Privacy Preserving String Pattern Matching On Outsourced Data

Thesis Presentation

Bargav Jayaraman
201207509
MS By Research in CSE
CSTAR Lab
Guide: Dr. Bruhadeshwar Bezawada

Presentation Outline

- Domain Background
- Motivation to Privacy Preserving String Pattern Matching
- System Model and Problem Statement
- Our Approach for Privacy Preserving String Pattern Matching
- Search Optimized Privacy Preserving String Pattern Matching
- Ranking of Search Results
- Experimental Results on Real-world Data sets

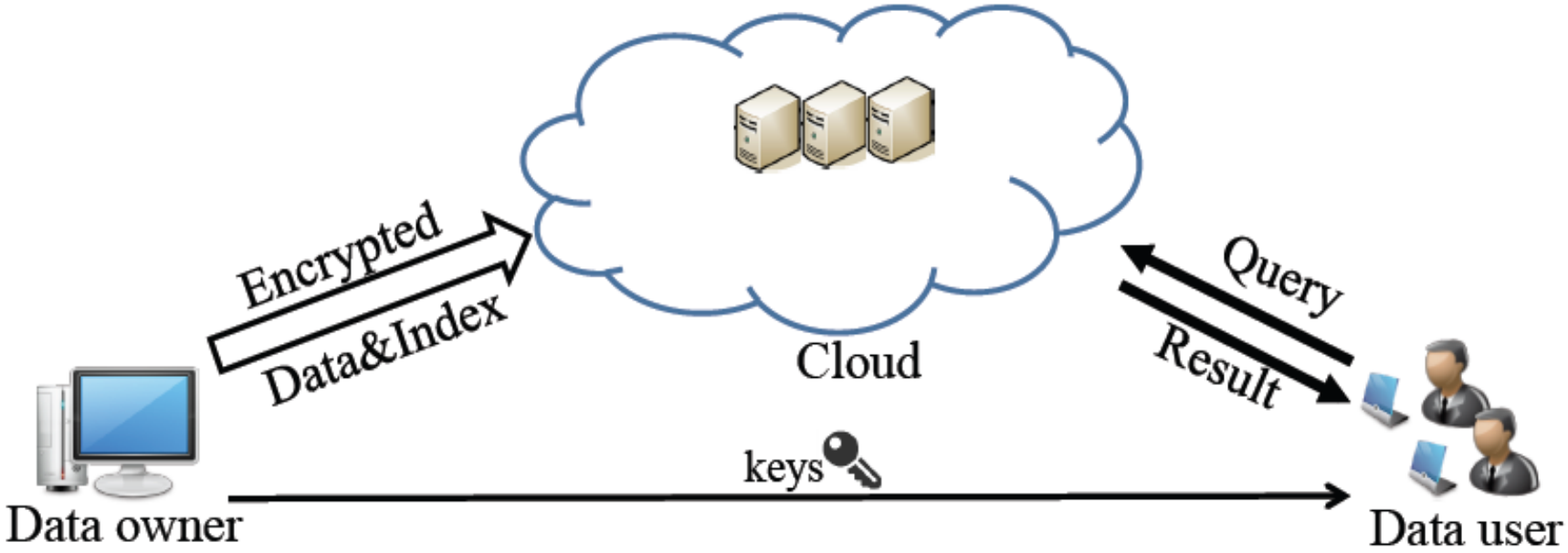
Domain Background

- Cloud storage is pervasive and popular
  - Dropbox; Google Drive ;Microsoft OneDrive ;
  - Apple iCloud; Amazon Simple Storage Service
- Storage format : documents indexed with keyword strings
  - Servers protect files by encrypting them at server-end
- How are documents of interest retrieved?
  - User sends a query string to cloud server
  - Server ***decrypts*** each document and matches the query string against each keyword in the document

- Major Drawback : Data is not private from the server!
  - The query strings reveal a lot of information about the user
  - Server learns the personal profiles of users and uses them for commercial gains like advertising, spam and so on
  - Corrupt employees can steal users' confidential data

Domain Background

- Solution : Data owner encrypts the files and authorizes (if any) users to search on them
- Data owner builds an **encrypted index**, which supports **encrypted queries**



Motivating Privacy Preserving String Pattern Matching

- String pattern matching problem is about checking if a query string ***occurs*** within a keyword

- String pattern matching enhances user search capability significantly by providing advanced search options *(Google™ search engine Advanced search options)*

- ***"terms beginning with these words"*** : When user only knows part of the matching keywords
  - Sample query: Find mobile numbers starting with string "**9940**"
  - Sample results: **9940**5040, **9940**5045, **….**

- ***"terms containing the words"*** : When query string is a sub-string of a keyword
  - Sample query: Find all genes with disease pattern "101"
  - Sample results: 11100**101**001, 100100**101**100100011

Problem Statement

- System Input (to cloud server):
  - Encrypted document set : D = { Enc(D1) ,…, Enc(Dn) } , where Enc is a symmetric key encryption algorithm
  - Note, each document has keywords : D(W) = {w1, w2,…, wm}
  - Encrypted index I, which is built over D(W)
  - An encrypted query string Enc(Tp), from data user

- Output:
  - List of documents: Lr ={ D1,…, Dk}, where string Tp is a *sub-string* of some keyword w in Di(w) for each document in Lr

- Example query string: *"late"*
- Desired output : all documents containing keywords like: *later, ablate, contemplate, plates, elated* … etc.

Adversary Model

- We adopt the Honest-but-Curious (HbC) adversary model for the cloud server and any passive adversaries

- The cloud server is:
  - *honest* in adhering to the communication protocols and the query processing algorithms
  - *curious* to learn additional information about the user by observing the data processed in the search protocol

Security Model

- We aim to achieve "IND-CKA", indistinguishability (or) *semantic security against adaptive chosen keyword attack* on symmetric key encryption algorithms

- In this model: given two document sets D0, D1, the owner builds an encrypted index : Ind_b
  - If b=0, Ind_b is index for D0
  - If b=1, Ind_b is index for D1

- After some chosen keyword queries to Ind_b adversary is challenged to output the value of : b

- This model does not hide number of keywords, documents accessed or the encrypted queries

Limitations of Prior Research

- Public-key based (PKE) approaches, reduce the problem to polynomial evaluation : the encrypted query string is one input and the cipher-text is the other input
  - These methods require multiple rounds of protocol interaction and is computationally expensive, making it impractical in the cloud server domain
- Some PKE methods are:
  - *Baron et al.'s* 5PM for DNA matching
  - *Katz et al.'s* Text processing protocol for DNA matching
  - *Hazay et al.'s* Pattern matching in presence of malicious adversaries
  - *Troncoso et al.'s and Mohassel et al.'s* DNA matching through DFA evaluation

Limitations of Prior Research

- Symmetric-key based (SSE) approaches focused on :
  - Exact keyword matching, which is a sub-set of the problem we are addressing
  - Fuzzy keyword matching using hamming distance errors, which is a variation of the keyword matching problem and tries to correct human errors in entering query keywords
- Some SSE methods are:
  - *Goh's* Bloom Filter Index
  - *Curtmola et al.'s* Inverted Index
  - *Cao et al.'s* Multi-keyword search protocol
  - *Wang et al.'s* Fuzzy keyword search protocol
  - *Vappas et al.'s* Blind Seer
  - *Seny et al.'s* KRB tree

Key Contributions

- **First approach** to support string pattern search in outsourced data under the symmetric encryption model
- Our index structure: Pattern Aware Secure Search Tree – PASStree, implements privacy preserving string pattern matching under the strong IND-CKA security model
- We describe an efficient ranking algorithm, to return the results in a best ranked manner
- Our prototype implementation works over a million key words with search time of few milliseconds

Our Approach

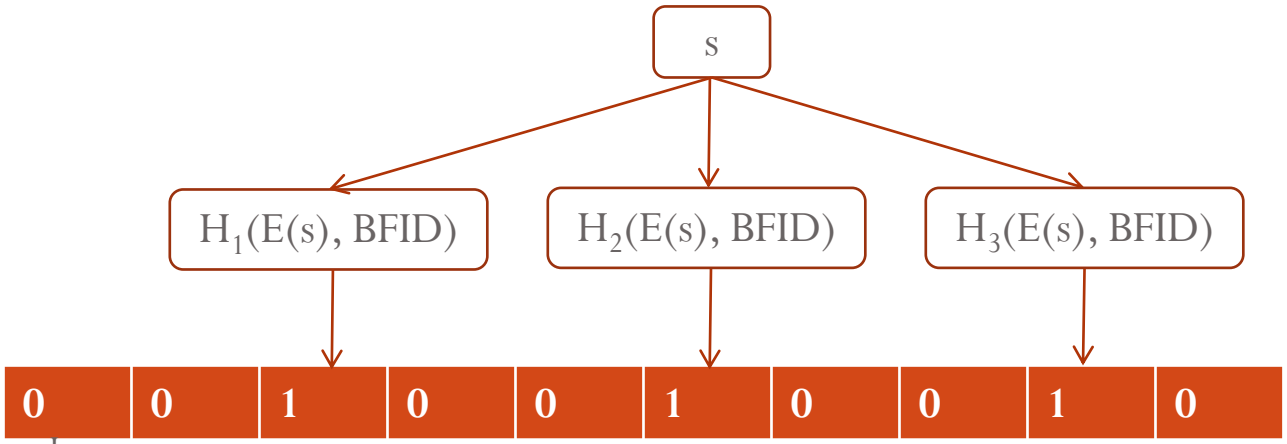
- Basic intuition : If a query pattern matches a keyword, then it implies that the query pattern *must* be a sub-string of the keyword
- We extract every possible sub-string of a keyword, encrypt and store the sub-strings inside a Bloom filter
- The index is the set of Bloom filters, one per keyword

- The problem is reduced to that of exact matching

- To search, the user generates an encrypted query, called *trapdoor*
- If trapdoor is found inside any Bloom filter, then the keyword is returned as match

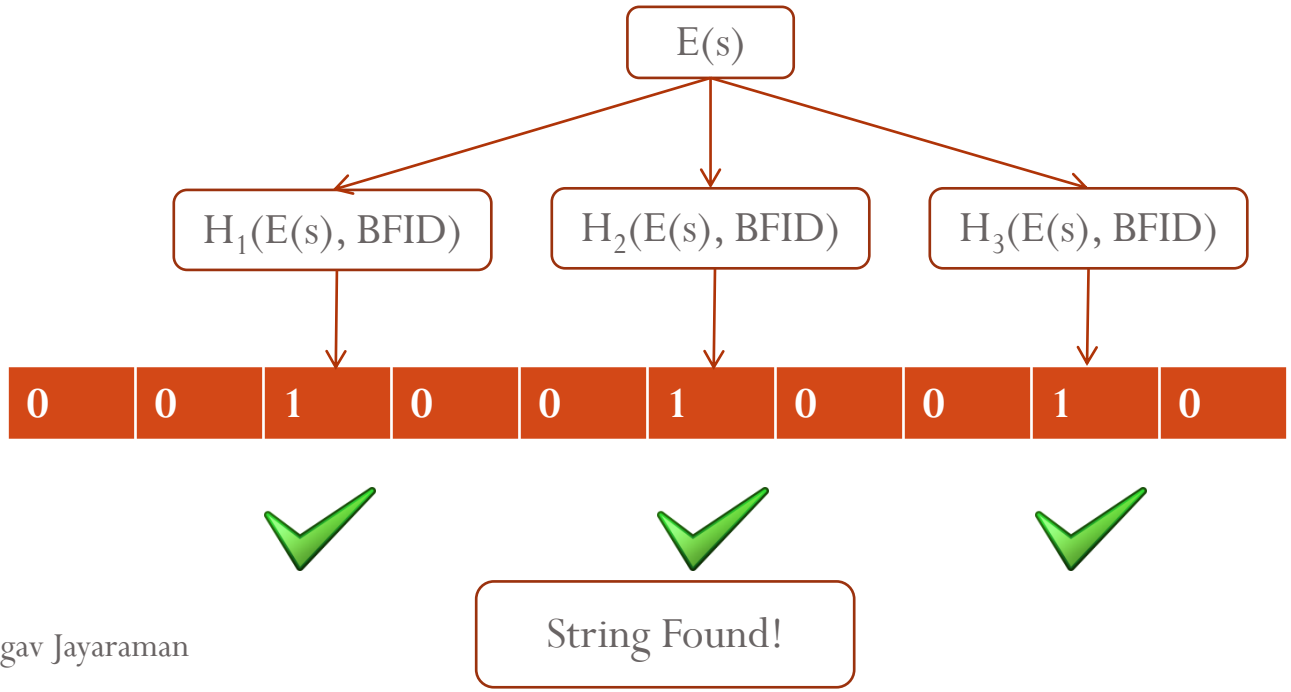
Our Approach : Bloom Filter Storage

- To store keyword: "Ship", extract all possible sub-strings:
  - *"Ship", "hip", "ip", "p", "Shi", "Sh", "hi", "h", "i"*
- A Bloom filter is a bit-array of size M, and has K hash functions which map into the range [0, M-1]
- To store a string in this array: we hash the string with each hash function and set the hash locations to 1
- Using encryption algorithm E we encrypt the sub-string before storage
- Each Bloom filter has a unique ID to provide randomness



Our Approach : Query Search

- To search a query string ‘s’ in a Bloom filter ID with BFID the user generates trapdoor: E(s)
- The cloud server hashes the trapdoor for each Bloom filter and reports a match if all positions are set to 1



Preventing Attacks on Bloom Filters

- Since Bloom filters are stored on cloud server, the adversary can try to learn about the contents in several ways

- First, the common bit locations can be used to infer common keywords across Bloom filters
  - We prevent this by using a random Bloom filter ID and ensuring that the same keyword is hashed into different locations

- Second, the number of bits in the Bloom filter can leak the number of strings stored
  - To prevent this, we add additional padding bits to each Bloom filter such that two Bloom filters with different number of keywords have nearly same number of 1s.

Technical Challenges in Basic Approach

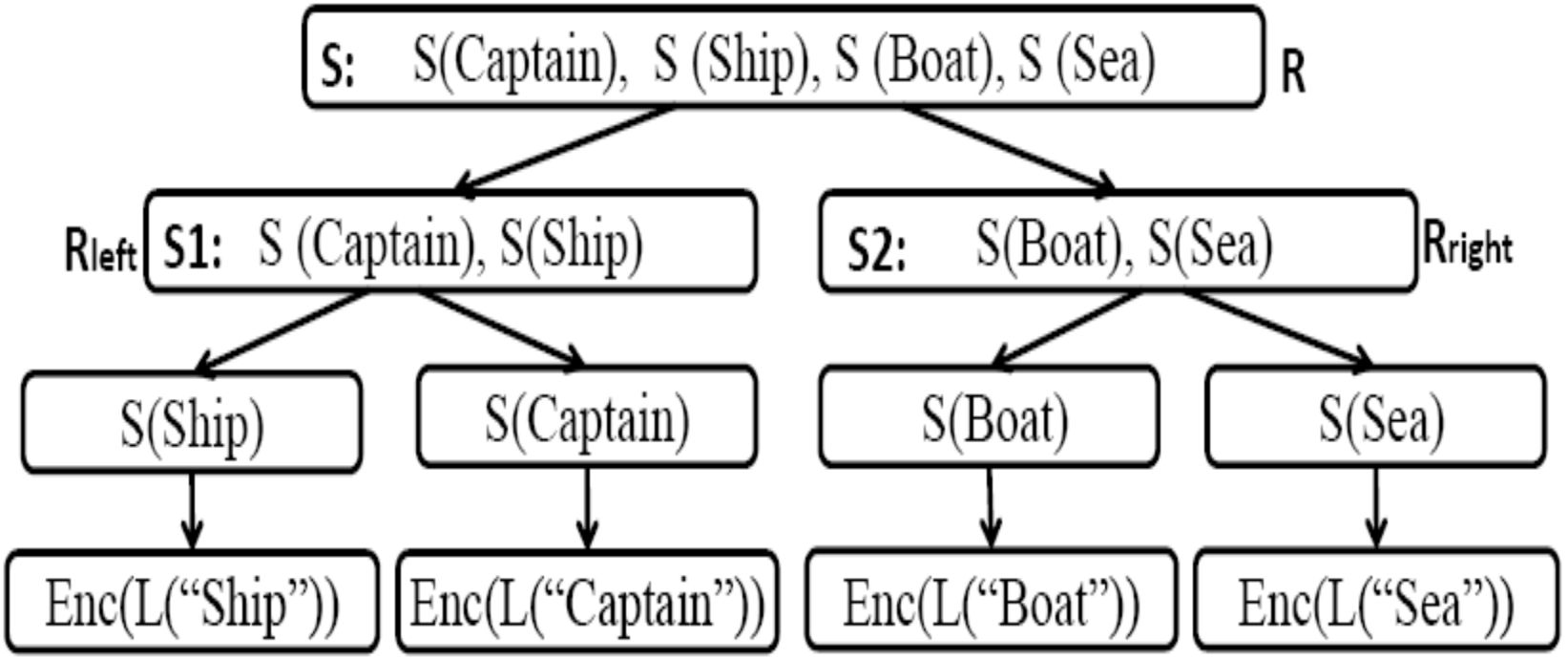
- First Challenge : Each document may have several keywords
- Matching each keyword in the basic approach is inefficient
- *Our solution :We build a binary tree like structure to perform efficient searching*

- Second Challenge : A query string can appear at different positions in a keyword
- The results need to be returned in a ranked order
- *Our solution :We record the matching positions along the length of the tree and rank the leaf nodes*

Improving Search Efficiency with PASSTree

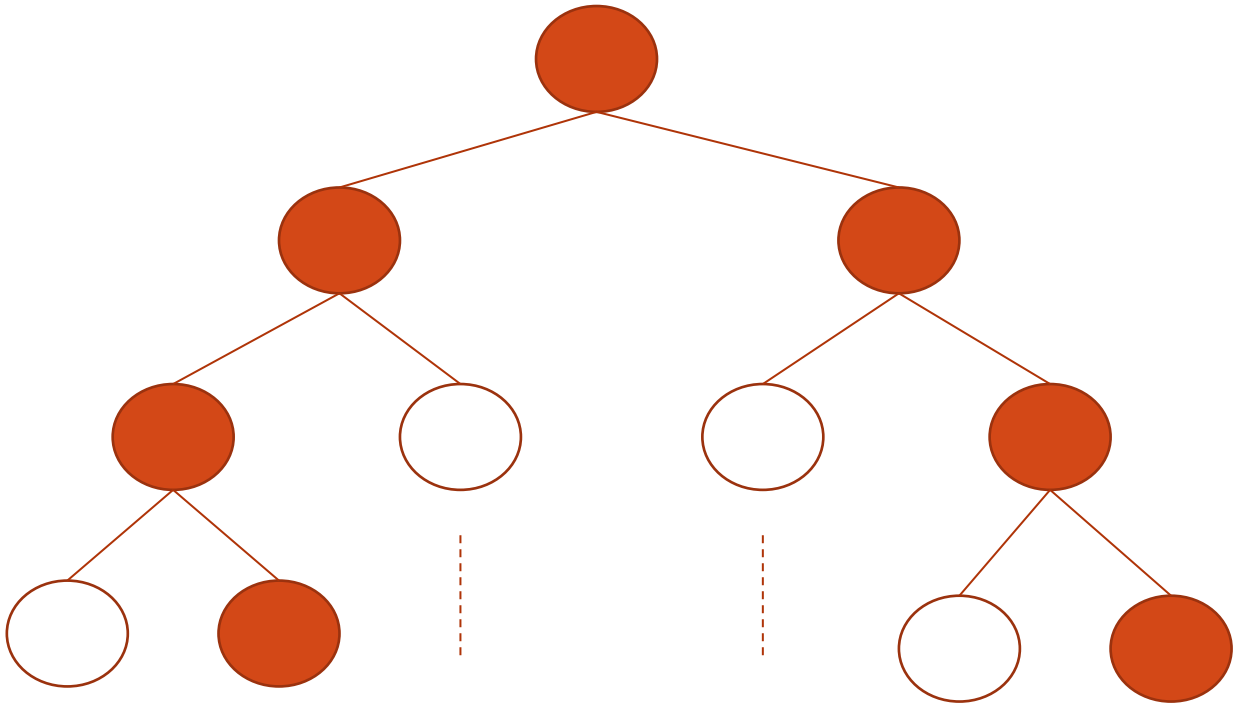
- PASStree arranges the keywords using Bloom filters arranged as nodes of a binary tree structure
  - The root node stores all the sub-strings of all the keywords in the document collection
  - The two children of the root node correspond to two equal sized sub-sets of the keyword collection in the parent node
  - Each child node stores the sub-strings of the keywords of the keyword set associated with it
  - Each leaf node contains a single keyword
- Each leaf node points to the ranked list of documents for the keyword
- The search proceeds along left and right sub-trees and returns all matching leaf nodes

PASSTree Illustration



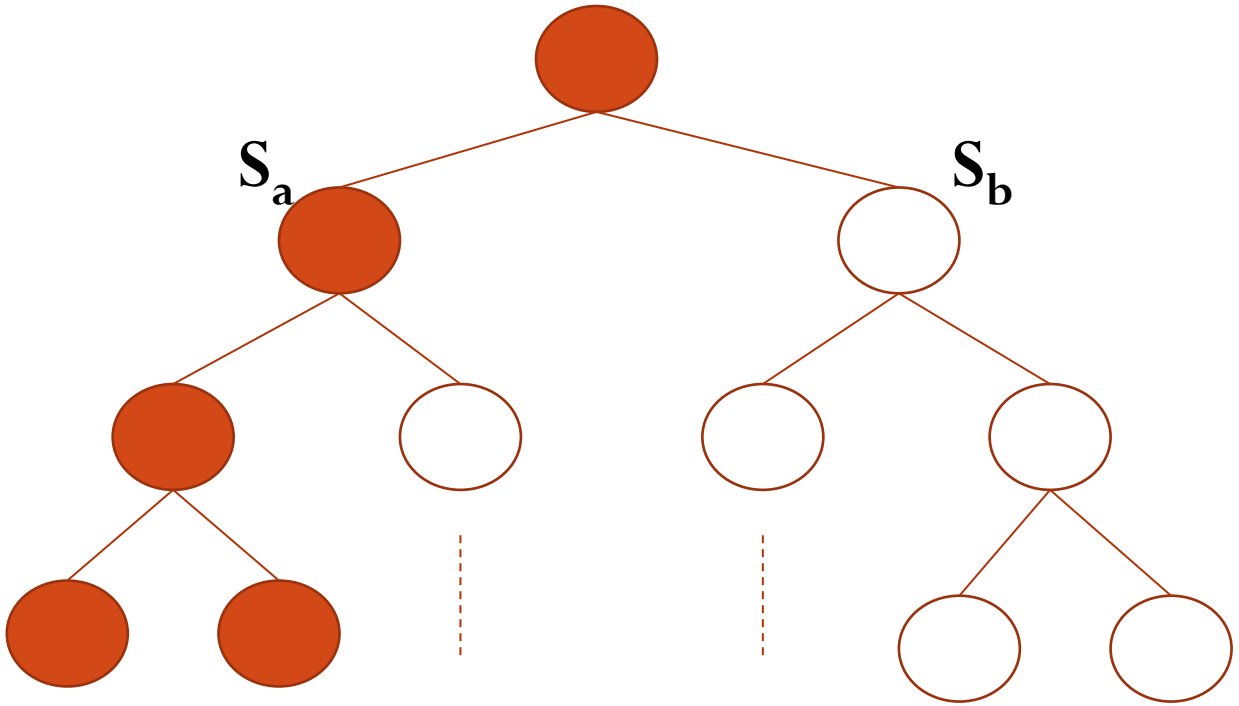
Problems with PASStree

- If a query is matched with two sub-trees, the search proceeds along both sub-trees
  - Likely that the search can span the entire tree



Problems with PASStree

- It is desirable to split the keywords set into two equal sized sub-sets $\mathbf{S_a}$, $\mathbf{S_b}$ such that any two keywords across $\mathbf{S_a}$ and $\mathbf{S_b}$ don't share too many sub-strings



PASStree+

- We cluster the keywords based on pre-defined similarity metrics
- We define following metrics of similarity
  - If two keywords share many sub-strings;
  - If two keywords appear as a phrase in same document;
  - If two keywords appear in the same document;
- We use Clustering LARge Applications (CLARA) clustering algorithm which is based on the standard k-medoids clustering algorithm

- Our experiments show that this approach improves search efficiency significantly

Ranking Search Results

- Simple heuristic: the position of the first occurrence of the pattern in the keyword determines the rank of the keyword with respect to the pattern
  - E.g., for a query string "Ship" : the set of matching keywords "Shipment", "Shipper", "Worship", will be ranked 1, 1 and 2

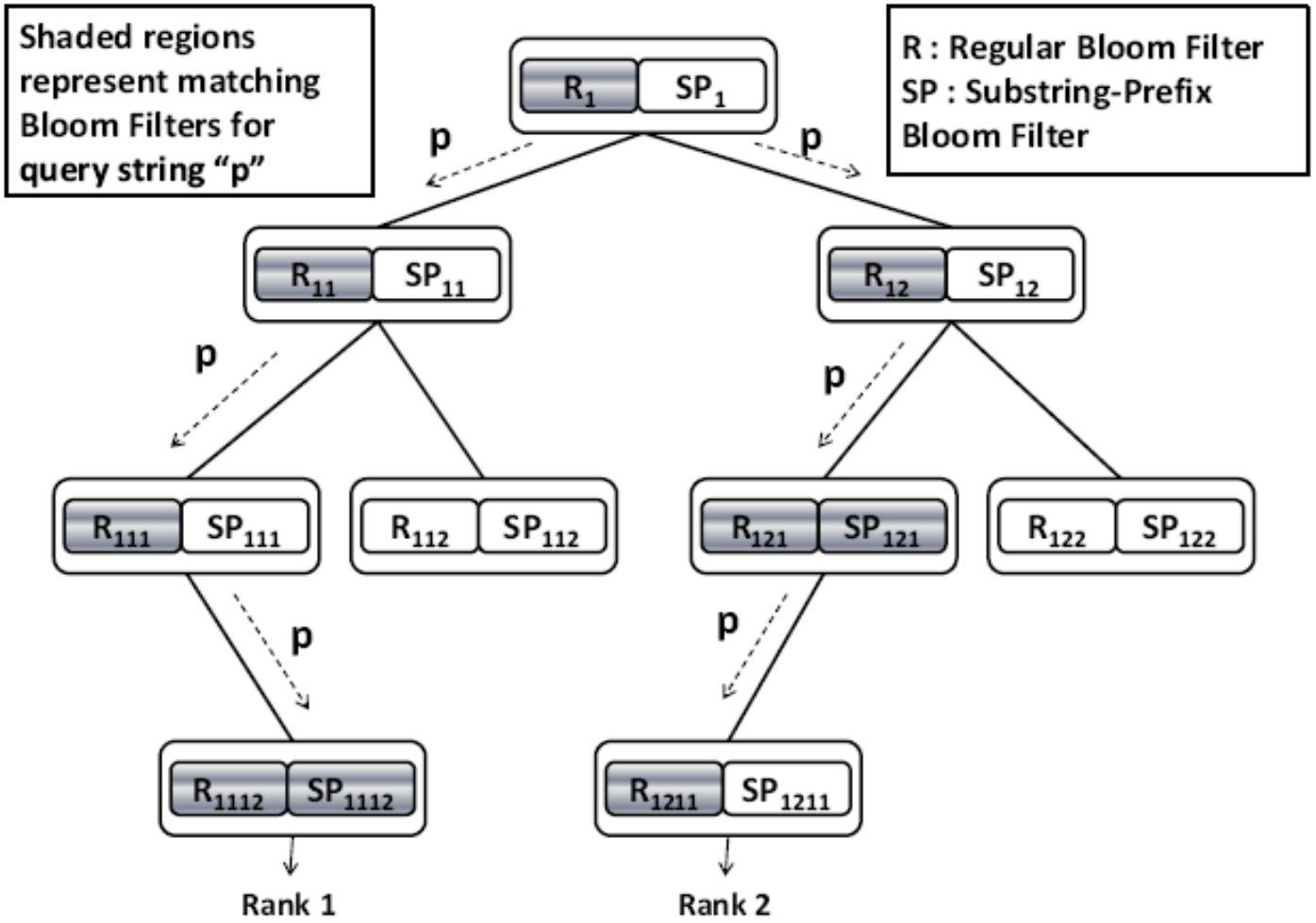
- We record the matching positions for a given pattern using an auxiliary Bloom filter called sub-string prefix (SP) Bloom filter
  - Each node in the PASStree+ gets one such SP filter

Ranking Search Results

- At a leaf node, the SP Bloom filter stores all the prefixes of the keyword
- At the next higher node, the SP Bloom filter stores all sub-strings of the keyword in the 2nd position and so on

- For example, for keyword "Shipment":
  - Leaf node's SP stores : "Shipment", "Shipmen" , "Shipme", "Shipm", "Ship", "Shi", "Sh", "S"
  - Parent node's SP stores : "hipment", "hipmen" , "hipme", "hipm", "hip", "hi", "h"

Ranking Example



Document List Encryption

- Document list of each keyword at leaf node is encrypted with a unique key revealed only by a valid trapdoor
- All valid trapdoors (sub-strings) of a keyword are encoded as roots of a polynomial
- Solving the polynomial reveals the decryption key for the corresponding document list
- Polynomial is padded with some random roots to thwart statistical analysis

Security Analysis

- PASStree does not reveal the sizes of the individual keywords, since we store all possible sub-strings of a keyword and we randomize the Bloom filters

- Some of the trapdoors are never searched, which means that guessing the set of legitimate trapdoors is not possible for the adversary, even after significant amount of searches

- Any two Bloom filters are indistinguishable from each other, since we apply sufficient padding to each Bloom filter at the same level and also garble the Bloom filter

Experimental Evaluation

Implementation Details

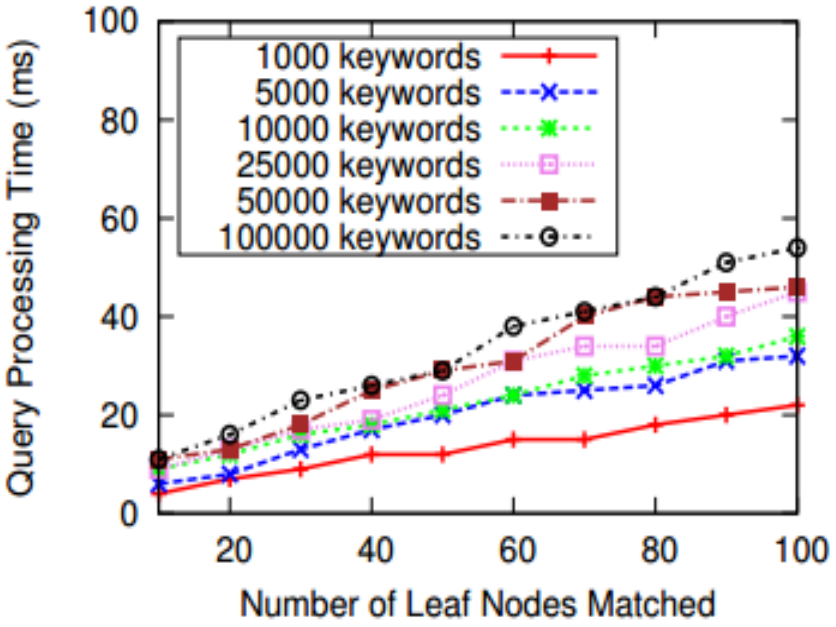
- Language: C++
- OS: Ubuntu 12.10
- CPU: Intel Core i3-2120k (3.3GHz)
- RAM: 4 GB
- Encryption: AES 128-bit key
- Hashing: HMAC-SHA2 256-bit key

Datasets

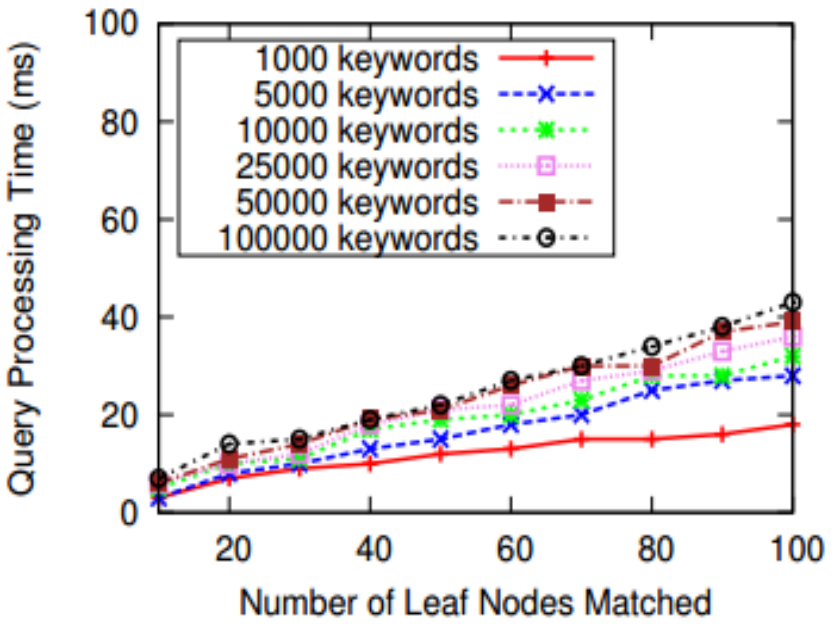
- WIKIPEDIA Dataset
  - 10 million plus web pages
  - 100 distinct keywords per file
  - *prefix and sub-string queries* on dataset sizes of 1k, 2k, …, 10k, 25k, 50k and 100k distinct keywords
- ENRON Dataset
  - 0.6 million plus emails
  - 10 distinct keywords per file
  - *multi-keyword Sender-Receiver queries* on dataset sizes of 4k, 6k, 8k, 10k and 12k distinct keywords
- Each configuration was tested 5 times using random sampling and results were averaged

Experimental Evaluation

PASStree



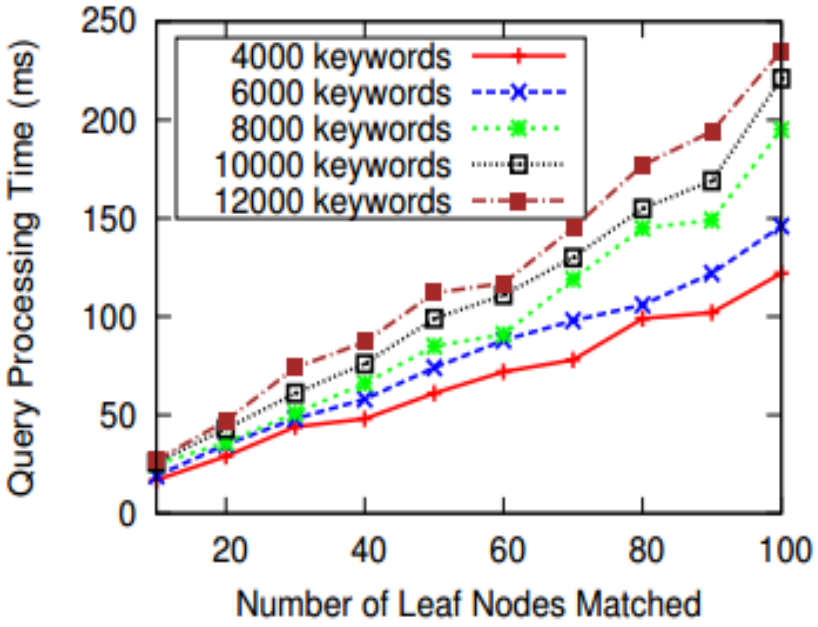
PASStree+



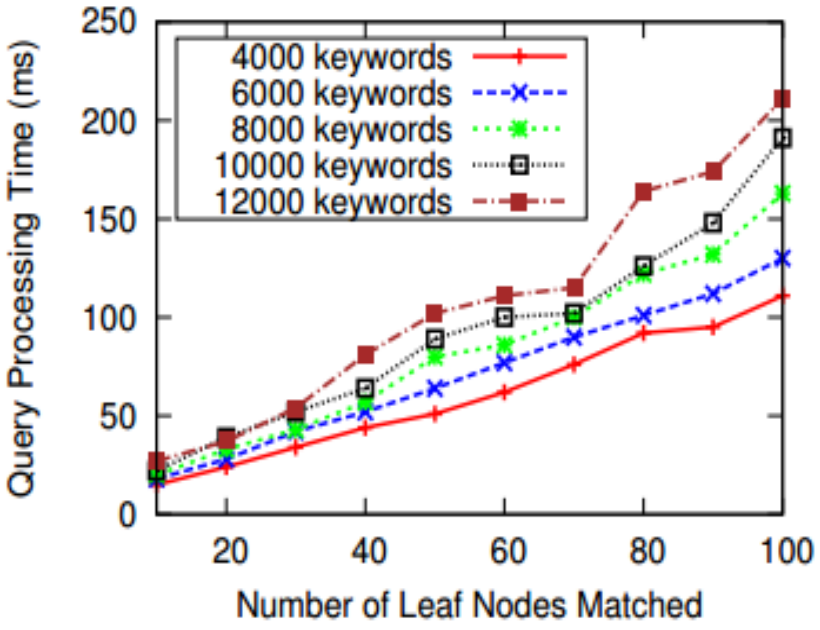
Query Execution Time for WIKIPEDIA Dataset

Experimental Evaluation

PASStree



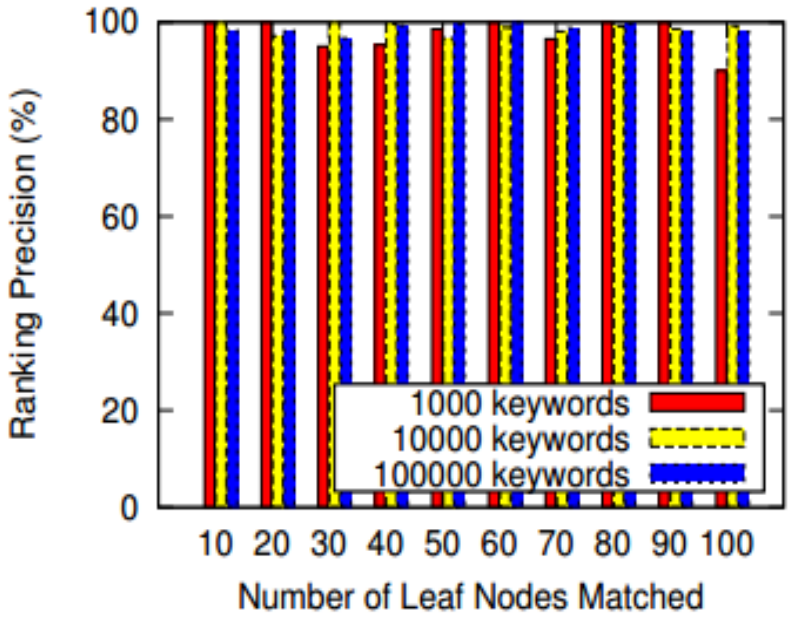
PASStree+



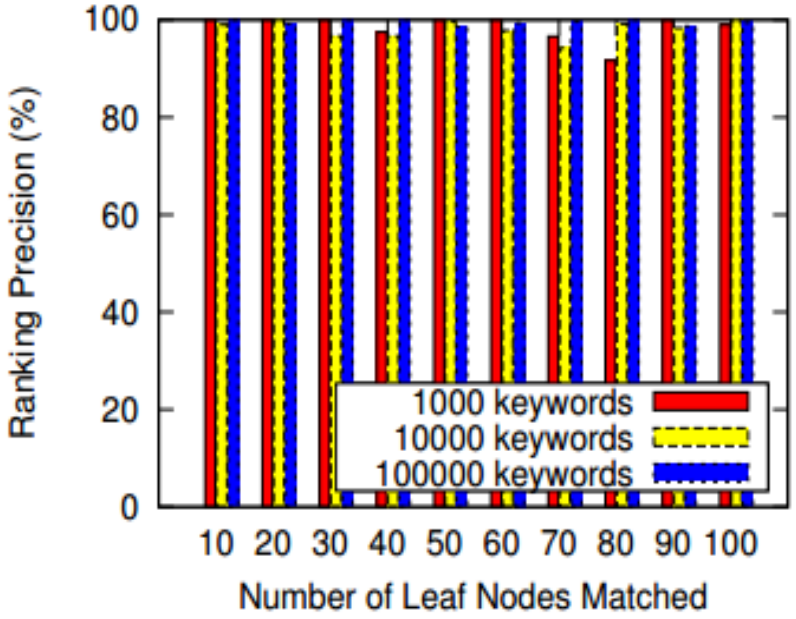
Query Execution Time for ENRON Dataset

Experimental Evaluation

PASStree



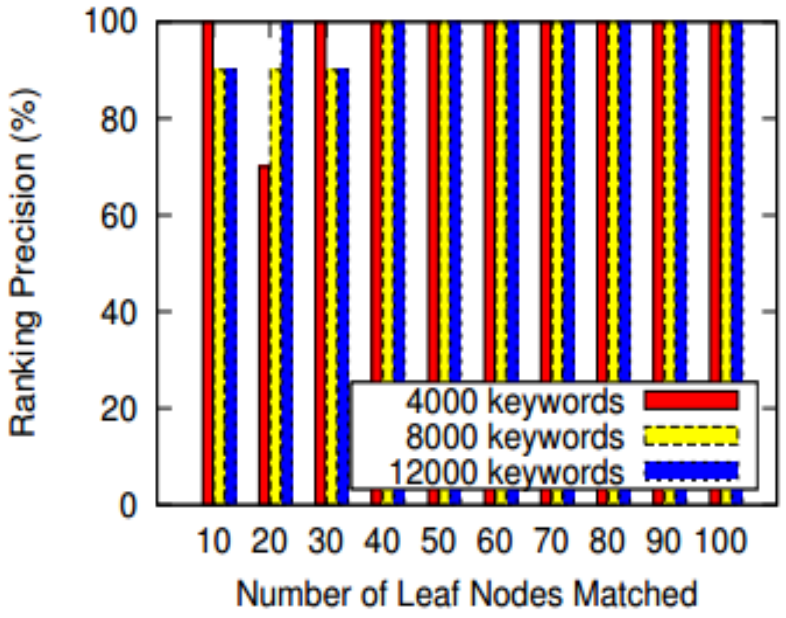
PASStree+



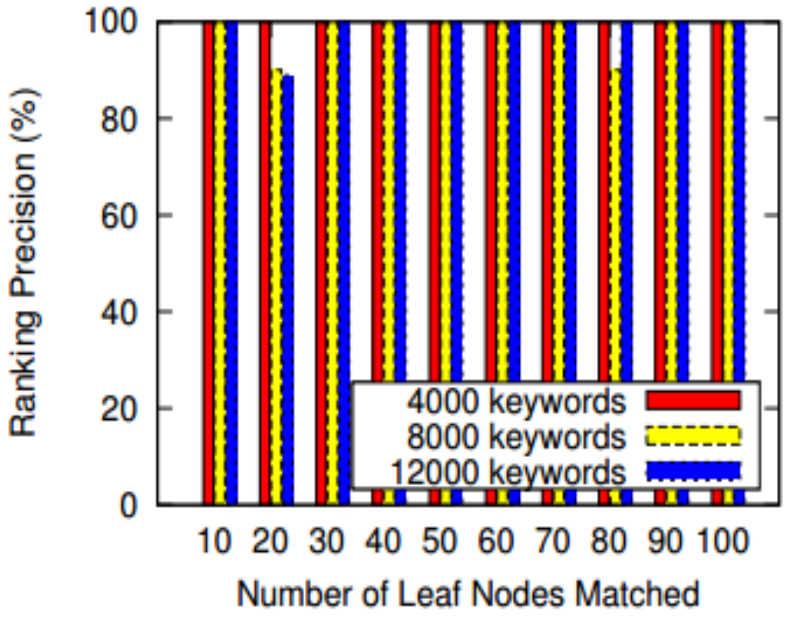
Ranking Precision for WIKIPEDIA Dataset

Experimental Evaluation

PASStree



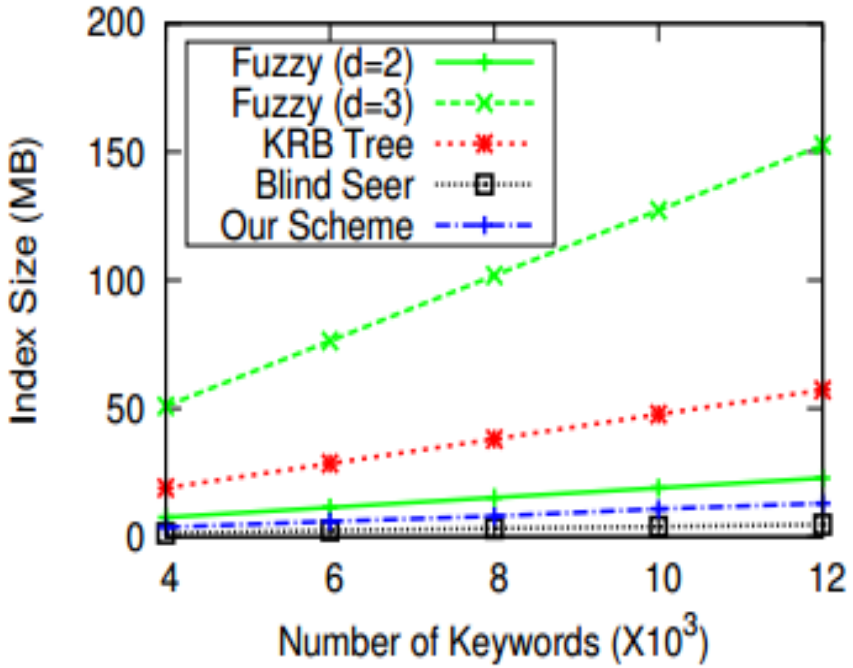
PASStree+



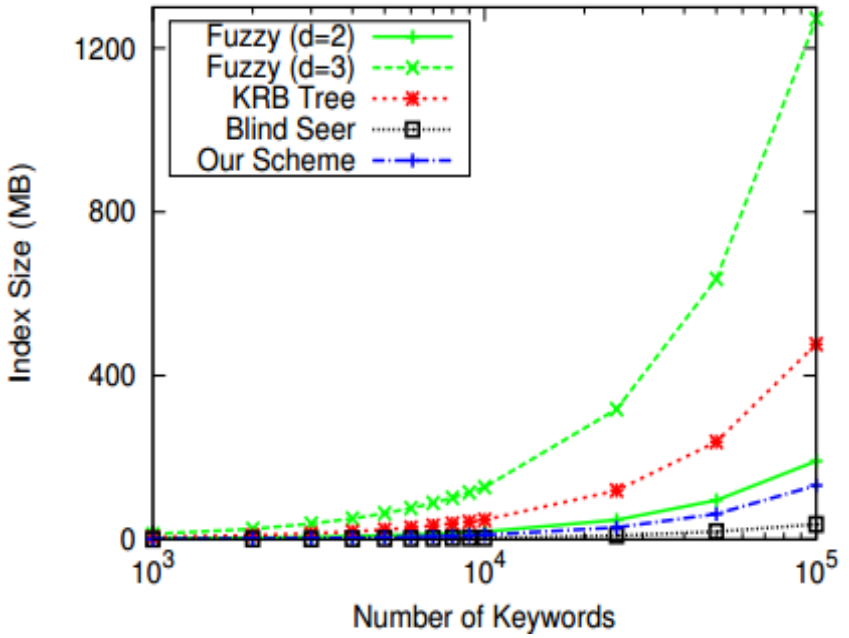
Ranking Precision for ENRON Dataset

Experimental Evaluation

ENRON Dataset



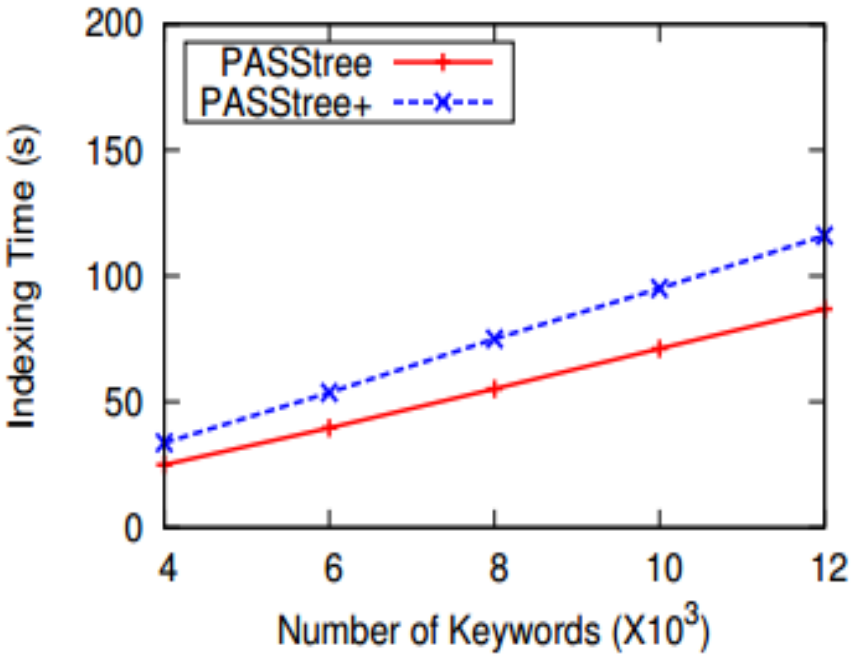
WIKIPEDIA Dataset



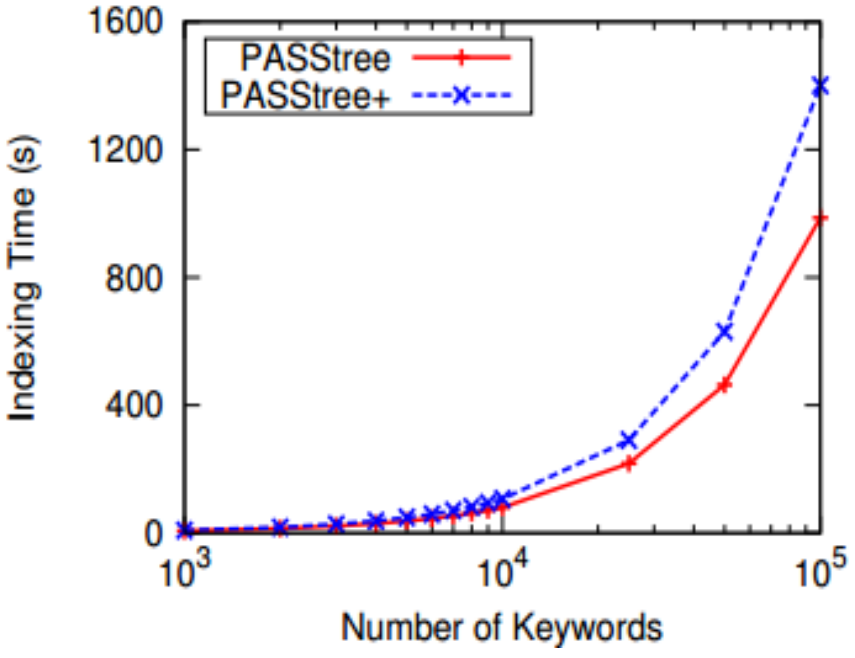
Index Size compared to other similar works

Experimental Evaluation

ENRON Dataset



WIKIPEDIA Dataset



Index Construction Time

Summary

- **First** symmetric key based privacy preserving string matching algorithm
- PASStree+ is an efficient search tree that optimizes the search complexity
- Provides strong privacy guarantees in the IND-CKA security model
- A ranking algorithm that is nearly as accurate as the plain-text version
- Experiments on real-world data sets indicate the practicality and feasibility of deployment

- Future work in this domain explores regular expression matching and secure indexes supporting multiple types of queries

Thank you for listening ☺