6.857 Computer and Network Security Final Project

Cocoon: Encrypted Substring Search

Eric Chen, Ismael Gomez, Brian Saavedra, Jonatan Yucra

Abstract: Homomorphic encryption schemes allow for computations to be done on encrypted data without the need to decrypt the file first. There is an assortment of real world applications in which such schemes would find themselves useful. However, current implementations of fully homomorphic systems add an impractical amount of computation for today's hardware to handle. This limitation can be side-stepped by instead using partially-homomorphic schemes, which limit the kinds of computations that can be done on the encrypted data in exchange for vast performance improvements. Our goal for this project is to develop such a system, particularly one that is able to do substring search. The system we have developed, called Cocoon[1], uses suffix trees to perform queries returning all the indices in the original text file in which the queried sub-string is present. Prior to being uploaded, text files are encrypted through a Python script which we provide to the user. This allows for a communication protocol between the server and the client in which the server is kept in the dark regarding the true contents of uploaded encrypted file.

Keywords: homomorphic encryption, substring search, suffix trees

[1]As of May 11, 2016, the application is deployed at https://cocoon-857.herokuapp.com and sourced at https://github.com/izzygomez/cocoon

Contents

1 Introduction

Standard symmetric-key encryption schemes allow for anyone who has the secret key to decrypt the entirety of any message that was encrypted with that key. However, the needs of today can often make demands that our traditional schemes can't meet on their own. For example, it has become exceedingly popular to do storage and computation on the cloud for a variety of reasons, whether it be ease of access, storage needs, or privacy. Thus we wish to be able to not only encrypt our files, but we want to also pass this encrypted file to third parties and have them do computations on this file without gaining any information.

1.1 Background

Homomorphic encryption schemes allow a balance between performance and information-hiding. In such a cryptosystem, computations can be done on encrypted data without the need to decrypt the dataset in the process. Therefore, it becomes safe to use third parties to do computational work on one's behalf without gaining significant insight on the decrypted contents of the file in the process.

Homomorphic schemes can have a variety of different real world applications. For instance, one could be used to implement a secure and private electronic voting system. Such a system would need to have votes be encrypted to protect the voter's identity. The encrypted votes can then be tallied up by a third-party server without the server ever knowing who voted how.

Another real world application would be substring search. Consider the case involving medical researchers working with data containing people's genomes. A very natural use of substring search would be to query the genomes for the existence of a certain DNA sequence that if present could mean that the patient is predisposed to an illness. There are two main criteria that need to be satisfied in this scenario. First, the data needs to be encrypted to protect the subjects' privacy. Second, the researchers need to be able to query the data efficiently.

Homomorphic encryption schemes manage to solve the first requirement. In the given case involving a medical lab the researchers would be able to store their data on AWS and perform the required computations without Amazon ever getting a glimpse at the decrypted version of their subject's genomic data. However, fully homomorphic schemes add impractical amounts of required computation. If we allow for homomorphic encryption schemes to only support a limited number of operations the extra computation required becomes much more manageable. Therefore, the second criterion put forth in the case of the medical lab can be solved by using a partially homomorphic scheme that only supports substring-search operations.

1.2 Problem Statement

Our goal is to implement a partially homomorphic encryption scheme that is capable of encrypted substring search. A user should be able to use Cocoon to upload their encrypted files and perform queries which return the indices of all occurrences of a specified substring. The following conditions should be met throughout this process:

1. Cocoon should return correct indices corresponding to the unencrypted original text file. If a substring appears more than once, all occurrences should be reported to the user.
2. The server should not be able to gain information about the original unencrypted contents of the files users upload.

2 Related Works

Much related work has been done in the homomorphic encryption arena. For instance, as part of his 2009 thesis dissertation Craig Gentry proposed the first fully homomorphic encryption scheme [3]. While this marked an important step in the field of homomorphic encryption, it also showed that there is much room to improve. Even though Gentry's implementation using ideal lattices would be functional, Gentry himself has acknowledged that it would increase the computation time to do things such as a Google search by a factor of one trillion.

Other research focuses on exploring partially homomorphic encryption schemes for their potential to be more performant and practical for real-world applications. Curtmola et al. [4] help define general searchability in symmetric encryption schemes. Such definitions were later used by Cash et al. [5] to devise highly-scalable and efficient implementations of a searchable symmetric encryption protocol. More specifically, the protocols support conjunctive boolean queries for matching documents. These efforts are similar to the goals we set out to achieve by building Cocoon, where we emphasized real-world practicality and usability. The difference is that Cocoon supports substring search queries, rather than boolean queries.

Chase and Shen [2] propose a partially homomorphic encryption scheme which enables substring-search queries on the encrypted data. Their scheme involves querying an encrypted dictionary which draws its structure from a suffix tree, a data structure used for fast substring lookups. The communication protocol between the client and server involves three rounds of communication. This is the encryption scheme that is implemented in Cocoon.

3 Algorithm

In the next parts, we will describe the methods and theory behind Cocoon, which is based largely on the queryable encryption scheme defined in Chase and Shen's paper [2].The approach for dealing with searching for substrings in an encrypted text draws inspiration from the approach that would be used for unencrypted text.

3.1 Suffix Tree Construction

Chase and Shen present encrypted suffix trees as an efficient way to find substrings in a large text. A suffix tree is just a way of storing all the different suffixes of a text in one data structure, that leads to efficient lookup times. In a suffix tree, the concatenation of the edge labels of any path from the root to a leaf yields a suffix of the original string. To construct the suffix tree, we use Ukonnen's algorithm [7], an efficient algorithm for constructing a suffix tree out of a text. Once the tree is created, we use its structure to find substrings of the text the tree is encoding. To analyze the structure of the suffix tree, and understand how it helps to find substrings of a text, Figure 1 will be used extensively. Figure 1 will be useful in understanding the examples in the following sections. The following sections are structured such that initially, a simple way to perform the operations required is given, and this is built up to describe a system which is secure, correct, and efficient. Define t to be the substring that is being searched for, and s to be the string that is being searched in the following sections.

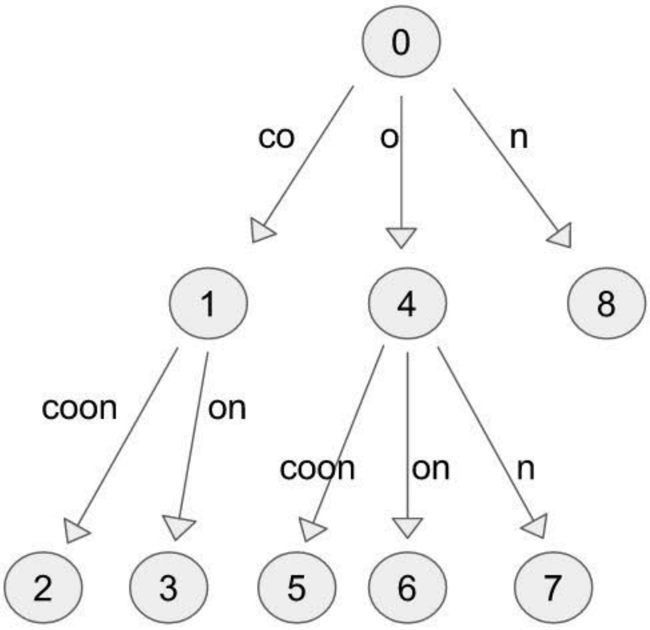


Figure 1. The suffix tree for the word cocoon

3.2 Suffix Tree Basic Search

Consider querying for $t =$ "oon", and $s =$ "cocoon". Define $path(u)$ for a node u to be the concatenation of all the edge labels starting at the root down to the node u. To perform this search, one would first start at the root of the tree, node 0, and then walk down. To decide which branch to walk down, the path of every node that is a child of node 0 has to be looked at. Since $t =$ "oon", we search down towards the node who's path is a prefix of t. Using this approach leads to node 4. Once at node 4, the paths of all the children of node 4 have to be looked at, and the one whose path is a prefix of t is chosen. At node 6 and all other nodes u is stored a value, called ind, which is the index at which $path(u)$ starts in the text. If none of the paths match t, this algorithm returns that it did not find t within s.

3.3 Problem with Using Path

The previous search worked fine, but in some cases, the algorithm described above would fail. Specifically, this would be the case then t was not exactly a $path(u)$ of some node u, but was still a substring of s. Consider the case where a user wants to know at which indices the substring "coco" occurs, so t is defined to be $t =$ "coco". If a similar process to the process described above is followed, the substring will not be found, since the only paths that would be checked against "coco" would be "co", and then "cocoon", none of which match the substring that was being looked for. However, it can be noticed that "coco" is a substring of the word "cocoon" and that "co" is a substring of "coco". This shows that the simple algorithm that is currently in place does not work, and it will have to be modified in order to find substrings that do not exactly match the path of some node in the suffix tree.

3.4 Solution: Initpath

To solve this problem, the algorithm will need to store something Chase and Shen's paper [2] calls $initpath(u)$, where u is a node of the suffix tree. Define $initpath(u)$ as the concatenation of all the edge labels up to the parent of node u, concatenated with the first letter of the edge label from the parent of node u to node u. For example, in the tree, the initpath of node 2 would be "coc." The reason why this is done is so that when a word like "coco" is queried, it will be found by looking for prefixes of t. Now that the initpath of a node is defined, it will be used to solve the previously stated problem. In similar fashion to before, the tree is traversed downwards, but this time comparing the initpaths of the nodes, rather than the paths of the nodes. The algorithm concludes that t is a substring of s when t is equal to the initpath of a node, but if the initpath of a node becomes a substring of t, then another process has to be followed, since the algorithm previously would have said that t was not in s. The process to follow is to take t, and remove the last character from it. Then, the same search in the suffix tree is performed for this modified t. Taking the last character away from t leaves us with "coc", which is then searched for in the suffix tree. This leads to node 2, since its initpath is "coc". (Note: If an initpath had not been found, the last character of the modified t would be removed again, and be searched for. This process would be repeated until the modified t was equal to an empty string, at which point the algorithm would declare that t is not a substring of s). After reaching this step, the algorithm is not necessarily certain that t is a substring of s. An example of this is if $t =$ "coca". In this case, the algorithm needs to find a way to verify that t is in s, even though a prefix of t did match the initpath of some node u in the suffix tree. To fix this, once the initpath is found, the server will send back a string starting at $ind(u)$ and ending at $ind(u) + length(t)$. The client will then check these returned letters, and check if these letters match t. If they do, then t is a substring of s, otherwise it is not. In the case of $t =$ "coca", the algorithm would return false, since the string returned by the server would be "coco", and not "coca", like in t.

3.5 Returning All Occurrences

The previous fix does not fix all of the problems with this algorithm. When a search to see if t is a substring of s is performed, the algorithm above only returns the first index at which t occurs in s, but in reality, t might occur many times in s, and a user might want to know all the times that this happens. The algorithm has to be modified accordingly, to return all indices at which t occurs in s.

3.6 Solution: Leaf Array

To do this, a leaf array has to be made, called L. L is an array of size n, where the i^{th} entry represents the index at which the path ending at the i^{th} leaf, counting from left to right, begins. Then, to each node u, two attributes are added. The first is $left(u)$, which gives the index of the leftmost leaf in the subtree rooted at node u. Furthermore, $num(u)$ is also added, which gives the number of leaves in the subtree rooted at node u. By doing this, looking up all the occurrences of a substring t in a string s is trivial. A look-up process like the one described above is performed, and the node u who's initpath matches the longest prefix of t is found. Then, to find all the indices, the algorithm just returns the $num(u)$ elements that start at index $left(u)$ in L.

3.7 The Need for a Dictionary and Encryption

Up to this point, it has been described how to explore and retrieve information regarding substrings from a suffix tree, but in reality, this can be inconvenient and not efficient to do. Instead of having to traverse part of the tree every time a user enquires about a particular string, what can be done instead is to store the information contained in the tree as a dictionary. By doing this, look-up is a lot more efficient, and no information that was in the tree is lost. Like the suffix tree, this dictionary would only have to be made once. With the improvements that have been made so far, the information the dictionary will contain needs to be defined. The dictionary, D, will contain as $(key, value)$ pairs $(Enc(initpath(u)), Enc((ind(u), left(u), num(u))))$ for all nodes u in the suffix tree. This changes the way in which the algorithm works slightly, because instead of traversing the tree to find the appropriate initpaths, the initpath can just be looked up by making a query to the table, which is more time efficient. Up until now, nothing in the suffix tree had been encrypted, but that will be changed in order to make this scheme secure. What will be going into the suffix tree will not be a text, but an encrypted version of the text. This way, the server will not have knowledge of what it is storing. In a similar way, the queries a client makes, t, are also encrypted. The server will also store C, a ciphertext of the string s.

3.8 Additional Security Measures

To protect against an adversarial server, several more enhancements were made to the way in which data was represented/transferred in this algorithm. The first of these enhancements was to add dummy nodes in the suffix tree, so that the number of children of a node remains unknown to the server. Another enhancement is permuting the ciphertext C and the leaf array L, so that the server gains very little from observing many queries going through. Only the client has access to the secret key, so only the client can know the full permutation. Finally, one of the last enhancements was to protect against a malicious server, which was done by adding authentication to messages sent back from the server, to make sure those messages were something that could have reasonably been expected by the client. Our implementation uses CBC-MAC for this purpose.

4 Implementation

Cocoon has been implemented as a web application [1]. A variety of libraries, frameworks, and environments were used to construct Cocoon. We cover exactly what and how we used in the following sections.

4.1 Web Stack and Dependencies

We use Node.js to create the server, Express.js as the framework, and MongoDB for data storage. On top of these web technologies we also made use of a Python script to take care of some of our client-side tasks.

4.2 Suffix Trees

We chose to construct and encrypt our suffix tree and related data structures using an external Python script. This choice was made because the Javascript libraries that we found capable of constructing suffix trees were very slow and often times caused the browser to time out. We decided on using Thomas Mailund's [10] Python wrapper for his suffix tree implementation written in C. The main reason why we chose this implementation is because it provides a substantial performance improvement over alternatives. Straight out of its packaging however this library did not meet all the requirements that we were looking to fulfill. Particularly we were looking for an implementation to support finding multiple occurrences. This proved to be a challenge, and in the end we decided to write our own code to expand upon Mailund's module by implementing the necessary changes to his module ourselves (i.e. creating a leaf array along with a dictionary, changing the dictionary values to tuples instead of numbers, etc.)

4.3 Symmetric Key Encryption and Authentication

In order to be able to keep the true contents of the client's text file hidden from the server we needed to use a symmetric key encryption scheme. For our implementation, we used AES in CBC mode as the symmetric encryption scheme, as it had a variety of implementations to choose from both in Python and Javascript libraries. For authentication, we included a tag at the end of every AES-encrypted piece of data that is the CBC-MAC. PyCrypto [9] was used to implement AES and CBC-MAC for the original encryption of the suffix tree, and we used the Stanford Javascript Crypto Library [8] to do client-side browser encryption and decryption.

4.4 Pseudorandom Function

Chase and Shen [2] also call for a pseudorandom function to be used to encrypt the keys in the dictionary. This function takes in a key and plaintext as input to produce a ciphertext.

For our impplementation we chose to use SHA256, as it is a secure noninvertible pseudorandom function itself. To include support for a private key as an additional input, we make our final pseudorandom function to be the hash of the concatenated hashes of the plaintext and key.

$$f_{\text{key}}(\text{PTXT}) = \text{SHA256}(\text{SHA256}(\text{PTXT}) \cdot \text{SHA256}(\text{key}))$$

We used the implementation of SHA256 provided by [8] for client-side browser hashing, and the Python's hashlib library for SHA256 during offline encryption.

4.5 Key Generation

The required random keys are created using Paul Wolf's [11] StringGenerator Python library.

4.6 Pseudorandom Permutation Function

The authors of [2] make use of a pseudorandom invertible permutation family of functions P to hide string indices and leaf positions, but do not construct or define a suitable P. Thus, we were faced with the challenge of implementing a P with the property that it should be computationally infeasible for any probabilistic polynomial time adversary to distinguish a function chosen randomly from P from a uniformly random function. We implemented the algorithm described in [6] to construct pseudorandom permutation functions given a pseudorandom function.

We decided to construct an instance of P by using the SHA-256 hash function as a pseudorandom function, and then proceeded to define the `permute` and `unpermute` functions as defined by [6]. Specifically, we are able to permute the inclusive range of values from 0 to $2^\gamma - 1$ (for some $\gamma \in \mathbb{Z}^+$) amongst itself. Whether $\gamma = 2n$ or whether $\gamma = 2n + 1$ defines the `permute` and `unpermute` functions as follows:

`permute` functions

$$p_k^{2n}(L \cdot R) = R \cdot [L \oplus f_k^n(R)]$$
$$p_{k'}^{2n+1}(L' \cdot R') = R' \cdot [L' \oplus \text{ first } n \text{ bits of } f_k^{n+1}(R')]$$

`unpermute` functions

$$u_k^{2n}(\alpha \cdot \beta) = [\beta \oplus f_k^n(\alpha)] \cdot \alpha$$
$$u_{k'}^{2n+1}(\alpha' \cdot \beta') = [\beta' \oplus \text{ first } n \text{ bits of } f_{k'}^n(\alpha')] \cdot \alpha$$

where $f_k^n(x) = \text{SHA256}(\text{SHA256}(x) \cdot \text{SHA256}(k))$, $|x| = |f_k^n| = n$, $(\cdot)$ is the concatenation operator, $|L| = |R| = |L'| = n$, and $|R'| = n + 1$. We then define $h = p \circ p \circ p$ as a pseudorandom permutation function, and thus $g = h^{-1} = u \circ u \circ u$. The functions (h, g) are implemented twice: once in the local Python script to permute the C and L arrays under the keys K_C and K_L, respectively; and another time in the client-side Javascript code to query the server for the appropriate permuted indices. The Python implementation made use of built-in libraries, while the Javascript implementation required using the Stanford Javascript Crypto Library [8].

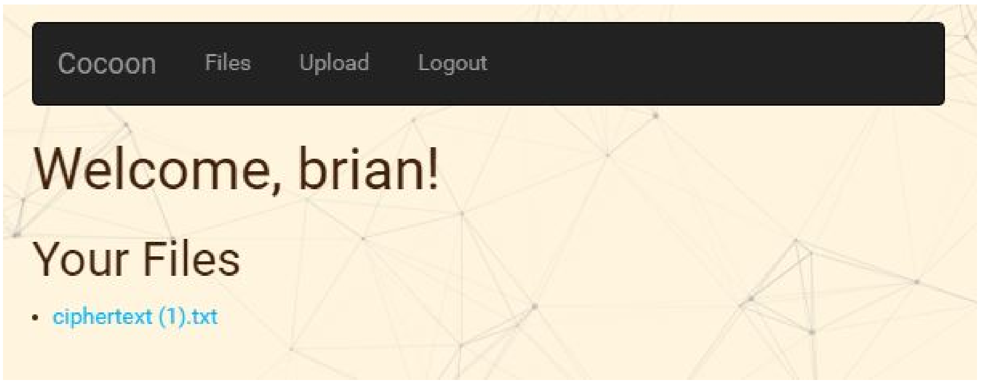
The implementations for the Python pseudorandom permutation functions are found in `crypto/crypto.py`, and the Javascript implementations are found in `public/js/crypto.js`.

4.7 Communication Protocol

There are up to three rounds of communication between the client and server in a single request. The communication protocol is implemented in `public/js/file.js` for the client and `routes/file.js` for the server.

5 Walk-through

Registered users will be presented with a simple interface upon log-in, which gives them the option to either upload a file or query one of their previously uploaded files.



Welcome, brian!

Your Files

- ciphertext (1).txt

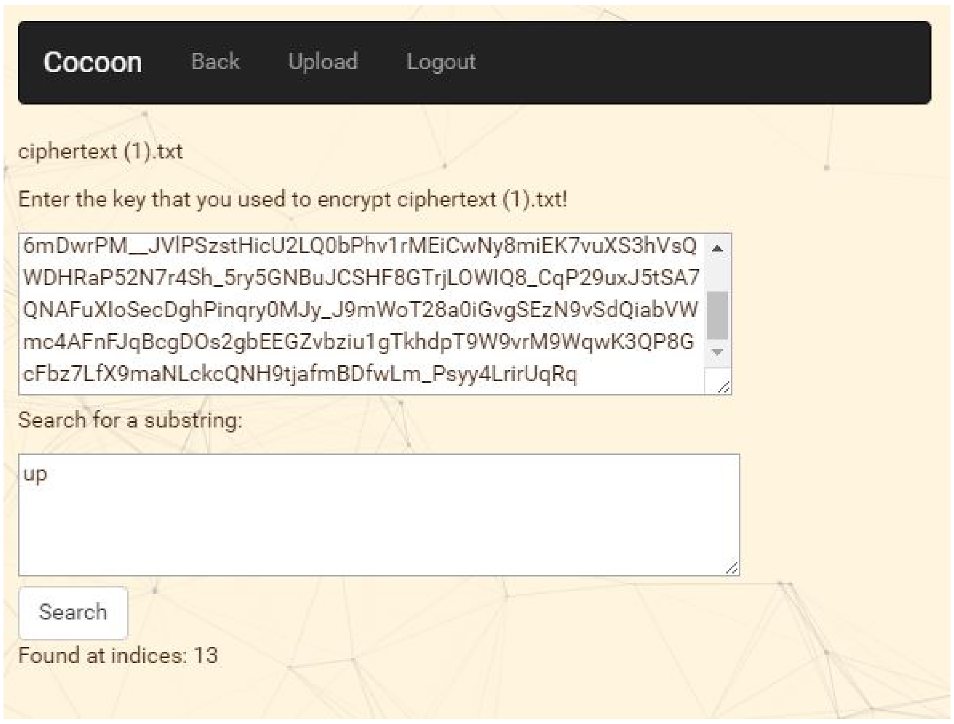
Figure 2. To choose a file to query, click on its hyperlink. To upload a new file, click upload.

5.1 Uploading a File

Before a client can upload their text file, they must first encrypt it locally. We provide users with an encryption Python script, **encrypt.py**, through our project repository that returns both an encrypted data structure for their file and also the key needed to make future queries to this encrypted file. The reason why we chose to use an external Python script as opposed to having encryption take place client-side in the browser is discussed in section 4.2. Before the user is able to run the Python encryption scheme successfully they must first install Python, as well as install the three Python libraries which are mentioned in the repository's README file. Instructions on how to do so can be found there. It is important that at the end of this encryption step the user records the key that has been randomly generated for their file somehow, as they will not be able to recover this later and it will be needed for queries.

5.2 Querying A File

To make a query, users first have to select one of their previously uploaded files. Following this step, they must input the key associated with that file and also the substring they wish to query for. If they don't input the correct key, nothing of value will be returned back. With the correct key in use, the user will see the indices of all occurrences of the substring in the user's original text file.



ciphertext (1).txt

Enter the key that you used to encrypt ciphertext (1).txt!

```
6mDwrPM__JVlPSzstHicU2LQ0bPhv1rMEiCwNy8miEK7vuXS3hVsQ
WDHRaP52N7r4Sh_5ry5GNBuJCSHF8GTrjLOWIQ8_CqP29uxJ5tSA7
QNAFuXIoSecDghPinqry0MJy_J9mWoT28a0iGvgSEzN9vSdQiabVW
mc4AFnFJqBcgDOs2gbEEGZvbziu1gTkhdpT9W9vrM9WqwK3QP8G
cFbz7LfX9maNLckcQNH9tjafmBDfwLm_Psyy4LrirUqRq
```

Search for a substring:

```
up
```

Found at indices: 13

Figure 3. The original text file was "hello what's up dog". After sending the query for "up" with the correct key, the user is told the indices in which "up" is found.

6 Analysis

Having fully described the encryption scheme and Cocoon's implementation, we now analyze security and performance characteristics of Cocoon. Adversarial behavior is first fully defined. Afterwards, we explore the limitations of Cocoon's current implementation, and propose future improvements.

6.1 Attack Model

For the purposes of exploring security, we assume malicious adversaries are actively trying to learn information about the stored text and the queries. Adversaries do this by analyzing any leaked information from the stored text, its associated data structures, and the query messages. Thus, in our implementation, the Cocoon server is a potential malicious adversary. By virtue of being a web application, however, Cocoon also has other potential adversaries. This includes users of Cocoon that attempt to gain access to the account of other users.

6.2 Security

As explored by Chase and Shen [2], the proposed substring-searchable encryption scheme that we implemented satisfies malicious $(\mathcal{L}_1, \mathcal{L}_2)$-CQA2 security. This security definition means that a malicious adversary, here representing the server storing encrypted user data, will be unable to deduce any useful information beyond already leaked information from the ciphertext and the communication transcript. This is despite the adversary given the ability to choose queries (CQA stands for "chosen query attack") and the pair of leakage functions, $(\mathcal{L}_1, \mathcal{L}_2)$. Therefore, the security of Cocoon's underlying communication protocol and storage mechanism is theoretically sound (formal proof found in [2]).

However, Cocoon is an actual implementation, thus making it susceptible to standard software engineering pitfalls. This means that the security of the system depends on the implementation being robust and bug-free. While we hope this is mostly the case, we acknowledge the possibility of some security vulnerabilities. We are nevertheless confident that in case of such security breaches, the contents of user text files and associated keys remain secret. This claim is justified by the fact that encryption and key generation is done locally on a user's computer with open-source software, so it is infeasible for an adversary to access either the original text or the generated keys unless a user's computer is compromised. Our contention is that such a situation is indicative of larger security issues that our system does not claim to address.

6.3 Performance

Ukkonen's algorithm gives an asymptotic suffix tree construction time of $O(n \log n)$ time in the general case, but for constant-size alphabets, it runs in $O(n)$ time, as is the case. Encrypting involves making linear passes through all the elements of the data structure and either performing enryption or permutation operations. Thus, if we let λ be the security parameter, then the overall construction time is $O(\lambda n)$.

We give the encryption times of our implementation (`encrypt.py`) for a few text files below.

Plaintext	Original Size	Enc. Time	Ciphertext Size (MB)
A Lover's Complaint	16 KB	1.848 s	7.8 MB
Romeo and Juliet	136 KB	27.004 s	84 MB
Metamorphosis	140 KB	26.946 s	85 MB

These run times are for encryption excluding the dummy children values in the suffix tree. Adding in dummy children values increases the run time and space complexity a great deal. For instance, encrypting A Lover's Complaint (16 KB) requires 29.225 seconds and results in a ciphertext of size 132 MB.

6.4 Improvements and Limitations

A big limitation of our implementation is that encryption, in general, runs very slowly. For the suffix tree construction step, we use a Python wrapper around C functions, so this step of the algorithm runs very quickly. However, for the encryption itself as well as construction of the C and L arrays, we use purely Python and its associated libraries. The run times of the encryption phase can be greatly improved by switching over to C or C++.

In addition, objects larger than 16 MB cannot be stored in MongoDB without using GridFS. Currently, GridFS is not integrated into Cocoon, so the user will be unable to upload large files.

Like Chase and Shen explain, there is a degree of data leakage that the server can obtain from our queries. For example, the server can tell the length of the substrings the client is trying to query. In addition, the server is able to deduce the ordering of permuted indices of the ciphertext and leaf array indices because of their ordering in the client's query. Future improvements for this project could include making changes to our protocol or data structures to decrease these kinds of information leakage.

The original encryption step could also be improved, as in its current state it is not entirely user friendly. Streamlining the installation of the Python dependencies would improve the user experience.

7 Conclusion

At the beginning of the project we set out to meet the two goals mentioned in the introduction. The crypto scheme we built, Cocoon, meets our expectations in regards to these two goals. User queries are responded with correct indices for all ocurrences of the specified susbtring. Through our communication protocol between the client and server, we also managed to keep the server from accessing unencrypted versions of the user's files or from getting access to the keys needed to decrypt them.

We would like to acknowledge Ronald Rivest and the rest of the staff for continued support and advisal of the project throughout its development.

References

[1] *Cocoon-857*, Heroku Web-app Deployment. Found live at: https://cocoon-857.herokuapp.com/. Source code at: https://github.com/izzygomez/cocoon

[2] Chase, M., & Shen, E. (2015). *Substring-Searchable Symmetric Encryption.* Retrieved from https://eprint.iacr.org/2014/638.pdf.

[3] Gentry, C. (2009). *A Fully Homomorphic Encryption Scheme* (Doctoral dissertation). Stanford University. Retrieved from https://crypto.stanford.edu/craig/craig-thesis.pdf

[4] Curtmola, R., Garay, J., Kamara, S., & Ostrovsky, R. (2006). *Searchable Symmetric Encryption: Improved Definitions and Efficient Constructions.* ACM Conference on Computer and Communications Security. Retrieved from https://eprint.iacr.org/2006/210.pdf

[5] Cash, D., Jarecki, S., & Jutla, C. (2013). *Highly-Scalable Searchable Symmetric Encryption with Support for Boolean Queries.* Crypto. Retrieved from https://eprint.iacr.org/2013/169.pdf.

[6] Luby, M., & Rackoff, C. (1988). *How to construct Pseudorandom Permutations from Pseudorandom Functions.* Society for Industrial and Applied Mathematics, 17(2), 373-386. Retrieved from http://epubs.siam.org/doi/pdf/10.1137/0217022

[7] Ukkonen, E. (1995). *On-line construction of suffix trees.* Algorithmica, 14, 249-260. Retrieved from https://www.cs.helsinki.fi/u/ukkonen/SuffixT1withFigs.pdf.

[8] Stanford Javascript Crypto Library. https://crypto.stanford.edu/sjcl/

[9] PyCrypto - The Python Cryptography Toolkit. https://www.dlitz.net/software/pycrypto/

[10] Thomas Mailund's Suffix Tree Library. http://www.daimi.au.dk/ mailund/suffix_tree.html

[11] Paul Wolf's String Generator Library. https://pypi.python.org/pypi/StringGenerator