RESeED: Regular Expression Search over Encrypted Data in the Cloud

Mohsen Amini Salehi^1, Thomas Caldwell, Alejandro Fernandez, Emmanuel Mickiewicz, Eric W. D. Rozier^2, and Saman Zonouz^3
Electrical and Computer Engineering
University of Miami
{*m.aminisalehi*^{1}, *e.rozier*^{2}, *s.zonouz*^{3}}*@miami.edu*

David Redberg
Fortinet Inc.
Sunnyvale, California 94086
dredberg@fortinet.com

***Abstract*—Capabilities for trustworthy cloud-based computing and data storage require usable, secure and efficient solutions which allow clients to remotely store and process their data in the cloud. In this paper, we present RESeED, a tool which provides user-transparent and cloud-agnostic search over encrypted data using regular expressions without requiring cloud providers to make changes to their existing infrastructure. When a client asks RESeED to upload a new file in the cloud, RESeED analyzes the file's content and updates novel data structures accordingly, encrypting and transferring the new data to the cloud. RESeED provides regular expression search over this encrypted data by translating queries on-the-fly to finite automata and analyzes efficient and secure representations of the data before asking the cloud to download the encrypted files. We evaulate a working prototype of RESeED experimentally (currently publicly available) and show the scalability and correctness of our approach using real-world data sets from `arXiv.org` and the IETF. We show absolute accuracy for RESeED, with very low (6%) overhead, and high performability, even beating `grep` for some benchmarks.**

***Keywords*-cloud computing; security; privacy; searchable encryption; regular expression**

I. Introduction

The establishment of algorithms which allow for keyword searches over a set of encrypted documents [3] has been critical to the development of privacy preserving clouds, and will likely grow in importance given predicted losses resulting from the recent damage done to global confidence in cloud security [16], [17]. During the last decade, there have been an increasing number of proposed solutions to address the users privacy and data confidentiality violation concerns while providing the capability to perform searches over encrypted data. These solutions can be categorized into two groups. First, cryptographic algorithms, initiated by the seminal work by Boneh et al. [3], make use of mathematical primitives such as public key encryption. The major advantage of using cryptographic techniques is the existence of theoretical proofs that they will not leak sensitive information as long as the underlying mathematical primitives are not broken. These approaches often suffer from performance and scalability aspects that limit their real-world deployment significantly. The second group of techniques make use of novel techniques from database research [18], resulting in search methods for large-scale data centers such as locality sensitive hashing techniques. Such solutions have shown promising results in real-world settings in terms of the efficiency of remote query processing on encrypted data. On the negative side, database approaches have been highly criticized for disclosing sensitive information that (even though indirectly) could potentially cause data confidentiality and user privacy violations.

Additionally, both of the existing cryptographic and database techniques for cloud data storage and processing fall short in two primary ways. First, they have mostly concentrated on the simplest categories of search, i.e., key-term search. Second, for real-world deployment, they require cloud provider cooperation in implementing their proposed algorithms within the cloud that could be a significant barrier in practice. While recent advances have extended the capability of encrypted keyword search, allowing searches for ranges, subsets, and conjunctions, increasing the power of search over encrypted data [5]. One powerful tool which has remained elusive, however, is the ability to apply regular-expression based search on encrypted data. A solution using current methods remains impossible in practice due to the exponential explosion of the space required for the storage of the resulting ciphertexts. As such, a regular expression based search on possible phrases in a document with even as few as 5,000 words would require more storage than will exist for the foreseeable future.

In this paper, we present RESeED a method that provides cloud providers with a scalable user-transparent capability for regular expression and full text search over encrypted files stored in the cloud. RESeED achieves this objective without any need for the cloud provider to cooperate, or change its infrastructure. To the best of our knowledge, RESeED *is the first solution to provide searchability over encrypted data using the power of regular expressions*, and the first that makes use of both database techniques such as local indexing, and cryptographic approaches such as symmetric encryption to satisfy the requirements of deployment efficiency and information leakage guarantees. Users who have RESeED installed on their client systems are able to upload files to the cloud for storage, remotely search for encrypted files using regular expressions, and download the data which matches these queries. the background so that the users see only plain-text data on their local machines.

The architecture of RESeED is illustrated in Figure 1. Queries are processed through the use of novel data structures which we call the *column store* and *order store*. A single column store that indexes keywords found in the files for the entire data-set, and a compact order store, which contains a fuzzy representation of keyword ordering, is created and stored for each new file. Queries are processed based on the

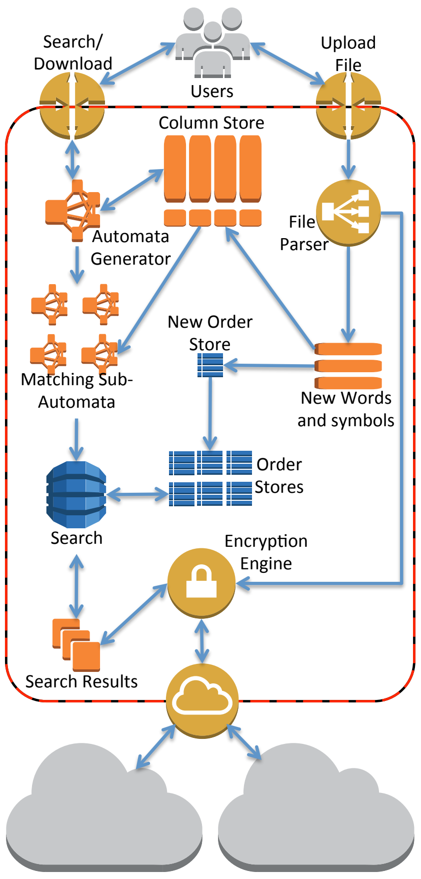

Figure 1. System Architecture

conversion of the regular expression to an automaton, which is then transformed to match sub-expressions. RESeED is deployed on a trusted hardware that facilitates secure access to Cloud providers. Currently, RESeED is built for Fortivault, a trusted gateway for accessing Cloud services provided by Fortinet Ltd[1]. Experimental results generated from a working prototype of RESeED on real data-sets, express promising results both in terms of correctness and performance.

In summary, the contributions of the paper are the following.

- We introduce a novel and scalable solution which processes regular-expression based searches over encrypted data. Our solution operates based on two novel data structures, a column store, and an order store, to achieve this scalability. We process searches over these data structures using algorithms described in Section III to transform the query problem into a domain covered by our new data structures.
- We present the first practically deployable solution to the search over encrypted data problem with zero information leakage guarantees, implement this solution and evaluate our prototype on real-world data-sets.
- We demonstrate the performability, scalability, and correctness of our results, demonstrating the low overhead incurred, even when compared with existing solutions, and demonstrating high performability, in some cases beating the run time of existing tools for regular expression search over unencrypted data.

[1]http://www.fortinet.com/

This paper is organized as follows. Section II reviews the most recent work in the literature, and establishes the need for our solution. Section III presents the algorithms used by RESeED for regular expression search on encrypted text. Section IV describes our implementation details followed by Section V which presents the empirical results using real data-sets. Finally, Section VI concludes the paper and lays out our plan for future work.

II. Related Work

Searchable encryption has been extensively studied in the context of cryptography [8], [9], [21], mostly from the perspectives of efficiency improvement and security formalization. Searchable encryption was first described in [21], but the methods provided are impractical and erode security due to the necessity of generating every possible key which the search expression can match. To reduce the search cost, in [9], Goh proposed to use Bloom filters to create per-file searchable indexes on the source data. However, these previous studies just consider exact keyword search.

One could construct regular expression based search over encrypted data using the scheme presented in [5], but again the results prove impractical, requiring ciphertext and token sizes of the order $O(2^{nw})$.

Recent work has focused significantly on enabling search over encrypted Cloud data. Wang *et al.,* [23] studied the problem of similarity search over the outsourced encrypted Cloud data. They considered edit distance as the similarity metric and applied a suppressing technique to build a storage-efficient similarity keyword set from a given collection of documents. Based on the keyword set, they provided a symbol-based tree-based searching index that allows for similarity search with a constant time complexity. In other research [14], privacy-preserving fuzzy search was proposed for encrypted data in Cloud. They applied a wild-card-based technique to build fuzzy keyword sets which were then used to implement a fuzzy keyword search scheme.

Ibrahim *et al.,* [12] provided approximate search capability for encrypted Cloud data that was able to cover misspelling and typographical errors which exist in a search statement and in the source data. For this purpose, they adapted the metric space [10] method for encrypted data to build a tree-based index that enables the retrieval of the relevant entries. With this indexing method, similarity queries can be carried out with a small number of evaluations. Our work contrasts with these studies due to the fact that our search mechanism allows search on the encrypted cloud data through the use of regular expressions, meaning that the searches enabled by our technique are not limited to a set of predefined keywords.

Making use of encryption techniques ensures that user privacy is not compromised by a data center [13]. But the problem then becomes, how can an encrypted database be queried without explicitly decrypting the records to be searched? At a high level, a searchable encryption scheme [11] provides a way to encrypt a search index so that its contents are hidden except to a party that is given appropriate tokens. The Public Key Encryption with keyword search primitive, introduced by Boone *et al.,* [3], indexes encrypted documents using keywords. In particular, public-key systems that support equality $(q = a)$, comparison queries $(q > a)$ as well as more general queries such as subset queries $(q \in S)$. Song *et al.,* [21] presented a symmetric cryptography setting for searchable encryption architectures for equality tests. Equality tests in the public-key setting are closely related to Anonymous Identity Based Encryption and were developed in [4] by Boneh *et al.,* In [5], Boneh *et al.,* introduce a new primitive called Hidden Vector Encryption that can be used for performing more general conjunctive queries such as conjunction of equality tests, conjunction of comparison queries and subset queries. This system can also be used to support arbitrary conjunctive queries without revealing any information on the individual conjuncts.

III. Regular Expression Based Search Over Encrypted Data

We introduce a new method for enabling regular expression search over encrypted documents which does not suffer from the exponential growth of the stored search keys. The method presented in this paper requires the storage of tokens for public-key encryption with keyword search (PEKS) as described in [3] whose storage requirement is on the order $O(nw)$. PEKS is a keyword-based encryption algorithm that does not reveal any information about the content, but enable searching for the keywords. We also use a novel structure we call an order store, which contains a fuzzy indication of the order of keywords within a document, whose storage requirement is on the order $O(n)$, and in practice has a fractional constant factor which keeps the overhead very low. When searching the entirety of `arXiv.org`, for instance, the order stores required only a 6.1% overhead.

A. Alphabet Division

The key to our algorithm for regular-expression based search over a set of encrypted data revolves around the ability to divide the alphabet Σ, of a non-deterministic finite automaton (NFA) representing some regular expression r into two disjoint subsets, C, the set of *core* symbols, and Ω, the set of *separator* symbols. Our methods work for any language composed of strings from $S_C \in 2^{\Sigma}$ separated by strings from $S_\Omega \in 2^{\Omega}$. Intuitively strings from S_Ω are collections of white space and other separator characters, while strings from S_C are words, numbers, or other important symbols that users tend to search. Based on a survey of the uses of regular expressions in practice [2], [7], [19], [24], given an alphabet where such a division is possible, searches for phrases in the language most often take this form. We note that this appears to be the case for languages other than natural language (e.g. genome sequences are composed of sets of codons which can be divided into start and stop codons, and the core codons which code for amino acids), but focus on the application of this technique to natural language within this paper.

B. Algorithm for Regular Expression Search

In order to apply regular expression based search to a document, we first perform two operations on the document creating two data structures. The first is to extract every unique string $s_i \in 2^{S_C}$, or the set of unique words. The second is to create a fuzzy representation of the order of the strings in the document, by hashing each word to a token of N bits, and storing them in the order in which they appear in the document.

For the set of all documents we create a single *column-store*, Ξ, which indexes the set of unique strings found in all documents, and indicating the files in which they can be found. The column-store can be generated using PEKS token for each keyword as described in [3]. For each file or document, F_i, we also create a unique *order store*, O_i, from the ordered set of fuzzy tokens. The order store is simply an ordered list of the fuzzy hashes of N bits for each word, with each hash appearing in the same order as the original word appeared in the original document.

Algorithm 1 Regular Expression Search

```
 1: procedure REGEXP($r, \Xi$)          ▷ Where $r$ contains the regular expression, and $\Xi$ is the column-store.
 2:     $n_0 \leftarrow$ REGEXPTONFA($r$)
 3:     $N' \leftarrow \omega$TRANSFORM($n_0$)
 4:     marking $\leftarrow$ ()          ▷ A bit matrix for each file, and each word in $\Xi$, initially all 0
 5:     for $n'_i \in N'$ do
 6:         $d_i \leftarrow$ NFATODFA($n'_i$)
 7:         for word $\in \Xi$ do
 8:             if $d_i$ ACCEPTS word then
 9:                 $F \leftarrow$ FILES(word)
10:                 MARK(F, marking)
11:             end if
12:         end for
13:     end for
14:     $P \leftarrow$ FINDPATHS($n_0, N'$, marking)
15:     matches $\leftarrow \emptyset$
16:     for $p_i \in F$ do
17:         $O_i \leftarrow$ GETORDERSTORE(File for $p_i$)
18:         if $p_i \in O_i$
19:         then matches $\leftarrow$ matches $\cup$ (File for $pi$)
20:     end for
21:     return matches
22: end procedure
```

Using these data structures, we can perform a search which is equivalent to one performed over the encrypted database via the procedure given in Algorithm 1. The regular expression r is first converted into a non-deterministic finite automaton (NFA), n_0 [15], this automaton is then partitioned into a set of NFAs defined by a transformation we call an ω-transformation, detailed in Algorithm 2.

The ω-transform generates a set of automata, N' from our original NFA n_0 by operating on the transitions labeled with elements of Ω in n_0. Each state in the original NFA with an incoming transition labeled with a symbol from Ω is

Algorithm 2 ω-Transformation

```
1: procedure ωTRANSFORMATION($n_0$)
2:     Let $T_{n_0}$ be the transitions in $n_0$
3:     Let $S_{n_0}$ be the states in $n_0$
4:     for Each state $s_i \in S_{n_0}$ do
5:         for Each incoming transition of $s_i$, $t_j \in T_{n_0}$ do
6:             if $t_j$ has label $l_k \in \Omega$ then
7:                 $S_{\text{Start}} \leftarrow s_i$
8:                 $T_\Omega \leftarrow t_j$
9:             end if
10:        end for
11:        for Each outgoing transition of $s_i$, $t_j \in T_{n_0}$ do
12:            if $t_j$ has label $l_k \in \Omega$ then
13:                set $s_i$ to an accepting state
14:            end if
15:        end for
16:    end for
17:    $T_{n_0} \leftarrow T_{n_0} \setminus T_\Omega$
18:    for each $s_i \in S_{\text{start}}$ do
19:        generate the automaton $n'_i$ from the reachable
   states of $s_i$
20:        if $s_i$ is an accepting state
21:        then remove $s_i$ from the accepting states of $n'_i$
22:        $N' \leftarrow n'_i$
23:    end for
24:    return $N'$
25: end procedure
```

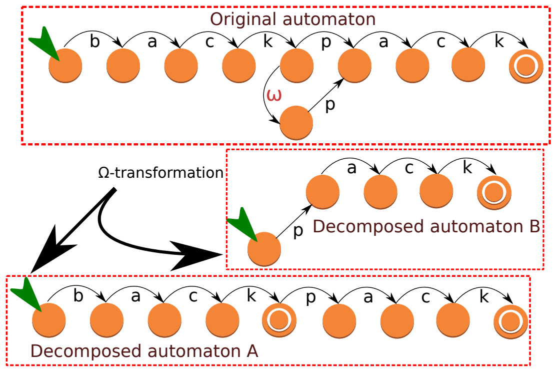

Figure 2. Example of an ω-transformation on a simple automaton for the regular expression $back[\backslash s]^\star pack$.

added the the set of start states. Each state with an outgoing transition from Ω is added to the set of accepting states. We then remove from n_0 all transitions labeled with symbols from Ω and for each start state, generate a new NFA from those states reachable from the new start state, as shown in figures 2. Intuitively, this new set of NFAs N' are the set of NFAs which match strings in C separated by strings in Ω, and are thus suitable for searching our column store for matches on individual tokens.

Once N' has been obtained, we minimize the NFA contained in the set, and for each word in the column store, Ξ, check for containment in each NFA. We maintain a bit matrix, `marking` that indicates for each file, if a word matched each NFA in N'. Once this matrix is obtained, we check the marking for each file to see if, for the marked NFAs in N', if there exists some $n'_i \in N'$ with the original start state of n_0, some $n'_j \in N'$ with an original accepting state of n_0, and a set of automata N_{path} such that there is an $n'_{i+1} \in N'$ whose start state is an accepting state of n'_i, and which has an accepting state which serves as the start state of some $n'_{i+2} \in N'$, and so on until we find an NFA n'_{j-1} with an accepting state that is the start state of n'_j, showing that the file contains the necessary words to form a path from the start state of n_0 to one of the original accepting states.

To confirm this path, we generate a new NFA from the relationship between n_0 and N' which we will call n''. This NFA has states equal to all states which were either a start state or accepting state in an NFA in N', and which has transitions for every NFA in N_{path} labeled with a hash of the words matched in Ξ by those NFAs. We set the start state and accepting states of n'' to be the same as n_0. We then obtain the order store, O_i, for each file F_i where such a path could exist, and check to see if there exists a string in O_i is accepted by n''. If there is, we indicate a match, and mark the matching encrypted file as part of the set of files which contain a match for the original regular expression r.

C. Practical Considerations and Challenges for Cloud-Based Encrypted Data

The results of our previous experimentation with cloud-based encrypted search [20] addresses the issue that there is often a serious penalty for making multiple requests to a cloud storage system for the results of queries on an individual basis. As such our column and order stores are maintained in aggregated tables, which are downloaded together, and searched locally to reduce the penalty for querying the cloud. In addition, the choice of the subdivision of Σ into C and Ω is partially subjective. While it does not effect the correctness of our results, it can effect the quality of the partition for usability reasons. As such, we define the division by allowing users to define the set Ω, and then defining $C = \Sigma \setminus \Omega$.

IV. Implementation

We discuss the implementations details of our framework regarding both the creation and maintenance of the column store, and the production and translation of all automata.

A. Column store creation and maintenance

When creating, updating, and maintaining the column store, we used three main structures. First, the column store table maps individual tokens, in the form of strings, to sets of file identifiers. Second, we map file identifiers to the names of files that have been processed by the program. and create an inverse file map which represents its inverse. This gives us a bijection from the set of file names of processed files to the set of file identifiers When the program finishes executing, we save the updated column store.

Updating the column store, given a list of new files, is handled as follows. For each file F_i in the list, if the file has not already been encountered, we assign a new numerical identifier for the file name and insert it into both the file map and inverse file map. We create a new file to contain the hashes of the individual tokens in f_i which is then parsed, and each previously unseen token is added to the column store. We hash these tokens and insert them into the file of hashes that corresponds to F_i. For our implementation, the tokens are hashed using SHA-1. We take the first b^2 bytes of the 20 bytes produced by the SHA-1 and discard the remainder. The string of these resulting bytes produced for a given file the order store and contains the hashes of all of the tokens contained in F_i in the order that they appear.

B. Automata Creation and Translation

In our implementation, each NFA is represented by a transition table given by a square adjacency matrix, and a start and end state. Transitions are represented by integers (to allow all ASCII characters to be used as transitions while still having room for special characters to represent epsilon transitions and common regular expression wild-cards such as ., \w, and \d).

The proposed framework implements a breadth first search algorithm to explore the Path NFA to create P, which is a vector of paths. Each path is represented as a vector of sub-automata (each sub-automata identified by an integer and only present once in the case of cycles) that would provide a valid path through the Path NFA. The allpaths structure is later used with the order stores to check if the file a given order store represents satisfies any valid paths. Since the size of the subNFA state space is on the order of the number of delimiters in the input regular expression, creating this structure and checking against are more efficient than scanning each file.

It is noteworthy that due to the nature of the implementation, the language of this program differs slightly from the `grep` command regular expression search. In fact, our current implementation is similar to the `grep` command with `w` flag. There are also minor differences between the `grep` regular expression language and the language RESeED works with. Table I describes the regular expression language we use in RESeED.

V. Evaluations

A. Experimental Setup

In order to experimentally evaluate the performance and correctness of our algorithm we tested our algorithm on two different data sets. The first data-set is the Request For Comments[3] (RFC) document series, a set of documents which contain technical and organizational notes about the Internet. This data-set contains 6,942 text files with total size of 357 MB and has been used in previous research works (*e.g.,* in [23]). The second data-set is a collection of scientific papers from the arXiv[4] repository. This data-set contains 683,620 PDF files with the total size of 264 GB. All experiments were conducted on a computer running Linux (Ubuntu 12.04), with 4 Intel Xeon processors (1.80 GHz) and 64 GB RAM. For the purposes of fairness in our experiments, we limited the algorithm to a single core, but note that our algorithm benefits heavily from parallelization, as many of the steps are trivial to parallelize.

We derived a set of ten regular expressions benchmarks based on those provided by researchers at IBM Almaden [6], translated into an abbreviated set of regular expression operators and symbols, as indicated in Table I. This set of benchmarks was initially compiled for searching Web contents. We adapted these benchmarks to be semantically interesting and popular for both the RFC and the arXiv data-sets. The regular expression benchmarks and their descriptions are listed in Figure 3. They are sorted based on the time they take to execute. That is, the first benchmark is the fastest and the last one is the most time consuming one.

B. Finding the Proper Hash-Width

Due to the fuzzy nature of the order store, there exists a chance that our search algorithm returns files that do not contain the regular expression that we are searching for, resulting in a false positive due to the pigeonhole principle. While a smaller hash-width results in a more compact order store, it also results in a higher rate of false positivies. We experimentally evaluated the effect of hash-width on the rate of false positives to determine an ideal hash-width for our data sets. We generated order stores for each file with hash-widths between one and 10 bytes and measured the rate of false-positives for the benchmarks listed in Figure 3. We used grep to confirm the observed rate of false positives returned by our method. The result of this experiment for the RFC data-set is illustrated in Figure 4. The horizontal axis shows different hash-widths and the vertical axis shows the percentage of false-positive rate for each benchmark.

As we can see in Figure 4, most of the benchmarks have high false-positive rate for a hash-width of one. However, for a hash-width of two, the false-positive rate drops sharply. In particular, the drop of the false-positive rate is sharper for benchmarks that have are less fuzzy (*i.e.,* contain specific words) such as benchmarks (B), (C), and (D). In contrast, the false-positive rate in benchmarks which are more fuzzy (*e.g.,* benchmark (G) and (J)) tend to drop slower. The reason is that for a these fuzzier regular expression, there are several words that can match to the expression and if any of these matching words have a hash collision, then it will lead to a false-positive. We observe that with a hash-width of three bytes the false-positive rate is almost zero for all benchmarks.

C. Evaluation of Regular Expression Benchmarks

To demonstrate the efficacy of our search method we also evaluated its performance for the RFC and arXiv datasets and compared this performance against the performance of the `grep` utility [1]. For each benchmark, we measured the overall search time in our method, indicating the time to construct automata; the time to match against the column store, and the time to match against the order store for the encrypted data. Additionally, we measure the total time that `grep` takes to search the same regular expression over the unencrypted data-set. To eliminate any measurement error that may happen due to other loads in the system,

[2]In our experiments, $b = 3$.

[3]http://www.ietf.org/rfc.html

[4]http://arxiv.org/

Table I
REGULAR EXPRESSION SYMBOLS USED IN THIS PAPER AND THEIR MEANINGS.

Symbol	**Definition**	**Symbol**	**Definition**
`.`	any character; Character . is shown as `\.`	`|`	OR statement
`*`	zero or more repetition of a character	`\s`	any Ω character
`+`	one or more repetition of a character	`\w`	any alphabetic character
`?`	zero or one repetition of a character	`\d`	any numeric character

(A) Words related to Interoperability:
`Interopera(b(le|ility)|tion)`

(B) All files that have "Cloud Computing" in their text:
`cloud(\s)^+computing`

(C) Structured Query Language or SQL:
`S(tructured)^?(\s)^+Q(uery)^?(\s)^+L(anguage)^?`

(D) All references to TCP/IP or Transmission Control Protocol Internet Protocol:
`((Transmission(\s)^*Control(\s)^*Protocol)|(TCP))`
`(\s)^*/^?(\s)^*((Internet(\s)^*Protocol)|(IP))`

(E) All files that reference "Computer Network" book of "Andrew Stuart Tanenbaum":
`Tanenbaum,(\s)^*(A\.|Andrew)((\s)^*S\.|Stuart)^?(,)^?`
`(\s)^*(\")^?Computer(\s)^*Networks(\")^?`

(F) All dates with YYYY/MM/DD format:
`(19|20)(\d\d)/(0(1|2|3|4|5|6|7|8|9)|1(0|1|2))/`
`(0(1|2|3|4|5|6|7|8|9)|(1|2)\d|3(0|1))`

(G) URLs that include Computer Science (cs) or Electrical and Computer Engineering (ece) and finished by .edu:
`http://((\w|\d)^+\.)^*(cs|ece)\.(\w|\d|\.)^+\.edu`

(H) All IEEE conference papers after the year 2000:
`(2\d\d\d(\s)^+IEEE(\s)^+(\w|\s)^*)|`
`(IEEE(\s)^+(\w|\s)^*2\d\d\d(\s))Conference`

(I) Any XML scripts in the papers:
`<(\?)^?(\s)^*(xml|html)(\s)^+.^*(\?)^?>`

(J) Papers from any US city, state, and possibly ZIP code:
`(\w)^+(\s)^*,(\s)^*(\w\w)(\s)^*\d\d\d\d\d(-\d\d\d\d)^?`

Figure 3. Regular expression benchmarks used for the performance evaluations.

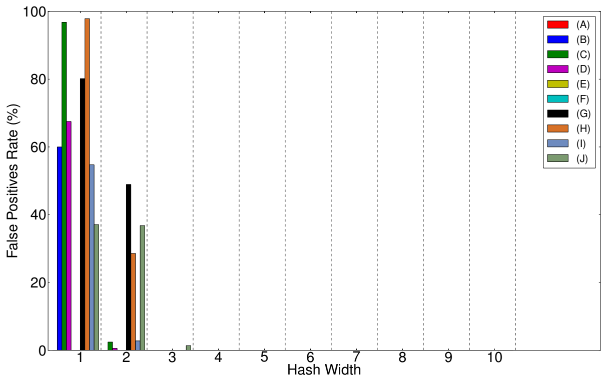

Figure 4. False positive rate for each benchmark as a function of hash-width.

we ran the experiment multiple times, reporting the mean and 95% confidence interval of the results. The confidence intervals are so small that they are not readily visible in the graph. Figure 5 shows the result of our evaluations using the benchmarks listed in Figure 3. The experiment shows the feasibility of searching complicated regular expressions within a limited time. More importantly, the figure shows that even though our method searches on the encrypted data, it performs faster for benchmarks (A)-(F) than `grep`. The reason for our method's improved performance is the fact that our method uses the column store first, searching fewer files compared to `grep` which scans the whole file set. We note that for the benchmarks that perform longer than `grep` (benchmarks (G)-(J)), our method spends a considerable amount of its time to match against the order store.

In general, our method performs faster than `grep` when given less fuzzy regular expressions or when the list of the order stores that need to be searched is small. In the former case, matching each entry of the column store against our generated automata is performed quickly and in the latter case, the number of files that have to be checked in the order store are few.

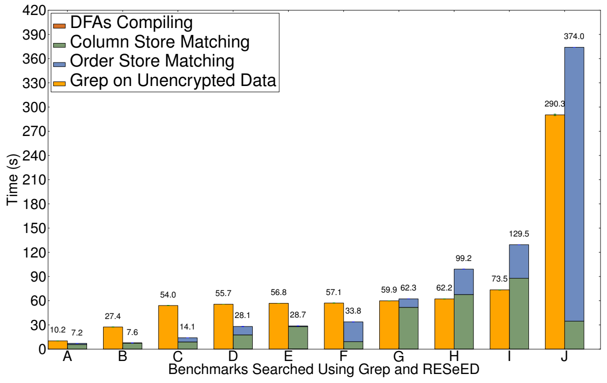

Figure 5. Time to search for matches for our benchmarks from Figure 3 for Grep and RESeED.

We also investigated how our proposed method scales for larger data-sets. We compared the performance of our method using the same benchmarks on the RFC data-set against the arXiv data-set, and normalize the results based on the size of the data-sets to show how well our method scales for larger data-sets. The result of this experiment is shown in Figure 6. The horizontal axis shows different benchmarks evaluated against the RFC and the arXiv data-sets and the vertical axis shows the time required to execute the search, normalized by the size of the data-sets in MB. The arXiv data-set is 730 times larger than RFC, however, as we can see in Figure 6, even for the most difficult benchmark (J) the normalized search time the arXiv data-set is remarkably less than the time for the RFC data-set. This experiment demonstrates the scalability of our approach, showing that as the size of the data-set increases, the normalized search time drastically decreases. The reason for this remarkable decrease is the scalable structure of the column store which enables effective identification of order stores which contain possible accepting paths in the automaton representing our regular expression. These features make our method highly scalable, and well suited for very large data-sets.

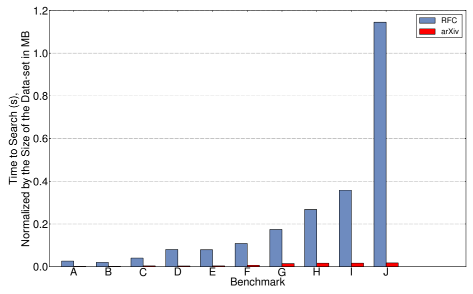

Figure 6. Size normalized search time for the RFC and arXiv data-sets.

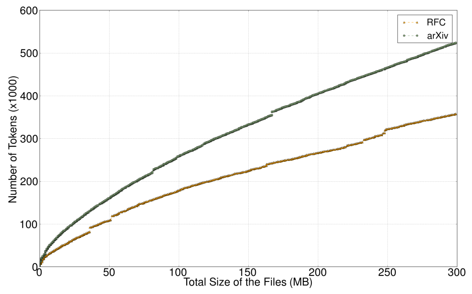

Figure 7. Column store size as a function of data-set size.

D. Analysis of the Overhead

Our proposed method has two main sources of overhead. The first is related to the space complexity of the column store. To investigate the growth of the column store, we have measured the number of tokens stored in the column store as files are added to the data-set. Figure 7 illustrates how the size of the column store scales as the size of the RFC and arXiv data-sets increases. For the sake of comparison, in this figure, we have shown the number of tokens (*i.e.,* entries) in the column store for the arXiv and RFC data-sets for the first 300MB of files to illustrate the trend. We expected the number of tokens in the column store to have a logarithmic growth as the size of data-set increases [22]. As can be seen in Figure 7, the increase in the size of the column store includes a linear and a logarithmic component. We that the logarithmic component is due to the addition of dictionary words to the column store, whereas the linear component is caused by unique words which exist only within a small handful of files such as name of the authors of a paper or name of cities. For the same reason we notice a more logarithmic trend in the number of tokens in the RFC data-set whereas growth of the number of tokens in the arXiv data-set has a larger linear component due to more unique names such as author names, university names, names in the references, etc.

However, the size of the column store is small compared to the size of the whole data-set. More importantly, this overhead becomes less significant as the size of the data-set increases. For instance, the size of column store in the RFC data-set is 13% of the whole data-set whereas it is only 2% of the arXiv data-set. We also note that the overhead from the column store is similar to that induced by searchable encryption techniques which cannot handle regular expressions, such as those in [5].

The second overhead is pertains to the time taken to update the column store and to generate new order stores upon the addition of a new file to the data-set. To measure this overhead we added files to the RFC and the arXiv data-sets one by one in a randomized order and measured the time of the update operation, completing this experiment several times. The result of this experiment is shown in Figure 8 for both the RFC (Sub-figure 8(a)) and the arXiv (Sub-figure 8(b)) data-sets. In this experiment, for the sake of comparison, we show the update time for only the first 300

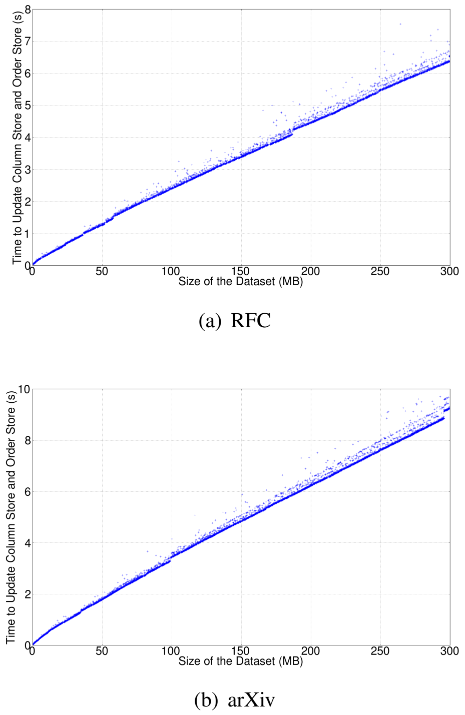

Figure 8. Time to update column store for RFC and arXiv data sets.

MB of added files for both the RFC and the arXiv data-sets. In both sub-figures as the size of the data-set increases, the time for the update operation increases linearly. A constant part of this increase is due to the overhead of generating the order store for each new file. However, the general linear trend is attributed to the time to insert multiple new entries into the column store. In our implementation, we have used an `unordered_set` data structure to store the list of appearances of a token in different files. The average time complexity of the insert operation for multiple entries (*i.e.,* multiple tokens) in this data structure in linear based on the number of tokens. We note that this performance overhead can be mitigated with multiple column stores, and benefits easily from parallelization.

VI. Conclusions and Future Works

In this paper, we presented RESeED, a method which provides the cloud providers with a user-transparent and cloud-agnostic capability to process regular expression based search over encrypted data residing in the cloud. RESeED improves upon current state-of-the-art techniques in the search over encrypted data, by providing a highly scalable, performable, and accurate solution with low storage and performance overhead. The proposed solution is also user-transparent, and cloud-agnostic. Our experiments with a working prototype of RESeED on real-world data-sets proves RESeED's deployability and practicality empirically.

Acknowledgment

The authors would like to thank Fortinet Inc. for their generous financial support for this project. The authors would also like to thank Cornell University for making the full repository of arXiv.org available for download, and the Internet Engineering Task Force for making the RfC repository available for download.

Availability

A demo of our tool, with uploads restricted to prevent abuse, can be found at the following URL:

http://www.performalumni.org/trust/Dragonfruit/demo/

References

[1] The open group base specifications issue 7. http://pubs.opengroup.org/onlinepubs/utilities/grep.html. Accessed: 2014-1-31.

[2] Alfred V Aho and Margaret J Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

[3] Dan Boneh, Giovanni Di Crescenzo, Rafail Ostrovsky, and Giuseppe Persiano. Public key encryption with keyword search. In *Advances in Cryptology-Eurocrypt 2004*, pages 506–522. Springer, 2004.

[4] Dan Boneh and Matt Franklin. Identity-based encryption from the weil pairing. In *Advances in CryptologyCRYPTO 2001*, pages 213–229. Springer, 2001.

[5] Dan Boneh and Brent Waters. Conjunctive, subset, and range queries on encrypted data. In *Theory of cryptography*, pages 535–554. Springer, 2007.

[6] Junghoo Cho and Sridhar Rajagopalan. A fast regular expression indexing engine. In *Proceedings of the 18th International Conference on Data Engineering*, pages 419–430, 2002.

[7] Charles LA Clarke and Gordon V Cormack. On the use of regular expressions for searching text. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(3):413–426, 1997.

[8] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In *Proceedings of the 13th ACM conference on Computer and communications security*, CCS '06, pages 79–88, 2006.

[9] Eu-Jin Goh. Secure indexes. Cryptology ePrint Archive, 2003. Report 2003/216.

[10] Gísli R. Hjaltason and Hanan Samet. Properties of embedding methods for similarity searching in metric spaces. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 25(5):530–549, May 2003.

[11] Dennis Hofheinz and Enav Weinreb. Searchable encryption with decryption in the standard model. *IACR Cryptology ePrint Archive*, 2008:423, 2008.

[12] Ayad Ibrahim, Hai Jin, Ali A Yassin, Deqing Zou, and Peng Xu. Towards efficient yet privacy-preserving approximate search in cloud computing. *The Computer Journal*, 56(5):1–14, May 2013.

[13] Seny Kamara and Kristin Lauter. Cryptographic cloud storage. In *Financial Cryptography and Data Security*, pages 136–149. Springer, 2010.

[14] Jin Li, Qian Wang, Cong Wang, Ning Cao, Kui Ren, and Wenjing Lou. Fuzzy keyword search over encrypted data in cloud computing. In *Proceedings of the IEEE International Conference on Computer Communications*, INFOCOM '10, pages 1–5, 2010.

[15] Peter Linz. *An introduction to formal languages and automata*. Jones & Bartlett Publishers, 2011.

[16] Mateo Meier. Prism expose boosts swiss data center revenues. http://www.dailyhostnews.com/prism-expose-boosts-swiss-data-center-revenues, July 2013.

[17] Andrea Peterson. Nsa snooping is hurting us tech companies' bottom line. http://www.washingtonpost.com/blogs/wonkblog/wp/2013/07/25/nsa-snooping-is-hurting-u-s-tech-companies-bottom-line/, July 2013.

[18] Raluca Ada Popa, Catherine 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*, pages 85–100. ACM, 2011.

[19] Deepak Ravichandran and Eduard Hovy. Learning surface text patterns for a question answering system. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, pages 41–47. Association for Computational Linguistics, 2002.

[20] Eric W. D. Rozier, Saman Zonouz, and David Redberg. Dragonfruit: Cloud provider-agnostic trustworthy cloud data storage and remote processing. In *Proceedings of the 19th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC 2013)*, PRDC '13, 2013.

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

[22] Ja Thom and J Zobel. A model for word clustering. *Journal of the American Society for Information Science*, 43(9):616–627, 1992.

[23] Cong Wang, Kui Ren, Shucheng Yu, and Karthik Mahendra Raje Urs. Achieving usable and privacy-assured similarity search over outsourced cloud data. In *Proceedings of the IEEE International Conference on Computer Communications*, INFOCOM '12, pages 451–459, 2012.

[24] Sun Wu and Udi Manber. Fast text searching: allowing errors. *Communications of the ACM*, 35(10):83–91, 1992.