SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 0000; **00**:1–21
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe

RESeED: A Secure Regular-Expression Search Tool for Storage Clouds

Mohsen Amini Salehi[1]*, Thomas Caldwell, Alejandro Fernandez, Emmanuel Mickiewicz, Eric W. D. Rozier[2], Saman Zonouz[3], and David Redberg

[1] *HPCC lab., School of Computing and Informatics, University of Louisiana at Lafayette, LA, 70503.*
[2] *Department of Electrical Engineering and Computing Systems at University of Cincinnati, OH, 45220.*
[3] *Electrical and Computer Engineering Department, at Rutgers University, NJ, 08854.*

SUMMARY

Lack of trust has become one of the main concerns of users who tend to utilize one or multiple Cloud providers. Trustworthy Cloud-based computing and data storage require secure and efficient solutions which allow clients to remotely store and process their data in the Cloud. User-side encryption is an established method to secure the user data on the Cloud. However, using encryption, we lose processing capabilities, such as searching, over the Cloud data. In this paper, we present RESeED, a tool that provides user-transparent and Cloud-agnostic regular-expression search functionality over encrypted data across multiple Clouds. Upon a client's intent to upload a new document to the Cloud, RESeED analyzes the document's content and updates its data structures accordingly. Then, it encrypts and transfers the document to the Cloud. RESeED provides the regular-expression search functionality over encrypted data by translating the search queries on-the-fly to finite automata and analyzing concise and secure representations of the data before asking the Cloud to download the encrypted documents. RESeED's parallel architecture enables efficient search over large-scale (and potentially big data scale) data-sets. We evaluate the performance of RESeED experimentally and demonstrate its scalability and correctness using real-world data-sets from `arXiv.org` and IETF. Our results show that RESeED produces accurate query responses with a reasonable ($\simeq$6%) storage overhead. The results also demonstrate that for many search queries, RESeED performs faster in compare to the `grep` utility that functions on unencrypted data.
Copyright © 0000 John Wiley & Sons, Ltd.

Received ...

KEY WORDS: Cloud storage; security; privacy; searchable encryption; regular-expression.

1. INTRODUCTION

Cloud providers offer scalable storage solutions to users and relieve the burden and costs of managing a data center. In spite of the advantages offered by storage Clouds, there is an increasing concern over the confidentiality of the user data in these environments [1–8]. A proven solution to the confidentiality concerns is the user-side cryptographic techniques for the data [9]. However, such techniques limit the Cloud users from several aspects. One limitation is that the cryptographic techniques usually are not transparent to end-users [10, 11]. More importantly, these techniques restrict functionalities such as searching and processing the users' data [12]. Although there are solutions for searching over encrypted data (*e.g.,* [13]), they do not scale for complicated search queries, such as regular-expression-based queries.

*Correspondence to: School of Computing and Informatics, University of Louisiana at Lafayette, LA, Email: amini@louisiana.edu

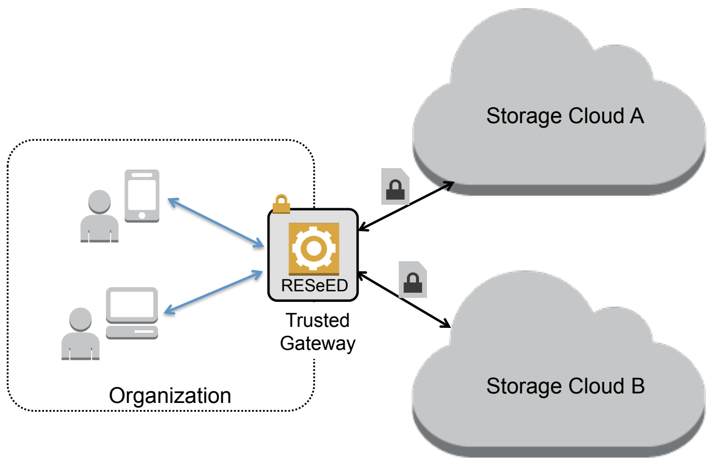


Figure 1. Motivating Scenario: An organization that utilizes multiple Clouds to store its sensitive data. The organization uses a trusted gateway that facilitates transparent access to the Clouds and provides regular-expression based search functionality for the end-users.

Our motivation, in this research, is an organization that owns an increasing volume of data documents (also termed *documents* in this paper) with sensitive information. For the sake of the fault tolerance and availability, the organization tends to utilize multiple Clouds (a.k.a. multi-Cloud [2, 14]) to store the documents. An example of such organization is a hospital that owns patients' health records. Another example, is a bank that keeps clients' credit history reports. The organization requires security for the documents and the ability to flexibly search their contents within a short time. It also expects user-transparency, meaning that the users should not be involved in any details of the encryption, decryption, storing, and searching data from multiple Clouds.

To address the above-mentioned requirements, our collaborators at Fortinet Ltd.[†] are working on a trusted user-side gateway that transparently encrypts all the data before transferring them to a destination Cloud (see Figure 1). The gateway requires a scalable system to flexibly search (particularly, regular-expression-based search) on the encrypted stored data.

Existing techniques for searching on encrypted Cloud data fall short in two primary ways. *First*, for real-world deployment, they require the Cloud provider cooperation to implement their algorithms [15] that could be a significant barrier in practice [16, 17]. *Second*, they mostly concentrate on the simplest categories of search, i.e., keyword search (*e.g.,* [18, 19]). Although recent advances have extended the capability of encrypted keyword search, allowing searches for ranges, subsets, and conjunctions [20, 21], one powerful tool which has remained elusive is the ability to apply regular-expression-based search on encrypted data. A solution using current methods (*e.g.,* [22]) remains impossible in practice due to the exponential explosion of the space required for the storage of the resulting ciphertexts.

In this paper, we present RESeED, a system that offers a scalable and user-transparent functionality for regular-expression search over encrypted documents stored in multiple Clouds. RESeED achieves this objective without any need for the Cloud provider to cooperate, or change their infrastructures (*i.e.,* it is Cloud-agnostic). The search system resides on the trusted gateway and provides a lightweight user interface that is accessible from users' terminals, such as laptops or smart-phones (see Figure 1). RESeED clients are able to upload data files (*i.e.,* documents) to a storage Cloud, remotely search for encrypted documents using regular-expressions, and download the documents that matches these queries.

[†]`http://www.fortinet.com/`

To the best of our knowledge, RESeED is the first solution to provide regular-expression searchability over the encrypted data. It is also the first system that makes use of two search techniques simultaneously, namely, database techniques [23], such as local indexing, and cryptographic techniques, such as symmetric encryption [24], to satisfy the requirements of deployment efficiency. RESeED is user-transparent and implements the necessary encryption and decryption steps in the background so that the users see only plain-text data on their terminals.

As the size of data is rapidly increasing and forming big data scale data-sets hosted on the Cloud (*e.g.,* [25–28]), RESeED needs to be scalable so that it can maintain its efficiency with the increasing size of the data-sets. For this purpose, in this research, we have proposed and implemented a parallel architecture for RESeED that enables it to perform the regular-expression search on large-scale data-sets in a short time.

Experimental results, generated from a working prototype of RESeED on real-world data-sets, demonstrates the correctness and performance of the proposed method. In summary, the contributions of this research are the following.

- We introduce a novel solution which processes regular-expression based searches over encrypted data. Our solution operates based on two novel data structures. We process searches over these data structures using algorithms to transform the query into a domain covered by our new data structures.
- We propose the parallel architecture of RESeED that enables a scalable regular-expression search for large-scale data-sets.
- We implement the first practically deployable solution to the search over encrypted data problem and evaluate it on real-world data-sets.
- We demonstrate the efficacy, scalability, and correctness of our results on a real setting scenario. Demonstrating the low overhead incurred, even when compared with existing solutions, and demonstrating efficacy, in some cases beating the run time of existing tools for regular-expression search over unencrypted data.

This paper is organized as follows. Section 2 reviews the most recent work in the literature, and establishes the need for our solution. Section 3 presents the algorithms used by RESeED for regular-expression search on encrypted data. Then, Section 4 describes the overall architecture of the system. Section 5 discusses the details on the Cloud agnosticism and multi-Cloud features of RESeED which is then followed by Section 6 that provides the security analysis of RESeED. Section 7 presents the parallel architecture of RESeED and Section 8 presents the empirical results using real-world data-sets. Finally, Section 9 concludes the paper and lays out our plan for future work.

2. RELATED WORK

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 the encrypted data [13, 19, 29, 30]. In storage Clouds, in particular, making use of encryption techniques ensures that user privacy is not compromised by the Cloud providers [31] and has established the definition of *trustworthy Clouds*. However, such solution eliminates any remote data processing capabilities, such as searching or database queries.

The solutions for keyword searches over a set of encrypted documents [22] are critical for privacy preserving Clouds and are rapidly growing in importance given the recent users private data disclosure [32]. Current solutions for searches over encrypted data are categorized into two main groups. The *first* group of techniques make use of novel techniques from database research [23], 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.

Second, cryptographic algorithms, initiated by the seminal work by Boneh et al. [22], 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.

Searchable encryption has been extensively studied in the context of cryptography [24, 30, 33], mostly from the perspectives of efficiency improvement and security formalization. Searchable encryption was first described in [30], 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 [33], Goh proposed to use Bloom filters to create per-document 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 [20], however, the results prove impractical, requiring ciphertext and token sizes of the order $O(2^{nw})$ where n is the number of documents and w is the number of tokens in the documents.

Wang *et al.,* [34] 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 symbolic tree-based searching index that allows for similarity search with a constant time complexity. In other research [19], 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.,* [29] 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 [35] 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.

A searchable encryption scheme [36] 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 Boneh *et al.,* [22], 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.,* [30] 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 [13]. In another work [20], 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.

Another direction of research involves the use of homomorphic encryption [37] to further obscure data from the cloud. The encryption method can be used to obscure information, such as keyword frequency, while still being able to use them in the mathematical calculations for searching without decrypting them. In [38] an exclusive-or (XOR) homomorphism encryption approach is proposed to support secure keyword searching on encrypted data on cloud storage. The approach provides a new data protection method by encrypting the keyword and randomizing it by applying XOR operation with a random bit-string for each session to protect access pattern leakage. The approach can reduce data leakage to service provider due to the homomorphic keys. However, the approach remains impractical for large-scale data-sets due to its time complexity. In addition, it still remains in the area of keyword-based search and not regular expressions.

In our previous works [39, 40], a preliminary version of regular expression search system for encrypted data was provided. The current study extends the preliminary system in four main directions. First, it provides an architecture to make the system scalable for big datasets. Second, it enables a transparent and secure search across multiple clouds. Third, we develop a detailed security analysis for RESeED. Fourth, we implement the system within the Fortivault gateway.

3. 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 operates based on two data structures that require the storage overhead in the order of $O(nw)$. The following subsections elaborate on how a regular-expression is searched against a set of encrypted documents.

3.1. Alphabet Division

The key to our method 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 method works for any language composed of strings from $S_C \in 2^C$ 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 (also termed *tokens* in this paper) that users tend to search. Based on a survey of the uses of regular-expressions in practice [41–44], given an alphabet where such a division is possible, searches for phrases in the language most often take this form. This appears to be the case even for languages other than natural languages (*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). However, we focus on the application of this method to a natural language within this paper.

It is worth noting that 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, in our implementation, we define the division by allowing users to define the set Ω, and then defining $C = \Sigma \setminus \Omega$.

3.2. 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 tokens. The second is to create a fuzzy representation of the order of the strings in the document, by hashing each token to a code of b 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 tokens found in all documents, and indicating the documents in which they can be found. The column store can be generated using the PEKS token for each keyword, as described in [22].

For each document, denoted 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 b bits for each token, with each hash appearing in the same order as the original token appeared in the original document.

Using these data structures, we can perform a search, via the procedure demonstrated in Algorithm 1. Based on this algorithm, the regular-expression r is first converted into a non-deterministic finite automaton (NFA), denoted n_0. Then, in step 3, this automaton is partitioned into a set of NFAs defined by a transformation we call an ω-transformation. It partitions a regular-expression NFA n_0 into a set of NFAs, denoted N', by operating on transitions on symbols in Ω. The result of ω-Transformation for the example of $back[\backslash s]^?pack$ regular-expression is shown in Figure 2. As we can see in this example, based on the appearance of Ω, the regular-expression NFA (n_0) is decomposed to three NFAs (called A, B, and C). In this example, $\Omega_1 \in \Omega$ represents a

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 document, and each NFA in $N'$, initially all 0
 5:     for $n'_i \in N'$ do
 6:         $d_i \leftarrow$ NFATODFA($n'_i$)
 7:         for token $\in \Xi$ do
 8:             if $d_i$ ACCEPTS token then
 9:                 $F \leftarrow$ FILES(token)
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:     $\Phi \leftarrow$ FILES(marking)
17:     for $f_i \in \Phi$ do
18:         $O_i \leftarrow$ GETORDERSTORE($f_i$)
19:         if $P$ ACCEPTS $O_i$
20:         then matches $\leftarrow$ matches $\cup$ ($f_i$)
21:     end for
22:     return matches
23: end procedure
```

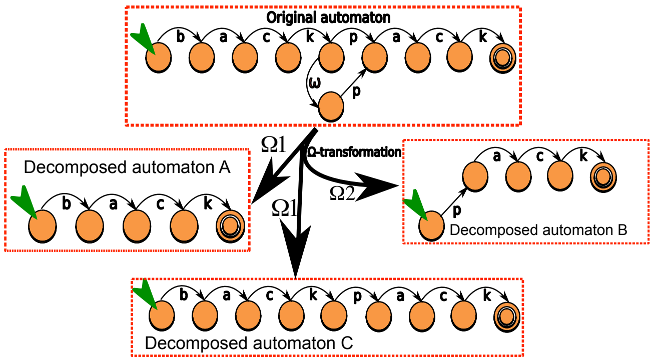


Figure 2. Example of an ω-transformation on an automaton for the regular-expression $back[\backslash s]^?pack$. The regular expression is decomposed to three NFAs namely, A, B, and C, based on $\Omega_1 \in \Omega$ that represents a separator for start of the regular expression and $\Omega_2 \in \Omega$ which represents for $[\backslash s]^?$ (shown as ω) in the original regular expression.

separator for start of the regular expression and $\Omega_2 \in \Omega$ represents for $[\backslash s]^?$ in the original regular expression. Details of ω-transformation algorithm is explained in subsection 3.3.

Once N' is obtained, for each token in the column store, Ξ, we check for containment in each NFA (steps 7 to 9 in Algorithm 1). We maintain a bit matrix, `marking`, that indicates for each document, if a token matched any NFA in N'. Once this matrix is obtained in step 10, we check the `marking` for each document, in step 14, to see which NFAs are satisfied in each document.

Although `marking` matrix determines which documents contain matching tokens with the search query, it cannot determine if the tokens are appeared in the expected order in the documents or not. To verify whether a series of NFAs satisfy r (*i.e.,* if the tokens are appeared in the expected order), an additional structure, called *path-NFA*, is created that operates on the order store of the

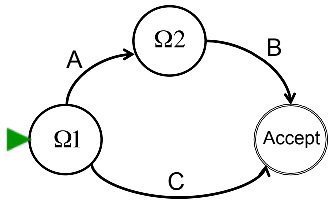


Figure 3. Path-NFA for the regular-expression $back[\backslash s]^{?}pack$. Transitions refer to the NFAs constructed in Figure 2.

documents in `marking`. The path-NFA, denoted P, is a finite automata that acts on NFAs in N' for transitions and accepts if those transitions represent a valid path through n_0. That is, path-NFA P is used to assure that a given document contains the necessary tokens to form a path from the start state of n_0 to one of the original accepting states.

The path-NFA has the same start and final states as n_0. For the example of $back[\backslash s]^{?}pack$ regular-expression (in Figure 2) that has three NFAs A, B, and C, the path-NFA is shown in Figure 3.

We then obtain the order store, O_i, for each document $f_i \in \Phi$, in step 18, where such a path could exist, and check to see if there exists a string in O_i that is accepted by P. If there is, then we indicate a match and mark the matching encrypted document as part of the set of documents which contain a match for the original regular-expression r (step 20 in Algorithm 1).

3.3. ω-Transformation Algorithm

To match a regular-expression against the tokens in the column store, a transformation on the original non-deterministic finite automaton (NFA), n_0, is required. The ω-transformation generates a set of automata, N' from n_0 by operating on the transitions labeled with elements of Ω in n_0. Intuitively, the 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.

Algorithm 2 shows the pseudo code for ω-transformation. Based on this algorithm, each state in the original NFA with an incoming transition labeled with a symbol from Ω is added to the set of start states (steps 9 to 14 in Algorithm 2). Then, each state with an outgoing transition from Ω is added to the set of accepting states (steps 15 to 19 in Algorithm 2). In the next step, we remove from n_0 all transitions labeled with symbols from Ω for each start state (step 21) and generate a new NFA from those states reachable from the new start state, as shown in Figure 2.

4. RESeED ARCHITECTURE

4.1. Overview

The modular design of the whole RESeED system is demonstrated in Figure 4. The User Interface (also termed UI) provides all the functionalities of the RESeED to different types of terminals (*e.g.,* smart phones, laptops, etc.). It also interacts with the internal components of the system.

The *Cloud Allocator* module includes the Cloud selection policy and decides the storage Cloud provider in a multi-Cloud scenario to upload a given document. The *RESeED core* receives the search query from the UI and carries out the main search functionality based on the method described in Section 3. This includes the production and translation of all automata and takes care of creating and updating the column store and the order store. Detailed architecture of the RESeED core is described in the Subsection 4.2.

Storage Cloud Interface provides the abstraction layer to access various Clouds. To add the ability of working with a new Cloud, this interface should be implemented for that Cloud. We have already

Algorithm 2 ω-Transformation

```
procedure ωTRANSFORMATION($n_0$)
    Let $T_{n_0}$ be the transitions in $n_0$
    Let $S_{n_0}$ be the states in $n_0$
    Let $\Omega$ the set of separators
    $S_{\text{Start}} \leftarrow \emptyset$
    $T_\Omega \leftarrow \emptyset$
    $N' \leftarrow \emptyset$
    for Each state $s_i \in S_{n_0}$ do
        for Each incoming transition of $s_i$, $t_j \in T_{n_0}$ do
            if $t_j$ has label $l_k \in \Omega$ then
                Add $s_i$ to $S_{\text{Start}}$
                Add $t_j$ to $T_\Omega$
            end if
        end for
        for Each outgoing transition of $s_i$, $t_j \in T_{n_0}$ do
            if $t_j$ has label $l_k \in \Omega$ then
                set $s_i$ to an accepting state
            end if
        end for
    end for
    $T_{n_0} \leftarrow T_{n_0} \setminus T_\Omega$
    for each $s_i \in S_{\text{start}}$ do
        generate the automaton $n'_i$ from the reachable states of $s_i$
        if $s_i$ is an accepting state
        then remove $s_i$ from accepting states of $n'_i$
        Add $n'_i$ to $N'$
    end for
    return $N'$
end procedure
```

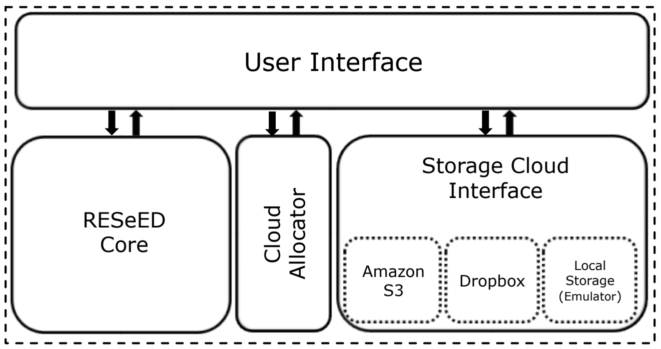


Figure 4. Modular Overview of the RESeED System.

implemented the interface for Dropbox[‡] and Amazon S3[§] storage Clouds. Also, for development and testing purposes, we have implemented the interface in an emulated mode that stores documents on the local storage.

‡`http://www.dropbox.com/`
§`http://aws.amazon.com/s3/`

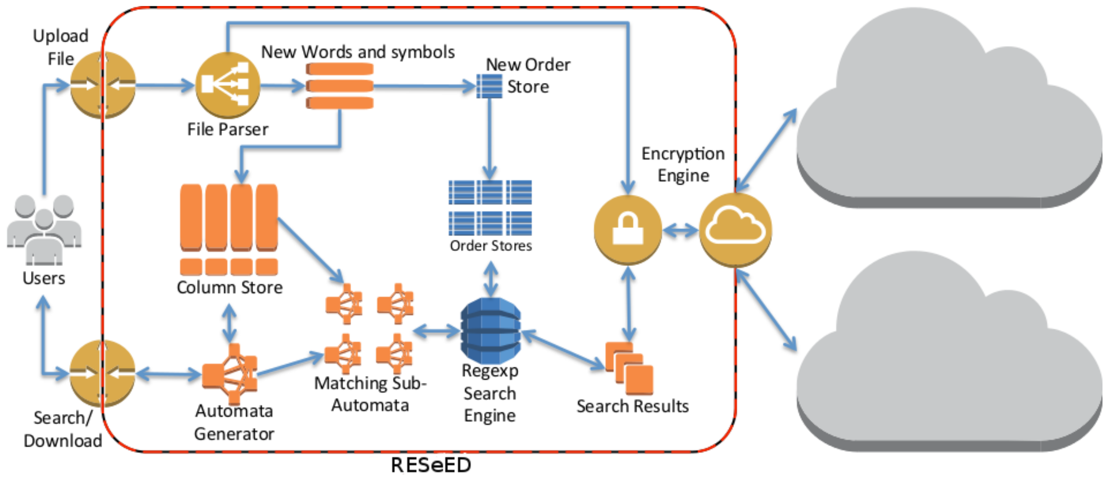
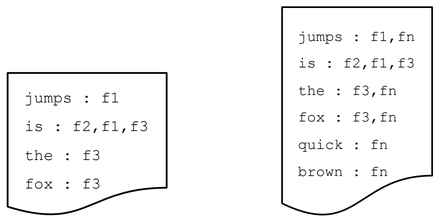


Figure 5. High-level Architecture of the RESeED Core.

4.2. *RESeED Core*

The high-level architecture of the RESeED core is illustrated in Figure 5. As we can see in this figure, the RESeED core has two main functionalities namely, uploading a new document (described in Subsection 4.2.1) and searching and downloading results (described in Subsection 4.2.2).

4.2.1. Uploading a New Document When a new document F is uploaded, its tokens are extracted by the document parser and the column store is updated accordingly. That is, for each extracted token, the column store is searched to see if the token already exists or not. If not, then the token and the document name are appended to the column store. In case the token already exists in the column store, we just insert the document name to the list of documents where the token has appeared. The current version of RESeED accepts the `pdf` and `txt` document formats for upload. For example, let the column store be as shown in Figure 6(a), before a new document fn is uploaded. Also, let the content of document fn as follows: `The quick brown fox jumps`. Then, the updated column store is as shown in Figure 6(b).



```
jumps : f1
is : f2,f1,f3
the : f3
fox : f3
```

(a) Initial column store

```
jumps : f1,fn
is : f2,f1,f3
the : f3,fn
fox : f3,fn
quick : fn
brown : fn
```

(b) Updated column store

Figure 6. Column store is updated with new tokens upon uploading a new document to a Cloud provider.

In the next step, all the tokens of the new document are hashed to create the order store. In our implementation, the tokens are hashed using the SHA-1 [45] hash function. We take the first b bytes[¶] of the 20 bytes produced by the SHA-1 and discard the remainder. The string of these resulting bytes produced for a given document F creates the order store that contains the hashes of all of the tokens contained in F in the order that they appear.

[¶]In our experiments, $b = 3$.

It is worth noting that RESeED also provides an off-line upload mode that is appropriate to index large-scale batch of documents that have been transferred through channels other than the RESeED.

Once the column store and order store are updated, the document is encrypted and uploaded to the appropriate storage Cloud through the Storage Cloud Interface component of the RESeED system (see Figure 4).

4.2.2. Searching and Downloading Results Contents of the documents uploaded to the Cloud are searched based on the user search query. The queries are constructed using regular-expressions formatted with the syntax shown in Table I. Queries are processed based on the conversion of the regular-expression to an NFA, which is then partitioned to a set of NFAs based on ω-transformation to match sub-expressions. After that, matching against order store occurs based on path-NFA (see Section 3.2).

Table I. Regular-expression symbols used in this paper and their meanings.

Symbol	**Definition**	**Example**
`.`	any character; Character . is shown as `\.`	`x.y` matches `xay`
`*`	zero or more repetition of a character	x* matches `xxx`
`+`	one or more repetition of a character	x$^+$ matches `xxx`
`?`	zero or one repetition of a character	x$^?$ matches `x`
`|`	OR statement	`x|y` matches `x` and matches `y`
`\s`	any delimiter character	`x\sy` matches `x y`
`\w`	any alphabetic character	`\w` matches any letter in `a-z` and `A-Z`
`\d`	any numeric character	`\d` matches any letter in `0-9`

In the implementation, each NFA is represented by a transition table given by a square adjacency matrix, and a start and end state.

Once the list of documents that match the input search query are found, they are displayed to the user. These documents can then be downloaded and decrypted for the user.

It is noteworthy that due to the nature of the implementation, the language of this program differs slightly from the `grep` utility regular-expression search. In fact, our current implementation is similar to the `grep` utility with `w` flag. Table I describes the regular-expression language we use in the RESeED system.

5. CLOUD AGNOSTICISM AND MULTI-CLOUD

In addition to search over encrypted data, RESeED provides an abstraction layer that offers *user-transparency* to the users. That is, users do not need to deal with the complexities of encrypting documents and storing/retrieving them to/from multiple Cloud providers. RESeED allocates documents to different Clouds based on a policy called the *Cloud selection policy*. The goal of this policy could be increasing the security or availability of the documents. Currently, the Cloud selection policy is set to minimize the storage cost by utilizing the storage Clouds based on the storage cost. As a future work, we are planing to extend the selection policy to increase the security and availability of the users' documents.

The search functionality of RESeED is independent of the Cloud provider architecture (*i.e.,* it is *Cloud-agnostic*) and does not require any adaptation or change in the Cloud provider infrastructure. The RESeED system just requires primitive operations (*e.g.,* upload, download, and delete) from a Cloud provider. The implementation of RESeED is pluggable so that any new Cloud storage can be plugged into that with a minimum implementation effort. RESeED currently supports Dropbox, Amazon S3 storage Clouds, and the local storage as an emulator used for testing purposes.

RESeED's search functionality is *location-independent* (*i.e.,* documents can be searched regardless of their storage location). Additionally, documents can be moved from one Cloud to another without any interruption on the search functionality. The reason for the location-independence is that RESeED search operates based on the column store and order store data structures which are updated when the documents are uploaded to a destination Cloud. In the case of moving/deleting a documents, RESeED updates the tokens in the column store and the location of the documents on the destination Cloud. The order store for the uploaded documents, however, does not have any dependence on the documents location.

6. SECURITY ANALYSIS

Our threat model assumes that adversaries may intend to attack the communication streams between client and the cloud storage. To explain what exactly the attacker could see or do using our system, we will first provide some definitions.

History: Let D be a collection of documents in the system $\{d_1, d_2, \ldots, d_n\}$. Let q be the regular expressions searched by the client $\{q_1, q_2, \ldots, q_n\}$. A history H_q can then be considered to be the tuple (D, q). In other words, this is a history of searches and interactions between client and server.

View: The view is whatever the cloud can actually see during any given interaction between client and the storage cloud. This includes the trapdoor of the file name searched and requested by the client and the collection of encrypted documents E that are uploaded to the storage cloud. Let $V(H_q)$ be this view.

Trace: The trace is the precise information leaked about the history H. For our system, this includes file identifiers that associated with the search results of the trapdoor T. It is our goal to allow the attacker to infer as little information about H_q as possible.

RESeED provides users with a trustworthy architecture to store their confidentiality-sensitive data securely in the Cloud while maintaining their ability to search the data for generic regular expressions. The sole Trusted Computing Base (TCB) in the RESeED framework is the trusted gateway that has access to all the sensitive information such as the column and order stores and the plain text data before the upload. Protected trusted gateway is a reasonable threat model in the real world settings because the user could keep it locally with minimal exposure to external (potentially untrusted) entities. Moreover, the trusted gateway does not violate the goal of this research, which is securing users' data on the cloud.

The View and Trace encompass all that the attacker would be able to see in RESeED. For the sake of this analysis, we assume that the chosen encryption and hashing methods are secure, and so E itself will not leak any information. Similarly, T only shows a listing of hashed files names requested.

An attacker monitoring the process during a search could see the resultant file identifiers that are requested with the given T. This would show an encrypted history as (E, T). However, since the attacker would not be able to discern the query, this data would be of little use.

Despite minimal trusted computing base, adversaries may still intend to intrude RESeED through man-in-the-middle, compromised honest Cloud providers, and untrusted Cloud providers to attack the confidentiality of the user data. In these cases, all the adversary could see is a set of encrypted documents and hence infer or extract users confidential information according to cryptographic encryption primitives. Such attackers can potentially target the integrity of the user data by modifying the content of data in E. However, such attacks can be verified in RESeED as the documents cannot be decrypted at the client side. In fact, we apply symmetric encryption on client's data (AES encryption was used in implementation) with keys stored by the user. In the event that the of encrypted data are altered by an attacker, the data cannot be decrypted at the client's side by the keys.

Additionally, the attackers with control over the Cloud infrastructures could potentially observe the search patterns that the user queries remotely. In particular, they could obtain sequences of document accesses in the Cloud. It is feasible that the intruders could launch a high-level frequency attack through investigation of long-term document accesses. Clearly, this assumes that *i)* the Cloud data are static and not updated dynamically that is rarely the case in practice; and *ii)* the Cloud is used for a long time frequently as such frequency analyses often require a large training data set before they could accurately infer useful information. Our assumption is that the updates on the data files in Cloud are not predictable. However, there are methods (e.g., [46]) that can be used to tackle frequency attacks when the updates are predictable. Also, note that even a successful frequency attack does not enable the attackers to access the plain text data and all the attackers gain is the knowledge of retrieved encrypted documents.

7. SCALABLE RESeED

As the size of data is rapidly increasing and forming big data-scale data-sets hosted on the Cloud (*e.g.,* [25–27]), one challenge is to come up with scalable data processing methods that can maintain their efficiency with the increasing size of the data-sets. To address this challenge, we propose and implement a parallel architecture that enables RESeED to perform the regular-expression search on large-scale data-sets in a short time.

To parallelize RESeED, we can consider its operation as a workflow with two major steps. The first step is the column store matching and the second step is the order store matching. Both of these steps process potentially large data-sets. However, these matching steps are highly prallelizable.

To parallelize matching against column store, we have to modify the way the column store is constructed. For that purpose, we create multiple independent column stores, each one called a *sub-column store*. Then, the tokens in each sub-column store are matched against NFAs in N' in parallel. To obtain the maximum speedup, the number of sub-column stores created should be based on the number of available processing units (p) and the sub-column stores should have zero intersection (*i.e.,* $\Xi = \Xi_1 \cup \Xi_2 \cup \Xi_3 ... \cup \Xi_p$ and $\Xi_1 \cap \Xi_2 \cap \Xi_3 ... \cap \Xi_p = \emptyset$).

Then, for a given token τ, with a hashed value $H(\tau)$, the destination sub-column store is determined based on Equation 1.

$$i = H(\tau) \bmod p \tag{1}$$

Given a hash function that uniformly maps each token to a hashed value, and for a data-set with large number of tokens, we can assure that the number of tokens in all sub-column stores are approximately the same. Therefore, the processing load of matching against the column store is equally distributed across the available processing units.

Once the sub-column stores are constructed, we match the tokens in each sub-column store against NFAs in N' (see Section 3) in parallel. As depicted in Figure 7, each sub-column store is mapped to a processing unit and its tokens are matched against each NFA and the documents that match each NFA are identified. In the next phase, for each document, we merge NFAs matched in different lists. The result of this phase is the list of documents that are matched to all NFAs in N', denoted ϕ, which is fed to the order store matching step.

This method is similar to the MapReduce [18] programming model that is extensively being used for big-data-scale processing. In this case, the input to the Map function is each line of the column store that includes a token and the list of documents, denoted F, where the token is appeared. The Mapper function processes the token by matching it against each $n'_i \in N'$ and if there is a match, it emits key-value pairs in form of $< f_i, n'_i >$ for all $f_i \in F$. Then, the Reduce function unions all n'_i for document f_i that constructs the `marking` matrix.

For the order store matching step, the pool of documents identified as the result of the column store matching are processed on the available processing units. Matching each hash document against the path-NFA (see Section 3) can be accomplished independently on each processing unit. Therefore, we map the corresponding hash documents of elements in ϕ to the available processing units for the order store matching. Finally, the list of documents that include a matching phrase to the search query are shown to the user.

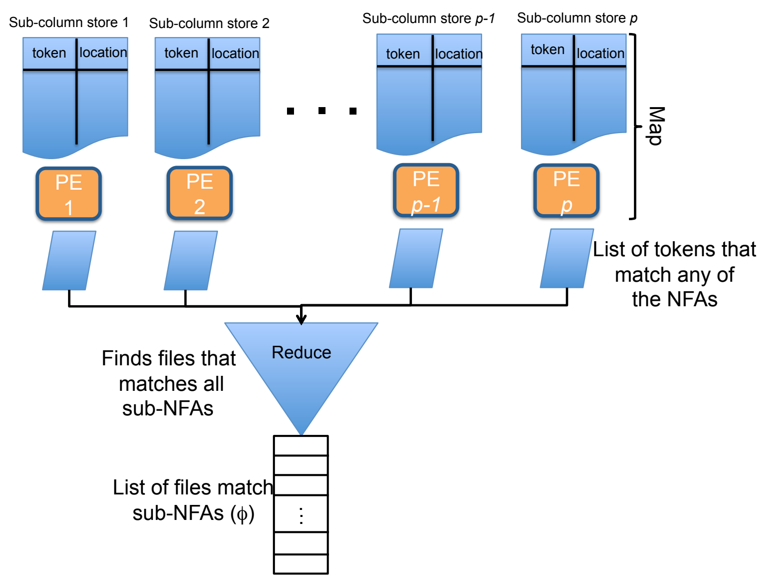


Figure 7. Workflow for Parallel Column Store Matching.

8. EVALUATIONS

8.1. *Experimental Setup*

To experimentally evaluate the performance and correctness of RESeED we tested it on two different data-sets. The first data-set is the Request For Comments[‖] (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 [34]). The second data-set is a collection of scientific papers from the arXiv[**] 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.

We derived a set of ten regular-expressions benchmarks based on those provided by researchers at IBM Almaden [47], 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 8. 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 computationally-heavy regular-expression.

8.2. *Finding the Proper Hash-Width*

The hash-width (*i.e.,* size of the hashed tokens in the order store) can be of any arbitrary size. We used SHA-1 hash function to create order store for documents. For each token in the document, SHA-1 creates a 20-byte-long hashed tokens. However, creating an order store with such hash-width imposes a considerable storage overhead. To reduce this overhead, we take the first b bytes of the 20 bytes produced by the SHA-1 and discard the remainder. In this case, there exists a chance that our search algorithm returns documents that do not contain the regular-expression that we are searching for, resulting in a *false positive* due to the pigeonhole principle. We define the false positive rate

[‖]http://www.ietf.org/rfc.html
[**]http://arxiv.org/

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

(B) All documents 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 documents 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 documents:
`<(\?)`$^?$`(\s)`*`(xml|html)(\s)`$^+$`.`*`(\?)`$^?$`>`

(J) Documents that include any US city, state, and possibly ZIP code:
`(\w)`$^+$`(\s)`*`,(\s)`*`(\w\w)(\s)`*`\d\d\d\d\d`
`(-\d\d\d\d)`$^?$

Figure 8. Regular-expression benchmarks used for the performance evaluations.

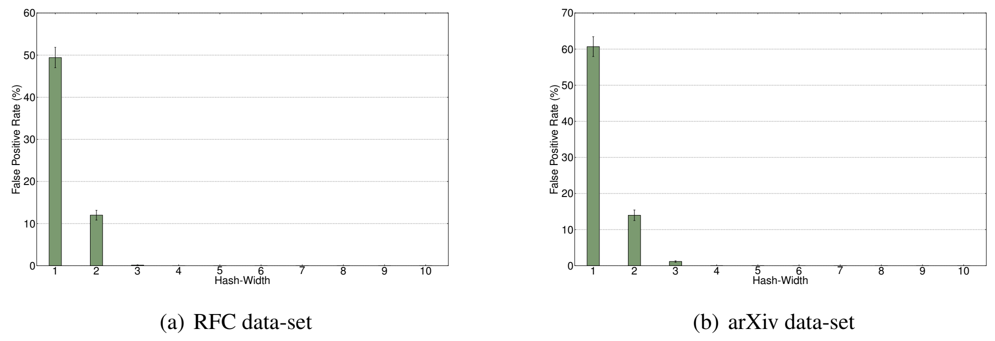
based on Equation 2.

$$Rate = \frac{f}{t} \cdot 100 \tag{2}$$

where f is the number of false positives and t is the total number of tokens found. While a smaller hash-width results in a more compact order store, it also results in a higher false positive rate.

We experimentally evaluated the effect of hash-width on the rate of false positives to determine an ideal hash-width for order store. We generated order stores for each token with hash-widths between one and 10 bytes and measured the rate of false-positives for the benchmarks listed in Figure 8. We used the `grep` utility to confirm the observed rate of false positives returned by our method. The result of this experiment for the RFC and arXiv data-sets are illustrated in Figure 9. The horizontal axis shows different hash-widths and the vertical axis shows the mean and 95% confidence interval of false-positive rate across all benchmarks.

As we can see in both Figures 9(a) and 9(b), there is a high false-positive rate for a hash-width of one. However, for a hash-width of two, the false-positive rate drops sharply. We notice that, the drop of the false-positive rate is sharper for benchmarks that have less fuzziness (*i.e.,* contain specific words) such as benchmarks (B), (C), and (D). In contrast, the false-positive rate in benchmarks



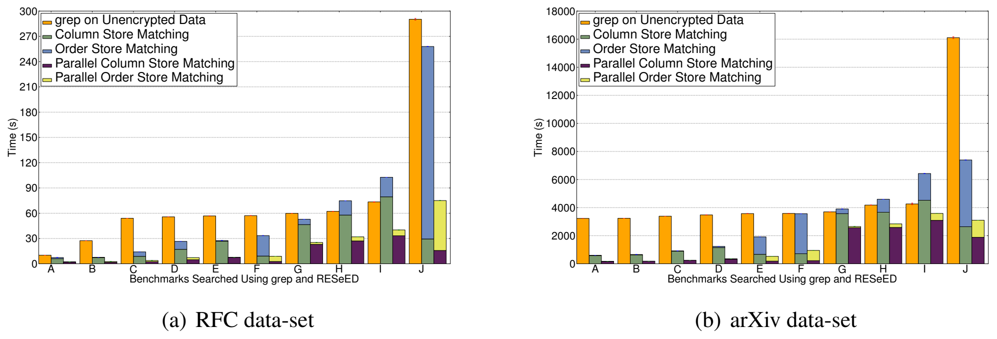
(a) RFC data-set (b) arXiv data-set

Figure 9. False positive rate of benchmarks as a function of hash-width.

which have more fuzziness (*e.g.,* benchmark (G) and (J)) tend to drop slower. The reason is that for such regular-expressions, there are several tokens that can match to the expression and if any of these matching tokens have a hash collision, then it leads to a false-positive. We observe that with a hash-width of three bytes the false-positive rate is close to zero on both data-sets. Hence, for the rest of our evaluations we consider the hash-width of the order store equals to three. We should note that hash-width three provides the best trade-off for the two evaluated datasets. In practice, we recommend to perform some initial tests on the applied dataset to find the efficient hash-width. However, we believe that the hash-width should be more than or equal to three.

8.3. *Evaluation of Regular-Expression Benchmarks*

To demonstrate the efficacy of RESeED, we evaluated its performance to search on the RFC and arXiv data-sets and compared the results against the performance of the `grep` utility [48]. For each regular-expression in the benchmarks listed in Figure 8, we measured the overall search time in RESeED, indicating the time to construct the automata and matching against the column store, and the time to match against the order store. We performed this evaluation on the sequential and parallel RESeED. We also measured the total time that `grep` takes to search the same regular-expression over the unencrypted data-sets. To eliminate any measurement error that can happen due to other loads in the system, we ran each experiment ten times, reporting the mean and 95% confidence interval of the results. However, the confidence intervals are so small that they are not readily visible in many cases in the graph.



(a) RFC data-set (b) arXiv data-set

Figure 10. Time to search for matches for our benchmarks from Figure 8 for the `grep` utility and RESeED.

Figure 10 shows the result of our evaluations using the benchmarks listed in Figure 8. The experiment shows the feasibility of searching complicated regular-expressions on the RFC (Figure 10(a)) and arXiv (Figure 10(b)) data-sets within a limited time. More importantly, the figures show that even though RESeED searches on the encrypted data, it performs faster for benchmarks (A)-(F) than grep. The reason for RESeED outperformance is the fact that it uses the column store first, searching fewer documents compared to grep which scans the whole documents. We note that for the benchmarks that perform longer than grep (benchmarks (G)-(I) in both figures), RESeED have their performance bottleneck on matching against the column store. This justifies the essence of parallel RESeED. Accordingly, we observe that for all of the benchmarks in both data-sets parallel RESeED significantly reduces the search time for all benchmarks. In fact, parallelization resolves the performance bottleneck of RESeED and makes it practically performable even on large-size datasets.

In general, sequential RESeED 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 the generated automata is performed quickly and in the latter case, the number of documents that have to be checked in the order store are few.

In the parallel RESeED, we observe that the parallelization is more effective in evaluating benchmarks that their search times are dominated by the order store matching. In other words, there is less parallelization gain where the bottleneck is the column store matching time. The reason is that in complex regular-expressions, time to match against column store is dominated by the matching the tokens against NFA not by the number of tokens to be matched.

8.4. RESeED Scalability

In this experiment, we investigate how RESeED scales as the size of the data-set increases. We evaluated the performance of the RESeED using the benchmarks listed in Figure 8 with different sample sizes from the arXiv data-set. For that purpose, we constructed data-sets with different sizes from 1 GB to 200 GB as can be seen in the horizontal axis of Figure 11. For each data-set size, we repeated the experiment for ten different samples and reported the mean search time for each benchmark. All of the experiments were executed on the four cores available on our server.

The result of this experiment is shown in Figure 11. The horizontal axis shows different data-set sizes evaluated and the vertical axis shows the time required to search each benchmark.

This experiment demonstrates that RESeED scales almost linearly, as the size of the data-set increases. The reason for this performance is the scalable structure of the column store which enables the effective identification of documents that contain possible accepting paths in the automaton representing our regular-expression. These feature makes our search method highly scalable, and well suited for large data-sets.

As illustrated in Figure 11, the search time for less fuzzy benchmarks (*i.e.,* benchmarks (A)-(E)) increases slower than more fuzzy benchmarks (*i.e.,* (G)-(J)). This is attributed to the extra time required for the column store matching for more fuzzy benchmarks. That is, as the size of the data-set increases, the number of entries in the column store grows and it takes more time to match the whole column store, specially for more fuzzy benchmarks, against a complicated NFA. Therefore, this experiment also signifies the dominance of matching against the column store step and its impact on the scalability of RESeED.

8.5. RESeED Speedup

In this experiment, we evaluated the speedup of parallel RESeED as the number of available processing units increases. For this experiment, we have used the arXiv data set. To access large number of processing units, we ran this experiment on the Pegasus2 [49] high performance computing facilities that we have which includes more than 10,000 X86 cores and 18 TB memory. Each user in this system is allocated up to 16 processing units and 30 GB memory. As Pegaus2 resources are shared amongst different users and there are coexisting workloads, we have executed each experiment ten times to remove any environmental impact and the mean speedup is reported.

The result of this experiment is illustrated in Figure 12. The horizontal axis shows different number of processing units (cores) that we deployed for this experiment and the vertical axis shows the speedup obtained for each benchmark based on different number of processing units. For the sake of readability we have connected the related points with dash lines.

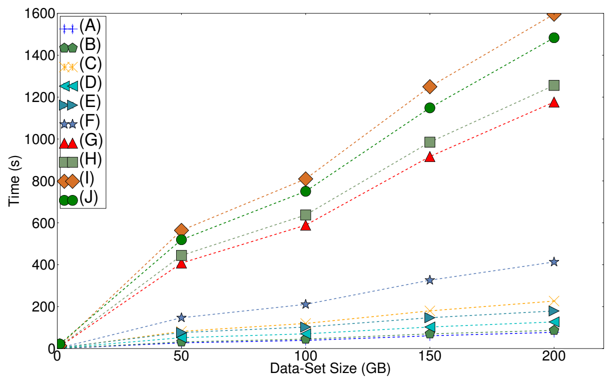


Figure 11. Scalability of RESeED in terms of time to search benchmark regular-expressions as the size of the data-set increases.

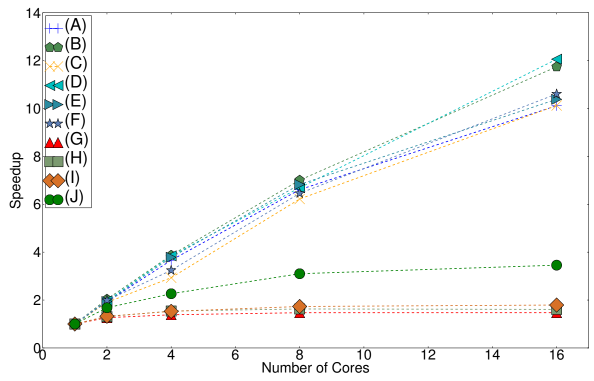


Figure 12. Speedup obtained by searching benchmark regular-expressions on different number of processing units (cores) using parallel RESeED.

As we can see in Figure 12, less fuzzy benchmarks are gaining more speedup as the number of processing units increases. However, due to the time for matching against the column store and order store, more fuzzy regular-expressions (*e.g.,* (G)-(J)) are not getting a high speedup by increasing the degree of parallelism. We observe that within the more fuzzy benchmarks ((G)-(J)), the one that its search time is more dominated by the order store matching (*i.e.,* benchmark (J), as demonstrated in Figure 10) gains a better speedup.

We can conclude that parallelizing RESeED is more effective for less fuzzy regular-expressions. Additionally, we notice that the benchmarks that their search time is heavily depends on the order store matching (*i.e.,* those that have a large document set to check, such as (F) and (J) benchmarks) benefit from parallelization better than those that their search time is dominated by the time to search the column store.

8.6. *Analysis of the Overhead*

RESeED has two main sources of overhead. In this experiment, we measure and analyze these overheads.

The *first* overhead is related to the space complexity of RESeED. To study the space complexity, we investigate the growth of the column store as the size of the data-set increases. For that purpose, we measure the number of tokens stored in the column store as documents are added to the data-set. To measure the imposed overhead accurately, for this experiment, we consider the sequential

RESeED that creates just one column store. In addition, we study the imposed space overhead of order store that is also needed to be stored.

Figure 13 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 300 MB 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 [50]. As can be seen in Figure 13, the increase in the size of the column store includes a linear and a logarithmic component. 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 within 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 (46.7 MB) whereas it is only 2% (5.4 GB) of the arXiv data-set. It is noteworthy that the overhead from the column store is similar to that induced by other searchable encryption techniques which cannot handle regular-expressions, such as those in [20].

The space overhead of the order store is linearly correlated with the size of the data-set. However, recall from Section 8.2 that we store a portion of hashed tokens in the order store that drastically reduces its overhead. Therefore, the exact overhead of the order store with hash-width 3 is 155 MB for the RFC and is approximately 10 GB for the arXiv data-set. We are currently working on the structure of order store to reduce its overhead further. Another data structure that is utilized in RESeED is `marking`. As mentioned earlier, it is an in-memory structure, used to store the file names and the corresponding NFA that they match with. Our measurements show that this structure will not take more than 200 MB for the most complicated benchmark evaluated. This can be well tolerated with the gateway hardware (*i.e.,* Fortivault) in which RESeED is executed.

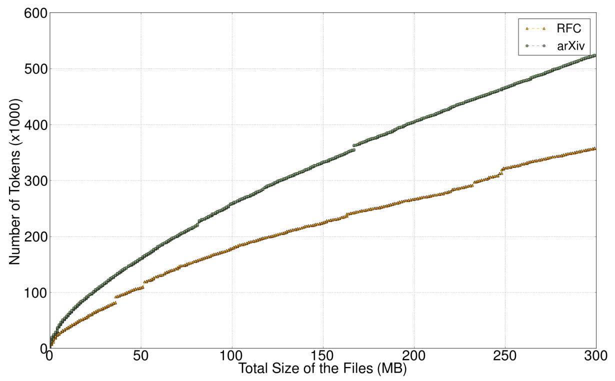
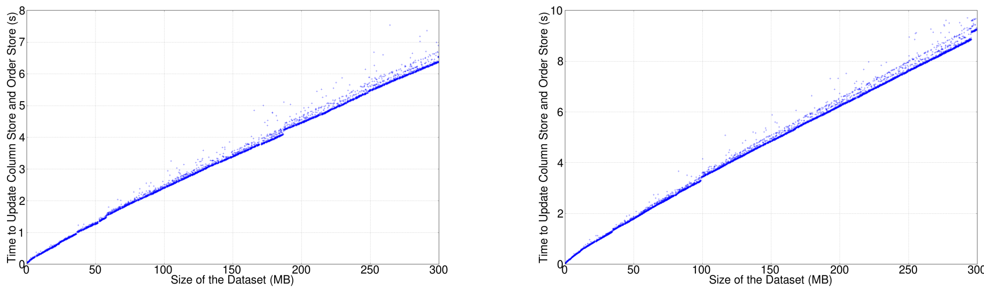


Figure 13. Column store size as a function of data-set size.

The *second* overhead pertains to the time taken to update the column store and to generate new order stores upon the addition of a new document 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. The result of this experiment is shown in Figure 14 for both the RFC (Sub-figure 14(a)) and the arXiv (Sub-figure 14(b)) data-sets. In this experiment, for the sake of comparison, we show the update time for only the first 300 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 document. 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



(a) RFC data-set (b) arXiv data-set

Figure 14. Time to update column store for RFC and arXiv data-sets.

`unordered_set` data structure to store the list of appearances of a token in different documents. 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.

9. CONCLUSIONS AND FUTURE WORKS

In this paper, we presented RESeED, a system that provides capability to process regular-expression based search over encrypted data residing potentially in multiple Clouds. RESeED improves upon current state-of-the-art techniques in the search over encrypted data, by providing a highly scalable, efficient, and accurate solution with low storage and performance overhead. The proposed solution is user-transparent, and cloud-agnostic. To support searching encrypted data over large-scale (and potentially big data scale) data-sets, we also proposed and implemented a scalable version of RESeED that works based on the MapReduce model.

Our experiments with a working prototype of RESeED on real-world data-sets empirically expresses RESeED's deployability and practicality. In particular, for several evaluated benchmarks, we noticed that RESeED can perform search queries on encrypted data faster than the `grep` utility that performs the search on unencrypted data.

In future, we plan to extend this research from different aspects. One interesting future direction is to extend RESeED to search encrypted genome sequences for existence of specific protein sequences. This work will enable biologists to search for particular protein sequences without revealing the whole patient's genome data to them. Another future research direction is to investigate optimal multi-Cloud allocation policies for the user documents in a way that increases availability and security of the data across several Clouds.

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. Preliminary version of portions of this material were presented at the IEEE Cloud 2014 [40] and IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid '14) [39] conferences.

AVAILABILITY

RESeED core is available for download at the following website:

`http://dataengineering.org/aminis/download/fts/FTS`

The manual document on how to run RESeED and its options are available here:

`http://dataengineering.org/aminis/download/fts/man.pdf`

Details of the test data we used for our experiments are available for use in the following address, which allows full recreation of our results.

`http://dataengineering.org/aminis/download/fts/data`

REFERENCES

1. C. Liu, J. Chen, L. T. Yang, X. Zhang, C. Yang, R. Ranjan, K. Ramamohanarao, Authorized public auditing of dynamic big data storage on cloud with efficient verifiable fine-grained updates, IEEE Transactions on Parallel and Distributed Systems 99 (PrePrints). doi:http://doi.ieeecomputersociety.org/10.1109/TPDS.2013.191.
2. Y. Zhu, H. Hu, G.-J. Ahn, M. Yu, Cooperative provable data possession for integrity verification in multicloud storage, IEEE Transactions on Parallel and Distributed Systems 23 (12) (2012) 2231–2244.
3. K. Yang, X. Jia, An efficient and secure dynamic auditing protocol for data storage in cloud computing, IEEE Transactions on Parallel and Distributed Systems 24 (9) (2013) 1717–1726.
4. Q. Wang, C. Wang, K. Ren, W. Lou, J. Li, Enabling public auditability and data dynamics for storage security in cloud computing, IEEE Transactions on Parallel and Distributed Systems 22 (5) (2011) 847–859.
5. A. 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).
6. M. Meier, Prism expose boosts swiss data center revenues, http://www.dailyhostnews.com/prism-expose-boosts-swiss-data-center-revenues (July 2013).
7. S. K. Pasupuleti, S. Ramalingam, R. Buyya, An efficient and secure privacy-preserving approach for outsourced data of resource constrained mobile devices in cloud computing, Journal of Network and Computer Applications 64 (2016) 12 – 22. doi:http://dx.doi.org/10.1016/j.jnca.2015.11.023.
8. R. Fathi, M. A. Salehi, E. L. Leiss, User-friendly and secure architecture (ufsa) for authentication of cloud services, in: Proceedings of the 8th IEEE International Conference on Cloud Computing, CLOUD '15, 2015, pp. 516–523.
9. E. W. D. Rozier, S. Zonouz, D. 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 '13, 2013, pp. 172–177.
10. R. Chow, P. Golle, M. Jakobsson, E. Shi, J. Staddon, R. Masuoka, J. Molina, Controlling data in the cloud: Outsourcing computation without outsourcing control, in: Proceedings of the ACM Workshop on Cloud Computing Security, CCSW '09, 2009, pp. 85–90.
11. F. Hu, M. Qiu, J. Li, T. Grant, D. Taylor, S. McCaleb, L. Butler, R. Hamner, A review on cloud computing: Design challenges in architecture and security, CIT. Journal of Computing and Information Technology 19 (1) (2011) 25–55.
12. B. Wang, B. Li, H. Li, Oruta: privacy-preserving public auditing for shared data in the cloud, IEEE Transactions on Cloud Computing 2 (1) (2014) 43–56.
13. D. Boneh, M. Franklin, Identity-based encryption from the weil pairing, in: Advances in Cryptology, CRYPTO '01, 2001, pp. 213–229.
14. H. Wang, Y. Zhang, On the knowledge soundness of a cooperative provable data possession scheme in multicloud storage, IEEE Transactions on Parallel and Distributed Systems 25 (1) (2014) 264–267.
15. M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, M. Zaharia, A view of cloud computing, Communications of the ACM 53 (4) (2010) 50–58.
16. D. Zissis, D. Lekkas, Addressing cloud computing security issues, Future Generation Computing Systems 28 (3) (2012) 583–592.
17. P. G. Dorey, A. Leite, Commentary: Cloud Computing - A Security Problem or Solution?, Information Security Technical Report 16 (3-4) (2011) 89–96.
18. J. Chen, W. Dou, X. Zhang, Kasr: A keyword-aware service recommendation method on mapreduce for big data application, IEEE Transactions on Parallel and Distributed Systems 99 (PrePrints). doi:http://doi.ieeecomputersociety.org/10.1109/TPDS.2013.2297117.
19. J. Li, Q. Wang, C. Wang, N. Cao, K. Ren, W. Lou, Fuzzy keyword search over encrypted data in cloud computing, in: Proceedings of the IEEE International Conference on Computer Communications, INFOCOM '10, 2010, pp. 1–5.
20. D. Boneh, B. Waters, Conjunctive, subset, and range queries on encrypted data, in: Theory of cryptography, 2007, pp. 535–554.
21. F. Zhangjie, S. Xingming, L. Qi, Z. Lu, S. Jiangang, Achieving efficient cloud search services: multi-keyword ranked search over encrypted cloud data supporting parallel computing, IEICE Transactions on Communications 98 (1) (2015) 190–200.
22. D. Boneh, G. Di Crescenzo, R. Ostrovsky, G. Persiano, Public key encryption with keyword search, in: Advances in Cryptology-Eurocrypt, 2004, pp. 506–522.
23. R. A. Popa, C. Redfield, N. Zeldovich, H. Balakrishnan, Cryptdb: protecting confidentiality with encrypted query processing, in: Proceedings of the 23rd ACM Symposium on Operating Systems Principles, 2011, pp. 85–100.
24. R. Curtmola, J. Garay, S. Kamara, R. Ostrovsky, Searchable symmetric encryption: improved definitions and efficient constructions, in: Proceedings of the 13th ACM conference on Computer and communications security, CCS '06, 2006.

25. E. E. Schadt, M. D. Linderman, J. Sorenson, L. Lee, G. P. Nolan, Computational solutions to large-scale data management and analysis, Nature Reviews Genetics 11 (9) (2010) 647–657.
26. D. Agrawal, S. Das, A. El Abbadi, Big data and cloud computing: Current state and future opportunities, in: Proceedings of the 14th International Conference on Extending Database Technology, EDBT/ICDT '11, New York, NY, USA, 2011, pp. 530–533.
27. J. Zhan, L. Zhang, N. Sun, L. Wang, Z. Jia, C. Luo, High volume throughput computing: Identifying and characterizing throughput oriented workloads in data centers, in: Proceedings of the 26th IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPSW '12, 2012, pp. 1712–1721.
28. W. Sun, X. Liu, W. Lou, Y. T. Hou, H. Li, Catch you if you lie to me: Efficient verifiable conjunctive keyword search over large dynamic encrypted cloud data, in: IEEE Conference on Computer Communications, INFOCOM '15, 2015, pp. 2110–2118.
29. A. Ibrahim, H. Jin, A. A. Yassin, D. Zou, P. Xu, Towards efficient yet privacy-preserving approximate search in cloud computing, The Computer Journal 56 (5) (2013) 1–14.
30. D. X. Song, D. Wagner, A. Perrig, Practical techniques for searches on encrypted data, in: 21st IEEE Symposium on Security and Privacy, S&P 2000, 2000, pp. 44–55.
31. S. Kamara, K. Lauter, Cryptographic cloud storage, in: Financial Cryptography and Data Security, 2010, pp. 136–149.
32. M. Theoharidou, N. Papanikolaou, S. Pearson, D. Gritzalis, Privacy Risk, Security, Accountability in the Cloud, in: Proceedings of 5th IEEE International Conference on Cloud Computing Technology and Science, CLOUDCOM '13, 2013, pp. 177–184.
33. E.-J. Goh, Secure indexes, Cryptology ePrint Archive, report 2003/216 (2003).
URL `http://eprint.iacr.org/`
34. C. Wang, K. Ren, S. Yu, K. M. R. Urs, Achieving usable and privacy-assured similarity search over outsourced cloud data, in: Proceedings of the IEEE International Conference on Computer Communications, INFOCOM '12, 2012, pp. 451–459.
35. G. R. Hjaltason, H. Samet, Properties of embedding methods for similarity searching in metric spaces, IEEE Transactions on Pattern Analysis and Machine Intelligence 25 (5) (2003) 530–549.
36. D. Hofheinz, E. Weinreb, Searchable encryption with decryption in the standard model., IACR Cryptology ePrint Archive 2008 (2008) 423–430.
37. A. López-Alt, E. Tromer, V. Vaikuntanathan, On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption, in: Proceedings of the 44th Annual ACM Symposium on Theory of Computing, STOC '12, 2012, pp. 1219–1234.
38. S. Q. Ren, B. H. M. Tan, S. Sundaram, T. Wang, Y. Ng, V. Chang, K. M. M. Aung, Secure searching on cloud storage enhanced by homomorphic indexing, Future Generation Computer Systems (FGCS)doi:http://dx.doi.org/10.1016/j.future.2016.03.013.
39. M. Amini Salehi, T. Caldwell, A. Fernandez, E. Mickiewicz, D. Redberg, S. Rozier, Eric W. D.and Zonouz, RESeED: A Tool for Regular Expression Search over Encrypted Data in Cloud Storage, in: Proceedings of the 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid '14, 2014.
40. M. Amini Salehi, T. Caldwell, A. Fernandez, E. Mickiewicz, D. Redberg, E. W. D. Rozier, S. Zonouz, RESeED: Regular Expression Search over Encrypted Data in the Cloud, in: Proceedings of the 7th IEEE Cloud conference, Cloud '14, 2014.
41. A. V. Aho, M. J. Corasick, Efficient string matching: an aid to bibliographic search, Communications of the ACM 18 (6) (1975) 333–340.
42. C. L. Clarke, G. V. Cormack, On the use of regular expressions for searching text, ACM Transactions on Programming Languages and Systems (TOPLAS) 19 (3) (1997) 413–426.
43. S. Wu, U. Manber, Fast text searching: allowing errors, Communications of the ACM 35 (10) (1992) 83–91.
44. D. Ravichandran, E. Hovy, Learning surface text patterns for a question answering system, in: Proceedings of the 40th Annual Meeting on Association for Computational Linguistics, Association for Computational Linguistics, 2002, pp. 41–47.
45. X. Wang, Y. Yin, H. Yu, Finding collisions in the full sha-1, in: Advances in Cryptology, CRYPTO '05, 2005, pp. 17–36.
46. P. Williams, R. Sion, B. Carbunar, Building castles out of mud: Practical access pattern privacy and correctness on untrusted storage, in: Proceedings of the 15th ACM Conference on Computer and Communications Security, CCS '08, 2008, pp. 139–148.
47. J. Cho, S. Rajagopalan, A fast regular expression indexing engine, in: Proceedings of the 18th International Conference on Data Engineering, 2002, pp. 419–430.
48. The Open Group Base Specifications Issue 7, `http://pubs.opengroup.org/onlinepubs/9699919799/utilities/grep.html`, accessed: 2014-7-11.
49. Center for Computational Science., `http://ccs.miami.edu/` (Accessed 2014-7-2).
50. J. Thom, J. Zobel, A model for word clustering, Journal of the American Society for Information Science 43 (9) (1992) 616–627.