Searchable encryption

Artūrs Bačkurs
backurs@mit.edu

Daniel Grier
grierd@mit.edu

Adam Sealfon
asealfon@mit.edu

Po-An Tsai
poantsai@mit.edu@mit.edu

1 Introduction

We built a system supporting encrypted substring search of a text document. The user can encrypt and upload files to a server. The client can also query the server to ask whether an encrypted file contains a particular string of characters. The query, like the file, is not sent in plaintext but is encrypted or hashed. The server should be able to respond correctly to queries without learning the contents of the file or the queries.

We designed and implemented two novel schemes for the problem of encrypted substring search. Both schemes require a copy of the file encrypted with a standard block cipher. The first scheme relies on an encrypted suffix array. In the second scheme, the server stores cryptographic hashes of all repeated substrings which occur in the file, as well as all substrings which occur once but would occur multiple times if the last character were deleted. It also stores the index in the file of each of these uniquely occurring substrings occurs. The encrypted suffix array scheme is simpler and has better memory usage and faster preprocessing time. The hashing scheme has better query time and security properties. Its query time can be made to be independent of the size of the file, assuming constant-time hash table lookups. The space usage on worst-case inputs is poor, but the space usage is reasonable on input files having the properties of English text or of randomly generated strings. We assume the following threat model.

Threat model. We assume that the server is honest but curious. That is, the server will follow the protocol and will respond appropriately to all user queries. However, the server will additionally seek to learn whatever it can about the client's data. The server does not have the capability to issue queries, and has no information about the strings being queried.

Previous work. There has been a sequence of recent papers on the problem of keyword search of encrypted data. In this setting, the file is assumed to consist of a sequence of words, e.g. whitespace-separated words in the file or any other atomic units the file can be broken up into. The server must be able to determine which encrypted files contain a particular queried word, which is also encrypted. Song, Wagner and Perrig [11] published the first solution to encrypted keyword search. Their scheme has provable security properties, linear-time encryption and search, and little space overhead. The CryptDB system [8] implements this encryption scheme and uses it to support keyword-based predicates in SQL queries. Several subsequent papers provide additional guarantees, alternate proposals, and improvements. For instance, Chang and Mitzenmacher [2] present an alternate scheme and prove that no information is leaked from search in addition to the set of files which contain the search keyword in common. Boneh *et al.* [1] present a solution in the public key setting. Kamara, Papamanthou and Roeder [3] provide a solution with a number of desirable properties, including sublinear search, strong security guarantees, and efficient upload and download of files. Popa and Zeldovich [9] present a practical encryption scheme for keyword search of data which is encrypted using different keys.

The techniques used in the papers mentioned above heavily rely on the fact that search is over *keywords*, which roughly correspond to the words that appear in the document. Building on this insight, it is possible to encrypt each unique keyword into a list of all keywords. Thus, to search for a keyword, the user can

simply send the encrypted keyword to the server, which is verified in linear time against the list. Another widely used technique is to use a Bloom filter to store the keywords. The idea of a Bloom filter is to use several hash functions to hash the (encrypted) keyword into an array. The filter starts with all entries 0. The locations where the keyword is hashed are set to to 1. To verify the presence of a query in the text, the server hashes the (encrypted) query using the predetermined hash functions. If all locations are 1, then the query is in the document with high probability. Otherwise, it is not in the document. By using Bloom filters, the cost of a search is approximately constant.

What makes these techniques good for time and space is the fact that there are $O(n)$ keywords in a given document. Unfortunately, the number of substrings in a document is $O(n^2)$, so directly applying the above techniques to substring search yields undesirable bounds on space and time. To our knowledge, there are no published proposals for practical substring search. We supply two schemes to circumvent this problem.

2 Hashing scheme

When the client uploads a file, the application on the client side prepares the following data necessary for hashing and encryption. It produces salt for hashing substrings. It produces an AES key and initialization vector for encryption of the file.

Now the client application hashes all substrings of the file that satisfy one of the following two conditions.

- The substring appears at least twice in the file; or
- The substring appears once in the file, but, if we remove the last letter of the substring, the resulting substring appears at least twice in the file.

Now the client applies the SHA-1 hash function to all these substrings and sends all hashes to the server with the following twist. For hashes of the second type, we store also the index of the unique position in the file where the substring appears. The client hashes a substring by first concatenating it with a salt value. Before sending the hashes to the server, the client dumps Python data structure containing all hashes and indices to a file and applies gzip compression on the resulting file to reduce the size of the file that needs to be sent, and then using base-64 encoding to produce a representation of the data which is safe to send over HTTP. The application of compression allowed us to reduce the time needed to send hashes by quite a large factor. For example, if we tried to send the corresponding data structure of Moby Dick without compression, it required 50 seconds. Using compression, the time needed to upload the data structure across the network was reduced to 25 seconds. SHA-1 hash function outputs 40 digits. To reduce the file size, we use prefix of the hash discarding remainder. The length of the suffix that we use depends on the file size that we want to encrypt. For example, when encrypting Moby Dick, we use only first 8 digits of the hash value.

We use a C implementation of the longest common prefix array [6, 4] to find all substrings with the stated properties. We compile it into dynamically linkable library and call it from Python to invoke necessary functionality. (The rest of our system is implemented in Python.)

The client also sends the file encrypted with AES in cipher feedback mode. We encrypt the file in the following way. We split the file in blocks of size 100. We encrypt every block with AES in cipher feedback mode. To achieve that, for different blocks, the resulting ciphertext is different, we compute an initialization vector for every block by applying the MD5 hash function on the concatenation of the block ID and the initialization vector of the file. We use the block ID when computing the initialization vector so that blocks with the same contents still have different ciphertexts.

To determine whether a query string appears in the file, we do binary search to find the longest prefix of the query that appears in the file. We first determine whether the hash of the exact query string appears in the list of hashes. If not, we begin the binary search. For each step of the binary search, we hash the current prefix of the query and send it to the server. The server searches over the list of hashes. The server returns whether the hash appears among the hashes of the first kind or the second kind, or does not appear at all. If the hash does not appear, the client restricts its search to the left half of the range. If the hash is of the first kind (i.e. the substring appears more than once in the file), the client restricts its search to the

right half of the range. Within $\log k$ steps for a query of length k, we can determine the longest prefix of the query which appears in the file. If this is shorter than the full query and corresponds to a hash of the first kind, then the query string does not appear in the file. If it corresponds to a hash of the second kind, the server also returns the position in the file and the client requests the needed blocks of the encrypted file at this index. The client decrypts these blocks and checks whether the query matches the decrypted substring at this index. The number of requests to the server is logarithmic in the length of the query, and the search and download is linear in the length of the query. Thus, for large enough files, our scheme is substantially faster than the trivial system where the client downloads the encrypted file from the server and searches for substrings locally. Running times are shown in Figure 1.

```
alice@cryptZoobar>>login alice 123 456
alice@cryptZoobar>>enc_upload c.txt c.txt
done: suffix array and lcp
done: make hashes
done:hash upload
done:enc block
done:enc_block upload
spent:3.16004300117
alice@cryptZoobar>>enc_search "House of Representative" c.txt
Found
spent:0.374740839005
alice@cryptZoobar>>enc_search "use of Repre" c.txt
Found
spent:0.374555826187
alice@cryptZoobar>>enc_search "usa of Repre" c.txt
Not Found
spent:1.49930000305
alice@cryptZoobar>>enc_share c.txt bob
success
alice@cryptZoobar>>ls
enc:
    c.txt
alice@cryptZoobar>>
```

Figure 1: System Diagram.

3 Encrypted Binary Search

In this section we start with a simple encryption search scheme based off binary search. We then present certain attacks on the original scheme and give fixes to mitigate their effectiveness.

3.1 Basic Scheme

The fundamental data structure used for this scheme is the suffix array. The suffix array is a sorted list of all suffixes in the document. Formally, the suffixes are all substrings in a document of length n of the form substring(i, n). For suffix array $A[1 \dots n]$, $A[i]$ gives the ith largest suffix.

The scheme relies heavily on the property that any substring in the document is a prefix of some suffix. Thus, to search for a substring in the document the client and server engage in a cooperative binary search. When uploading the document to the server, the client constructs the suffix array and encrypts the document, sending both to the server. If the client later wants to search the document, the client requests the middle element of the suffix array. The server then accesses the middle element of the suffix array, finds the

corresponding index in the file, and returns as many encrypted blocks as needed to the client that would contain the keyword.

The client then decrypts the blocks sent by the server. If the keyword is found, the client requests the document. Otherwise, the client knows that his keyword is either smaller than or larger than (in lexicographical order) the text sent to it. Thus, the client asks the server to either search left or right in the suffix array to grab the next block. The server and client proceed in this fashion until the keyword is found or search is no longer possible. The time it takes to search a document is therefore roughly $[\text{RTT}] \cdot m \cdot \log_2(n)$ where RTT is the round trip time of the network and m is the length of the keyword.

One potential limitation to this scheme is the time needed to preprocess the document to construct the suffix array. The naïve algorithm that considers all suffixes and then uses a standard sorting algorithm requires $O(n \log n)$ many comparisons with the average comparison running in time $O(n)$. This leads to a rather undesirable $O(n^2 \log n)$ time bound for preprocessing. However, due to the importance of this data structure, the problem has already been explored extensively. In [7], Nong et al give an algorithm that has asymptotic complexity $O(n)$ and uses space very efficiently. We use a particular implementation of this algorithm by Yuta Mori that appears to be the one of the fastest known implementations of the suffix array [5].

3.2 Mitigating Frequency Analysis

Unfortunately, the scheme as described above is quite vulnerable to frequency analysis. Suppose a curious server suspects that the document that the client uploaded is English text. Since the server knows the distribution of letters in English text, the server can guess, for instance, the letter that is lexicographically in the middle of the text. This is precisely the letter that will be referenced in the middle of the suffix array. The server knows then, the location of that letter in the text. Repeatedly applying this procedure, the server may be able to reconstruct the entire text.

The key problem in this scenario is that when the server searches left or right during a binary search over the suffix array, it knows in which direction the smaller lexicographic suffixes are and in which directions the larger ones are. Thus, in our implementation we are careful to ensure that the server does *not* get this information. In particular, after the client constructs the suffix array, it should not immediately send it to the server.

First the client converts the sorted array into a balanced binary search tree. The client then randomly exchanges the left and right nodes in the tree. Thus, the server could only know which child is the smaller one by decrypting the text contained at that index. Fortunately, because the decryption key is kept private, the server cannot do this. To make this practical, the client then serializes the binary search tree into an array in the standard fashion (i.e. for array $A[1 \ldots n]$, the left child of node i is at position $2i$ and the right child is at position $2i + 1$.), and sends this array to the server.

The interactive protocol between the client and the server is now slightly different. That is, the server must now send both the left and the right child of a given request to client. The client then decrypts both children, decides which one is lexicographically smaller, and tells the sever to go left or right in the suffix array based on this information. Very little extra information is exchanged during this process, so the running time is roughly equivalent to the running time of the basic scheme, yet has much better security.

3.3 Further Security Analysis

Unfortunately, we still cannot achieve complete cryptographic security with the refinement mentioned above. A first observation is that the very first block accessed by the protocol is still predictable in that it probably contains the letter that occurs above exactly half of the probability distribution mass of English text (this letter happens to be an 's'). Using the same technique the server may be able to guess that certain letters are contained in blocks that are close to the root of the binary search tree. For instance, the server has a 50% chance of guessing a block that contains the letter that occurs above exactly a fourth of the probability distribution mass of English text (this letter happens to be an 'a') However, the deeper the server probes the binary search array (i.e. the suffix array), the less confidence it can be about the letters contained at

those levels since the probability that it guesses the right ordering of the blocks decreases exponentially with depth.

Unfortunately, the suffix array leaks more information about the document as he continues to search it. Namely, for a large file one would expect that they are many commonly shared substrings (In fact, this is precisely the property harnessed in the other search scheme described in this paper. See Section **??**). This implies that for a large document, the positions in the document referenced at the furthest levels of the suffix search tree are likely to contain similar phrases. However, it is not clear a priori how an adversary would use this information to actually decrypt the document without significant prior knowledge of what might be contained in the document.

4 Discussion and extensions of the two schemes

We now discuss in greater detail the performance of the two schemes.

The encrypted binary search scheme has fast preprocessing time and smaller ciphertext size. Preprocessing requires linear time, since constructing the suffix array, turning it into a scrambled balanced binary search tree, and encrypting the file using a block cipher are all done in linear time. The server stores only an array containing every index into the file, and the block cipher encryption of the file. The main drawback to this scheme is its runtime for search. Searching for a string of length k in this scheme takes $O(k \log n)$ time with $O(\log n)$ rounds of communication between the client and the server, since one round of communication corresponds to a level of the binary search tree and on each round the client must download an encrypted length-k substring from the block cipher ciphertext stored by the server. There are also potential security vulnerabilities explore in Section 3.

The hashing scheme performs well when it comes to search time and security. Preprocessing still can be done in linear time, but in practice it is substantially slower than our other scheme. The preprocessing time could be improved by rewriting time-intensive portions of the code in a lower-level language such as C, using a more efficient function to hash substrings, and using a more concise encoding of the data to send over the network. We already use C libraries to compute the suffix array and longest common prefix array and running gzip on files before sending them over the network, but additional improvements can be made to these components to reduce preprocessing time. Our current implementation hashes a substring by calling a hash function with the substring as part of the input. This requires time proportional to the length of the substring. It is possible to modify the hashing scheme so that linearly many substrings can be hashed in time linear in the length of the file. Using such a scheme should also speed up preprocessing.

The memory usage of the hashing scheme is probably its main drawback. In addition to a block cipher encryption of the file, the scheme requres storing hashes of certain substrings. The number of hashes varies depending on the input, but for typical input files it is generally between $2n$ and $3n$ for an input file of length n, as we discuss in Section 5. The server also stores indices associated with n of these hashes, but the indices are the values 1 through n, so we could avoid storing them by storing the corresponding hashes in order. (If we do this space optimization, then we would need to read through all n hashes to construct a hash table before we can perform $O(1)$ hash lookups.) Consequently for a file of length n this scheme requires storing a block cipher encryption of the file and between $2n$ and $3n$ hashes. Seven or eight byte hashes are probably enough to make collisions sufficiently rare, so this scheme can be implemented with a roughly $20x$ space blowup.

The hashing scheme is quite practical and has excellent asymptotic behavior for search. The client performs binary search on the length of the keyword, so given a keyword of length k, the client issues $\log k$ requests to the server. The server can answer each of these requests in constant time. On the last of these requests, the client must download an encrypted length-k string from the block cipher stored in the server. Consequently search can be performed in $O(k)$ time, with $\log k$ rounds of communication across the network between the server and the client. In fact, for some keywords search will be much faster. For instance, searching for a string which occurs multiple times in the file takes constant time with only a single round of communication between the client and the server. A modified scheme in which all k hashed prefixes of the query are sent in a single round would achieve $O(k)$ runtime with a single round of communication at

the expense of weaker security for the reasons discussed in the next paragraph. For networks with high latency, having even a small number of rounds of communication between the client and the server produces a substantial slowdown. However, if latency is not a huge problem, the scheme performs quite well. It is particularly desirable in settings where the search client has limited bandwidth or processing power, since the messages sent are small and little processing is required by the client for search. For instance, our scheme would work quite well if the search client is running on a phone with a data plan, since here bandwidth is expensive and computation is limited.

The scheme also seems to be fairly secure. It does leak the information of whether the query string occurs zero, one, or more than one time in the file, but this is not a problem if the server does not know what queries the client is making. Intuitively, assuming that the server cannot invert hashes, the only other information leaked is the index into the file of the shortest prefix of the query which occurs exactly once in the file, if such a prefix exists. As currently written this intuition is not quite correct, since the sequence of accesses used by the scheme may also reveal that some other hashes correspond to strings that are a prefix of a particular hashed string. With enough queries, the server might be able to carry out a linking attack in which it infers that two hashed strings have a prefix in common and therefore corresponding indices in the file contain the same letter. With a very large number of queries, the server may be able to use frequency analysis to gain information about the file. However, it seems that an enormous number of queries would be needed to carry out this attack, and the scheme would still be secure if a reasonable number of queries are issued. In addition, it is straightforward to modify the scheme to avoid this linking based attack. Instead of performing binary search to determine the shortest unique prefix of the query, we instead do this in linear time by shaving off one letter at a time until the hashed substring is found in the list of hashes. Under this modified scheme, for each query only a single value in the server's list of hashes is ever revealed to the server, so no linking attack can be performed if the server has no external information about what is being queried. The disadvantage of this modification is that now up to k rounds of communication may be needed to search for a query of length k, although search can still be performed in $O(k)$ time. However, in this modified scheme we believe that as long as the server has no prior information about what queries are being issued and the server is unable to invert hashes or break the block cipher encryption, the server is unable to learn additional information about the file or the queries.

5 Language assumptions of the hashing-based scheme

The server space usage of the hashing scheme is dominated by the space needed by the hash table. Recall that in the hashing scheme, we store a hash of every repeated substring in the file, as well as a hash of every unique substring in the file which would be repeated if we removed the last character. There can be at most one hash of the second type starting at each location in the file, so the number of hashes of the second type is at most the length of the file. Consequently in order to show that this scheme has reasonable space usage it is enough to consider the number of hashes of the first time, that is, the number of substrings that occur more than once in the file. We will show that the number of repeated substrings is small for the classes of input we are interested in.

In fact, on worst-case inputs the number of repeated substrings can be quite large. For instance, consider the file $A^nBA^nBA^n$ consisting of n As, followed by a B, followed by n As, followed by another B, followed by another n As. Any string of the form A^iBA^j for $0 < i, j \leq n$ occurs twice in the file. Consequently there are at least n^2 repeated substrings in the file even though the length of the file is only $3n + 1$. While it is not difficult to construct "bad" inputs like this, the number of substrings we need to hash is much more reasonable on typical inputs we are interested in, such as English text.

In Figure 2 we present the number of hashes which would be stored in encryptions of files containing English text. The inputs are works of literature in English. The last column, $Substrings/filesize$, is between 2 and 3 for every text file we tested. This column gives the number of hashes (of both types) which must be stored per character in the file. Rather than quadratic behavior, we see that on a typical English file of length n, the number of hashes we must store is between $2n$ and $3n$, which is quite reasonable.

Another type of inputs on which encrypted substring search would be useful is genetic data. The genome

Document in file	**Size**	**Size (bytes)**	**Substrings hashed**	**Substrings/ file size**
U.S. Constitution	41 kB	42724	117527	2.751
The Iliad	809 kB	829302	2103009	2.536
Ivanhoe	1.1 MB	1126890	2359574	2.094
Moby Dick	1.2 MB	1235162	2472071	2.001
Don Quixote	2.2 MB	2307734	5334241	2.311
Les Miserables	3.1 MB	3254531	6998035	2.150
King James Bible	4.2 MB	4351843	12654378	2.908
Complete works of Shakespeare	5.2 MB	5465099	12343184	2.259

Figure 2: Space usage of our hashing scheme for English text files of various lengths. For English text files, the number of substrings hashed per character of input is typically between 2 and 3. Input files downloaded from Project Gutenberg

Organism genome in file	**Size**	**Size (bytes)**	**Substrings hashed**	**Substrings/ file size**
M. pneumoniae	798 kB	817244	3030599	3.708
P. falciparum, chr. 14	3.1 MB	3291951	7498023	2.278
E. coli	4.4 MB	4641733	26815885	5.777
C. elegans, chr. 1	14.4 MB	15072512	92931692	6.166
Drosophila melanogaster, chr. 2R	20.2 MB	21146787	350902213	16.594
E. coli with cap 100	4.4 MB	4641733	10764364	2.319
C. elegans with cap 100	14.4 MB	15072512	41715289	2.768
D. melanogaster with cap 100	20.2 MB	21146787	61197516	2.894

Figure 3: Space usage of our hashing scheme for nucleotide sequences of various lengths. For genetic data, the number of substrings hashed per character of input is much more variable than for English text, and is frequently significantly larger. However, if we wish only to support searches for strings of length less than 100, the number of substrings hashed per character of input is much smaller, as shown in the last three rows. Input files downloaded from the National Center for Biotechnology Information

sequences of individuals represent highly sensitive data, and we would likely want to store them encrypted and avoid revealing information about them to an untrusted server. Nucleotide sequences are simply long strings of letters (for DNA, "A", "C", "T", and "G"), and do not break naturally into keywords. Consequently keyword search is not useful in this setting. On the other hand, substring search capability like that provided by our scheme would be useful since it would allow a user to query whether a particular sequence of nucleotides occurs in the file.

The space usage of our scheme on genetic data is presented in Figure 3. Genome sequences frequently contain long repeated sequences, so unsurprisingly, many more substrings must be hashed by a file of constant length. Unlike for English text, where we saw that between two and three substrings must be hashed per character in the input file, for genetic data the number of hashes per input character is highly variable. However, if we constrain our scheme to allow search only of substrings which are 100 characters or less and only store hashes of substrings of at most this length, the memory usage is much more comparable. As shown in the last three rows of Figure 3, the memory needed to allow search for strings of at most 100 characters is comparable to the memory needed for search in English text files of the same length—that is, the number of hashes needed is between two and three times the size of the input file.

6 System description

6.1 Overview

Our system adheres to the typical client-server model. Since we wish to stress the encryption scheme that supports substring search, we decided to keep the server architecture relatively simple. That is, we extend zoobar as our server to support a primary encrypted file system and keyword search, as was it's state after Labs 2 and 3. Thus, the system benefits from the privilege separation achieved in those labs.

On the client side, we implement a shell-like python script to communicate with zoobar by sending HTTP requests. Although it would be more user friendly to apply the encryption browser-side, we have several reasons for not doing this. First, we cannot trust the browser not to leak users keys. Second, private keys would be vulnerable to web attacks like XSS. Finally, encryption in Javascript is not secure for many reasons [10]. Therefore, we used python to implement our system.

This system requires users to provide two passwords, one for logging into zoobar and the other for generating a master key by MD5 hashing. The master key will be used for encrypting users profile. Every user will have a profile that includes important information of that user, such as which file that have been upload, the key, the initialization vector (IV), and the encrypted file name. This profile will be uploaded to server side after it has been encrypted by master key. Every time a user logs in, he will first download the profile before performing any action. The detailed system diagram is in Figure 5.

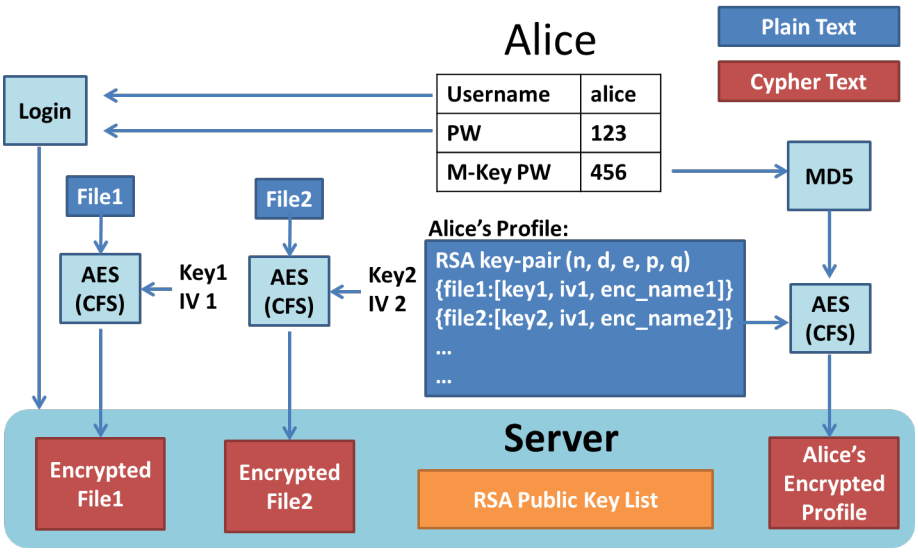

Figure 4: System Diagram.

When a user, Alice, want to upload a file, the system will generate a key and IV pair to encrypt its content as well as its filename with AES-CFS and store this key and IV pair in users profile. Since the filename is encrypted, other users cannot gain access to the file even if they know the filename in plain text. For the hashing scheme, the system will generate the hash table for this file, and for the binary search scheme it will make the binary search tree in a array. Both hash table and binary search tree will be json-encoded and sent to server.

If a user wants to search for a keyword in a file, he will need the key and IV pair to perform substring search and encrypted filename to let server target the right file. The client-side program will retrieve the corresponding key and IV pair from the users profile and send an HTTP request containing encrypted filename. Once the server gets the request, it will perform either hash table look up for first scheme or binary search for second scheme on the corresponding file.

To support file sharing, a public-key infrastructure is used. When a user registers a new account, the

system will automatically generated a RSA key pair for the new user, register the public key on server, and store private key in users profile. Suppose user Alice wants to share her file to another user Bob. She will first ask for Bobs public key from server and use it to encrypt the file name, key and IV. Then Alice will send the encrypted file information to server. When Bob logs in, he will download his profile and check if there are messages for sharing file from other users. Assuming there is, Bob will decrypt the message with his private key and update his profile. In this way, both Alice and Bob can access the encrypted file. The detailed flow diagram is in Figure **??**.

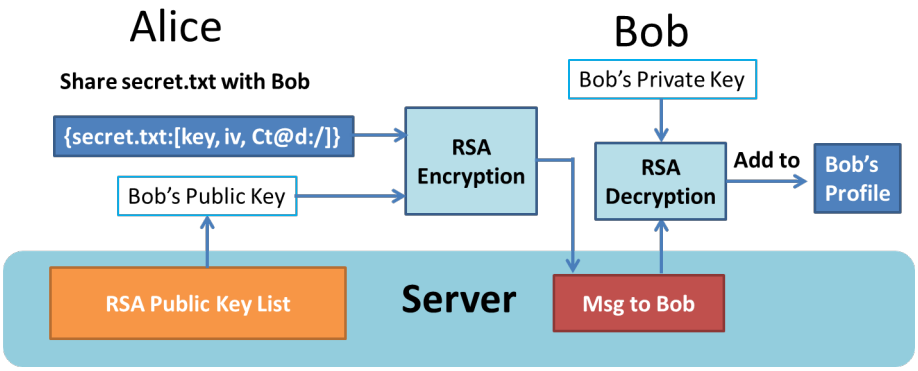

Figure 5: Sharing flow.

6.2 Optimizing Substring Search

Most of the scheme details have been described in previous sections. Here, we briefly go through some implementation tricks that make the scheme more practical. We only detail the hashing scheme because the techniques used in the binary search scheme are identical.

The most time consuming parts in this scheme are making the hash table and uploading it. Although this is indeed one-time action for every file, we would like to ensure that the scheme allows for reasonable pre-processing time.

Firstly, on the client-side, the system performs the function to make suffix array in C so that it would take less time comparing with implementation in Python. In fact we build our own python bindings for the implementation of fast suffix array. In fact, if we would like to further accelerate pre-processing, we can write the whole function in C. However, writing everything in C will also render sending HTTP requests difficult. Since we want to emphasize the scheme's themselves to show that they can be made practice, we feel that this is an acceptable trade-off.

Second, before we upload the hash table, the system first JSON-encodes, then gzips, then b64-encodes to make it smaller and safer to send over the network. Although the size of hash table is still much bigger than the size of file, weve successfully decreased 50% of the uploading time for hash tables that are several megabytes long.

7 Conclusion

We have designed and implemented two novel schemes for encrypted substring search, a problem which to our knowledge has no published practical solutions. In both schemes search can be performed in time sublinear in the length of the file, and search in our hashing scheme requires time independent of the length of the file and linear in the length of the query. While there is some ciphertext blowup, in both schemes it is linear for typical inputs and may be practical in some settings. We discuss some potential security vulnerabilities of our binary search scheme, but we believe that desirable security properties can be proven for our hashing

scheme. We also suggest two modifications of the hashing scheme, one of which sacrifices some security for improved search time while the other sacrifices some search time for improved security guarantees.

References

[1] Dan Boneh, Giovanni Di Crescenzo, Rafail Ostrovsky, and Giuseppe Persiano. Public key encryption with keyword search, 2003.

[2] Yan-Cheng Chang and Michael Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. In *Proceedings of the Third International Conference on Applied Cryptography and Network Security*, ACNS'05, pages 442–455, Berlin, Heidelberg, 2005. Springer-Verlag.

[3] Seny Kamara, Charalampos Papamanthou, and Tom Roeder. Dynamic searchable symmetric encryption. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 965–976, New York, NY, USA, 2012. ACM.

[4] Yuta Mori. An implementation of the induced sorting algorithm. `https://sites.google.com/site/yuta256/`. [Online; accessed 13-December-2013].

[5] Yuta Mori. Sais: An implementation of the induced sorting algorithm. `https://sites.google.com/site/yuta256/sais`.

[6] Ge Nong, Sen Zhang, and Wai Hong Chan. Two efficient algorithms for linear time suffix array construction. *Computers, IEEE Transactions on*, 60(10):1471–1484, 2011.

[7] Ge Nong, Sen Zhang, and Wai Hong Chan. Two efficient algorithms for linear time suffix array construction. *IEEE Transactions on Computers*, 60(10):1471–1484, 2011.

[8] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: Protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 85–100, New York, NY, USA, 2011. ACM.

[9] Raluca Ada Popa and Nickolai Zeldovich. Multi-key searchable encryption. Cryptology ePrint Archive, Report 2013/508, 2013. `http://eprint.iacr.org/`.

[10] Matasano Security. Javascript cryptography considered harmful // `http://www.matasano.com/articles/javascript-cryptography/`.

[11] Dawn Xiaoding Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *Security and Privacy, 2000. S P 2000. Proceedings. 2000 IEEE Symposium on*, pages 44–55, 2000.