gates and eventually wires in the computer. These logic gates are the physical way that the computer runs algorithms and returns results. The transpiler makes this switch from code to hardware at compile time. This means that before the code has even been run, the transpiler needs to know exactly what operations will need to be performed. This means that some changes need to be made to make C code compatible with the transpiler.

A general difference between regular C code and the C code used in the transpiler examples, including the ones written for this project, is the use of two pragma directives throughout the transpiler-friendly code [12].

The first of these directives is "#pragma hls_top". This line can be found directly above the function where the transpiler should start from when it compiles the code. If there is more than one function in a code file, the transpiler needs to know from which one to start. Usually, C programs tell compilers this by having a main function which is automatically where compilation begins. Because the transpiler does not require a main function, it uses this pragma directive to indicate the function in which the function begins [12].

The other pragma directive is used multiple times throughout each example. This one is "#pragma hls_unroll yes". It appears directly above every instance of a loop throughout the code. The purpose of this directive is to unroll every loop [12]. Loop unrolling is essentially just writing the loop body in sequence the number of times that the loop is supposed to run. Why would we do this? The point of loops in code is to make code more compact and easier for the programmer. Instead of typing the same line of code multiple times, it is typed once and the loop conditions tell the computer how many times to execute it. When we unroll a loop, we check the loop conditions to see how many times it is supposed to execute and rewrite the lines from the loop body that many times.

Consider the following loop:

```
for (int i=0; i<5; i++) {
     do something;
}
```

The loop header (the first line) initializes a new variable, i, and sets it equal to zero. On each iteration of the loop, i is incremented by one, and the loop continues until i is equal to five. At that point, the loop is over and the program continues to the next line of code. This means that the loop will run exactly five times. Once when i is zero, once when i is one, once when it is two, once when it's three, and finally once when it's four. After this fifth iteration, i is incremented to five. At that point, the loop condition has been met and we fall out of the loop to continue to the rest of the program. This is a loop with a non-variable end condition. It will run exactly five times no matter what.

To unroll this loop, the program first analyzes the header and determines that this loop will run five times in total. In other words, the body of the loop (the "do something" line in this case) will be run five times. Another way to accomplish this is to simply write the following code:

```
do something;
do something;
do something;
do something;
do something;
```

When a loop is unrolled, the compiler takes a loop like the code we started with and treats it as consecutive lines of code like these. Unrolling all loops is important for the FHE transpiler that turns code into physical circuits and logic gates that are used in FHE. To do this, it needs to know exactly what each line of code is going to do and how often it needs to be run. Unrolling loops is what allows the transpiler to translate the code as it needs to.

It is clear from the above example that if a loop has what's called a variable end condition, it cannot be unrolled. If, for example, the loop condition is based on another variable that is updated throughout the rest of code, it is impossible to know at compile time exactly how many times the loop will run in total. If the compiler doesn't know how many times the loop will be run, then it cannot be unrolled. This becomes relevant as we try to implement string algorithms into FHE code.

Another important feature of the FHE transpiler is that all code examples that involve strings define them as having length MAX_LENGTH. This is a constant that is always set in the header file in the example directory. All strings in C code are really just arrays of characters. Generally, variable-length arrays are allowed because the code can check how many items are in the array before trying to access them. That way it can make sure we are not trying to access an element that doesn't exist, like the fifth element of a three-element array. However, this does not work with FHE where all variables are encrypted. There is no way to check the contents of an array before accessing it [12]. For this reason, the FHE transpiler does not support variable length arrays [12] [2]. Instead, we choose to set the maximum length of strings that the transpiler will allow, based on the biggest possible string we might need. No matter how many characters are actually in any given string, the transpiler will treat each one as though it is of length MAX_LENGTH. Because the transpiler can't check the contents of the string, it has to run algorithms on every single character from zero to MAX_LENGTH, even if not all the available characters are being used. This slows down the code significantly. For this reason, although the built-in MAX_LENGTH is thirty-two characters, we lowered it to sixteen for our algorithms that use only small strings.

[2]This also is beneficial for data privacy. Because the transpiler treats all strings as being the same size regardless of their actual length, the actual sizes of the strings remain secret and protected.

Algorithms

Many different string algorithms can be used for the blocking and scoring processes in entity resolution. For this project, we have chosen to focus on four of them, two for blocking and two for scoring.

The first algorithm we will discuss is known as the Soundex algorithm. This is a phonetic algorithm that turns strings into codes based on how the words sound when spoken aloud. This helps catch slight spelling variations like “Smith” and “Smyth”. The U.S. Census uses this algorithm to help look up names [13]. Soundex operates on a single string and converts it into a four-character code [13]. To use it on a database record like our John Smith above, we must decide what string version of the record to pass into the algorithm. Perhaps we decide to pass in the first name concatenated with the last name, or perhaps the last name concatenated with the zip code. Or perhaps we decide to block the records in more than one way using Soundex and compare the results. Whatever the input string is, we then feed it through the Soundex algorithm. We keep the first letter of the string unchanged [14]. Then, the rest of the string is transformed as follows. All consonants are replaced with numbers according to a phonetic coding system. Letters ‘B’, ‘F’, ‘P’, and ‘V’ become the number ‘1’, ‘C’, ‘G’, ‘J’, ‘K’, ‘Q’, ‘S’, ‘X’, and ‘Z’ ‘2’, ‘D’, ‘T’, ‘3’, ‘L’, ‘4’, ‘M’, ‘N’, ‘5’, ‘R’, ‘6’ [14]. Then all vowels and other consonants are removed from the string [14]. This leaves a string that begins with a letter and is followed by a series of numbers. A Soundex code should contain exactly four characters. If the string is too short, we pad it with zeros at the end to make it four characters, and if it is too long, we truncate it after the fourth character [14]. We can now block records based on the Soundex code.

The regular C soundex algorithm looks like this:

```
// Function to generate Soundex encoding for a word
char* soundexEncoding(const char* word, char* encoding) {
   int len = strlen(word);
```

```
    encoding[0] = word[0]; // First letter is always included
    char prevDigit = '\0';
    int encIndex = 1;
    for (int i = 1; i < len && encIndex < 4; ++i) {
        char digit = soundexDigit(word[i]);
        if (digit != '\0' && digit != prevDigit) {
            encoding[encIndex++] = digit;
            prevDigit = digit;
        }
    }
    // Pad or truncate the encoding to ensure it has 4 characters
    while (encIndex < 4) {
        encoding[encIndex++] = '0';
    }
    encoding[4] = '\0'; // Null-terminate the encoding string
    return encoding;
}
```
[15]

The bulk of the code happens in the for loop. The number of loops depends on an outside variable (encIndex), making it a loop with a variable end condition. This means that until the loop is run, it's impossible to know how many times it will need to be run. This does not work with the transpiler's need to turn all code into logic gates at compile time. The regular Soundex code also uses a variable called prevDigit that is set to the digit from the previous iteration of the loop. It uses this variable to determine if the same character is repeated more than once in a row. Repeated characters do not affect the phonetics of a word so Soundex disregards runs of the same character. If the current character is the same as the previous one, we remove it from the string. The dependence on this outside variable also interferes with the transpiler's ability to translate code into logic gates. There is again no way to know the value of prevDigit on each individual loop iteration at compile time. To account for these issues, we need a very simplified version of the Soundex algorithm:

```
// Function to generate Soundex encoding for a word
#pragma hls_top
```

```
void CapitalizeString[3](char my_string[MAX_LENGTH[4]]) {
 const int len = StrLen(my_string);
   #pragma hls_unroll yes
 for (int i = 1; i < MAX_LENGTH; i++) {
   if (i < len){
     my_string[i] = soundexDigit(my_string[i]);
   }
 }
}
```

This code does not remove any runs of the same letter, nor does it truncate or pad the result to get a Soundex code of exactly four characters. To work with the transpiler's need for simple loops, we don't change the size of the string. This simplified Soundex acts as a basic proof of concept, showing that the general idea of the Soundex algorithm can be accomplished in the FHE transpiler. This version of the algorithm takes in a string and returns a string of equal length, in which every character has been replaced with the correct Soundex coded digit[5]. Any character in the string that does not have a valid Soundex code is replaced with a zero. This simplified version is enough to show that FHE has a way to go before it is ready for commercial production. Even this basic program, which would take a few microseconds to compile and run with a regular C compiler, took four minutes and forty seconds to run using FHE. If the most basic programs take this long to run, the necessary more robust programs in commercial use would run even slower, much too slow to be practical. Even the added security benefits are not enough to justify such a computationally expensive program.

For this algorithm, we used the transpiler built-in example called string_cap [12]. This example takes in a string as input and returns the same string with the first letter of every word capitalized. The Soundex algorithm also takes a string as input and returns a string so the only

[3] The function name is taken from the built in transpiler example that this code modified. Changing the name of the function affects the build file and how the program runs, so we left the name unchanged.
[4] As mentioned above, the transpiler version of the algorithm takes only strings of exactly length MAX_LENGTH as opposed to the regular C code version that takes strings of any length, both shorter and longer.
[5] This is true of all characters in the string except the first one, which remains unchanged by the Soundex algorithm.

changes needed to this example to make it work for Soundex was to change the actual algorithm in the main code file.

Another method of blocking records for entity resolution is to simplify the string to try and account for slight variations in spelling and convention. One way to do this is to change a record with the name field in the form “FIRST_NAME LAST_NAME” to be the first initial of the first name concatenated with the first four letters of the last name. This would turn “JOHN SMITH” into “JSMIT”. This shorter version of the name becomes the blocking key to group records together. We would then score all records with the blocking key “JSMIT” against each other to determine how many of them refer to the same real-world entity.

This method seems easy enough and indeed, in standard C code, it is very simple:

```
// Assumes name in format 'FIRST LAST' where first contains between
// 1 and 14 characters
char* concat(char my_string[]) {
   int len = strlen(my_string);
   int space = findSpace(my_string, len);
   int i;
   for (i=1; i < 5; i++) {
     my_string[i] = my_string[i+space];
   }
   my_string[i] = '\0'; //null terminate the string after the code
   return my_string;
}
```

First we find the index of the space in the string so that we know where the last name begins. Then, in a loop, we set the second, third, fourth, and fifth positions in the string to the first, second, third, and fourth characters after the space respectively. We then null terminate the string after the last character of the blocking key.

However, things become, predictably, more complicated when we use the FHE transpiler. When we try the loop from the standard code in the transpiler, we get an error. This is because the loop we created references a variable, namely the space variable that contains the index of

the space in the string, within the loop. Because this variable is determined as part of the code, at compile time the transpiler has no way of knowing what the value of space will be. For this reason, it cannot turn lines of code that reference this unknown value into logic gates.

To fix this problem, we have to make the code much clunkier and less elegant. The point of variables and loops in a program is to consolidate code and make it more streamlined and easier to read. But in this case, the combination of the loop and our space variable makes it impossible for our code to function with the transpiler. For this reason, we must remove the ambiguity that the space variable adds to the loop. Namely, we have to create a loop that does not depend on the space variable.

But how can we do that? We need to know where the space in the string is to know where the characters that make up the last name begin. The only way to use the information from the space variable without using it in the loop is to write the program in the following manner:

```
#pragma hls_top
void CapitalizeString[6](char my_string[MAX_LENGTH[7]]) {
 // Assumes name in format 'FIRST LAST' where first contains between 1 and 14 characters
 int space = findSpace(my_string);
 if (space == 1) {
   int i;
#pragma hls_unroll yes
   for (i=1; i < 5; i++) {
     my_string[i] = my_string[i+1];
   }
#pragma hls_unroll yes
   for (int j=i; j<MAX_LENGTH; j++) {
     my_string[j] = ' ';
   }
 }
```

[6] The function name is taken from the built in transpiler example that this code modified. Changing the name of the function affects the build file and how the program runs, so we left the name unchanged.

[7] As mentioned above, the transpiler version of the algorithm takes only strings of exactly length MAX_LENGTH as opposed to the regular C code version that takes strings of any length, both shorter and longer.

```
 if (space == 2) {
   int i;
#pragma hls_unroll yes
 for (i=1; i < 5; i++) {
     my_string[i] = my_string[i+2];
   }
#pragma hls_unroll yes
 for (int j=i; j<MAX_LENGTH; j++) {
   my_string[j] = ' ';
   }[8]
 }
...
 if (space == 13) {
   int i;
#pragma hls_unroll yes
   for (i=1; i < 3; i++) {
     my_string[i] = my_string[i+13];
   }
#pragma hls_unroll yes
   for (int j=i; j<MAX_LENGTH; j++) {
     my_string[j] = ' ';
   }
 }
 if (space == 14) {
     my_string[1] = my_string[15];
#pragma hls_unroll yes
   for (int j=2; j<MAX_LENGTH; j++) {
     my_string[j] = ' ';
   }
 }
}
```

This code is significantly longer and less elegant than our original version. In this one, we have to have a separate if statement to handle each possible value of the space variable. In reality, the space variable can have any value between one and fourteen. We only show the first and last two such cases here because including the entire code takes up four pages[9]. It is

[8] Note that in this version of the code, instead of null terminating the string after the blocking key, we blank out the rest of the string to kep it the same length. This keeps the program consistent with the example it is based off of and therefore allows it to run without changing the example Bazel build file.
[9] The full code can be found in the Appendix

generally considered bad coding practice to write a program so repetitive and primitive, but in this case, it's the only way to implement this algorithm in a way compatible with the transpiler. Because all numbers are either part of the loop variable or hardcoded, the transpiler can translate it directly into circuits at compile time. With this simplified yet clunky implementation of the algorithm, we can take in a string of up to sixteen characters that have at least one character for first name and one for last name and transform it from the form “FIRST_NAME LAST_NAME” to “FLAST”.

This is a simple program that takes microseconds to run under normal circumstances. But again, when compiled through the transpiler using FHE, it takes three minutes and fifteen seconds for encryption, compilation, run, and decryption. This again is a significant slowdown for a very simple program. Anything more complicated would surely be significantly worse.

For this algorithm, we also used the string_cap example that comes with the transpiler [12]. It takes a single string as an input and returns another string. There was minimal editing required to get this program running properly.

We now transition to discuss two scoring algorithms. The cosine similarity algorithm is a way to compare the similarity of two strings by converting them into vectors. This is done by creating a list of unique words that can be found in both strings [16]. Each string is turned into a vector whose order is the same as the length of the list of unique words [16]. The value of each vector at each position is determined by the number of times the word of that position in the unique words list can be found in the string that this vector contains [16]. Once we have the vectors, we calculate the dot product of the two and the individual magnitudes of each one [16]. We then divide the dot product by the product of the magnitudes [16]. This will give us a value between zero and one known as the cosine similarity [16]. A cosine similarity of exactly one

means that the strings represented by the vectors are exactly the same while a score of zero indicates that the vectors have nothing in common [16]. This can be used in the scoring process to determine how similar records that were blocked together actually are. We can choose a cosine similarity cutoff where we consider any value below it to be dissimilar and anything above it as similar enough to group the records into a single entity.

A trivial example of this algorithm is as follows. Considering the strings "hello world" (String1) and "hello there" (String2), we want to determine the cosine similarity between them. The list of unique words across both strings would be "hello", "world", and "there". String1 becomes the vector [1, 1, 0][10] and String2 is [1, 0, 1]. The dot product of these vectors is 1, the magnitude of the String1 vector is √2, and the magnitude of the String2 vector is also √2. We now divide the dot product (1) by the product of the magnitudes (√2 * √2). 1/ (√2 * √2) = 1/√4 = ½ = 0.5. The cosine similarity between our strings is 0.5. This makes sense because our strings contain two words each and have one in common. Half of the strings are the same which gives us a cosine similarity of one half.

While cosine similarity is a very useful scoring algorithm that can be used in the Entity Resolution process, unfortunately at this time it cannot be integrated with the Google transpiler. Because the transpiler is meant only to be a proof of concept, it only has very basic functionality. Two limitations of the transpiler directly affect the implementation of cosine similarity. One is that the transpiler does not support floating point numbers [12]. Otherwise known as decimal numbers, these are essential in the implementation of cosine similarity. The most basic reason for this is that the result of the cosine similarity algorithm is itself a decimal between zero and one. But there are other calculations done throughout the process that also result in decimals. Without decimal numbers, the algorithm loses its precision and usefulness. The other factor that makes it

[10] Because it contains the words "hello" and "world" once and "there" not at all.

impractical to code cosine similarity on the current transpiler is that it does not allow for any standard C header files to be included in the code. Because of this, the square root function is defined in the standard C header math.h, cannot be used. To find the magnitude of the vectors as part of the algorithm, we must find the square root of each of the components of each vector. Without the built-in, C-implemented square root function from math.h, we would have to implement our own square root algorithm to be used in the code which is beyond the scope of this project. As the transpiler stands today, it is not compatible with the cosine similarity algorithm. But hopefully, as more research is done on the subject of FHE, the transpiler will be updated and improved and cosine similarity will be an option for an entity resolution system that uses FHE.

The final string algorithm we will discuss is minimum edit distance. This is a much more complicated algorithm that either relies on recursion or nested loops. It is a way to determine how many changes one string must undergo to become another string. For example, the string "John Smith" can be turned into "Jon Smith" with just one change: dropping the H. The smaller the minimum edit distance between two strings, the more similar they are. There are three possible edit operations that the algorithm recognizes. Characters can either be inserted, removed, or replaced from the original string. A recursive approach to finding the least number of edits required to transform one string into another works as follows. Starting from the rightmost character, compare both strings character by character [17]. If the characters match, no action is required, but otherwise, we recursively try each of the three possible operations on one string to make both match and then continue the process until the end of both strings [17]. The algorithm keeps track of how many operations are performed in each of the different recursive paths that actually result in the two strings matching and returns the smallest of these numbers

[17]. In this way, we try every single combination of operations to turn one string into the other to discover the minimum number of operations required to do so, i.e. the minimum edit distance.

There is another way to calculate the minimum edit distance between two strings that does not require recursion. This method is simpler in terms of code but slightly less intuitive. In this version, we keep track of the minimum edit distance in two arrays. One holds the edit distance thus far (previous) and the other is where the updated minimum edit distance is calculated (current) [17]. Both of these arrays are the length of the first string plus one [18]. We iterate over the length of the first string and then within that loop, we iterate over the length of the second string [17]. The first loop represents what character we are up to in the first string and the second loop represents what character we are up to in the second string. In each iteration of the outer loop, we iterate over the entire second string and calculate for each character in the second string, what is the minimum amount of edits that can be made to get from our current character in the first string to the character we are up to in the second string [18]. For example, if our strings are “kittens” and “mittens” and we are on the second iteration of the outer loop, that means we are up to the substring “ki” in “kittens” and we loop over the characters in “mittens”. The first element in the current array is the minimum amount of edits required to get from “ki” to the empty string (two deletions). The second element is the minimum number of edits required to get from “ki” to “m”(one replacement and one deletion). The next is the minimum number of edits to get from “ki” to “mi” (one replacement). And so on. The minimum edit numbers are calculated by comparing the number of edits that would be required at each step, which can be determined in the following manner: an insertion ‘costs’ the value found at one less than the loop iteration number in the current array plus one, deletion is the value of the loop iteration in the previous array plus one, and substitution is the value at one less than the loop iteration in the

previous array (plus one if the previous characters in both strings are not the same) [19]. After each iteration over the second string, we set the previous array to be equal to the current one to save those results for the next iteration's calculations [17]. When both loops are complete the minimum edit distance between the strings can be found in the last position in the previous array [18].

The smaller the edit distance the more similar the strings are. When comparing strings as part of the scoring process, we can choose an edit distance threshold that is small enough that the strings can be considered similar. If the edit distance between two strings (or fields) falls within this range we can assume that they refer to the same real-world entity.

The recursive approach to minimum edit distance is not compatible with the transpiler because all the branches of the recursion cannot be known at compile time. The program doesn't know all of the computation it will have to do until it actually does it. This kind of code is incompatible with the FHE transpiler which needs to know how every line of code can be translated into hardware before the code begins. This leaves us with the iterative version of the algorithm. A regular C code implementation looks as follows:

```
int edit_distance(char *s1, char *s2) {
    int len1 = strlen(s1);
    int len2 = strlen(s2);
    int *prev_row = (int *)malloc((len2 + 1) * sizeof(int));
    int *current_row = (int *)malloc((len2 + 1) * sizeof(int));
    // Initialize previous row
    for (int j = 0; j <= len2; j++)
        prev_row[j] = j;
    for (int i = 1; i <= len1; i++) {
        current_row[0] = i;  // cost of deleting characters from s1 to match
empty s2
        for (int j = 1; j <= len2; j++) {
            int cost = (s1[i - 1] == s2[j - 1]) ? 0 : 1;
            current_row[j] = min(current_row[j - 1] + 1,         // insertion
                                 prev_row[j] + 1,               // deletion
```

```
                                prev_row[j - 1] + cost);      // substitution
            }
            // Swap rows for next iteration
            int *temp = prev_row;
            prev_row = current_row;
            current_row = temp;
        }
        int distance = prev_row[len2];  // The final row will contain the answer
        free(prev_row);
        free(current_row);
        return distance;
    }
```
[19]

An important note about this code is that it contains a nested loop. This causes transpiler complications in terms of referencing outside loop variables inside the loop because the inner loop refers to the outer loop variable. To avoid this complication, we simplified the algorithm to only calculate the first iteration of the outer loop. What this means practically, is that instead of our code calculating the minimum edit distance between the entire first and second strings, it only calculates the minimum edit distance between the first character of the first string and the second string in its entirety. While not a complete implementation of the algorithm, it does demonstrate the basic idea of iteratively calculating the minimum edit distance.

```
#pragma hls_top
int simple_sum[11](char s1[MAX_LENGTH], char s2[MAX_LENGTH[12]]) {
    int len1 = StrLen(s1);
    int len2 = StrLen(s2);
    int prev_row[MAX_LENGTH + 1];
    int current_row[MAX_LENGTH + 1];
#pragma hls_unroll yes
    for (int j = 0; j < MAX_LENGTH; j++) {
      if (j <= len2) {
        prev_row[j] = j;
      }
```

[11] The function name is taken from the built in transpiler example that this code modified. Changing the name of the function affects the build file and how the program runs, so we left the name unchanged.
[12] As mentioned above, the transpiler version of the algorithm takes only strings of exactly length MAX_LENGTH as opposed to the regular C code version that takes strings of any length, both shorter and longer.

```
    }
    if (len1 > 0) {
        current_row[0] = 1;  // cost of deleting from s1[0] to match empty s2
#pragma hls_unroll yes
        for (int j = 1; j < MAX_LENGTH; j++) {
      if (j <= len2) {
            int cost = (s1[0] == s2[j - 1]) ? 0 : 1;
            current_row[j] = min(current_row[j - 1] + 1,         // insertion
                                 prev_row[j] + 1,               // deletion
                                 prev_row[j - 1] + cost);       // substitution
      }
    }
    }
    return current_row[len2];
}
```

This code is mostly the same. The key differences are the removal of the outer loop and the addition of the #pragma lines to unroll all the loops. Even this dramatically simplified version of the code that only finds a partial minimum edit distance takes five minutes and five seconds to run on the FHE transpiler. The full minimum edit distance algorithm would take even longer. This is in contrast to the regular C code version of the entire algorithm that takes only microseconds to run.

For this implementation, we modified the transpiler example simple_sum [12]. This example takes in two integers and returns their sum as an integer. We chose this example because it takes in two inputs and returns a single integer input just like minimum edit distance. We tweaked the example so that it took in two string inputs instead of integers and changed the actual algorithm in the main code file to do minimum edit distance instead of simple addition.

Conclusion

In a world full of data, it is important to make sense of it all and to be able to determine which data points refer to the same real-world entities. Blocking and scoring is a system of entity resolution that minimizes the amount of computationally expensive fuzzy string comparisons

that need to be performed by blocking the records that are likely to be the same entity together. This method can be useful in comparing data across databases, either to determine common entities between separate databases or to use connecting data across databases to connect records within a single database. This potential benefit is limited by confidential and protected data that cannot be shared between databases. Using FHE to encrypt the data would mitigate this problem by allowing the entity resolution to be computed on data that remains encrypted throughout the process.

In this paper, we have detailed some basic implementations of string algorithms with FHE that are useful in blocking and scoring entity resolution systems. Integrating these string operations with FHE is valuable not only as a step to implementing the blocking and scoring method of entity resolution with FHE but also for computer science in general. In its current state, FHE cannot be used commercially because it is far too slow. But as the hardware for FHE continues to improve, there will be more and more practical applications for FHE. These simple string operations serve as primitives in different search and database applications. They have universal value to any process that requires string matching. Here we have taken the first steps to implementing them with FHE.

Appendix

Below please find the full FHE transpiler implemented version of the string algorithms discussed in this paper, including all utility functions.

Soundex Code:

```
#include "string_cap.h"
int StrLen(char my_string[MAX_LENGTH]) {
 int len = MAX_LENGTH;
#pragma hls_unroll yes
 for (int i = MAX_LENGTH - 1; i >= 0; i--) {
   if (my_string[i] == '\0') {
     len = i;
   }
 }
 return len;
}
char toupper(char c) {
   if (c > 96 && c < 123) {
       c -= 32;
   }
   return c;
}
// Function to convert a character to its Soundex digit
char soundexDigit(char c) {
   switch (toupper(c)) {
       case 'B': case 'F': case 'P': case 'V':
           return '1';
       case 'C': case 'G': case 'J': case 'K': case 'Q': case 'S': case 'X':
case 'Z':
           return '2';
       case 'D': case 'T':
           return '3';
       case 'L':
           return '4';
       case 'M': case 'N':
           return '5';
       case 'R':
           return '6';
       default:
           return '0'; // Not a valid Soundex digit - use '0' not '\0'
```

```
    }
  }
  // Function to generate Soundex encoding for a word
  #pragma hls_top
  void CapitalizeString(char my_string[MAX_LENGTH]) {
   const int len = StrLen(my_string);
     #pragma hls_unroll yes
   for (int i = 1; i < MAX_LENGTH; i++) {
     if (i < len){
       my_string[i] = soundexDigit(my_string[i]);
     }
   }
  }
```

Concatenation Algorithm Code:

```
  #include "string_cap.h"
  int findSpace(char my_string[MAX_LENGTH]) {
  #pragma hls_unroll yes
   for (int i=0; i<MAX_LENGTH; i++) {
     if (my_string[i] == ' ')
       return i;
   }
   return -1;
  }
  #pragma hls_top
  void CapitalizeString(char my_string[MAX_LENGTH]) {
   // Assumes name in format 'FIRST LAST' where first contains between 1 and 14
characters
   int space = findSpace(my_string);
   if (space == 1) {
     int i;
  #pragma hls_unroll yes
     for (i=1; i < 5; i++) {
       my_string[i] = my_string[i+1];
     }
  #pragma hls_unroll yes
     for (int j=i; j<MAX_LENGTH; j++) {
       my_string[j] = ' ';
     }
   }
   if (space == 2) {
     int i;
```

```
#pragma hls_unroll yes
 for (i=1; i < 5; i++) {
     my_string[i] = my_string[i+2];
   }
#pragma hls_unroll yes
 for (int j=i; j<MAX_LENGTH; j++) {
   my_string[j] = ' ';
   }
 }
 if (space == 3) {
   int i;
#pragma hls_unroll yes
   for (i=1; i < 5; i++) {
     my_string[i] = my_string[i+3];
   }
#pragma hls_unroll yes
   for (int j=i; j<MAX_LENGTH; j++) {
     my_string[j] = ' ';
   }
 }
 if (space == 4) {
   int i;
#pragma hls_unroll yes
   for (i=1; i < 5; i++) {
     my_string[i] = my_string[i+4];
   }
#pragma hls_unroll yes
   for (int j=i; j<MAX_LENGTH; j++) {
     my_string[j] = ' ';
   }
 }
 if (space == 5) {
   int i;
#pragma hls_unroll yes
   for (i=1; i < 5; i++) {
     my_string[i] = my_string[i+5];
   }
#pragma hls_unroll yes
   for (int j=i; j<MAX_LENGTH; j++) {
     my_string[j] = ' ';
   }
 }
```

```
 if (space == 6) {
   int i;
#pragma hls_unroll yes
   for (i=1; i < 5; i++) {
     my_string[i] = my_string[i+6];
   }
#pragma hls_unroll yes
   for (int j=i; j<MAX_LENGTH; j++) {
     my_string[j] = ' ';
   }
 }
 if (space == 7) {
   int i;
#pragma hls_unroll yes
   for (i=1; i < 5; i++) {
     my_string[i] = my_string[i+7];
   }
#pragma hls_unroll yes
   for (int j=i; j<MAX_LENGTH; j++) {
     my_string[j] = ' ';
   }
 }
 if (space == 8) {
   int i;
#pragma hls_unroll yes
   for (i=1; i < 5; i++) {
     my_string[i] = my_string[i+8];
   }
#pragma hls_unroll yes
   for (int j=i; j<MAX_LENGTH; j++) {
     my_string[j] = ' ';
   }
 }
 if (space == 9) {
   int i;
#pragma hls_unroll yes
   for (i=1; i < 5; i++) {
     my_string[i] = my_string[i+9];
   }
#pragma hls_unroll yes
   for (int j=i; j<MAX_LENGTH; j++) {
     my_string[j] = ' ';
```

```
    }
  }
  if (space == 10) {
    int i;
#pragma hls_unroll yes
    for (i=1; i < 5; i++) {
      my_string[i] = my_string[i+10];
    }
#pragma hls_unroll yes
    for (int j=i; j<MAX_LENGTH; j++) {
      my_string[j] = ' ';
    }
  }
  if (space == 11) {
    int i;
#pragma hls_unroll yes
    for (i=1; i < 5; i++) {
      my_string[i] = my_string[i+11];
    }
#pragma hls_unroll yes
    for (int j=i; j<MAX_LENGTH; j++) {
      my_string[j] = ' ';
    }
  }
  if (space == 12) {
    int i;
#pragma hls_unroll yes
    for (i=1; i < 4; i++) {
      my_string[i] = my_string[i+12];
    }
#pragma hls_unroll yes
    for (int j=i; j<MAX_LENGTH; j++) {
      my_string[j] = ' ';
    }
  }
  if (space == 13) {
    int i;
#pragma hls_unroll yes
    for (i=1; i < 3; i++) {
      my_string[i] = my_string[i+13];
    }
#pragma hls_unroll yes
```

```
    for (int j=i; j<MAX_LENGTH; j++) {
      my_string[j] = ' ';
    }
  }
  if (space == 14) {
      my_string[1] = my_string[15];
#pragma hls_unroll yes
    for (int j=2; j<MAX_LENGTH; j++) {
      my_string[j] = ' ';
    }
  }
}
```

Minimum Edit Distance Code:

```
#include "simple_sum.h"
int StrLen(char my_string[MAX_LENGTH]) {
 int len = MAX_LENGTH;
#pragma hls_unroll yes
 for (int i = MAX_LENGTH - 1; i >= 0; i--) {
   if (my_string[i] == '\0') {
     len = i;
   }
 }
 return len;
}
int min(int x, int y, int z) {
   if (x <= y && x <= z) return x;
   if (y <= x && y <= z) return y;
   return z;
}
#pragma hls_top
int simple_sum(char s1[MAX_LENGTH], char s2[MAX_LENGTH]) {
   int len1 = StrLen(s1);
   int len2 = StrLen(s2);
   int prev_row[MAX_LENGTH + 1];
   int current_row[MAX_LENGTH + 1];


#pragma hls_unroll yes
   for (int j = 0; j < MAX_LENGTH; j++) {
     if (j <= len2) {
       prev_row[j] = j;
```

```
    }
  }
  if (len1 > 0) {
      current_row[0] = 1;  // cost of deleting from s1[0] to match empty s2
#pragma hls_unroll yes
      for (int j = 1; j < MAX_LENGTH; j++) {
    if (j <= len2) {
          int cost = (s1[0] == s2[j - 1]) ? 0 : 1;
          current_row[j] = min(current_row[j - 1] + 1,         // insertion
                               prev_row[j] + 1,               // deletion
                               prev_row[j - 1] + cost);       // substitution
    }
  }
  }
  return current_row[len2];
}
```

References

[1]B. Dean, “Backlinko,” *Backlinko*, Dec. 12, 2023. https://backlinko.com/facebook-users (accessed Apr. 25, 2024).

[2]T. Masui, “Entity Resolution: Identifying Real-World Entities in Noisy Data,” *Towards Data Science*, Sep. 21, 2023. Accessed: Apr. 25, 2024. [Online]. Available: https://towardsdatascience.com/entity-resolution-identifying-real-world-entities-in-noisy-data-3e8c59f4f41c#b64f

[3]“Types of Homomorphic Encryption,” *IEEE Digital Privacy*. https://digitalprivacy.ieee.org/publications/topics/types-of-homomorphic-encryption (accessed Apr. 25, 2024).

[4]C. Stouffer, “What is encryption? How it works + types of encryption,” *Norton*, Jul. 18, 2023. Accessed: Apr. 25, 2024. [Online]. Available: https://us.norton.com/blog/privacy/what-is-encryption

[5]J. Lake, “Exploring RSA encryption: a comprehensive guide to how it works,” *Comparitech*, Dec. 10, 2018. Accessed: Apr. 25, 2024. [Online]. Available: https://www.comparitech.com/blog/information-security/rsa-encryption/

[6]stash, “PPT - Public Key Encryption and the RSA Public Key Algorithm PowerPoint Presentation - ID:2390093,” *SlideServe*, Jul. 26, 2014. https://www.slideserve.com/stash/public-key-encryption-and-the-rsa-public-key-algorithm (accessed Apr. 25, 2024).

[7]M. Creeger, “The Rise of Fully Homomorphic Encryption,” *ACM Queue*, Sep. 26, 2022. https://queue.acm.org/detail.cfm?id=3561800 (accessed Apr. 25, 2024).

[8]“Fully Homomorphic Encryption,” *IBM Research*. https://research.ibm.com/topics/fully-homomorphic-encryption#publications (accessed Apr. 25, 2024).

[9]C. Gentry and S. Halevi, “Implementing Gentry’s Fully-Homomorphic Encryption Scheme,” *IBM Research*, Feb. 2011. Accessed: Apr. 25, 2024. [Online]. Available: https://eprint.iacr.org/2010/520.pdf

[10]T. Ghai, Y. Yao, Srivatsan Ravi, and P. Szekely, “Evaluating the Feasibility of a Provably Secure Privacy-Preserving Entity Resolution Adaptation of PPJoin Using Homomorphic

Encryption," 2022. Accessed: Apr. 25, 2024. [Online]. Available: https://arxiv.org/pdf/2208.07999.pdf
[11]Z. S. Borgs Lars Kolb, Erhard Rahm, Rainer Schnell, Christian, "Privacy Preserving Record Linkage with PPJoin".
[12]Shruthi Gorantala, Rob Springer, Sean Purser-Haskell, William Lam, Royce Wilson, Asra Ali, Eric P. Astor, Itai Zukerman, Sam Ruth, Christoph Dibak, Phillipp Schoppmann, Sasha Kulankhina, Alain Forget, David Marn, Cameron Tew, Rafael Misoczki, Bernat Guillen, Xinyu Ye, Dennis Kraft, Damien Desfontaines, Aishe Krishnamurthy, Miguel Guevara, Irippuge Milinda Perera, Yurii Sushko and Bryant Gipson, "A General Purpose Transpiler for Fully Homomorphic Encryption," *Cryptology ePrint Archive*, Paper 2021/811, 2021. Accessed: Apr. 25, 2024. [Online]. Available: https://eprint.iacr.org/2021/811
[13]"Soundex System," *National Archives*. https://www.archives.gov/research/census/soundex (accessed Apr. 25, 2024).
[14]GeeksforGeeks, "Implement Phonetic Search in Python with Soundex Algorithm," *GeeksforGeeks*, Aug. 01, 2022. Accessed: Apr. 25, 2024. [Online]. Available: https://www.geeksforgeeks.org/implement-phonetic-search-in-python-with-soundex-algorithm/
[15]ChatGPT. (GPT-3.5). OpenAI. Accessed: Apr. 18, 2024. [Online]. Available: https://chat.openai.com/chat
[16]N. Tahir, "Fuzzy string matching using cosine similarity," *Another Dev's Two Cents*, Sep. 20, 2015. Accessed: Apr. 25, 2024. [Online]. Available: https://nishtahir.com/fuzzy-string-matching-using-cosine-similarity/
[17]GeeksforGeeks, "Edit Distance," *GeeksforGeeks*, Jul. 06, 2011. Accessed: Apr. 25, 2024. [Online]. Available: https://www.geeksforgeeks.org/edit-distance-dp-5/
[18]T. André, "Unraveling the Levenshtein Algorithm: How Edit Distance Transforms Text Analysis," *Medium*, Feb. 15, 2024. Accessed: Apr. 25, 2024. [Online]. Available: https://medium.com/@thiagoandre.dev/unraveling-the-levenshtein-algorithm-how-edit-distance-transforms-text-analysis-734f7bf58c55
[19]ChatGPT. (GPT-4). OpenAI. Accessed: Apr. 18, 2024. [Online]. Available: https://chat.openai.com/chat