AN AFFINE BASED HOMOMORPHIC ENCRYPTION SCHEME

An Undergraduate Research Scholars Thesis

by

KYLE LOYKA

Submitted to the Undergraduate Research Scholars program at
Texas A&M University
in partial fulfillment of the requirements for the designation as an

UNDERGRADUATE RESEARCH SCHOLAR

Approved by Research Advisor: Dr. Sunil Khatri

May 2017

Major: Electrical Engineering

TABLE OF CONTENTS

ABSTRACT

An Affine Based Homomorphic Encryption Scheme

Kyle Loyka
Department of Electrical and Computer Engineering
Texas A&M University

Research Advisor: Dr. Sunil Khatri
Department of Electrical and Computer Engineering
Texas A&M University

Standard encryption protocols do not allow for native searching or modification of data, without decryption. In this scenario, whoever is performing the operations on this file must also have the encryption key. Homomorphic encryption aims to solve this problem. Using homomorphic encryption, an encrypted file could be searched or modified without having to decrypt it, while still preserving the privacy of the file. This feature would allow users to outsource their processing needs to third parties, while still retaining information secrecy. These third parties would be able to perform operations on behalf of the user without attributing meaning to the user's input and output values.

The homomorphic encryption model proposed in this paper works with plain text files by representing characters using the ASCII encoding scheme. These characters will be encrypted using an affine cipher. This scheme supports string search and concatenation, and integer addition and subtraction operations.

ACKNOWLEDGEMENTS

I would like to thank Dr. Sunil Khatri for his guidance in my research and in my career.

I would also like to thank my parents and brother for their support.

CHAPTER I

INTRODUCTION

With the rapid adoption of computers, the amount of information stored digitally is continuously rising. Individuals and corporations have more incentive than ever to protect their data. Encryption serves an important role in ensuring that only authorized users can access certain data. Encryption allows users to obfuscate the content of their files by encrypting them with a unique encryption key. Without access to the keys, the content of the user's encrypted files appears to be random and cannot be understood. Only with access to the unique encryption keys can the files be decrypted and readable. Modern encryption protocols have limitations that affect the ease to which encrypted files can be accessed or modified. This problem becomes significant when considering the use of cloud storage. If a user (who has access to the encryption key) wanted to modify their encrypted file that is stored in the cloud, they would have to download their encrypted file, decrypt it, make modifications to the decrypted (unencrypted) file, encrypt the modified file, and upload the encrypted modified file to the cloud. Figure 1 shows this process visually. This process is inefficient, since most of the time and computation is spent on downloading, decrypting, encrypting, and uploading. A disproportionate amount of time is spent accomplishing the user's task, which is to modify the data in unencrypted file. Additionally, the number of steps in this process decreases its ease of use for the user. The problem with modern encryption is that it is not fast or easy to modify the data in encrypted files (assuming the user has access to the encryption keys). Homomorphic encryption aims to solve this problem.

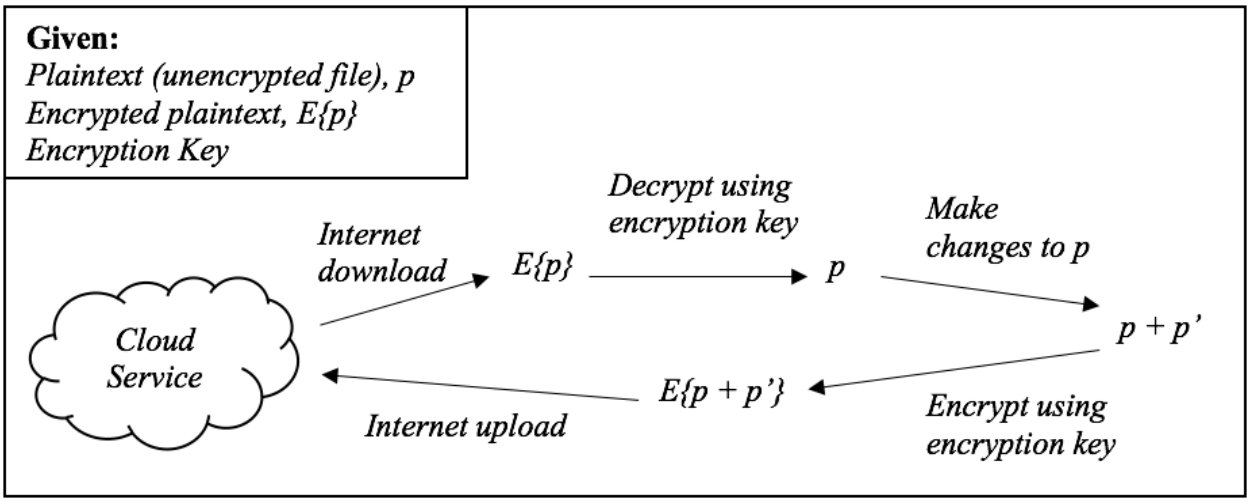

Figure 1. Modifying an encrypted file stored in the cloud using current encryption technologies. This process is inconvenient to the user since they must download and decrypt each file that they wish to modify. The changed files must then be re-encrypted and uploaded to the cloud.

Using homomorphic encryption, an encrypted file could have its contents modified without decrypting the file. Figure 2 shows how a homomorphic encryption scheme could be used to modify data stored in an encrypted file in the cloud.

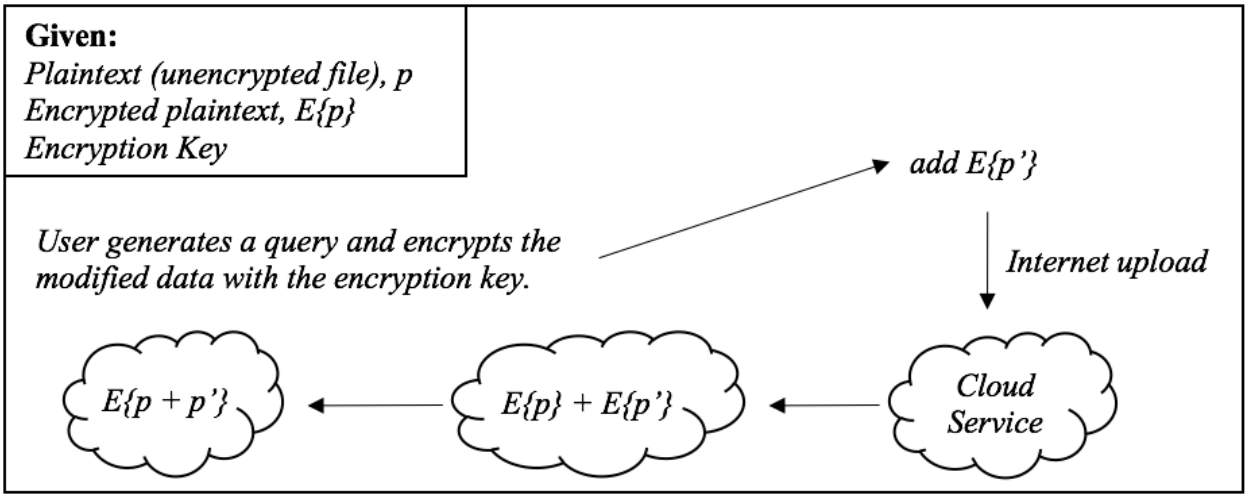

Figure 2. A homomorphic encryption scheme with cloud computing. Modifications to a file are processed on the user's computer, encrypted, and sent to the cloud. Once in the cloud, the encrypted data can be modified. The cloud service cannot derive meaning from any of the encrypted data. The cloud knows that a modification was made, but does not know the contents of the original or modified files.

Using this scheme, the user can generate a query to send to the server. This query contains an encrypted piece of data and instructions on what to do with it. For example, one of

the possible instructions could be to modify a file in the cloud based on the given encrypted piece of data. Once the cloud receives this query, the modification operation can be performed on an encrypted file using the given encrypted piece of data. The result of this operation would produce a new encrypted file. When this file is decrypted, it would contain the unencrypted version of the modifications that the user requested. It is important to note that since the file and changes to the file were encrypted, no one except for the user would have access to the original file, file changes, or modified files. This scheme provides an opaque interface through which the cloud can operate on data at the request of the user, without deriving any meaning from the data it handles.

Previous Work

In 1978, the researchers Rivest, Adleman, and Dertouzos published a concept paper formalizing the idea of homomorphic encryption [7]. That same year, Rivest, Shamir, and Adleman published the RSA encryption scheme specifications [8]. One of the incidental properties of RSA was that it was partially homomorphic, with respect to multiplications [8, 10]. However, no other operations were homomorphic. Then, in 2009, Craig Gentry published his thesis describing his design the first fully homomorphic encryption scheme [2, 3]. While it was functional, the scheme was trillions of times slower than unencrypted operations [4, 9]. Since then, companies such as IBM and Microsoft have developed their own homomorphic schemes, which have made improvements on the time it takes to compute encrypted operations [5, 11]. Despite these improvements, these schemes are still billions of times slower than unencrypted operations.

In our approach to homomorphic encryption, we rely on the affine cipher to provide cryptographic security [6]. The strength of the affine cipher can be increased by extending the input and output range and grouping data into packets to provide more entropy [12]. The goal of our research is to adapt the affine cipher to be homomorphic, that is, to support string searches and concentrations, and integer addition and subtractions.

Importance

The combination of homomorphic encryption and cloud data storage would provide massive value to consumer and enterprise users [1]. Using homomorphic encryption, a user could upload their files to Dropbox and make changes to these files in the cloud with total information secrecy. Users could keep all their files stored in the cloud, where they could have access or modify them from anywhere, and still protect the content of their files. This feature may also be attractive to businesses who want to keep their data secure, but also want to outsource their IT needs. A business using homomorphic encryption could keep sensitive financial information stored in the cloud, reducing IT costs, while still preserving information secrecy and the ease of access that employees are used to (with no special or added steps to modify their data). Another possible benefit from homomorphic computing is the privacy of internet searches (e.g. Google search). A user could encrypt their search query and receive a result from Google. Through this process, Google would not know what the user searched for and would not know what the search returned. However, the user, who has access to the encryption key, can decrypt the search results and derive meaning from them. Users may find benefit in this scheme since their searches remain private.

CHAPTER II

METHODS

This homomorphic encryption scheme is based on ASCII text encoding and the affine cipher function. Together, they allow for the searching and concatenation of encrypted text files, as well as addition and subtraction operations on encrypted integers. This approach treats strings the same way as integers which makes encrypted string data indistinguishable from encrypted integer data. These features will be expanded in the following sections.

ASCII Text Encoding

The ASCII text encoding scheme is a popular way to represent text characters on computer systems. In this scheme, 256 text-characters are each represented by a unique number code. For example, an ASCII encoded text document is stored as a file containing a sequence of ASCII codes. When a user views the text document, each ASCII code is converted into its corresponding character and displayed on the screen. Since there are 256 possible ASCII characters/codes, and each is unique, every ASCII character can be stored in a unique 8-bit (1 byte) sequence.

Affine Cipher

The other part of this implementation includes the affine cipher, which is based on modular arithmetic. Modular arithmetic is a type of math that deals with the remainders from the division of numbers. Part of modular arithmetic, and a key feature in the affine cipher, is the operation of modular reduction. This operation is denoted by using the *mod* symbol. The number

to the right of *mod* is known as the *modulus*. When performing modular reduction, the input (the number to the left of *mod*) is divided by the modulus. The remainder of this division is the result for the modular reduction operation. The principles of division show that the remainder, and therefore the result of a modular reduction, can range from [0, *modulus* - 1], for a given modulus. Any integer input can be modularly reduced, but its output will be bounded within this range.

The affine cipher is used to encrypt integer values, and in this implementation, will be used to encrypt ASCII character integer representations. Equation 1 shows a generic affine cipher encryption equation.

Equation 1 – Affine Cipher Encryption: $\varepsilon = (\alpha\ \lambda + \beta)\ \%\ \rho$

This equation transforms the input, λ, according to the values of α, β, and ρ. For this scheme, the values of α, β, and ρ are considered the encryption key. The integer input, λ, is multiplied by α and then has β added to it. This value is then modularly reduced by ρ. In the affine equation, ρ is the modulus since it appears to the right of the *mod* operator. The resulting number represents the encrypted version of λ, which is denoted by the term ε. It is important to note that for the affine cipher, gcd (α, ρ) must equal 1. This is due to the properties of modular reduction and to ensure that every input results in a unique output. If this condition is not met, it is possible that the output may not be unique. The encrypted output can be decrypted by using the inverse of Equation 1. Equation 2 shows the inverse of Equation 1. This inverse is also known as the affine cipher decryption equation.

Equation 2 – Affine Cipher Decryption: $\lambda = (\alpha^{-1}(\varepsilon - \beta))\ \%\ \rho$

To decrypt, take the encrypted value, ε, and subtract β. Then, multiply the result by the modular-inverse of α, α^{-1} (by definition, the modular inverse, α^{-1}, is the value which satisfies the equation $(\alpha\ \alpha^{-1}) \equiv 1\ mod\ \rho$). Finally, take this value and modularly reduce it by ρ. The result will be equal to λ.

From inspection, the affine cipher appears to be a linear equation. However, this is not quite the case. The affine cipher would behave similarly to a linear equation if it weren't for the *mod* ρ. As discussed previously, the size of the modulus determines the number of unique outputs. In this case, the *mod* ρ operation limits the number of unique outputs to ρ possible values. Since the affine cipher is based on a linear equation, this means there can be only ρ possible input values that each result in a unique output value. It is important to preserve uniqueness in inputs and outputs. Otherwise, if two inputs gave the same output, there would be no way to determine which input was intended. This effectively limits the number of unique inputs and outputs of the affine cipher to ρ possible values, ranging from [0, ρ -1].

In our implementation, encrypted data is stored in a data structure that contains two members: a list of integers (for representing encrypted data), and a counter (for counting the number of arithmetic operations that were performed on the data). The steps for handling string and integer data, as well as why this data structure is needed, will be described in the following sections.

Encrypting Strings

By using the ASCII encoding scheme to represent text characters as integers, text characters can be encrypted using the affine cipher. For each character in a text file, the

character's ASCII number code would be inputted into the affine cipher. This results in an encrypted number, which is then appended to a new encrypted file. Once the entire text file has been traversed, this new encrypted file would represent the encrypted version of the original text file. It should also be noted that instead of writing these outputs directly to a file, the encrypted numbers may be stored in computer memory as a list of integers. The list datatype makes it very simple to concatenate encrypted strings. To concatenate two encrypted strings, append the list of one encrypted string onto the end of the other encrypted string. The resulting list represents the encrypted version of the concatenation of the two strings.

Since there are 256 ASCII characters, there are only 256 possible inputs to the affine cipher, and therefore the modulus, ρ, must equal 256. To increase the security of the encryption scheme, the input and output range could be increased. By increasing ρ, the affine cipher would then have a larger range of unique input and output values. To utilize the larger input space, the number of characters encrypted at a time could be increased. Instead of encrypting one character at a time (which results in 256 input values), a user could encrypt two characters at a time. By encrypting two characters at a time, the input and output range grows to 256^2 unique values (individually, for both inputs and outputs). In general, when encrypting K characters at a time, the affine cipher has 256^K unique input and output values (for each). This means that the modulus, ρ, must be equal to 256^K, where K, is the number of characters encrypted at a time. Increasing the input and output ranges substantially increases the number of ways in which data can be encrypted, thus making the scheme more secure.

To encrypt K characters at a time, first, K characters are read from the original (plain text) file. Then for each of the K characters, their ASCII encodings are looked-up and converted to an 8-bit binary string. This results in K (8-bit) binary strings. These binary strings are then placed

and joined together based on the order in which their corresponding characters were read from the plain text file. This results in a single 8**K* bit binary string. This binary string is then cast back to an integer, which is unique for this series of *K* ASCII characters. This number is then encrypted using the affine cipher. Setting $\rho=256^K$ will guarantee unique outputs for every input. The output of the affine cipher can then be saved in a list of integers or in a file. The next *K* characters would be read from the plain text file, and this process would be repeated. In the case where the plaintext is not fully divisible by *K* (the last few characters of a file may not fill an entire *K* sized block), extra space characters will be appended to the end of the file so that a full *K* sized block can be formed.

With $K=1$, our scheme is susceptible to frequency analysis attacks. An attacker could analyze the frequency in which certain encrypted values occur within an encrypted file, and correlate this to the frequency in which characters appear in the English language (or other language). This gives the attacker useful information in determining the unencrypted meaning of an encrypted value. By grouping $K>1$ characters together, the frequency analysis is more ambiguous since there are more possible inputs and outputs. This does not fully mitigate the threat, but does make the attack more difficult.

Decrypting Strings

The decryption process follows the reversed sequence of steps from the encryption process. First, the encrypted value, ε, is loaded from the encrypted file or encrypted list. It is decrypted according to Equation 2. The resulting integer value is cast into an 8**K* bit binary string. This binary string is then split into *K* 8-bit blocks. Each 8-bits is cast into an ASCII value and appended to a new unencrypted file (or list) based on the order in which they appeared in the

8*K bit binary string. The next index from the encrypted file is loaded and this process is repeated. This process continues until every element in the encrypted file has been decrypted.

Encrypted String Search

This homomorphic implementation supports search operations on strings. To search for text in an affine encrypted file, the user must encrypt the string they would like to search for (this will be denoted as the *query-string)*. It is important that the query-string must be encrypted using the same parameters (K, α, β, and ρ values) that were used to encrypt the file they would like to search. The result of this encrypted query-string would be stored in memory as a list of integers. Since the encrypted file is a list of integers, it is searched by trying to find a sub-list within it that matches the list of the encrypted query-string. If the list of the encrypted query-string exists within the encrypted file, then it is known that the query-string exists within the unencrypted file. The location where the encrypted query-string's sub-list starts (in the encrypted file) is then returned to the user. This process is straightforward for the case of $K = 1$. In cases where $K > 1$, the search process becomes more complex. As K, the number of characters grouped in one encrypted block, increases, the possible ways in which a substring can be encrypted also increases. See Figure 3 for an example of how many possible ways the string "HELLO" may be encrypted with K=3.

Since the file was encrypted in K-character blocks, it is possible that the search query exists but is offset by some amount of characters in the encrypted file. This means that there is a possibility that the search string doesn't fit perfectly into K-character chunks. In this case, leading or trailing characters of the search string may be encrypted in blocks with other unknown characters. To mitigate this issue, all possible offset combinations of the query-string are

generated. Then, for a possible query, the algorithm encrypts the blocks with known values (the blocks with no *stars* in them) and stores their values in a list. If the list of known encrypted-block values match within the encrypted file/list, the index location of the match (in the encrypted file) is stored. If no match is found, the search moves on to the next possible query. If there are no remaining queries, the search would fail, and the user would be notified that no matches were found. If this first search did find a match, then the searching process continues. For the blocks

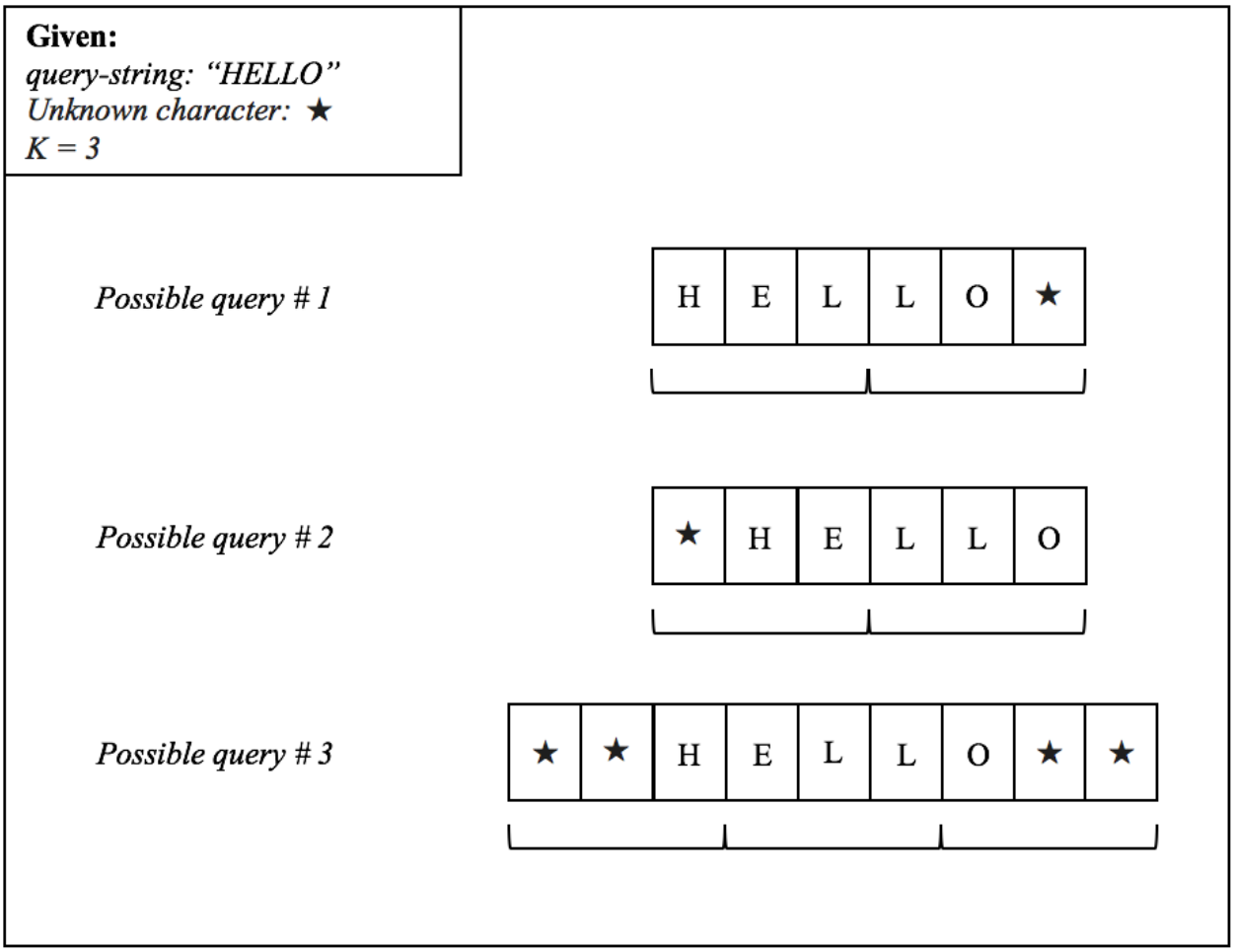

Figure 3. Variations in forming search queries. Assuming the string "HELLO" exists within an encrypted file, it is encrypted as one of the three combinations above. To fully search the file, each of these search combinations must be utilized (or until a match is found). Some of the search queries have character groupings in which one or more of the characters is unknown. Unknown characters are represented with the star symbol. For these cases, the K-block must be brute-forced.

with a *star*, block combinations are generated where the *star* is replaced with every ASCII character. In the case of two or more *stars* in one block, each *star* is varied independently. These

combinations are encrypted and compared to the index in the encrypted file (+/- some amount). This index is chosen based on the index found in the known-block search. If the *star* block is the beginning block of a query, then the index-1 is compared with all possible *star* block combinations. If a match is found and there are no other star blocks to search, then the search was successful and the user is notified of the encrypted-string's location in the encrypted file. If no match is found, then the program will attempt to search for the next possible query (if there is one). If no more possible queries exist, then the search will notify the user of no match. If a match is found and another star block needs to be searched, such as a trailing star block (see Figure 3: *Possible query #3*), then all possible combinations for that block are generated. These combinations are then encrypted and compared to the value stored at the encrypted file's index with an offset value based on the number of the known encrypted K-blocks. If a match is found, then the user is notified of the encrypted string's location in the encrypted file. If no match is found, then the next possible search query is attempted. If no other possible queries are left to attempt, then the user will be notified that the query-string was not found.

Arithmetic

One of the key principles of this encryption scheme is that integer data and string data are treated the same way. To encrypt an integer, it is first cast into a string. The string can be padded with leading "0" characters to increase its length, which, when encrypted, will obscure the magnitude (number of digits) in the integer. These padded "0"'s do not affect the unencrypted value of the object. The string is then encrypted using the same process as described the section "*Encrypting Strings"*.

To support arithmetic, a special data structure is needed. As described previously, this

data structure contains two members: a list of numbers (representing the encrypted string) and counter variable. This structure is used to represent both text data and number data. The counter is necessary when decrypting data that are the results of encrypted addition or encrypted subtraction. This is because, when adding affine encrypted values, the number of β terms increments with each addition. For subtraction, the number of β values decreases with each subtraction operation (there can be negative multiples of β values). By keeping track of the number of additions and subtractions, the extra β values can be accounted for in the decryption process. The counter is initially set to zero and is incremented for every addition and decremented for every subtraction. The counter is also useful for interpreting the unencrypted output.

Due to the way digits ("0", "1", "2", ..., "9") are encoded in ASCII, the value of an integer can be determined by subtracting 48 from any ASCII encoded numeric character. For example, since "1" is encoded as 49, the ASCI encoded value can have 48 subtracted from it to determine that 49 represents (49-48=1), the integer 1. When adding or subtracting two ASCII values, this ASCII offset value accumulates in the same way as β. When decrypting, the accumulation of 48's can be removed to reveal the integer result of addition or subtraction. If the resulting integer is greater than nine, any digits in the tens, hundreds, thousands (etc.) places act as carry values when interpreting the decrypted value. The maximum unencrypted representation for each element is limited by ρ. Specifically, the highest value for each element (that has been decrypted) is $(\rho/2 - 1)$, and the lowest is $(-\rho/2)$. If the value of an element exceeds $(\rho/2 - 1)$ then its value "rolls-over" into the negative interpretation. If the value of an element is less than $(-\rho/2)$, then it's value "rolls-under" into the positive interpretation. This behavior is based on the behavior of bitwise one's compliment.

To add two encrypted objects, first the length of each list is compared. If the lengths do not match, an addition cannot be performed. This means that if a user wanted to add “1” + “100”, they would have to encrypt these values as “001” and “100” respectively so that their string lengths matched. This is done due to constraints on the counter value and the decryption process. If the lengths were not equal, it would be possible that during an operation, that some elements in the list would have more β accumulations than others. Requiring equal lengths ensures that for each operation, each element in the list has the same number of β values. Additionally, by keeping operations restricted to lists of the same length, it is more difficult to discern the magnitude of an encrypted number based solely on its length. For example, “0001” has zeros padding the number, this obscures the magnitude of the number when observing the length of the encrypted string. After both lists are verified to have the same length, the next step in the addition process is to add the corresponding elements of each list. The results are stored in a new list. The counters for each list are added together and incremented by one. The result of the list and counter additions are stored in a new data structure. Figure 4 illustrates this process. Subtraction works in a similar way as addition, except that the corresponding elements of each list are subtracted and the counter values are subtracted and decremented by one.

Addition and subtraction are only supported for K=1 due to a limitation of the encryption process and the ASCII encoding scheme. With K>1, K characters are grouped at a time, and their bit representations are concatenated to form an 8*K bit binary string. The problem arises from how the ASCII values for the characters “0” through “9” are represented. Take for example, how ASCII “9” is encoded as 57. If you were to add the ASCII representation of “9”, 57, four times, (57+57+57+57= 228), the resulting value can be represented in eight bits. But if the ASCII representation of “9” was added five times (57+57+57+57+57 = 285), the resulting value would

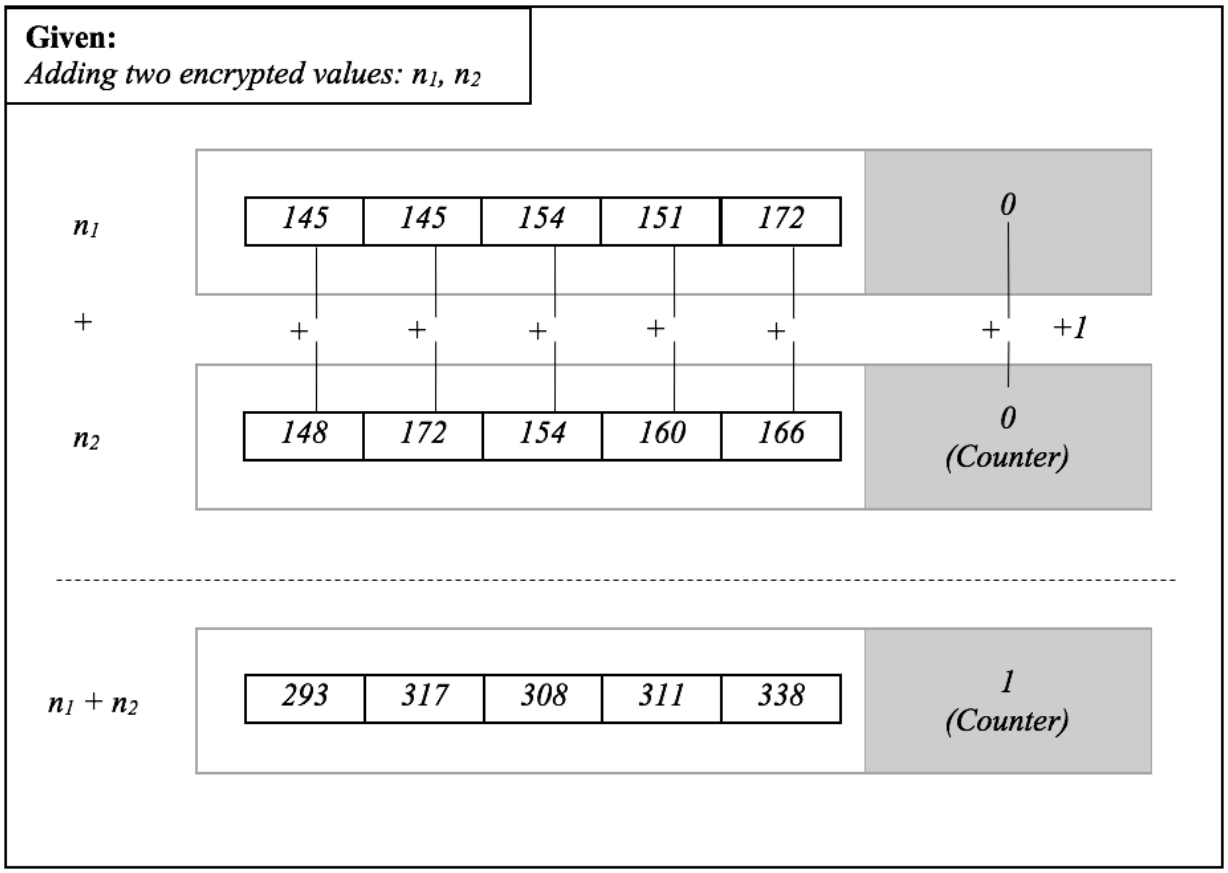

Figure 4. Addition of two encrypted objects. The corresponding elements of each list are added together and stored in a new list. The corresponding counter values are added together and incremented by one.

overflow in an 8-bit representation. In this case, when considering how each character's bits are concatenated with its neighbors' bits, the overflown bit would get carried to the next character's allocation of bits. For subtraction, subtracting the ASCII value of "9" from the ASCII value of "0" (48-57=-9) would result in the bits under-flowing, which also affects the bits of its neighbors. This alters the integrity of the data in a way that is not recoverable. This issue also occurs in the encrypted space (with encrypted integers), and cannot be mitigated without sacrificing access to the encryption keys. If the cloud had access to the encryption keys, they could know which 8-bit boundaries to watch for carries, and could make note to help in decrypting and encoding the data. However, this sacrifices access to the encryption key, which defeats the purpose of this homomorphic encryption scheme. This issue could also be mitigated

by casting ASCII values into 16 bits (which would leave an extra 8 bits for padding and overflow), but ultimately, the number of operations would still be limited. K=1 works for arithmetic since the character's bits are not concatenated with anything else. Every element in the list represents only one character value, and therefore all the bits in that element are known to represent a single character. An overflow or underflow does not disrupt other encrypted neighbors, since each element in the list is independent of the elements surrounding it. In the case of overflow, the carry digit is appended to the front of the bit representation and the size of the element container grows to contain it.

Note that it would be possible to add two non-numeric ASCII characters together. A user could encrypt "q" and encrypt "r" and tell the computer to add the two encrypted values. It would yield a result that would be indistinguishable from an encrypted integer result. Of course, when decrypting, the result would be garbage data. This is feature is beneficial though, because a user could tell the cloud to add two encrypted integers together or two encrypted characters together, but the cloud would not be able to tell between what was garbage and what was a meaningful addition/subtraction operation, providing a level of obscurity.

CHAPTER III

RESULTS

In this section, the results of our homomorphic implementation will be discussed. The code was written in the Python (2.7) programming language and tested on a system running macOS 10.12 with a 2.7GHz Intel i7 processor and 16GB RAM. Data was collected for the tables by running each test case ten times. The results for each case were then averaged and consolidated into tables based on the parameters that were varied.

Strings – Varied α

Table 1 shows the results of an experiment where the value of α was varied and all other parameters (K, β, and ρ) were held constant. These results demonstrate how varying α has negligible influence on the encryption scheme's performance. Through significant variations in α, there was no significant changes to encryption time, decryption time, encrypted file size, or encrypted search times. This behavior is expected since α acts as a multiplier in the affine cipher, but the cipher's output is limited in range by the *mod* ρ operation. The fixed ρ constrains the output values to a fixed range. Therefore, the output (encrypted) file size does not change. The encrypted file size is larger than the unencrypted file size. This is due to the (fixed) value of ρ, will be explained in the section "*Strings – Varied* ρ*"*.

Table 1: Strings – Varying α with fixed (K, β, and ρ)										
K	***α***	***β***	***ρ***	**Enc. Time (ms)**	**Dec. Time (ms)**	**Unenc. File Sz. (MB)**	**Enc. File Sz. (MB)**	**Enc / Unenc. File Sz.**	**Enc. Search Time (ms)**	**Unenc. Search Time (ms)**
1	1	71041	2^64	3517.713	5055.423	1.24	9.91	7.99	607.731	24.864
1	3	71041	2^64	3532.508	4971.240	1.24	9.91	7.99	603.580	24.219
1	199	71041	2^64	3509.908	5021.792	1.24	9.91	7.99	608.542	24.670
1	71041	71041	2^64	3519.551	5030.228	1.24	9.91	7.99	607.196	24.338
1	230917123411231	71041	2^64	3539.511	5064.104	1.24	9.91	7.99	605.245	24.177
1	12302342342342341231	71041	2^64	3511.854	5092.807	1.24	9.91	7.99	606.697	24.744

Strings – Varied *β*

Table 2 shows the results of an experiment where the value of β was varied while keeping all other parameters (K, α, and ρ) constant. These results demonstrate how variations in β do not significantly affect the performance of the encryption scheme. This behavior is expected since the affine cipher's output range is limited by the *mod* ρ operation, which limits the range of output integers, and therefore limits the size of the file.

Table 2: Strings – Varying β with fixed (K, α, and ρ)										
K	***α***	***β***	***ρ***	**Enc. Time (ms)**	**Dec. Time (ms)**	**Unenc File Sz. (MB)**	**Enc. File Sz. (MB)**	**Enc. / Unenc File Sz.**	**Enc. Search Time (ms)**	**Unenc Search Time (ms)**
1	12302342342342341231	1	2^64	3517.713	5055.423	1.24	9.91	7.99	607.731	24.864
1	12302342342342341231	3	2^64	3532.508	4971.240	1.24	9.91	7.99	603.580	24.219
1	12302342342342341231	199	2^64	3509.908	5021.792	1.24	9.91	7.99	608.542	24.670
1	12302342342342341231	71041	2^64	3519.551	5030.228	1.24	9.91	7.99	607.196	24.338
1	12302342342342341231	230917123411231	2^64	3539.511	5064.104	1.24	9.91	7.99	605.245	24.177
1	12302342342342341231	12302342342342341231	2^64	3511.854	5092.807	1.24	9.91	7.99	606.697	24.744

Strings – Varied ρ

Table 3 shows the results of an experiment where the value of ρ was varied. As ρ increased, the range of possible output values also increased. Since K=1, this meant that as ρ increased, the size of the representation of a single character also increased. This affected the encrypted file size. Since an ASCII character can be represented in 8-bits, the case where $\rho=2^8$ means that every output of the affine cipher could be represented in 8-bits. This is the same size as an unencrypted ASCII value, therefore each encrypted value was the same size as an unencrypted value. With this principle applied to every encrypted character, this meant that for $\rho=2^8$, the encrypted file size was the same as the unencrypted file size. The case where $\rho=2^{16}$ means that the encrypted character is now represented in 16 bits. Since K=1, this means that an 8-bit ASCII value was now represented in 16-bits. This results in the doubling of the size of each character representation. This change propagates through the entire encrypted file, causing the encrypted file size to be two times as large as the unencrypted file. In the case where $\rho=2^{32}$, each ASCII character is encrypted and represented as 32-bits, which is four times as large as a standard ASCII character representation. In this case, the encrypted file size was four times the size of the unencrypted file. This pattern continued for all variations of ρ. Since K=1, each element in the encrypted list represented one character, so the length of the list remained constant. Search time is proportional to the length of the encrypted list, therefore search times for encrypted files did not vary meaningfully.

Table 3: Strings – Varying ρ with fixed (K, α, and β)										
K	***α***	***β***	***ρ***	**Enc. Time (ms)**	**Dec. Time (ms)**	**Unenc. File Sz. (MB)**	**Enc. File Sz. (MB)**	**Enc. / Unenc. File Sz.**	**Enc. Search Time (ms)**	**Unenc. Search Time (ms)**
1	12302342342342341231	71041	2^8	3567.828	5117.220	1.24	1.24	1.00	608.065	24.161
1	12302342342342341231	71041	2^16	3545.013	5121.104	1.24	2.48	2.00	608.289	25.328
1	12302342342342341231	71041	2^32	3635.781	5069.406	1.24	4.96	4.00	610.958	24.689
1	12302342342342341231	71041	2^64	3534.376	5084.582	1.24	9.91	7.99	612.763	24.496
1	12302342342342341231	71041	2^128	3437.900	5144.918	1.24	19.83	15.99	604.506	24.210

Strings – Varied *K*

Table 4 displays the results for an experiment where the value of *K* was varied. This table shows how varying *K* influences encryption time, decryption time, file size, and search time. *K* is the number of characters grouped in one encrypted element of the encrypted list. As *K* increased, the time to encrypt and decrypt the file decreased. This is because less affine cipher calculations must be performed as more characters are grouped together. From these results, it is shown that it is faster to concatenate the bits of *K* characters and encrypt the block, rather than to encrypt each character individually. For the first 8 trials in the table, ρ was fixed at 2^{64}. This meant that each element in the list was represented by a 64-bit number. With K=1, this meant that one character was represented by a single 64-bit number. With K=2, this meant that two characters were represented in a single 64-bit number. This process of increasing *K* continued until K=8, the maximum *K* value for $\rho=2^{64}$. When K=8, and $\rho=2^{64}$, 8 ASCII characters where represented in 64 bits. This is a full usage of the 64 bits (8 characters * 8-bits), which resulted in an encrypted file size that was equal to the unencrypted file size. As *K* increased (for a fixed ρ), the size of the encrypted file decreased until it approached the size of the unencrypted file. In general, the most spatially optimal configuration is when all the bits in an encrypted element are used to represent

ASCII characters. When all bits in the encrypted element are used to represent ASCII characters, the encrypted file size is the same as the unencrypted file size. It can also be noted how when K=8 and ρ=2^{128}, the encrypted file size increased from the previous trial, this is because with a 128-bit representation, only 64-bits (8*(8-bit)) were utilized to store ASCII values. The search time increased as K increased, this is due to the phenomenon shown in Figure 3, which is that as K increases, there are more possible variations of the encryption of a substring that may exist within the encrypted files. More of these combinations must be searched as K increases, which is computationally intensive. For the cases where K=12 and K=16, the search timed-out after ten seconds of searching. Three trials were also done using ρ =2^{128} and K=8,12,16 to show that the implementation does scale to arbitrary values, the trends of decreasing encryption/decryption time, and search time continue, and that the principle of optimized spatial use still holds.

Table 4: Strings – Varying K and *ρ with fixed (α and β)*

K	***α***	***β***	***ρ***	**Enc. Time (ms)**	**Dec. Time (ms)**	**Unenc. File Sz. (MB)**	**Enc. File Sz. (MB)**	**Enc. / Unenc. File Sz**	**Enc. Search Time (ms)**	**Unenc. Search Time (ms)**
1	12302342342342341231	71041	2^64	3506.866	5088.674	1.24	9.91	7.99	697.752	29.519
2	12302342342342341231	71041	2^64	2343.175	3084.572	1.24	4.96	4.00	480.281	30.617
3	12302342342342341231	71041	2^64	1954.405	2403.911	1.24	3.30	2.66	572.623	30.883
4	12302342342342341231	71041	2^64	1764.176	2047.302	1.24	2.48	2.00	6173.163	30.037
5	12302342342342341231	71041	2^64	1611.689	1902.300	1.24	1.98	1.60	8029.314	31.242
6	12302342342342341231	71041	2^64	1538.749	1763.987	1.24	1.65	1.33	4234.235	31.251
7	12302342342342341231	71041	2^64	1482.355	1656.448	1.24	1.42	1.15	8047.905	31.070
8	12302342342342341231	71041	2^64	1450.028	1540.178	1.24	1.24	1.00	8019.927	30.207
8	12302342342342341231	71041	2^128	1420.536	1578.278	1.24	2.48	2.00	8019.803	30.544
12	12302342342342341231	71041	2^128	1346.238	1410.276	1.24	1.65	1.33	10000 + (timeout)	30.435
16	12302342342342341231	71041	2^128	1307.981	1290.979	1.24	1.24	1.00	10000 + (timeout)	30.325

Integers – Varied number of operations on an encrypted integer

Table 5 shows how as the number of encrypted integer operations influences the performance of our encryption scheme. The number of operations performed did not have a major impact on encryption or decryption times. However, as the number of operations increased, the size of the encrypted file also increased. This is because as encrypted values are repeatedly added or subtracted, the magnitude of the value increases (when subtracting, values can go negative). As these magnitudes increase, more space is needed to store their values. In this case, data size was measured by iterating through the encrypted array and summing the minimum number of bytes it would take to store each element, this includes the counter variable. The encrypted and unencrypted operations time measures how long it took the program to perform the number of operations for that trial.

Table 5: Integers – Varying number of operations with fixed (K, α, β, ρ, and # of encrypted digits)

Op.	*K*	*α*	*β*	*ρ*	Num. of Digits in Unenc. Num.	Num. of Ops.	Enc. Time (ms)	Dec. Time (ms)	Enc. Op. Time (ms)	Unenc. Op. Time (ms)	Enc. / Unenc. Op. Time	Enc. Sz. (B)	Unenc. Sz. (B)	Enc. / Unenc. Sz.
add	1	12302342342342341231	71041	2^32	1	2^1	0.015	0.028	0.015	0.002	7.600	5.00	1.00	5.00
add	1	12302342342342341231	71041	2^32	1	2^2	0.024	0.020	0.026	0.003	10.440	5.00	1.00	5.00
add	1	12302342342342341231	71041	2^32	1	2^8	0.017	0.017	1.178	0.048	24.540	7.00	2.00	3.50
add	1	12302342342342341231	71041	2^32	1	2^16	0.016	0.029	279.892	12.084	23.163	9.00	3.00	3.00
add	1	12302342342342341231	71041	2^32	1	2^20	0.026	0.028	4408.540	190.454	23.148	9.00	3.00	3.00
sub	1	12302342342342341231	71041	2^32	1	2^1	0.026	0.026	0.016	0.002	7.524	5.00	1.00	5.00
sub	1	12302342342342341231	71041	2^32	1	2^2	0.014	0.021	0.023	0.003	9.120	5.00	1.00	5.00
sub	1	12302342342342341231	71041	2^32	1	2^8	0.018	0.019	1.171	0.051	23.047	7.00	2.00	3.50
sub	1	12302342342342341231	71041	2^32	1	2^16	0.015	0.031	279.330	11.803	23.666	9.00	3.00	3.00
sub	1	12302342342342341231	71041	2^32	1	2^20	0.022	0.030	4449.754	195.439	22.768	9.00	3.00	3.00

Integers – Varied ρ

Table 6 shows the varying the ρ value for encrypted numbers with an arbitrary number operations performed on them. This shows how the encrypted integer file sizes behave like the encrypted strings' file sizes with varied ρ values. Since ρ limits the range of output values, increasing ρ increases the output range. This means that the magnitude of output values can be larger, and therefore more space is needed to store each output value.

Table 6: Integers – Varying ρ with fixed (K, α, β, # of ops, and # of encrypted digits)

Op.	K	α	β	ρ	Num. of Digits in Unenc. Num.	Num. of Ops.	Enc. Time (ms)	Dec. Time (ms)	Enc. Op. Time (ms)	Unenc. Op. Time (ms)	Enc. / Unenc. Op. Time	Enc. Sz. (B)	Unenc. Sz. (B)	Enc. / Unenc. Sz.
add	1	12302342342342341231	71041	2^8	1	2^20	0.015	0.029	4280.924	191.507	22.354	6.00	3.00	2.00
add	1	12302342342342341231	71041	2^16	1	2^20	0.025	0.029	4276.283	191.078	22.380	7.00	3.00	2.33
add	1	12302342342342341231	71041	2^32	1	2^20	0.024	0.029	4281.739	193.066	22.178	9.00	3.00	3.00
add	1	12302342342342341231	71041	2^64	1	2^20	0.027	0.030	4275.275	195.507	21.868	13.00	3.00	4.33
sub	1	12302342342342341231	71041	2^8	1	2^20	0.027	0.030	4327.357	193.945	22.312	6.00	3.00	2.00
sub	1	12302342342342341231	71041	2^16	1	2^20	0.027	0.029	4323.665	193.351	22.362	7.00	3.00	2.33
sub	1	12302342342342341231	71041	2^32	1	2^20	0.028	0.030	4302.107	194.888	22.075	9.00	3.00	3.00
sub	1	12302342342342341231	71041	2^64	1	2^20	0.029	0.033	4306.704	192.540	22.368	13.00	3.00	4.33

Integers – Varied number of digits in unencrypted number

Table 7 shows how varying the number of digits in the unencrypted number impacts the scheme's performance. To give an example of the varied testing parameter, the numbers "1", "2", and "3" are all have a number-of-digits-value equal to one. The numbers "11", "24", and "76" all have a number-of-digits-value equal to two. Also, this scheme counts front-padded zeros as the number of digits in a number. So if a user wants to encrypt "1", the number of digits equals one. But if a user wants to encrypt "01", then the number of digits equals two.

The results from this test show how encryption and decryption time increases as the number of digits also increases. Since K=1, the number of digits equates to the number of affine encryptions that occur. This means it takes longer to encrypt the entire number representation. This also means that as the number of digits increases, the length of the encrypted list grows, which increases the encrypted file size.

Table 7: Integers – Varying # of encrypted digits with fixed (K, α, β, # of ops, and ρ)

Op.	*K*	*α*	*β*	*ρ*	Num. of Digits in Unenc. Num.	Num. of Ops.	Enc. Time (ms)	Dec. Time (ms)	Enc. Op. Time (ms)	Unenc. Op. Time (ms)	Enc. / Unenc. Op. Time	Enc. Sz. (B)	Unenc. Sz. (B)	Enc. / Unenc. Sz.
add	1	12302342342342341231	71041	2^32	1	2^20	0.017	0.033	4900.348	204.147	24.004	9.00	3.00	3.00
add	1	12302342342342341231	71041	2^32	2	2^20	0.031	0.037	5889.808	203.979	28.875	15.00	3.00	5.00
add	1	12302342342342341231	71041	2^32	3	2^20	0.036	0.046	6848.432	207.804	32.956	21.00	3.00	7.00
add	1	12302342342342341231	71041	2^32	4	2^20	0.038	0.048	7806.896	204.266	38.219	27.00	3.00	9.00
sub	1	12302342342342341231	71041	2^32	1	2^20	0.030	0.033	4908.887	205.043	23.941	9.00	3.00	3.00
sub	1	12302342342342341231	71041	2^32	2	2^20	0.032	0.037	5931.079	201.218	29.476	15.00	3.00	5.00
sub	1	12302342342342341231	71041	2^32	3	2^20	0.043	0.045	6773.765	201.137	33.677	21.00	3.00	7.00
sub	1	12302342342342341231	71041	2^32	4	2^20	0.040	0.048	7690.135	207.497	37.061	27.00	3.00	9.00

CHAPTER IV

CONCLUSION

In conclusion, the encryption scheme described here is homomorphic with respect to string searches and concatenations, and integer addition and subtraction operations. It is based on the ASCII text encoding and the affine cipher, with moderate improvements made to increase security. The scheme treats string and integer data equally, to avoid leaking information as to whether encrypted objects are of string or an integer types. The scheme does have some drawbacks. Integer operations are limited to K=1, which is the lowest level of security. Additionally, the scheme is susceptible to frequency analysis attacks. Due to the deterministic nature of this encryption scheme, an attacker could analyze the frequency in which certain encrypted values appear within an encrypted file. This frequency information could then be used to infer the unencrypted value of the encrypted value.

Future Development

The string search feature is very inefficient for type K>1 encrypted strings. This method could be improved by searching for known and fixed possible substrings of the search query. If a possible match is found, instead of brute-forcing star-combinations, the encrypted elements neighboring the matched query could be sent to the user to decrypt. Once decrypted, the user could verify if the contents and compare to what they had searched. If the contents contain the searched characters in the sequence that they were expecting, then the search would be successful. If the contents did not match what was expected, then a different *possible-query* would be searched.

This work could also be modified to support non-traditional encoding schemes and bit lengths, which could help alleviate its deficiencies in arithmetic, allowing for variable K support with integers. Work could also be done to derive a more secure affine cipher, possibly using the quadratic formula in a way that it could support math operations and string searches. Additionally, work could be done to support multiplication and division in the encrypted space.

REFERENCES

[1] Florentine, Sharon. "Cloud Adoption Soars, but Integration Challenges Remain." *CIO*. CIO, 05 Jan. 2016. Web. 09 Apr. 2017.

[2] Gentry, Craig. "A FULLY HOMOMORPHIC ENCRYPTION SCHEME." Thesis. Stanford University, 2009. Web.

[3] Gentry, Craig. "Computing Arbitrary Functions of Encrypted Data." *Communications of the ACM* 53.3 (2010): 97-105. *ACM*. Web. 23 Dec. 2016.

[4] Gentry, Craig, and Shai Halevi. "Implementing Gentry's Fully-Homomorphic Encryption Scheme." (2011): 1-29. *International Association for Cryptologic Research*. Web. 24 Aug. 2016.

[5] Gilad-Bachrach, Ran, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. "CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy." *Microsoft Research* (2016): n. pag. 8 Feb. 2016. Web. 9 Apr. 2017.

[6] Kraft, James S., and Lawrence C. Washington. "Cryptographic Applications." *An Introduction to Number Theory with Cryptography*. Boca Raton: CRC, 2014. 170-75. Print.

[7] Rivest, Ronald L., Len Adleman, and Michael L. Dertouzos. "On Data Banks and Privacy Homomorphisms." Ed. Richard A. DeMillo, David P. Dobkin, Anita K. Jones, and Richard J. Lipton. *Foundations of Secure Computation* (1978): 165-79. Academic Press. Web. 21 Jan. 2017.

[8] Rivest, Ronald L., Adi Shamir, and Leonard M. Adleman. "A Method for Obtaining Digital Signatures and Public-key Cryptosystems." *Communications of the ACM* 21.2 (1978): 120-26. *ACM Digital Library*. Web. 21 Jan. 2017.

[9] Schneier, Bruce. "Homomorphic Encryption Breakthrough." *Schneier on Security*. N.p., 9 July 2009. Web. 09 Apr. 2017.

[10] Teske-Wilson, Edlyn. *Homomorphic Cryptosystems*. Ottawa: University of Waterloo, 27 June 2011. PDF.

[11] Thomson, Iain. "Microsoft Researchers Smash Homomorphic Encryption Speed Barrier." *The Register*. N.p., 9 Feb. 2016. Web. 09 Apr. 2017.

[12] Wu, Yongfeng. "Improvement Research Based on Affine Encryption Algorithm." *2015 14th International Symposium on Distributed Computing and Applications for Business Engineering and Science (DCABES)* (2015): n. pag. *IEEE Xplore*. Web. 22 Dec. 2016.