FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

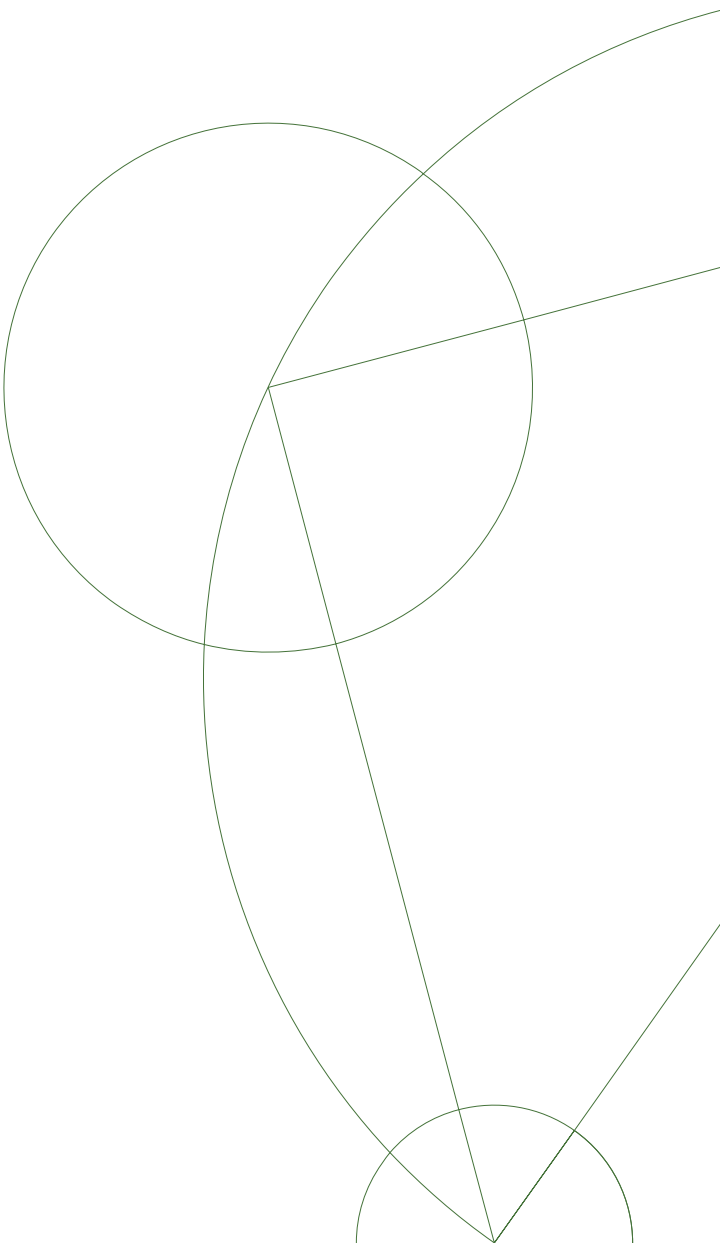
DEPARTMENT OF COMPUTER SCIENCE

Bachelor

Suffix Arrays In Intrusion Detection

April 2017

Mark Roland Larsen <fvg932@alumni.ku.dk>



Supervisors
Michae Thomsenl <m.kirkedal@di.ku.dk>
Troels Larsen <gvl981@di.ku.dk>

Contents

1 Abstract

2 Limitations

Limits: Network intrusion detection, SACA comparison

3 Introduction

In 1973, Peter Weiner presented a linear-time solution to the pattern matching problem, as long as the alphabet was fixed. His structure, which he denoted bi-tree could identify any substring of a text, without specifying all of them [1]. His method processed the text from right to left, such that the bi-tree would be updated to contain longer and longer suffixes. By doing so, he consequently defined the notation of textual inverted index, that would bring forth improvements, analyses and new applications for more than 40 years after the introduction [1]. About three year later Ed McCreight presented a left to right algorithm, and change the data structure name to suffix tree. A name that has stuck ever since [1]. In 2013, the Combinatorial Pattern Matching symposium marked the 40th anniversary of Weiner's work, with a special session. This data structure and those that followed had a deep impact on string matching. Their range scope extends to music equipment, AI (artificial intelligence), plagiarism detectors, in the field of biology, to name a few [2, 1]. In biology, string matching algorithms significantly aid biologists in retrieving and comparing DNA strings, reconstructing DNA strings from overlapping string fragments and looking for new or presented patterns occurring in a DNA[3]. Text-editing applications also adopt string matching algorithms, whenever the application has to acquire an unambiguous event of a user-given pattern, such as a word in some document[4, 3].

Suffix trees can be up to 20 times larger than the original source in terms of gigabytes for a genome, consequently being an irritant for applications, where suffix trees were desirable. In 1990, Udi Manber and Eugene W. Meyers introduced the suffix array, which abolished the majority of the structure of the suffix tree, storing only the suffix indices of the string input in lexicographical order. A suffix array could be created by a preorder traversal of a suffix tree, and can thus be seen as sequences of leaves' labels of the suffix tree [1]. Although the suffix array could be seen as a dissimilar structure than the suffix tree, it was later shown, in 2001 by Turo Kasai et al. [1], that suffix arrays are space efficient and concise representation of suffix trees [1, 3]. In 2003 the first linear construction algorithm was presented for direct suffix array construction. In 2004 Mohamed Ibrahim Abouelhoda et al. [5] show that every algorithm which utilize a suffix tree structure, can be systematically replaced with an equivalent algorithm based on a suffix array structure and some additional information [5]. Several fast suffix array construction algorithms trailed, thus the most notable was presented in 2009 by Nong et al. guaranteeing asymptotic linear time construction, almost optimal space requirement and is fast in practice [6, 7, 1]. Data compression and suffix trees are tightly interlocked and in 2000, Paolo Ferragina introduced the FM index x, which is a compressed suffix array based on the Burrows-Wheeler transform. This compressed data structure, which may be smaller than the original, supports searching without decompression [1]. Suffix arrays today are the most pervasive in software systems, of the suffix tree variants [1].

String matching can furthermore be found various software applications in the vein of virus scanners (anti-virus) or intrusion detection systems to secure data security over the internet. Commercial anti-malware systems exploit unique malware signatures to detect malicious software as part of their implementation [8, 2]. One method of detecting malicious software on an already infected system , is use of a malware signature repository. Malware signatures could, amongst others, consist of unique fixed length hashes of known malicious software. These unique malware fingerprints are then used to recognize malicious software that are infecting a system, hence the repository are searched to find a distinct match, if any. This predicament is clearly an exact string matching problem which permit usage of a suffix tree or a suffix array data structure. The purpose of this paper, is to study pro and cons of using a suffix array as main data structure for an implementation of an intrusion detection system, exploiting the signature-based technique. We first give an introduction to suffix trees and suffix arrays, before moving on to an extensive clarification of the linear time construction algorithm SAIS. We illustrate examples of applications using suffix trees which can systematically be replaced by suffix arrays and some extra information. We explain the Burrows-Wheeler transform which is used to create compressed suffix arrays.

We implement an intrusion detection system which is capable of determine maliciousness of a file on a Windows 10 operated system, using a repository consisting of over 27.000.000 malware MD5 hashes from VirusShare [9]. Using Autoruns for Windows, which contains comprehensive auto starting information of program that are configured to run during system boot or login, the intrusion detection system is capable of automatically scanning all startup programs for maliciousness during system boot [10]. Furthermore, we implement a live process scanner which runs in the background and scan all new stating processes for maliciousness. We compare construction time, applications and space requirements of suffix trees and suffix arrays. We introduce yet another space requirement reduction algorithm SAIS-FLS, for applications using exact string matching on a suffix array data structure, constructed from a concatenation of fixed length distinct strings. SAIS-FLS is capable of reducing the number of suffix indices with a factor of the fixed string length. We will highlight advantages and disadvantages of the signature-based technique and compare it with its cousin, the behavior-based techniques and give an elaboration to the term malware.

4 String Matching

Exact string matching is both an algorithmic problem and data structure problem [11]. The static data structure consist of preprocessing some predefined large text $T = \{t_1, t_2, \ldots, t_m\}$, and query some smaller pattern $P = \{p_1, p_2, ..., p_n\}$ [11]. The objective is to preprocess text T and query pattern P in text T in linear time, $O(m), m \in |T|$ and $O(n), n \in |P|$, respectively [11].

Problem:

> Given a pattern P and a long text T, the problem consist of finding all occurrences of pattern P, if any, in text T [3].

The occurrences of pattern $P = \{ana\}$ in text $T = \{banana\}$ are found at $T[1, 3]$ and $T[3, 5]$, as illustrated in Figure 1. Note that pattern P may overlap.

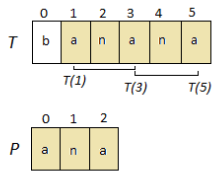


Figure 1: The text T={banana} and pattern P={ana} over the alphabet Σ={abn}. The pattern P occours in T in, at position $T[1]$ and $T[3]$. Notice that occurrences of P may overlap.

Since most discussions of the exact string matching paradigm, begins with a naive method, this paper adopt the tradition, both presented by Gusfield et. al and by many others [3]. The naive method forms a basic understanding and insight to the more complex exact string matching algorithms presented in the paper.
The method align left end of P with left end of T and the scan from left to right, comparing characters of P in T, until either there is a mismatch or P is exhausted, in which case an occurrence of P in T is reported. P is then shifted one place to the right, and the character comparison is restarted from the left end of P which repeats until P shifts past right end of T [3].
Let n denote the length of P and let m denote the length of T, then the worst-case time complexity of the naive method, is $\Theta(nm)$. This is particular clear if P and T consists of the same repeated characters, such that the is an occurrence of P in T for each of the first $m - n - 1$ positions.
Since most discussions of the exact string matching problem begin with the naive method. This paper adopt this tradition, as it form a basic insight to the more complex exact string matching algorithms presented later on [3].

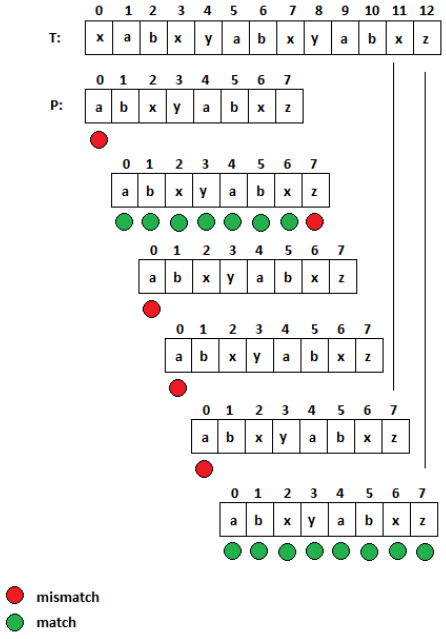


Figure 2: The naive method, where P is shifted one character to the right after each mismatch.

Let pattern P= abxyabxz and let text T= xabxyabxyabxz.

Then the naive method align left end of P with left end of T and scan from left to right, comparing the characters of P with T, until either two disparate characters are located or P is exhausted, in which case an occurrence of P in T is reported. If a character mismatch happens, P is shifted one place to the right, until P exceeds T, as illustrated in Figure 2 [3]. The worst-case bound of the naïve method is Ω(nm), which can be reduced to Ω(n + m) with the basic idea of shifting P more than one character at a time. This means that the number of character comparisons are reduced, due to P moving through T more rapidly. Some methods even exploit skipping over parts of the pattern after P has shifted, further reducing character comparisons [3].

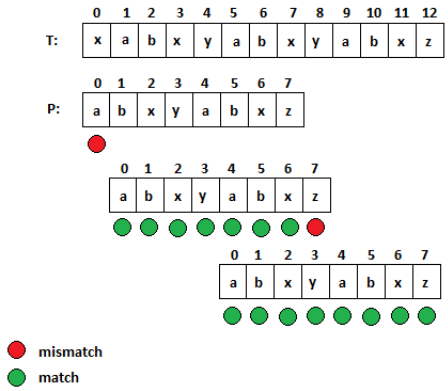


Figure 3: After a mismatch, P is shifted to the next occurrence of a at position 5 in T, moving through T more rapidly

Figure 3 illustrates the idea of shifting P more than one character to the right. At initialization , the left end of P aligns with left end of T, here comparing each character from P with T from left to right.
Let $P[0]$ denote the starting character of P found at position 0, such that $P[0] = a$

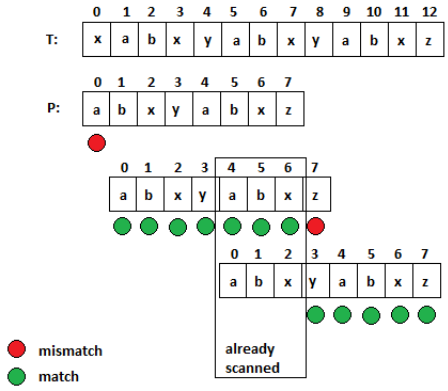


Figure 4: Characters that have already been scanned are stored, so when P is shifted to position 5 in T, abx have already been scanned and can be skipped, and the character scanning is resumed from position 8 and 3, in P and T, respectively.

When comparing characters, if a character in T match $P[0]$, store the location. If a mismatch occur, shift P to the stored location, here position 5 in T and restart the character comparison, as in Figure 3. This is doable for the reason that $P[0] = a$ does not occur in T before position 5, such that $T[5] = P[0] = a$. The method in Figure 3 can be improved further, knowing that the next three characters are abx after P has shifted to position 5 in T. Knowing this, the first three characters are skipped, and character scanning are resumed from position 8 in T and position 3 in P, as illustrated in Figure 4 [3].
The three methods presented exemplifies the basic idea of comparison based algorithms. More efficient algorithms have been developed, such as the Boyer-Moore and Knuth-Morris-Pratt algorithm, which have been implemented to run in linear time ($O(n + m)time$) [3]. These are without a doubt interesting algorithms to analyze, however this paper merely delivers a short and precise description of the paradigm. Another approach to the comparison based method is the preprocessing approach, where comparisons are skipped by first spending a small amount of time, learning about the internal structure of pattern P or text T. Some methods preprocess pattern P to solve the exact string matching problem, where the opposite approach is to preprocess text T, such as algorithms based on suffix trees [3].

4.1 Suffix trees

The classic application for suffix tree is the substring problem [3, 12], which is both a data structure -and an algorithmic problem [11]. That is, given a long text T over some alphabet Σ, and some pattern P, the substring problem consist of preprocessing T in linear time $O(m)$, and hereafter T should be able to take any unknown pattern P, and in linear time $O(n)$ determine occurrences of P, if any, in T [3]. The preprocessing time is here proportional to the length of text T, and the query is proportional to the length of pattern P [3].
This paper adopts the approach of Gusfield et al., by not applying the denotation of pattern P and text T, in respect to describing suffix trees. By using the general description and denotation of suffix trees, there will be less confusion, since input string can take different roles and vary for application to application [3].

Conceptually a suffix tree is a compressed trie [11].

> **Definition** A trie contains all suffixes of string S, where each edge is labeled with a character from some alphabet Σ. Each path from root to leaf represent a suffix, and every suffix is represented with some path from root to leaf [11, 12].

Figure 5 illustrates two tries, left of the string banana and the right over the string *banana*\$. Note that right trie has the termination character \$ appended to the end. This is due to the fact that the definition of a trie dictates that every suffix is represented with some path from root to leaf. Suffix *ana* in left trie does not have a path from root to leaf, but appending a termination character to S that exists nowhere else in the string, will eliminate the problem.
Creating a compressed trie, one takes each non-branching nodes and compress them, such that edge-labels from non-branching nodes concatenates into a new edge-label, as illustrated in Figure 6. Here node 1 is a non-branching node, one then concatenate a to n, to form a new edge-label na, deleting the non-branching node [11]. The number of non-branching nodes in a trie is at most the number of leaves. By compressing, we know have

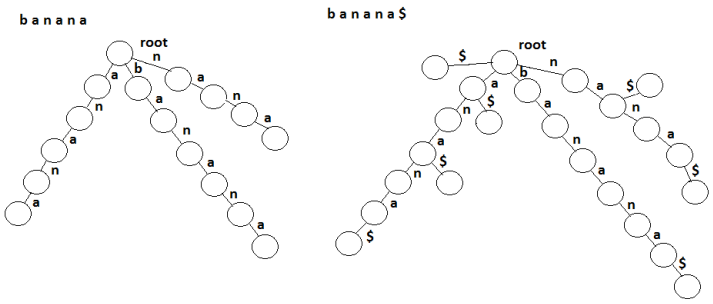


Figure 5: Left is a trie of the string *banana* and the right is a trie of the string *banana*\$.

that the number of internal nodes is at most the number of leaves, having O(k) nodes total.

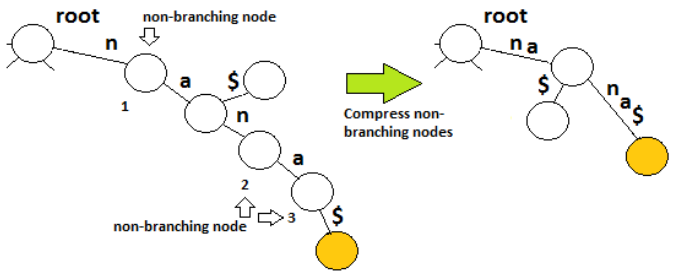


Figure 6: Compressing a trie.

Definition A Suffix tree, T, is a m-character string S concatenated with a termination character \$, that is represented as a directed rooted tree with exactly m leaves, numbered 1 to m. Except the root, each internal node contains at least two children, with each edge labeled with a nonempty substring of S. No two edges exiting a node can have labels beginning with the same character. The concatenation of edge-labels on the path from the root to leaf i, unerringly spells out the suffix of S that starts at position i, such that it spells out $S[i..m]$. The termination character \$ is assumed to appear nowhere else in S, such that no suffix of the consequential string can be a prefix of any other suffix[3].

The suffix tree for the string *banana*\$, in lexicographical order, is illustrated 7. Each path from the root to a leaf i, unerringly spells out a suffix of S, starting at position i in S. As an example, leaf numbered 2 spells out *nana*\$, starting at position 2 in the S, such that $S[2..6] = nana\$$. Each node has at least two children, and no two edges exiting a node begins with the same character.
To dive into the substring problem using linear preprocessing time, $O(m)$, and linear search time, $O(n)$ we follow the tradition, and starts with a naive and straightforward algorithm to building suffix trees before verturing into the linear time preprocessing approach [3].

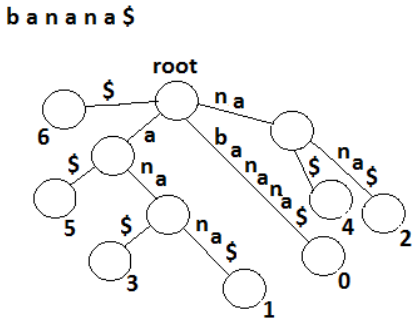


Figure 7: A suffix tree T for string *banana*\$.

4.2 Suffix Trees To Suffix Arrays In Linear Time

Firsrt, suppose we used a suffix tree as datastructure for an application that resign on a device with abundance of space, but needed it to run a smaller device where the current datastructure would be too large. Then we could convert the suffix tree to the space efficient suffix array, provided that the current operations on the suffix tree is supported by the suffix array [3].

Deffinition

> Let an $edge(v, u)$ in T be lexically less than an $edge(v, w)$ if, and only if the first character in $edge(u, v)$ is lexically less than the first character in $edge(v, w)$ [3]

As no two edges out of v have labels beginning with the same character, edges out of v lexical ordered. So the path from the root of T following the lexically smallet edge out of each node leads to a leaf in T which represents the lexically smallet suffix in T. Then the suffix array SA of T can be constructed in linear time. For suffix tree $T=$ banana in Figure 7, the lexical depth-first traversal visites the nodes 6, 5, 3, 1, 0, 4, 2 in that order [3].
Latter, suppose that we have no datastructure constructed for our new application, but already know that a suffix tree would be too large for our new device. Then it would be easier if we could construct the suffix array without the need of a pre-constructed suffix tree, especially if we could construct it in linear time. Luckily we can, but besides of being more space effecient a more detailed description is needed.

4.3 Suffix Arrays

Suffix array are space efficient alternatives to suffix trees [3, 7]. Before Manber and Meyers in 1990 introduced the first direct suffix array construction algorithm – SACA, suffix arrays were constructed using lexicographical-order traversal of suffix trees [3, 7, 13, 6]. Manber and Meyers made suffix trees obsolete in respect to constructing suffix arrays, and their approach is known as a doubling algorithm, where with each sorting pass, doubles the depth to which each suffix are sorted. This means that suffixes are sorted in logarithmic number of passes, providing a worst case bound of $O(nlogn)$ and $O(n)$ expected, assuming linear sort, reminiscent of Radix Sort [13] and queries can be answered in $O(P + logn)$ with use of Binary Search [3].

With the discovery of four different SACAs requiring only $O-(n)$ time worst case in 2003, the situation drastically changed. SACAs have since been the focus of intense research [13, 6]. In 2005 Joong Chae Na introduced more linear time SACAs, where two stood out, the Ko-Aluru (KA) algorithm for supplying good performance in practice and the Kärkkäinen-Sanders algorithm for its elegance [6].
According to a survey paper, SACAs have to fulfill three important requirements:

1. The algorithm should run in asymptotic minimal worst case time, where linear is an optimal way [6].

2. The algorithm should run fast in practice [6].

3. The algorithm should consume as less extra space in addition to the text and suffix array as possible, where constant amount is optimal [6].

Although no current SACAs fulfill the requirements in an optimal way, research into faster and more space reducing SACAs continued [6]. Later on, in 2009, Nong et al. introduced two new linear time construction algorithms, one which outperformed most known and existing SACAs, called Suffix Array Induced Sorting SA-IS algorithm, guaranteeing asymptotic linear time and almost optimal space requirements [6].

4.4 SAIS - Suffix Array Induced Sorting Algorithm

The SA-IS algorithm is a divide and conquer and recursion algorithm, using variable-length leftmost S-type substrings and induced sorting [7]. In view of the fact that the SA-IS algorithm is unsophisticated to comprehend, implement and guarantees asymptotic linear time construction and close to optimal space, SA-IS has been chosen as the single algorithm for the implementation of a malware detection system and the experiments which follow.

```
SAIS(S, SA)
    (* Step 1 : Initialization & classification*)
    SA <-- suffix array of S
    t <-- type array
    P <-- LMS indicies array
    B <-- bucket array
    Scan S once from either left or right and classify all characters as
        S-type or L-type and place them in t.
    Scan t once from  either left or right and locate all LMS substrings
        in S and put them into P_1
    (* Step 2 : Induced sort LMS-substring)
    Induced sort all LMS substrings using P_1 and B
    Name each LMS substring in S by its bucket index to get a
        new shortened string S_1
    (* Step 3 : Uniqueness - recursive step)
    if T_1 is distinct, hence all characters are unique
       then
           Directly compute SA_1 from S_1
       else
           SAIS(S_1, SA_1)
    (* Step 4 : Induce SA from SA_1)
    Induce SA from SA_1
    return
```

Basic notations

Let S be a string or text of n-characters stored in an array $[0 \ldots n-1]$ and let $\Sigma(s)$

be the alphabet of S.

Let $S\$$ be a string S concatenated with the termination symbol \$, where \$ is not contained in S and is the lexicographical smallest character in S. For S containing concatenation of multiple strings, let $S = S_0\$S_1\$...S_{n-1}\$$, where \$ is the termination symbol for each concatenated string in S, and is the lexicographical smallest character in $S_0, S_1, ..., S_{n-1}$. Furthermore, S may not be contained in $S_0, S_1, ..., S_{n-1}$. String S is supposed to be concatenated with the unique termination symbol \$, if not explicit stated otherwise [7].

Let $suf(S, i)$ be some suffix in S starting at $S[i]$ running to the termination symbol \$. $suf(S, i)$ is of S-type or L-type if $suf(S, i) < suf(S, i+1)$ or $suf(S, i) > suf(S, i+1)$, respectively [7].

Let $suf(S, n-1)$ be the termination symbol and of S-type [7].

Let $S[i]$ be S-type or L-type, if suf(S,i) is S-type or L-type, respectively [7].

Observation

- $S[i]$ is S-type if $S[i] < S[i+1]$ or $S[i] = S[i+1]$ and $suf(S, i+1)$ is S-type [7].
- $S[i]$ is L-type if $S[i] < S[i+1]$ or $S[i] = S[i+1]$ and $suf(S, i+1)$ is L-type [7].

The properties defined in the observation suggest that scanning from right to left, determing the type of each suffix or character can be done in constant time, $O(1)$, and that the type array t, can be filled in linear time, $O(n)$ [7].

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S:	m	m	i	i	s	s	i	i	s	s	i	i	p	p	i	i	$
t:	L	L	S	S	L	L	S	S	L	L	S	S	L	L	L	L	S

Figure 8: Type array t is filled from right to left

Figure 8 illustrates the filled type array, t, for text S=mmiissiissiippii\$, where text S is scanned from right to left, determining the type of each suffix and character. Going from right to left in Figure 8 we have that $suf(S, 16)$=\$ is a S-type, $suf(S, 15)$=i\$ > $suf(S, 16)$=\$ and is L-type, $suf(S, 14)$=ii\$ > $suf(S, 16)$=i\$ and is L-type and so forth, filling the type array t in linear time.

Let $S[i]$ be a left most S-typeLMS character, if $S[i]$ is S-type and $S[i-1]$ is L-type, and let $suf(S, i)$ be a LMS suffix, if S[i] is a LMS character [7].

Let $S[i..j]$ be a LMS substring if both S[i] and S[j] are LMS characters, and there exists no other LMS characters in the substring, and $i \neq j$ or it is the sentinel itself [7].

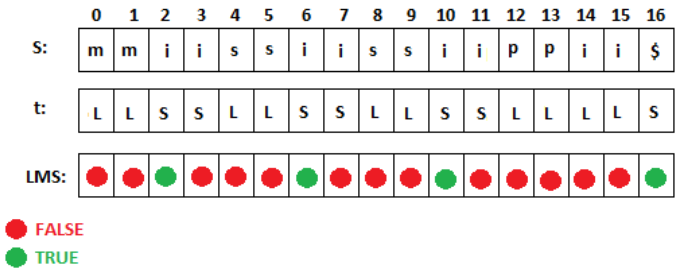


Figure 9: Type array t and LMS array defined for S=mmiissiissiippii$

As Figure 9 exemplify, four LMS characters are defined for S=mmiissiissiippii$, here at position 2, 6, 10 and 16 in S. Furthermore, four substrings and suffixes exists in S, namely S[2..6], S[6..10], S[10..16] and S[16..16], and S[2..16], S[6..16], S[10..16] and S[16..16], respectively. After defining the S-types, L-types and LMS, the induction process of LMS substrings commence.

Deffinition

> Determining the order of any two substrings, the corresponding characters are compared from left to right, comparing their lexicographical values first, and next their types, where S-type is considered highter priority than L-type [7].

Induced sorting LMS substrings

This part address the challenging problem of sorting the variable length LMS substrings. The basic idea is to create a new array, SA, and bucket sort the LMS substring into their equivalent buckets. Each bucket is named corresponding to the alphabet $\Sigma = \{\$, i, m, p, s\}$ in lexicographical order, such that SA contains four buckets, named \$, i, p and s in that order, as shown in Figure 10 [7]. S is scanned from left to right, and indices for each LMS substring is appended to the end of its corresponding bucket in SA. The first LMS substring index is placed at the end of bucket for i, here at position 8 in SA and forwards the bucket end one to the left, hence the bucket end for i now rest at position 7 in SA. This process is repeated until all LMS substring indicies are placed in their buckets [7].

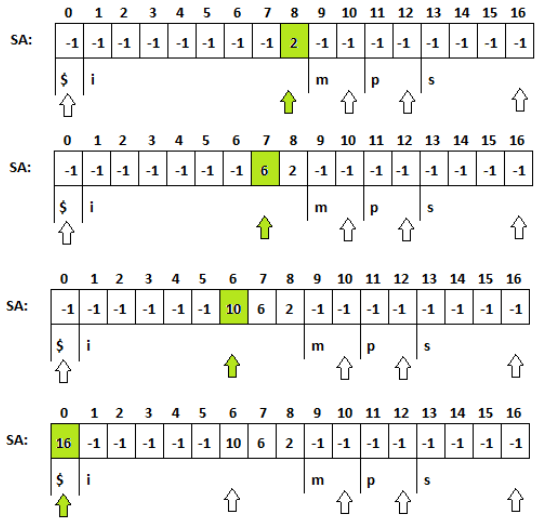


Figure 10: Induced sort of LMS substring

When the LMS substrings are placed, then scan SA from left to right and for each nonnegative value $S[i]$, if $S[i]-1$ is L-type, then place $SA[i]-1$ in the corresponding bucket for $suf(S, SA[i]-1)$, and lastly forward the bucket head one to the right [7].

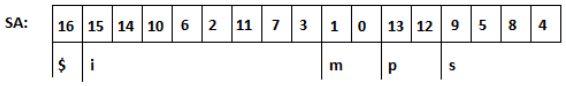


Figure 11: SA after the induced sorting for LMS substring.

Roughly equivalent, when all L-types are placed, scan SA from right to the left for each nonnegative value S[i], if $S[i]-1$ is S-type, then place $SA[i]-1$ in the corresponding bucket for $suf(S, SA[i]-1)$, and forward the bucket end one to the left. The above operations are demonstrated in Appendix B and the final result is displayed in Figure 11 [7].

It is now the matter of determine if all LMS substrings are correctly sorted in SA, hence the uniqueness step in the SAIS algorithm. This is done by scanning SA from left to right, and obtaining each LMS substring, and comparing the lexicographical values and types, and place them in buckets named accordingly to the lexicographical order they appear, starting from 0. So scanning from left to right in SA given in Figure 11 gives the following bucket $B = \{\{0; \$\}, \{1; iippii\$\}, \{2; iissi, iissi\}\}$. The bucket keys are then placed in S_1 in the order as they appear in the original string S, hence $S_1 = \{2, 2, 0, 1\}$ as illustrated in Figure 12. If each character in S_1 is unique, hence does not exists any where else in S, then SA_1 can be computed directly from S_1, else fire the recursive step $SAIS(S_1, SA_1)$. S_1 for S in Figure 12 is not distinct, since 2 exists twice in S_1, hence 2 is not unique, consequently a recursive step, $SA(S_1, SA_1)$, is needed. Before venturing into the recursive step, save the original positions of the LMS substrings as they appear in S_1 into P_1, where $S_1[0] = 2$ points at position 2 in S, $S_1[1] = 2$ points at position 6 in S, $S_1[2] = 1$ points at position 10 in S and finally $S_1[3] = 0$ points at position 16 in S, such

that $P_1 = [2; 6; 10; 16]$ [7].

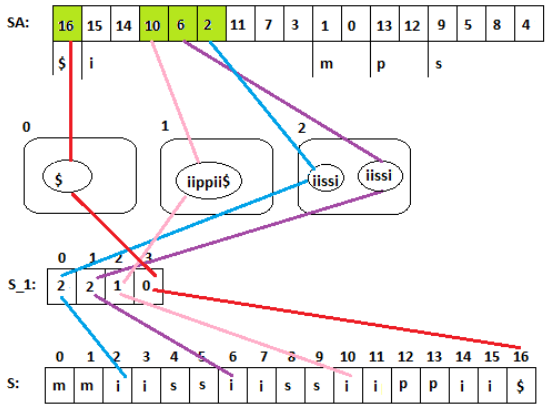


Figure 12: Building S_1 from SA, using the original positions of the LMS substrings in S.

In the recursive step for $SAIS(S_1, SA_1)$, locate S-types, L-types and LMS characters/-substrings and determine if S_1 is distinct. In this case, as demonstrated in Figure 13, there is only one LMS substring in S so the new string S_1 is trivially distinct.

Induce sort SA from SA_1

Either SA_1 has been computed directly from S (if S is distinct) or returned from one or more recursive steps. In either case, SA can be induced sorted from SA_1 using information bound in P_1 [7].

First initialize all indicies in SA with -1 and find the bucket ends. Then scan SA_1 from right to left and place $P_1[SA_1[i]]$ at the corresponding bucket end, and forward the bucket end one item to the left [7].
Then sort L-types by scanning SA from left to right for each non-negative item $SA[i]$. If $SA[i-1]$ is L-type, place $SA[i-1]$ in the corresponding bucket head for SA and forward the bucket head one item to the right [7].
Last, sort all S-types by scanning SA from right to left for each non-negative item SA[i]. If $SA[i-1]$ is S-type, place $SA[i-1]$ in the corresponding bucket end, and forward the bucket end one item to the left [7].

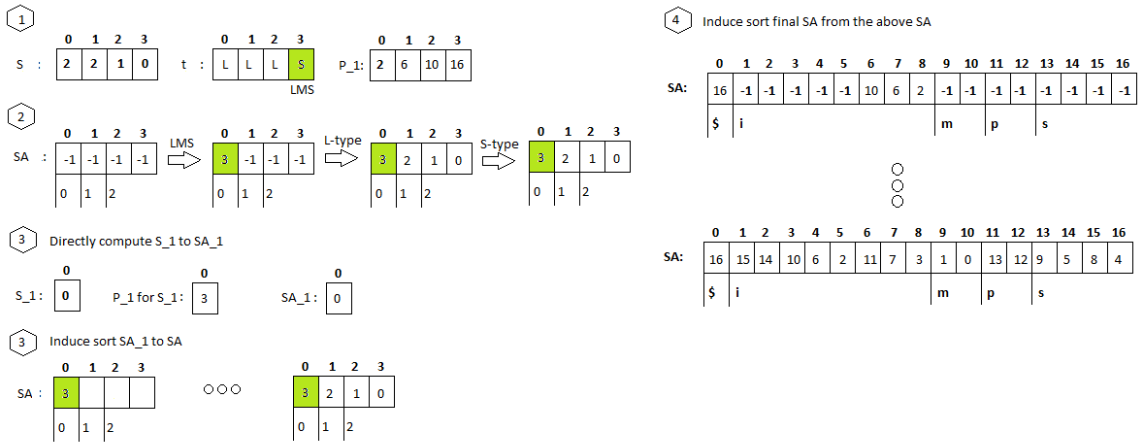


Figure 13: The recursive step $SA(S_1, SA_1)$. First S-types, L-types and LMS characters/substrings for S_1 is localized. Secondly, induced sort LMS substrings, such that $SA_1 = [3; 2; 1; 0]$. Last, scan SA_1 from right to left and place $P_1[SA_1[i]]$ into the buckets end of SA as described for induced sorting LMS Substrings, for each item in SA_1. The final result is displayed in step 4.

This procedure is demonstrated in Figure 39 where SA is induced sorted from S for the text T=mmiissiissiippii\$. SAIS returns the suffix array SA=[16; 15; 14; 10; 6; 2; 11; 7; 3; 1; 0; 13; 12; 9; 5; 8; 4] for text T=mmiissiissiippii\$ which is indeed sorted in lexicographical order as illustrated in Figure 14 [7].

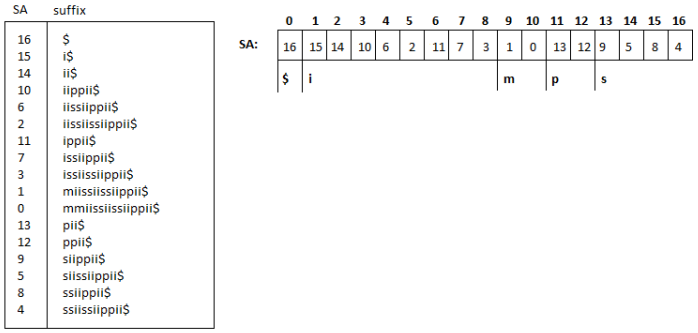


Figure 14: The suffix array for the text T=mmiissiissiippii\$.

Correctness, completeness and performance

We shown a thorough run-through of the SA-IS algorithm, now we tend to correctness, completeness and the performance. We claimed that SAIS is a linear time suffix array construction algorithm using divide-and-conquer and recursion techniques. Furthermore we claimed that the consequential constructed suffix array contains indices of all suffixes of a legal input S according to their lexicographical order. Ge Nong et al. [7] already proved these attributes of SA-IS, we merely elaborate with examples. Recall that S must be terminated with the sentinel and of such, is not a legal input, if S is not terminated by

the sentinel. Furthermore, the sentinel must be the unique smallest character in S, and exist nowhere else in S.

Induced Sorting LMS-Substrings & Inducing SA from S_1

Step two and three in both induced sorting LMS-substring and inducing SA from S_1 are equivalent. In induced sorting LMS-substrings we scan S from right to left and append LMS indices in their correspobding buckets in SA and in inducing SA from S_1, S_1 is scanned from right to left and we append $P_1[SA_1[i]]$ into their corresponding bucket's in SA. The sorting of the variable-size LMS-substrings is the most challenging problem. Ge Nong et al. [7] solves the problem by induced sorting. We have earlier shown this procedure with an example for S=mmiissiissiippii$ and will continue down this road to elaborate the proof presented by Ge Nong et al. [7].

> **Theorem 4.4** Induced sorting LMS substrings will correctly sort all LMS prefixes of S into SA. [7].

First, let LMS prefix $pre(S, i)$ of $suf(S, i)$ to be either the sentinel itself when $i = n-1$ or prefix $S[i..k]$ in $suf(S, i)$ where $i \neq n-1$, $k > i$ and $S[k]$ is the first LMS character after $S[i]$. LMS prefix $pre(S, i)$ is defined as L-type of S-type if $suf(S, i)$ is L-type or S-type [7].

Then suf(S,5) is L-type, hence pre(S, 5) is L-type and pre(S, 5)=$S[i \dots k] = S[5 \dots 6]$ = si, since $S[k = 6] = i$ is the first LMS character after $S[i]$.

S	index	type	suffix	LMS-prefix for suf(S, 0)	LMS prefx substring
m	0	L	suf(S,0)={mmiissiissiippii$}	pre(S,0) = S[i..k] = S[0..2]	{mmi}
m	1	L	suf(S,1)={miissiissiippii$}	pre(S,0) = S[i..k] = S[1..2]	{mi}
i	2	S	suf(S,2)={iissiissiippii$}	pre(S,0) = S[i..k] = S[2..6]	{iissi}
i	3	S	suf(S,3)={issiissiippii$}	pre(S,0) = S[i..k] = S[3..6]	{issi}
s	4	L	suf(S,4)={ssiissiippii$}	pre(S,0) = S[i..k] = S[4..6]	{ssi}
s	5	L	suf(S,5)={siissiippii$}	pre(S,0) = S[i..k] = S[5..6]	{si}
i	6	S	suf(S,6)={iissiippii$}	pre(S,0) = S[i..k] = S[6..10]	{iissi}
i	7	S	suf(S,7)={issiippii$}	pre(S,0) = S[i..k] = S[7..10]	{issi}
s	8	L	suf(S,8)={ssiippii$}	pre(S,0) = S[i..k] = S[8..10]	{ssi}
s	9	L	suf(S,9)={siippii$}	pre(S,0) = S[i..k] = S[9..10]	{si}
i	10	S	suf(S,10)={iippii$}	pre(S,0) = S[i..k] = S[10..16]	{iippii$}
i	11	S	suf(S,11)={ippii$}	pre(S,0) = S[i..k] = S[11..16]	{ippii$}
p	12	L	suf(S,12)={ppii$}	pre(S,0) = S[i..k] = S[12..16]	{ppii$}
p	13	L	suf(S,13)={pii$}	pre(S,0) = S[i..k] = S[13..16]	{pii$}
i	14	L	suf(S,14)={ii$}	pre(S,0) = S[i..k] = S[14..16]	{ii}
i	15	L	suf(S,15)={i$}	pre(S,0) = S[i..k] = S[15..16]	{i}
$	16	S	suf(S,16)={$}	pre(S,0) = S[i..k] = S[16..16]	{$}

Figure 15: Suffixes with corresponding LMS prefixes T=mmiissiissiippii$.

Suppose we appended all LMS-substring indices in SA for S=caa$ with the following L-types and S-types illustrated in Figure 16. Then for each non-negative item in SA we encounter when scanning from left to right, we ask if suf(S, i) has an immediate left neighbor suf(S, i-1) that is lexicographical larger than suf(S, i) and therefore should be appended to the right of suf(S, i) in SA. As illustrated in Figure 16, each immediate left neighbor suf(S, i-1) in S scanning from right to left in S is lexicographical larger than suf(S, i). The only LMS-character in S is $S[3]$=$, the sentinel itself, and its immediate

left neighbor is of L-type, and should be appended to its appropriate bucket head, and the head is forwarded one item to the right. Next non-negative item we encounter in SA is suf(S, 2), it immediate left neighbor suf(S,1) is of L-type and is therefore lexicographical larger than suf(S, i), so it should be added to the right of suf(S,i). Since S[1]=a, suf(S,1) should be appended in the bucket head for a, but should also be to the right of suf(S,2) which is already in bucket a in SA. Luckily we forwarded the bucket's head one item to the right, thus suf(S,1) will be appended after suf(S, 2) in SA. This entails that any immedite left neighbor $SA[i-1]$ of $SA[i]$, $suf(S, SA[i-1])$ is lexicograpical larger than $suf(S, SA[i])$.

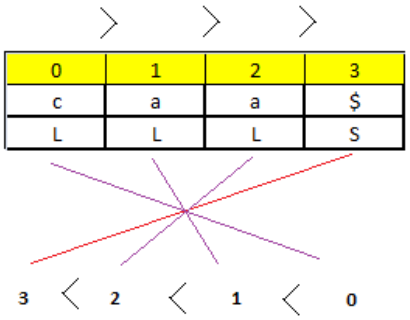


Figure 16: L-types and S-types for S=caa\$, where $S[0\ldots3] > S[1\ldots3] > S[2\ldots3] > S[3\ldots3]$.

When inserting L-type pre(S,i) of suf(S,i) in SA, we must ensure that each L-type $pre(S,i)$ for $suf(S,i)$ is sorted correctly for all S-type $pre(S,i)$ of $suf(S,i)$ in SA [7]. Figure 15 illustrate suffixes and LMS-prefixes for $S = mmiissiissiippii\$$. Suppose that k number of L-type LMS-prefixes are correctly sorted into SA and suppose that we append the (k+1)th L-type LMS-prefix $pre(S,i)$ to the head of its corresponding bucket and there is already a greater $pre(S,j)$ in front of $pre(S,i)$, e.g. to the left of $pre(S,i)$. This consequently signify that character $S[i] = S[j]$ since they are in the same bucket, $pre(S,j+1) > pre(S,i+1)$ and $pre(S,j+1)$ is in front of $pre(S,i+1)$. If this is the case, we must have scanned SA from the left to the right, and discovered LMS-prefixes that are not sorted correctly. This is a contradiction and imply that all L-type LMS-prefixes are sorted in correct order in SA [7].
In the last step we append S-type LMS-prefixes and consequently sort all LMS-prefixes in SA. We scan SA from right to left, and for each non-negative index i, if suf(S,i-1) is of S-type, append it to the bucket end of its corresponding bucket, and forward the bucket end one to the left.
In similar manner to the prove of L-type, by contradiction, suppose that k S-type LMS-prefixes are appended to their corresponding bucket ends and sorted correctly. Suppose that when we append the $k+1$th LMS-prefix pre(S, i) to its corresponding bucket end there is already a smaller S-type LMS-prefix pre(S, j) behind pre(S, i), e.g. to its right. This must entail that S[i] = S[j], since they are in the same bucket, pre(S, j+1) < pre(S, i+1), and pre(S, j+1) is behind pre(S, i+1) in S. This implies that when SA where scanned from right to left, before adding pre(S, i) to its bucket, we must have stumbled by LMS-prefixes in SA that were not sorted correctly [7].

bucket	index	type	SA	LMS prefx
$	0	s	16	{$}
i	1	l	15	{i}
	2	l	14	{ii}
	3	s	10	{iippii$}
	4	s	6	{iissi}
	5	s	2	{iissi}
	6	s	11	{ippii$}
	7	s	7	{issi}
	8	s	3	{issi}
m	9	l	1	{mi}
	10	l	0	{mmi}
p	11	l	13	{pii$}
	12	l	12	{ppii$}
s	13	l	9	{si}
	14	l	5	{si}
	15	l	8	{ssi}
	16	l	4	{ssi}

Figure 17: Sorted S-type LMS-prefixes of T=mmiissiissiippii$.

When appending S-type LMS-prefixes to their equivalent buckets, we may overwrite already appended S-type LMS-prefixes. Figure 17 show the appended S-type LMS-prefixes and as proven, all LMS-prefixes are now sorted in SA applying the same logic as with L-type LMS-prefixes [7]. We observe that every LMS-substring is a LMS-prefix, hence from the above proofs we can derive the following properties. All LMS-substrings are as well LMS-prefixes, therefore sorting LMS-prefixes will thus correctly sort all LMS-substrings. Furthermore, every substring in S is a prefix of a LMS-prefix, hence sorting all LMS-prefixes will consequently sort every substring in S [7].

Divide-and-conquer & recursion

As LMS-substrings are treated as basic blocks of the string in S, sorting all LMS-substrings will consequently sort all substrings in S, as proven. Whenever two (or more) LMS-substrings is contained in the same bucket, we have a case where we have equal LMS-substrings. We then use the order index of each LMS-substring and replace LMS-substrings in S with their names, as illustrated in 12. S can then be represented by a shorter string S_1, consequently solving the problem in a divide-and-conquer manner using recursion [7]. So we divide the problem into subproblems that are smaller instances of the original problem. Solve the subproblems recursively and combine the solutions of the subproblems into a solution for the original problem **??**. Ge Nong et al. [7] proves that $||S_1||$ is at most $||S||$ eg. $n_1 \leq \lfloor n/2 \rfloor$, thus cutting the problem in half with each recursion, by sorting the LMS-substrings. In Figure 13, S=mmiissiissiippii$ generated a shorter string S=3210. Here $||S_1||$ is at most half of $||S||$.

Space and time complexity

Ge Nong et al. [7] proves that all L-type or S-type suffixes of S can be sorted in O(n) time, here with the knowledge that if all S-type suffixes have been sorted correctly in SA, all S-type andL-type suffixes can sorted in linear time by traversing SA once from left to right [7]. Furthermore, knowing that the problem is reduced by at most half $\lfloor n/2 \rfloor$at each

recursion, we must have that:

T(n) = T($\lfloor n/2 \rfloor$) + O(n) = O(n)

Here the first O(n) counts for reducing the problem and inducing the final result from subproblem [7] and the last gives the result as an upper bound of O(n). Space needed to store the sufix array for each reduced problem iteration is the akeelesheel for the space complexity. The first iteration is bound by $n\lceil logn \rceil$ bits, and descrease at most half for each iteration . This imply that the upper bound is governered by the first iterration and therefore the space complexity is O(n log n) bits [7].

4.5 Binary Search

4.6 Longest Common Prefix - LCP

The basic task of analysing a single genome, is to characterize and locate repeated elements of the genome. When comparring two or more genomes the task is to find similar subsequences of genomes. This makes repeat analysis to play a key role in the study, analysis and comparison of complete genome. A prefix in an definition for an affix placed before a word. Suppose we have S=abekat, then *abe*, *ab* and *abek* are prefixes, since they are an affixes placed before *kat*, *ekat* and *at*, respectively. The longest common prefix amongst two string *lcp(*S_0, S_1*)* is the length of the longest word both strings share, from left to right.
Let S_0=abekat\$ and S_1=abe, then the longest common word is *abe*, hence lcp(S_0, S_1)=3.

4.7 Burrows-Wheeler Transform

The BWT - Burrow-Wheeler transform, invented by Burrow and Wheeler in 1994, also known as block sorting, is a lossless data compression algorithm and produces a permutation bwt(S) of an input string S, such that S can be reversed from bwt(S), but is easier to compress [14].
BWT is a very powerful tool in data compression and even simple algorithms that implement BWT have good performance and achieve a good compression ratio using relative small space. Furthermore, even more powerful BWT-based compression tools, such as Bzip and Szip are still used today [14].

Besides data compression, BWT have a remarkable and practical property, namely that it can build a data structure which is sort of a compressed suffix array for a input string S [14]. To fully map and understand BWT, and how it succeed to create permutation bwt(S) of an input string S that is easier to compress, this paper gives a precise description of the idea behind cyclic shifts and reversible lossless data compression, before venturing into property of building a data structure resembling compressed suffix array and its importance to the topic at hand.

BWT consist of reversible transformation of input string S denoted bwt(S). This reversible transformation, bwt(S), consist of exactly the same characters as in S over same alphabet Σ, but is usually easier to compress. The idea is to form a conceptual matrix M whose rows consist of cyclic shifts of S sorted in a left to right lexicographical order [14].

Let F denote the first column in M and let F_i denote the i-th character of column F. Almost equivalent, let L denote the last column in M and let L_i denote the i-th character of column L [14].

Then the following properties of M can be defined:

- Every column of M is a permutation of S.
- For $i = 2, \ldots, |2| + 1$, the character L_i is followed by the character F_i in S.
- Any character α, the i-th occurrence of α in F correspond to i-th occurrence of α in L.

We assume that string S is concatenated with the termination symbol \$. We first find each rotation of S and place these in a matrix M, which can be done by repeatedly taking the end character of S and sticking it to the front of S, until all rotations of S is exhausted, as illustrated in Figure 18. Then we sort the rows in lexicopgrapical order from top to buttom as in Figure 19 [15].

F										L
\$	a_0	a_1	a_2	b_0	b_1	a_3	a_4	b_2	a_5	a_6
a_6	\$	a_0	a_1	a_2	b_0	b_1	a_3	a_4	b_2	a_5
a_5	a_6	\$	a_0	a_1	a_2	b_0	b_1	a_3	a_4	b_2
b_2	a_5	a_6	\$	a_0	a_1	a_2	b_0	b_1	a_3	a_4
a_4	b_2	a_5	a_6	\$	a_0	a_1	a_2	b_0	b_1	a_3
a_3	b_4	b_2	a_5	a_6	\$	a_0	a_1	a_2	b_0	b_1
b_1	a_3	b_4	b_2	a_5	a_6	\$	a_0	a_1	a_2	b_0
b_0	b_1	a_3	b_4	b_2	a_5	a_6	\$	a_0	a_1	a_2
a_2	b_0	b_1	a_3	b_4	b_2	a_5	a_6	\$	a_0	a_1
a_1	a_2	b_0	b_1	a_3	b_4	b_2	a_5	a_6	\$	a_0
a_0	a_1	a_2	b_0	b_1	a_3	b_4	b_2	a_5	a_6	\$

Figure 18: Unsorted

Then bwt(aaabbaabaa\$) = aab\$baaaaba, hence the string from L in M_{sorted} read from top to buttom. We notice that the characters tend to stick together in coloumn L. This feature is more obvious with longer strings. As an example, take string S="tomorrow and tomorrow and tomorrow", where bwt("tomorrow and tomorrow and tomorrow")= "wwwdd nnoooaatttmmmrrrrrooo \$ooo", if we were to compress this using run length encoding (RLE) we would get the shortened string "3w2d2 2n3o2a3t3m5r3o2 \$3o", hence we would have successfully compressed the original string using bwt(S) and RLE, such that RLE(bwt(S)) shortened the string from 34 to 23 characters, which is a reduction of appoximetly 26 % [15]. It is easy to see how one could reverse RLE compression back to bwt(S), but it is not obvious how one would reverse bwt(S) to the original string S [15].

F										L
\$	a_0	a_1	a_2	b_0	b_1	a_3	a_4	b_2	a_5	a_6
a_6	\$	a_0	a_1	a_2	b_0	b_1	a_3	a_4	b_2	a_5
a_5	a_6	\$	a_0	a_1	a_2	b_0	b_1	a_3	a_4	b_2
a_0	a_1	a_2	b_0	b_1	a_3	b_4	b_2	a_5	a_6	\$
a_3	b_4	b_2	a_5	a_6	\$	a_0	a_1	a_2	b_0	b_1
a_1	a_2	b_0	b_1	a_3	b_4	b_2	a_5	a_6	\$	a_0
a_4	b_2	a_5	a_6	\$	a_0	a_1	a_2	b_0	b_1	a_3
a_2	b_0	b_1	a_3	b_4	b_2	a_5	a_6	\$	a_0	a_1
b_2	a_5	a_6	\$	a_0	a_1	a_2	b_0	b_1	a_3	a_4
b_1	a_3	b_4	b_2	a_5	a_6	\$	a_0	a_1	a_2	b_0
b_0	b_1	a_3	b_4	b_2	a_5	a_6	\$	a_0	a_1	a_2

Figure 19: Sorted

Recall that bwt(aaabbaabaa) = aabbaaaaba and lets introduce LF-mapping, which provides a subscript (rank) for each character in S, hence each chraracter in S is given a number, equal to the number of times that character accoured previously in S.This proceedure is called S-ranking, since we rank accordingly to S. Ranks are already provided in Figure 19. Notice that the first coloum F and last coloum L have characters that occour in the same order, which is represented by color in Figure 20. This is actually not surpricing feature, since occourences of character c in S are sorted by its right-context in both F and L [15].

F	L
\$	a_6
a_6	a_5
a_5	b_2
a_0	\$
a_3	b_1
a_1	a_0
a_4	a_3
a_2	a_1
b_2	a_4
b_1	b_0
b_0	a_2

Figure 20: Characters in same order

Suppose we want to decompress and find the original string of RLE = 2ab\$b4aba, using bwt. First we start with the trivial step expanding the RLE compression to get bwt(aab\$baaaaba). Then we count the appearences of each character in $bwt(aab\$baaaaba)$ and place these in a coloum F in matrix M, in lexicographical order. Put bwt(aab\$baaaaba) in a coloumn L in matrix M and re-rank according to the number of time each character occour in bwt(S) for both F and L, which we will refer to as B-ranking - as in Figure 21.

F	L
\$	a_0
a_0	a_1
a_1	b_0
a_2	\$
a_3	b_1
a_4	a_2
a_5	a_3
a_6	a_4
b_0	a_5
b_1	b_2
b_2	a_6

Figure 21: B-ranking

We can now reverse bwt(aab\$baaaaba) using matrix M. Start at the first row of M, which have \$ in the first coloumn, and since rows are rotations of the original string S, the coloum to the right of \$, F, must contain the character to the left of \$ in S: a_0 in this case. Hence, we are building the original string from right to left, starting with the termination symbol \$. Now we are in the first row of L containing character a_0, we then find a_0 in F, which mean that the next character in the original string S must be to the right of this index. Continiuing this approach, we end with the string S=aaabbaabaa\$ which is indeed the original string, as demonstrated in Figure 22.

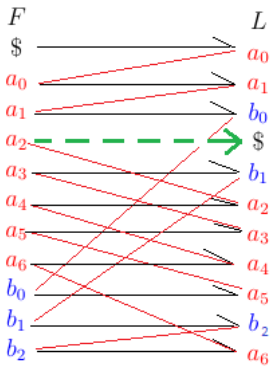


Figure 22: From F–> L to S

The FM-Index provide an opportunity for searching in compressed *bwt* files without full decompressing, which is important for the "Big Data" era of sequencing. We can search in a compressed $bwt(S)$ file using FM-index. Suppose that we want to find an occurrence of pattern P=aab in $bwt(S)$=bc\$ababaaa, so we build a FM-index, as illustrated in Figure . We search P from right to left, and find all occurrences of b in F. Luckily these are grouped together as part of the design and can be found in index 6 through 8 in F. Next we look at index 6 through 8 in L, and find any characters $c = a$, since b is preceded by a in P. Index 7 and 8 in T both have a's, so we look at the ranks of these, and locate the same characters in F with these ranks, which is in index 3 and 4. Then we find occurrences of a at index 3 and 4 in L, since a is preceded by a in P. P is now exhausted, and we

can therefore confirm that there is an occurrence of P=aab in $bwt(S)$=bc$ababaaa. This searching approach find the number of all occurrences of P in $bwt(S)$, but it does not tell us where in the text these occur. FM-index using bwt is a sort of a suffix array, but is easier to compress.

4.8 Application

We have seen two static datastructures, here the suffix tree and the space efficient suffix array, both which can be constructed in linear time complexity and searched in O(n) and O(n log m), respectively. Now we tend to the question regarding applications and operation, here if suffix trees and suffix arrays can be used in the same applications and supports equal operations.
The suffix tree is undouptly one of the most important datastructures in string processing and comparative genomics and once constructed, can efficiently solve a myriad of string processing problems, as demonstrated by Gusfield with almost 70 pages devoted to applications on suffix tress [3, 5]. The application demonstrated by Gusfield can be classified into three kinds of traversals [5]:

- a buttom-up traversal of the complete suffix tree
- a top-down traversal of a subtree of the suffix tree
- a traversal of the suffix tree using suffix links

Figure 23 shows some of the application discussed in Gusfield, with their traversal kind. Although suffix trees are asymptotically linear and plays a huge role in datstructure algorithmics, they are not as widespread in todays software application as expected. There are a few reasons for that. Space consumption of suffix tree are large and act as a bottleneck in large scale applications, since they require 20 bytes per input character. Moreover, it suffers from poor locality of memory references, which causes significant effeciency loss on cached processor architectures and makes it difficult to store in secondary memory [5]. In several geonem analysis and geonem comparison applications the above problems have been identified for suffix trees. But as seen, a more space effecient datastructure exist, hence the suffix array which only consumes $4n$ bytes per input character[5].
Mohamed Ibrahim Abouelhoda *et al.* [5] show that *every* algorithm that uses suffix trees as data structure, can systematically be replaced with an enchanced version of a suffix array (enchanced suffix arrays) and furthermore, solve the same problems in same time complexity. Enchanced suffix array is a datastructure consisting of the suffix array, and some other information or additional tables. Futhermore, enchanced suffix arrays are fast and easy to implement [5].

Application	Type of tree traversal		
	Bottom-up	Top-down	Suffix-links
Supermaximal repeats	√		
Maximal repeats	√		
Maximal repeated pairs	√		
Longest common substring	√		
All-pairs suffix-prefix matching	√		
Ziv–Lempel decomposition	√		
Common substrings of multiple strings	√	√	
Exact string matching		√	
Exact set matching		√	
Matching statistics		√	√
Construction of DAWGs		√	√

Figure 23: Application on suffix trees and their traversal kind [3, 5].

At this time it is now clear how every algorithm using a sufix tree, can be systematically replaced by an algorithm based on suffix arrays [5]. So we look at how enchanced suffix arrays conquer the three kinds of travesals which were classified earlier and whos applications we see in Figure 23.

Buttom-up traversal

We will first look at problems that are usually solved with buttom-up traversal, and show these can be solved with enchanced suffix arrays, using maximal repeated pairs and Zip-Lempel decomposition of a string, as examples [5].
Maximal repeated pairs plays an important role in genome analysis, and the algorithm presented by Gusfield was implemented in the *REPuter*-program [3, 5]. *REPuter*-program uses maximal repeated pairs to find approximate repeats in $O(\|\Sigma\|n + z)$ time, where z is the number of maximal repeated pairs and is based on space effecient suffix trees [5]. Repetitative structures are seen in the field of biology (but not limited to), where genetic mapping requires the identification of certain markers in a DNA that is variable between individuals, called tandem repeats. What varies, is the number of times a substring repeats in an array, refered to as *variable number of tandem repeats* (VNTR) [3]. The VNTR markers are used during the genetic level to search for specific defective genes or used in forensic DNA fingerprinting, since a small set of VNTR can uniquely charactirize an individual in a population [3].

> **Definition**
> A maximal pair in a string S is identical sbstring pairs α and β such that the character at the immidiate left (right) of α is different from the character to its immidiate left (right) of β. So extending α or β in any direction would destroy the equality of both strings [3].

> **Definition**
> A maximal pair (or maximal repeated pairs) is given by the triple (p_1, p_2, n'), where p_1 and p_2 is starting positions for two substrings and n' is their length. We define $R(S)$ as the set of all triples of maximal pairs in S [3].

Given string $S = xabcyiizabcqabcyrxar$, there are three occurrences of the substring *abc*. The first and third occurrences of *abc* do not form a maximal pair, but the first and second form the pair (2, 10, 3), and the second and third forms the pair (10, 14, 3). Note that the definition allow maximal pairs to overlap each other, so to model that case we assume that S has a symbol attacted to the start and end that exists nowhere else in S [3].

> **Definition**
> A maximal repeat α is defined as a substring of S that occours in a maximal pair in S, hence α is a maximal repeat in S is there is a triple $(p_1, p_2, |\alpha|)$ $\in R(S)$ and α occurs in S at startposition p_1 and p_2. We define $R'(S)$ as maximal repeats in S [3].

In the given string $S = xabcyiizabcqabcyrxar$, both *abc* and *abcy* are maximal repeats. Gusfield presents a algorithm using suffix trees, that finds all maximal pairs in $O(n)$ time for a string of length n and we presented the definition for such pairs [3].
To compute maximal pairs, the implementation presented by Mohamed Ibrahim Abouelhoda *et al.* [5] requires three tables: *suftab*, *lcptab* and *bwttab*. The *bwttab* contains the

Burrows and Wheeler transformation (Section 4.7), *lcptab* contains the longest common prefix (Section 4.6, though here it is a tree representation), *suftab* is the suffix array (Section 4.3 and 4.4)[5]. These tables are accessed in sequencial order, which leads to an improved cache coherence and reducing running time, such that maximal repeated pairs can be computed in $O(|\Sigma|n + z)$ time [5]. Mohamed Ibrahim Abouelhoda *et al.* [5] and Kasai *et al.* [16] proved that it is posible to compute maximal repeated pairs with suffix arrays and some additional information. We will not dwell into the algorithm for computing maximal repeated pairs, we use this to emphecies that suffix arrays can be used in the same applications which usauly solve problems with buttom-up traversal on suffix trees. Although as an example, we will show how to compute the Ziv-Lempel decomposition using suffix arrays, which is a lossless compression algorithm [17].

$S[i]$	a	c	a	a	a	c	a	t	a	t	\$
i	0	1	2	3	4	5	6	7	8	9	10
s_i	0	0	0	2	0	1	0	0	6	7	0
l_i	0	0	1	2	3	2	1	0	2	1	0

aca ⇩ aacatat\$

$S[i]$	a	c	a	a	a	c	a	t	a	t	\$
i	0	1	2	3	4	5	6	7	8	9	10
s_i	0	0	0	0	0	1	0	0	6	7	0
l_i	0	0	1	1	3	2	1	0	2	1	0

Figure 24

We follow the example conducted by Mohamed Ibrahim Abouelhoda *et al.* doing a Ziv-Lempel decomposition of the string S=acaaacatat\$, but with a minor correction. Mohamed Ibrahim Abouelhoda *et al.* claim that there exist a prefix of i=3 of S where l_i=2 is the longest prefix of $S[i \dots n]$ that is also a substring of S starting at some position $j < i$, hence $S[0 \dots i-1]$. So $S[3 \dots n]$ = aacatat\$ and $S[0 \dots 2]$= aca, which must mean that the longest prefix of $S[3 \dots n]$ which is also a substring of $[0 \dots 2]$ have to be a, with a length of 1, hence l_i=1 and s_i=0 and not l_i=2 and s_i=2 as claimed. This correction should yield the Ziv-Lempel decomposition in Figure 25 [5].

B	1	2	3	4	5	6	7	8
i_B	0	1	2	3	4	7	8	10
B-th block	a	c	a	a	aca	t	at	\$

Figure 25: Ziv–Lempel decomposition of S=acaaacatat [5].

An interesting application of suffix trees is the *lca* (Lowest Common Ancestor) problem, that is, finding the lowest common ancestor of node i and j in tree T. Lowest common ancestor was first obtained by Harel and Tarjan (1984, published online 2006 [18]) and later on simplified by Schieber and Vishkin (1988, published online in 2006 [19])[3].
Lowest common ancestor is an interesting application given that it is used in application as exact matching with wild cards and the k-mismatch problem, amongst others [3]. More interesting is the fact that *lca* of leaves i and j identifies the longest common prefix of suffixes i and j, which will be discussed later on.
By consuming linear time amount of preprocessing a suffix tree, that is a rooted tree, any

two nodes can be identified and their *lca* can be found in constant time, $O(1)$ [3, 20]. This paper will not dwell into the different linear time preprocessing algorithms for the *lca* predicament, but delivers an overview and clarification of the problem by introducing a simpler but slower algorithm. (maybe linear in the appendix?).

> **Definition** In a rooted tree T a node u is an ancestor of node v, if u is an unique path from the root to v [3].

> **Definition** In a rooted tree T, the lowest common ancestor of two node u and v, is the deepest node in tree T that is an ancestor of both u and v [3].

Let's suppose for simplification that an application is allowed preprocessing time of an upper bound of $\theta(nlgn)$, which is an acceptable bound for most applications [3]. Then, in the preprocessing state of tree T, perform a deept-first traversal of tree T and create a list L of nodes in order as they are visited. Then locating the *lca* of node 2 and 8, $lca[2, 8]$, in fig. 26, one only have to find any occurrences of 2 and 8 in L. Then take the lowest value in interval between $L[1] = 2$ and $L[12] = 8$. This value is the lowest common ancestor for node 2 and 8 in T, $lca[2, 8] = 1$.

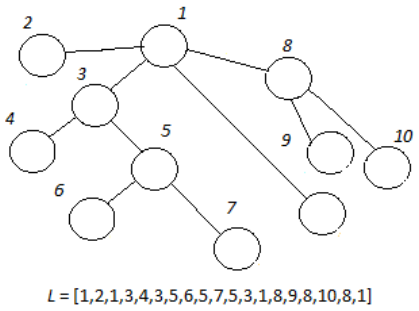


Figure 26: Rooted tree - deept-first travesal with $L = [1, 2, 1, 3, 4, 3, 5, 6, 5, 7, 5, 3, 1, 8, 9, 8, 10, 8, 1]$

Ibrahim Abouelhoda *et al.* did an experiment with two different programs computing maximal repeated pairs, using different files (listed here **??**)[5]. *REPuter* is based on suffix tree and *esarep* is based on enchanced suffix arrays [5]. The result is displayed in Figure 27. The program *esarep* used almost half of the space required for *REPuter* and in this case 2-3 times faster and in 4-5 times faster. This comparison emphzise the advantages of enchanced suffix arrays over suffix trees [5].

ℓ	Running time for *E. coli* ($n = 4{,}639{,}221$) in sec.					Running time for *Yeast* ($n = 12{,}156{,}300$) in sec.				
	maximal repeated pairs			*esasupermax*		*maximal repeated pairs*			*esasupermax*	
	#reps	*REPuter*	*esarep*	#reps		#reps	*REPuter*	*esarep*	#reps	
20	7799	3.28	0.79	899	0.16	175455	9.71	2.23	6432	0.47
23	5206	3.28	0.78	642	0.15	84115	9.63	2.16	4069	0.47
27	3569	3.31	0.79	500	0.15	41400	9.72	2.14	2813	0.45
30	2730	3.30	0.80	456	0.15	32199	9.69	2.14	2374	0.46
40	840	3.29	0.79	281	0.15	20767	9.57	2.13	1674	0.44
50	607	3.29	0.79	196	0.14	16209	9.64	2.12	1354	0.44

ℓ	Running time for Hs21 ($n = 33{,}917{,}895$) in sec.				
	maximal repeated pairs			*esasupermax*	
	#reps	*REPuter*	*esarep*	#reps	
20	40193973	54.63	24.00	188695	1.50
23	19075117	51.78	14.62	138523	1.44
27	8529120	47.97	9.88	98346	1.39
30	4787086	46.54	8.15	77695	1.34
40	732822	45.06	6.21	35719	1.23
50	149482	44.33	5.85	16392	1.19

Space requirement in megabytes			
	REPuter	*esarep*	*esasupermax*
E. coli	61	31	31
Yeast	160	83	83
Hs21	446	227	227

Figure 27: Table for computing maximal repeated pairs and supermaximal repeats. Running times are in seconds and space requirements are in megabytes. The number of repeats of length $\geq$ l is displayed in column titled *#reps*. Space requirement is independant of l [5].

Top-down traversals

Exact string matching is usualy computed in a top-down approach when using suffix trees as datastructure as illustrated in Figure 23. The starting positions of P in T are is displayed on every leaf in the subtree below the point of the last match, demonstrated in Figure 28. We match the characters of P down the unique path in T until P is exhausted or a character in P can not be matched. So if P is fully matched from the root along some path in T, we can find all occurrences of pattern P in T by buttom-up travesing the subtree below the end of the matching path and note the leaf indexs encountered. We can do this because every internal node has at least two children, so the number of leaves is proportional to the number of traversed edges [3].

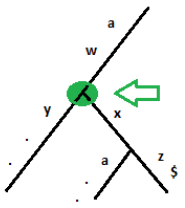


Figure 28: Suffix tree T for S=awyawxawzz, where there are three occurrences of the pattern P=aw in T, with their positions accordingly [3].

Another application exploiting the top-down approach is the exact set matching, where the problem consist of locating all occurrences from a set of patterns P_{set} in some string S, where the set is sent as an input all at once [3].
We define a *keyword tree* as:

Definition

The *keyword tree* T_{key} for the set of patterns P_{set} is a rooted directed tree, which must satisfy three conditions:

- All edges are labeled with exactly one character
- Any two labels comming from the same node, must have distinct labels
- Any pattern P_i in P_{set} maps a node v of T_{key} in a way that all characters from the root to node v spells out pattern P_i [3].

Suppose that we have a set of patterns P_{set} = {potato, poetry, pottery, science, school} and its keyword tree defined as illustrated in Figure 29. Since no two labels comming from the same node have identical labels, we can use the keyword tree to search for any occurrences of P_{set} in string S. Notice that we preprocess P_{set} into a *keyword tree* P_{key}, such that that we can find all occurrences of patterns in P_{set} in S by taking each position p in S and follow the unique path from r in T_{key} which matches a substring in S starting at character p. The dictionary problem is one of which where the set matching effeciently solves the problem. In this problem, a known set of strings, forming a dictionary is preprocessed to which a sequence of individual string is presented to the dictionary. Each of these strings is then either contained or not contained in the dictionay. The keyword tree solves ecactly that [3]. Mohamed Ibrahim Abouelhoda *et al.* [5] proved that any application that uses top-down traversals on suffix trees, can be solved using suffix arrays with some extra information [5]. Mohamed Ibrahim Abouelhoda *et al.* conducted experiments for answering enumeration queries, with three different programs: *streematch* (linked list representation of suffix trees), *mamy* (suffix arrays with additional buckets) and *esamatch* (based on enchanced suffix arrays) [5].

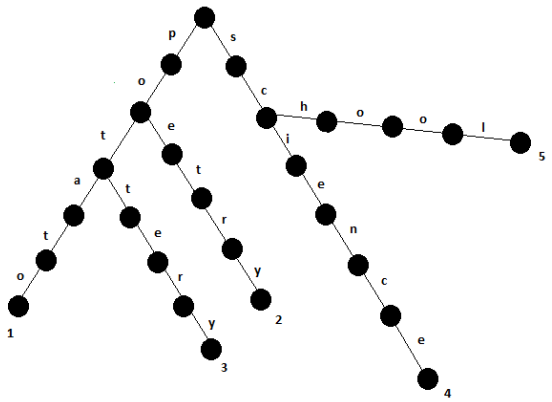


Figure 29: The *keyword tree* T_{key} over the set of patterns P_{set}={potato, poetry, pottery, science, school}.

The expriments were conducted on five different files listed below:

labelFILESE. coli : The complete genome of *Escherichia* bactirium (DNA) - Σ=4, length 4,639,221 Yeast : The complete genome of *Saccharomyces cerevisiae* (DNA) - Σ=4, length 12,156,300 Hs21 : A complete collection of protein sequences - Σ=4, length 33,917,895 Swissprot : Complete collection of protein sequences - Σ=20, length 29,165,964 Shaks : A collection of the complete work of William Shakespear - Σ=92, length 5,582,655 bytes

As illustrated in Figure 30 the program using enchanced suffix array *esamatch*, outperforms all other programs in both time and space consumption, except the file *Sharks*, which is explained by the large alphabet size. The program *esamatch* can thereforee compete with the other programs for small alphabets [5].

File	Running time for *minpl* = 20, *maxpl* = 30			Running time for *minpl* = 30, *maxpl* = 40		
	streemach	*mamy*	*esamatch*	*streemach*	*mamy*	*esamatch*
E. coli	9.47	5.56	**4.48**	9.63	5.70	**4.69**
Yeast	12.42	8.26	**5.37**	12.56	8.46	**5.80**
Hs21	20.15	12.50	**7.23**	20.43	12.69	**7.30**
Swissprot	41.78	9.55	**6.22**	40.80	10.09	**6.25**
Shaks	15.61	**4.29**	72.44	15.78	**4.37**	66.60
	Running time for *minpl* = 40, *maxpl* = 50			Space requirement		
	streemach	*mamy*	*esamatch*	*streemach*	*mamy*	*esamatch*
E. coli	9.86	5.87	**4.85**	56	40	47
Yeast	13.34	8.63	**5.74**	146	106	120
Hs21	21.22	12.88	**7.61**	407	296	327
Swissprot	42.96	9.83	**6.39**	320	288	281
Shaks	15.88	**4.49**	67.16	52	48	60

Figure 30: Space requirements and running times in megabytes and seconds, respectively. The programs did one million enumeration queries searching for exact patterns in the input strings. Minimal (*minpl*) and maximal (maxpl) are the minimal and maximal size of the patterns, respetively [5].)

Suffix links

Size of suffix trees can be reduced with the help of matching statistics, which is needed in more complex problems than exact string matching. Matching statistics are central to a fast approximate matching method designed for rapid database searching, it furthermore provide a bridge between exact matching methods and appromate string problems [3].

- **Notation** Let ms(i) denote the matching statistics for some i in string S

Preprocess suffix tree T for the fixed short string S_p and keep suffix links duing the construction of the tree. We define a suffix link as:

> **Definition** Let an arbitrary string be denoted by $x\alpha$, where x is a single character and α a possible empty substring. For an internal node v with a path label $x\alpha$, if there exists another node $s(v)$ with the path label α, then a pointer from v to $s(v)$ is a suffix link.

The suffix can then be used to find sm(i) for all i in S. Let S_p be a string of length m, then a matching statistics ms$i$ of S_p for all i in S, is a table of pairs (l_p, p), where $0 \leq j \leq m-1$ and the following holds:

- $S_P[i \dots j + t_p - 1]$ is the longest prefex of $S_p[j \dots m-1]$ that is a substring of S.
- $S_p[j \dots j + j + l_p - 1] = S[p_j \dots p_j + l_j - 1]$

Supose we have S= cacaccc and S_p=caacacacca, then we would construct the suffix tree for S keeping the suffix links. Then for each position p in S_p we would find the length

of the longest common prefix starting at some position in S. If $S_p[0]$=caacacacca, then the longest common prefix have length of 2 and accour in position 0 in S. Next one is $S_p[1]$=aacacacca, where the length of the longest commmon prefix is 1 in $S[0]$ and $S_p[2]$=cacacca with LCP length of 4 occouring in $S[1]$. Continuing this approach we get the matching statistic table in Figure 31 [5].

j	0	1	2	3	4	5	6	7	8	9
(l_j, p_j)	(2, 0)	(1, 1)	(4, 1)	(6, 0)	(5, 1)	(4, 2)	(3, 3)	(2, 4)	(2, 2)	(1, 3)

Figure 31: Matching statistic table for S= cacaccc and S_p=caacacacca [5].

The algorithm provided by Chang and Lawler [5] for computing matching statistic used suffix links and could solve the problem in $O(n + m)$ time. Later on Mohamed Ibrahim Abouelhoda *et al.* [5] proved that any problem that used suffix links with a suffix tree datastructure, could be solved using suffix arrays with some extra information in same time complexity as the algorithm presented by Chang and Lawler[5]. Mohamed Ibrahim Abouelhoda *et al.* performed an experiment where two programs computed the matching statistics on a pair of genomes, using a suffix tree and an enchanced suffix array as data structure, respecively [5]. The result displayed in Figure 32 revealed that *esams* consumed 30-40% less space in respect to *streems*, although the latter were up to three times faster [5].

Genome pair	Total length	*streems*		*esams*	
		time	*space*	*time*	*space*
Streptococuss 2	4,199,453	4.1	30	11.1	21
E. coli 2	10,107,957	13.3	65	18.9	43
Yeast 2	24,690,687	41.0	170	43.4	109
Human 2	67,739,601	169.2	472	314.0	294

Figure 32: Running time in seconds and space consumption in megabytes for matching statistics. The program *streems* uses suffix tree as data structure, where the tree construction time is not included. The program *esams* uses enchanced suffix array as datastructure, here a suffix array with some extra information [5].

4.9 SAIS-FLS - Space requirement reduction for fixed length strings

Suppose that string S consists of fixed length strings concatenated together, where each string $S_0, S_1, ..., S_{n-1}$ is terminated with the sentinel \$ such that $S = S_0\$S_1\$...S_{n-1}\$$.

Let $S = S_0\$S_1\$...S_{n-1}\$$ consist of concatenated fixed length distinct strings, such that $|S_0| = |S_1| = \ldots = |S_{n-1}|$ and each string in S is terminated by the sentinel.

Let the length of pattern P, $|P|$, be same length as the fixed length distinct string in S, such that $|P| = |S_0| = |S_1| = \ldots = |S_{n-1}|$.

Suppose that the string $S = jazz\$fuzz\$quiz\$$ is given and suffix array SA for S has

been computed, such that $SA = [14, 4, 9, 1, 5, 12, 0, 10, 11, 6, 13, 3, 8, 2, 7]$ as illustrated in Figure 33.

S=jazz$fuzz$quiz$

SA	Suffixes
14	$
4	$fuzz$quiz$
9	$quiz$
1	azz$$fuzz$quiz$
5	fuzz$quiz$
12	iz$
0	jazz$fuzz$quiz
10	quiz$
11	uiz$
6	uzz$
13	z$
3	z$fuzz$quiz$
8	z$quiz$
2	zz$fuzz$quiz$
7	zz$quiz$

Figure 33: Suffix array and suffixes for S=jazz$fuzz$quiz$

By means of the Binary Search algorithm, suppose we want to find pattern $p_0 = jazz\$$, $p_1 = fuzz\$$ and $p_2 = quiz\$$ in the suffix array for S, such that $SA[i]$ pattern$p_0 = jazz\$$,$p_1 = fuzz\$$ must be a suffix of $T[SA[i]]$. Then $p_0 = jazz\$$ is a suffix of $T[SA[0]]$, $p_0 = fuzz\$$ is a suffix of $T[SA[5]]$ and $p_0 = quiz\$$ is a suffix of $T[SA[10]]$. Now suppose, that we are only interested in exact matching, and do not care for unnecessary suffixes, then notice that we can match all fixed strings in S, with merely three indices in SA for S, which leads to 12 indices in SA for S that are never used when exploiting exact string matching.

> Let N_DS denote the number of fixed length distinct strings in $S = S_0\$S_1\$...S_{n-1}\$$, where n is number of characters in S.

This paper introduce an algorithm that reduce suffix array size for fixed length exact matching, from $O(n)$ to $O(N_DS)$ space complexity with linear time construction.

```
SAIS–FLS(string S, array SA, int len)
        let SA_FLS = new array[int]()
        for i=0 to i < length(SA) - 1 do:
                if(SA[i] NOT EQUAL TO length(S) - len
               && S[SA[i] + len] EQUALS ''$'')
                        then PUT i in SA_FLS
        return SA_FLS
```

Suppose that the length of the strings in S, len, is known, then scan SA once, from left to right, and find any index where $T[SA[i] + len] = \$$ and add the elements to the new array SA-FLS in $O(m)$ time, where m is the length of SA. Constructing the new suffix array for S using SAIS-FLS, all unnecessary indices in SA are removed and the new array maintain the lexicographical order.

> **Lemma 6.9-1** $SAIS - FLS$ return a new array $SA - FLS$ that is sorted in lexicographical order.

Proof By Contradiction

Let S be a string of strings, where each string is concatenated with the termination symbol \$.

Let SA be the suffix array for string S and let n denote the number of characters in SA. Suppose SA is sorted lexicographical for all suffixes in S.

Suppose that S[SA-FLS[i]] to S[SA-FLS[j]], where i < j < |SA-FLS| is sorted in lexicographical order. Suppose that $S[SA - FLS[j + 1]]$ is lexicographical smaller than $S[SA - IS - FLS[j]]$, that would suggest that SA for S is not sorted lexicographical for all suffixes in S, which is a contradiction. Furthermore, since SA is scanned from left to right and supposed sorted in lexicographical order, each item put in $SA-FLS$ must have been appended in lexicographical order.

> **Lemma 6.9-2** $SAIS-FLS$ return a new suffix array, $SA-FLS$, containing all indices from SA for $S = S_0\$S_1\$...S_{n-1}\$$ where $S[SA - FLS[i] + len] = \$$, $len =$|S$_0$| = $|S_1|$=...=$|S_{n-1}|$\$ and $0 < i < n$.

Proof By Contradiction

Suppose that there exist some i and j, $i < j$, in SA, $0 < i < j < |SA|$ and $len = S_0$ in $S = S_0$,S$_0$ =\$, where $S[SA[i] + len] = \$$ and $S[SA[j] + len] = \$$. Suppose that SA-FLS contain one item, that would suggest that $i = j$ which is a contradiction.

Suppose that SA is sorted in lexicographically order for all suffixes in $S = S_0\$S_1\$...S_{n-1}\$$ where $S_0, S_1, \ldots, S_{n-1}$ does not contain the termination symbol \$ and $len = |S_0| = |S_1| = \ldots = |S_{n-1}|$.

Suppose that all indices from SA, where $S[SA[i] + len] = \$$, $0 < i < n$, has been successfully added to the array $SA-FLS$. Suppose that there exists some j in $SA-FLS$ where $S[SA - FLS[j] + len] = \$$, that would suggest that there exists an index $SA[i] = SA[j]$ where $S[SA[i] + len] = \$$, but that is a contradiction, since only indices that are bound by $S[SA - FLS[i] + len] = \$$ was added to $SA - FLS$.

Lemma 6.1-1 and Lemma 6.1-2 suggest that $SA - FLS$ contains indices in lexicographical sorted order and are bound by $S[SA - FLS[i] + len] = \$$. Furthermore, the length of $SA - FLS$ is proportional to the number of the fixed length distinct string in $S = S_0\$S_1\$...S_{n-1}\$$. For large fixed length strings such as SHA1, SHA256 or MD5 hashes, $SA - FLS$ concededly reduce the number of indices stored. A string consisting of 27.000.000 MD5 hashes would produce a suffix array consisting of 27.000.000 x 33 = 891.000.000 indices, while SA-FLS contains only 27.000.000 indices, which is a reduction factor of 33. For the Sha256, the reduction factor would be 257, hence the length of the hash plus the termination symbol.

4.10 Suffix Array Construction Algorithm (SACA) comparison

We saw examples of preprocessing unstructured text into suffix trees and suffix arrays, examples of searching in $O(n)$ or O(n log m) time complexity and transformed strings that

are easier to compress using Burrows-Wheelers Transform. We introduced an algorithm that could reduce space requirements for fixed length strings with a factor proportional to the length of the fixed length string in S. Space usage is a factor, since sizes of data sets in some applications, as molecular biology, data compression, data mining and text retrieval, to name a few, can be incredibly large. In many cases the alphabet size |E| is typically a fixed constant, such as ASCII E=256 or E= 4 for DNA sequences. In such cases suffix trees and suffix arrays are larger than the text by a multiplicative factor of $O = (log_{\|E\|}n) = O(logn)$. To illustrate this, suppose that we have a DNA sequence of n symbols ($\|E\| = 4$) which we would store on a computer with 2n bits. A suffix array for that DNA sequence would use 4 bytes for each n words or 32 bits, which is 16 times larger than the text itself. The question of space usage is important in both theory and practice, since data sets can be incredibly large, especially in the "Big Data" era of next generation sequencing (SAIS-OPT). Nataliya Timoshevskaya et al. presented in 2014 a characterization and optimization of the SA-IS algorithm for suffix array construction, focusing on irregular memory access, using concepts from Burrow-Wheeler Transform, which achieves a 27% improvement in performance on tested data on the already tuned implementation of SA-IS in the Burrow Wheeler aligner used by the bioinformatics community (SAIS-OPT). Roberto Grossi et al. presented compressed suffix arrays and suffix trees that as a concrete example could compress a typical suffix array of 100 megabytes ASCII file to 30-40 megabytes or less, while the raw suffix array would require 500 megabytes (compressed suffix trees).

5 Malware - Malicious Software

The term malware spring from the two words of malicious and software, and are used to designate any unwanted software [21]. Malware is also referred to as malicious software, malicious code (MC) and malcode [22]. It was defined by G. McGraw and G. Morrisett as "any code added, changed or removed from a software system in order to intentionally cause harm or subvert the intended function of the system." and virus was defined as " a generic term that encompasses Viruses, Trojans, Spyware and other intrusive code." [22, 21]. A more canonical example of malware include viruses, worms and Trojan Horses and are characterized by the ability of replication, propagation, self-execution and corruption of a computer system which compromise confidentiality, integrity and effect denial of service [21]. The most common way of infecting a system is to transfer malware from a polluted system, using either network or local file system, to an uninfected system [21]. However, the variety of known and unknown malware is key to understanding the difficult problem of detecting malware. Categorizing malware have become more complex, as new versions appear, which is a combination of those who belong to an existing category [22]. Replication and invisibility are the property and characteristic of malware, respectively. Invisibility is used to evade themselves from detection by anti-malware and replication is used to ensure existence and in some cases the replication exhaust computer recourses in the vein of RAM and hard disk [21]. Malware exploit operation system vulnerabilities, software bugs and foliage itself such that it starts with the same lifecycle as the system. In other cases, malware are remotely controlled on an supplementary system, controlling the infection [21].
The first anti-malware program, Flushbot Plus, was invented by Ross Greenberg in 1987 and was used to prevent viruses and Trojan horses making unwanted changes to files [21]. John McAfee released VirusScan(TM) program in 1989 which could detect and repair

viruses simultaneously [21]. Los Alamos National Laboratory developed statistic-based anomaly detector which were rule-based on statistical analysis used with anomaly detection [21]. In 1990, inductive learning of sequential user patterns combined with anomaly detection was used by the Time-based Inductive Machine (TIM), masking on access matrices for anomaly detection was used by the Network Security Monitor (NSM) and the Information Security Officer's Assistant (ISOA) deployed statistics, profile checker, and a expert system, amongst others [21].
Although reports claim that malware continue to grow, researchers and manufactures evolve new methods to improve techniques for building anti-malware [21], here categorizing malware into groups. Furthermore, technological solutions increase the effectiveness and performance of malware detection systems with use of cloud computing, network-based detection, web, virtual machines, agent technologies or hybrid methods and technologies [21]. There are essentially two phases in the of software lifecycle in which malware is inserted, a pre-release and a post-release phase. In the pre-release phase, a internal threat or insider, usually a trusted developer within an organization, insert malicious code into software before it is released to end-users. Code inserted after a release to its end-users, is called post-release phase. The most popular names for users or creators of malware are black-hats, hackers and crackers and could be external/internal threats, a foreign government or an industrial spy [22]. Malware creation generally employ obfuscation or behavior addition/modification, here hiding the true intention of malicious code without behavior extensions exhibited by the malware where behavior addition/modification effectively creates new malware, respectively [22]. Researchers suggest that reuse of code is a major component of developing malware which plays a critical role in some signature based malware detection systems, here misuse detection methods [22]. Detection systems are developed by software companies, analyze and keep track of new programs, where valid programs are put in so-called white lists and malicious programs in grey lists. The program figuring on the grey list are scanned and classified in a controlled environments. If a grey list program-analysis result in a new malware, the software company release an update for end-user product databases [21].

5.1 Malware naming and classes

According to Microsoft [23], a specific malware with particular behavior can take more than one name depending on antivirus vendors naming, which depends on number of samples collected as well as the particular malware behavior [23]. CARO is a common method for naming malware and is a malware naming scheme developed by antivirus companies and researchers [23]. CARO does not solve the ambiguous class label problem but tries to address the incoherent labeling in general. CARO offers an universal standard for malware naming to prevent confusion between users and antivirus vendors [23]. The naming scheme is as follows:

<malware type>://<platform>/<family name>.<group name>.<infective length>.<sub variant><devolution><modifiers>

A structural example of using CARO is as follows (detailed illustration in Figure 34):

Backdoor: Win32/Caphaw.D

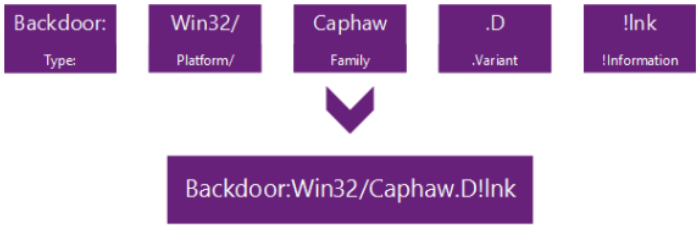


Figure 34: CARO naming scheme with a detailed description of a backdoor [23].

Imtithal A. Saeed et al. [21] divides malware into two classes: ordinary malware and network-based malware. Supplementary classifications are made depending on malware characteristics, here to facilitate authorship, correlation, information and identifying new variants. This classification is made to categorize malware into groups depending on network and web usage [21]. In Figure 35 lists the major malware families, which is explained below.

Factors of comparison \ Malware family		Spyware	Adware	Cookies	Trapdoor	Trojan horse	Sniffers	Spam	Botnet	Logic bomb	Worm	Virus
Creation techniques	Pattern	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
	Obfuscated	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
	Polymorphic	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
	Toolkit	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
Execution environment	Network	✔	✔	✔	✔	✖	✔	✔	✔	✔	✔	✖
	Remote execution through web	✔	✔	✔	✔	✔	✔	✔	✔	✖	✖	✖
	PC	✖	✖	✖	✖	✖	✖	✖	✖	✔	✔	✔
Propagation media	Network	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
	Removable disks	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
	Internet downloads	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
Negative impacts	Breaching confidentiality	✔	✖	✔	✖	✔	✔	✖	✖	✖	✖	✖
	Inconveniencing users	✖	✔	✖	✖	✖	✖	✔	✖	✖	✖	✖
	Denying services	✖	✖	✖	✔	✖	✖	✔	✔	✔	✔	✔
	Data corruption	✖	✖	✖	✔	✖	✖	✔	✔	✔	✖	✔

Figure 35: A comparison scheme of major malware families [21].

Network-based malware

Spyware:
A malware installed secretly on a users computer with the purpose of gathering information without the awareness and consent of the user, is called spyware. Microsoft and Google intentionally collect information from users using their own developed spyware [21].

Cookies:
Information stored on user's computer by their web browser with the purpose of authenticating the user depending on the information stored. Cookies stores site preferences and server-based sessions and are not executables but text formatted files. Cookies are not harmful by themselves but are in cooperation with spyware [21].

Adware:
Advertising-supported software that automatically play advertisement on user's computer without desire. Financial profit in the main objective of adware and are not harmful, but form pop-up windows that interrupts user's. Adware integrated with key loggers and other privacy-invasive software can though be harmful [21].

Backdoors:
Backdoors are malicious code integrated in an application or operating system with the purpose of granting programmers access to the system bypassing ordinary authentication methods. Trapdoors are a security problems because they give full access to the system without authentication and can be remotely accessed by attackers. Though backdoors are written by experts and specialized developers for friendly usage [21].

Trojan horse:
Code that appears useful but actually steals information [21].

Sniffers:
Programs that intercepts and record network traffic. Packages are intercepted and captured, here to decode and extract raw data, to gain access to fields and their content. Information gathered this way can later be used to launch an intrusion attack [21].

Spam:
Spam is a software package that sends identical email messages to numerous email addresses. This form of spam can cripple systems, as possibly thousands emails consume bandwidth [21].

Botnet:
A collection of infected computers which all contain embedded bot software and are controlled by a hacker. The collection of bot-infected computers are then used to execute malicious functions, bypassing any ordinary authentication that would normally be needed to acquire control of the computers. Denial-of-service (DoS) attacks uses botnet software [21].

Ordinary malware

Virus is software code that can potentially replicate itself during infection, to other applications of documents. Virus is code attached to application software using three different methods: pre-pending, embedding, and post-pending. As an example, the Autorun.inf file is target by malware developers with the purpose of replacing or adding malicious code. The Autorun.inf resign in a removable disk or storage device with the job of playing the disk or storage device automatically. Whenever a storage device or disk enters the system,

the operation system searches for the Autorun.inf file and executes it. Consequently, the virus will be infecting the system as the operation system execute the Autorun.inf file [21]. Another ordinary malware, are Worms, which are self-replicating software, that does not require a host program, but work independently. Worms create copies of themselves to increase the spread rate, though this characteristic is used by antivirus scanners, to locate numerous files with identical attributes which may indicate a malware infection. Worms roaming on a server can likewise consume bandwidth, preventing normal users to access the server [21].
Malicious code that remains quite until some unambiguous condition is met, typically a date and time, is called a logic bomb. Consequently, the logic bomb activates and executes when the condition is met and can have a huge impact on a systems confidentiality, or preventing online services or just sabotage files [21].

5.2 Current threat

Everyday services are accessible, as web-banking, e-shopping, social media and general communication, through the internet and thus plays a vital role in our daily life. In an increasing growing and global market, the internet has become a enormous information and communication network. People do transactions and relations everyday through the internet which consequently make malware, a program that aid people achieve their malicious intentions and goals [23].
Developers and criminals will exploit threats and vulnerabilities as long as they exists, which seems they do not. The largest and most noteworthy vulnerability found, was the so-called Heartbleed-bug. The bug was discovered in the Transport Layer Security (TLS) heartbeat function and was announced in 2014 [23]. TLS (SSL version 3) is an enhanced version of TCP with security services, including confidentiality, data integrity and end-point authentication [24]. Attackers and criminals could potential exploit this vulnerability and get access to web application memory. The web application memory could potentially contain sensitive data as usernames and passwords, emails and documents [23].
Another recent threat for organizations and businesses is the ransomware Cryptowall which locks and encrypts all programs and files on the system and hereafter demands a pay or ransom from users or groups in order to unlock the files and programs. This payment is usually in bitcoins, which is a peer-to-peer electronic cash system that allow online payments to be sent to one party to another without going through a financial institution [23] [25].

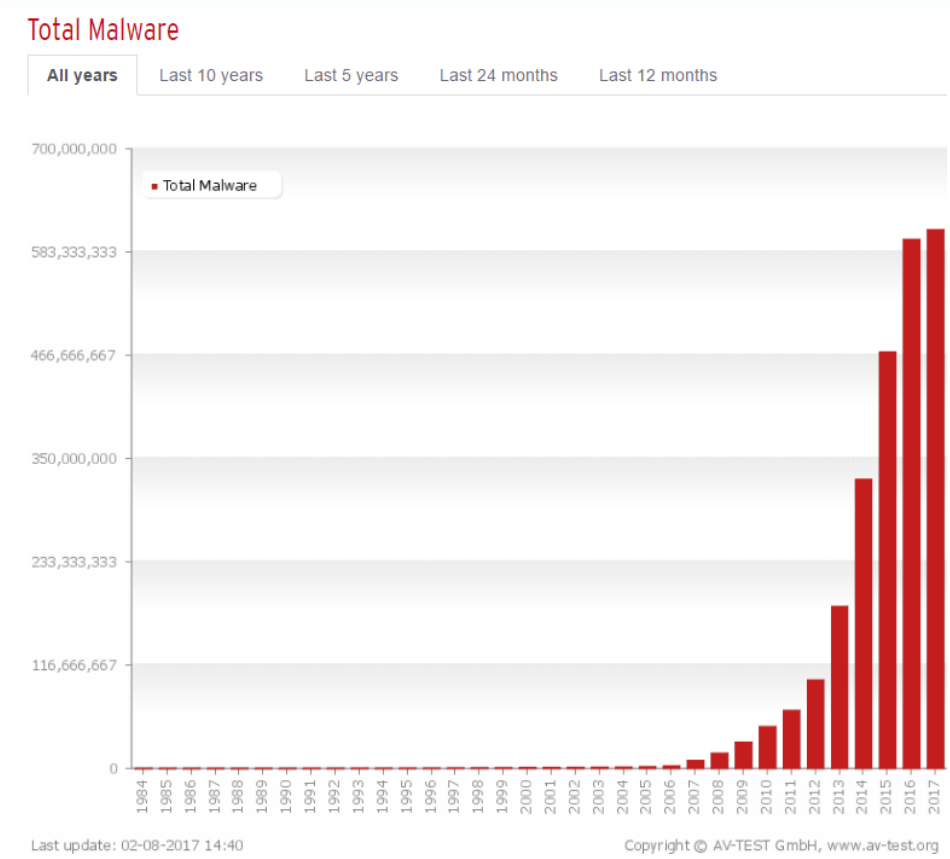


Figure 36

With the exponential growth of the internet, new malware is created every day. According to AV-TEST, an independent IT-security institute, 390.000 new malicious programs are identified each day, bringing the total malware count above 580.000.000 so far as illustrated in Figure 36 [26]. It therefore imperative for security companies to detect and analyze new malware and notify users and companies about new vulnerabilities and threats [23].

5.3 Detection techniques

Malware detection techniques can roughly be categorized as either anomaly-based detection or signature-based detection. Signature-based techniques scrutinize and evaluate programs against a dictionary of malware signatures in a database. The characterization of malware signatures are key to a signature-based technique and the benefit is effectiveness. The shortcoming is that signature-based techniques cannot shield against unknown malware [21, 22]. Anomaly detection techniques exploit knowledge of what is constituted normal behavior to scan programs and decide the grade of maliciousness [22]. If a program violate the constituted normal behavior it is considered malicious [22]. The two detection techniques employ one of three different approaches: static, dynamic, or hybrid. Each approach determine the information-gathering technique for the two categories, here the information to detect malware. The static approach uses syntax or properties of the program under inspection to determine if the program is malicious. In a signature-based technique using static-analysis approach an attempt is done to detect if a program is ma-

licious before it executes, while a dynamic approach would attempt to detect malicious behavior during program execution or after program execution [22]. A hybrid approach combine both static and dynamic information to detect malware [22].

5.4 Anomaly-based detection

Two phases usually occur when exploiting anomaly-based detection, a training phase and a detection phase. In the training phase a detector attempt to learn the normal behavior and the detection phase determine if a program exhibits unwanted behavior from the normal behavior-information gathered in the training phase [22]. An ability and advantage of anomaly-based technique is to detect unknown malware, reminiscent of zero-day attacks. But this technique is not without limitations. Determining normal behavior-features during the learning phase is complex and have high false-positive rate [22]. As an example, if some exception is not seen and analyzed in the training phase, but monitored during a scanning, it could lead to a false-positive. Therefore, a system can exhibit beforehand unobserved behavior in the monitoring phase which can lead to false-positive, here categorizing a normal non-dangerous program as malicious [22].
As mentioned, detection techniques can exploit three different approaches of information-gathering: static, dynamic, or hybrid. We give examples on all three information-gathering approaches for anomaly-based techniques.

Static anomaly-based detection exploit characteristics gathered about the file structure of a program currently under examination, here to determine malicious code. This make it possible to detect malware without executing the malware program where during the training phase, derived models attempt to determine and characterize different file types on a system based on structural composition [22]. These models are derived from learning file types generally roaming a system. Files are given predictable regular byte composition for their respective types, such that .pdf files have unique byte composition that are different from other file types, as .exe or .doc files [22]. If a file is considered to vary to much from a given model or models it is marked as suspicious and sends it to further analysis by other some other procedure where it is determined if the file is actual malicious [22]. This method is called Fileprint Analysis (n-gram) and applying 1-gram analysis to PDF files implanted with malware had a detection rates between 72.1 and 94.5 percent. Though implanted malicious code in PDF files is embedded either in the head or tail, it is still possible to embed malicious code in the middle of a PDF in such a way that it is possible to execute and open the PDF, hence deploy the malicious embedded code [22].
The dynamic anomaly-based technique take advantage of information gathered during a file execution to determine malicious code. Consequently the detection phase check for inconsistencies from the information learned in the training phase. We will look at some different methods that exploit the dynamic anomaly-based technique. One approach developed by Wang and Stolfo [22] is calculation of expected payloads for each port (service) on a system. This is done by creating a byte frequency distribution which allow for a centroid model. Centroid clustering is to separate a set of information into subsets, here to uncover the location of a centre for each subset, such that the dissimilarity distance between the information and the centre is minimized [27]. Each incoming payload is compared against the centroid model created during the training phase, where the Mahalanobis distance is measured between those two. A Mahalanobis distance is the

distance between two distinct groups (populations). Suppose we have two groups with boys and girls, respectively. Then consider relevant characteristics between individuals of these two groups, say height or weight [28]. Then we let a random vector X contain the measurement made on a given individual [28]. As we are interested in measuring the difference between groups the assumption is to let the random vector X have same variation about its mean, such that the difference between the groups can be considered as the mean vectors of the random vector X [28]. The Mahannobis distance acquiesce a strong statistical measurement of similarities [22]. A payload under investigation is analyzed against the centroid model, measuring the Mahalanobis distance, such that a large Mahalanobis distance consider a payload malicious [22]. Wang and stolfo [22] tested their technique against three weeks of trained data. After two weeks of testing, the approach detected 57 of 97 attacks, given a success rate of approximately 60% and a false-positive rate of under one percent [22]. Taylor and Alves-Foss present a technique that focuses on network protocol vulnerabilities and relies on the assumption that malicious network packets tend to have large number of SYN, FIN and RST packages and low amount of ACT packages [22]. The ACT, SYN, FIN and RST bit is contained in the flag field which is in the TCP segment structure. The acknowledgment bit ACK indicate that the value carried in the acknowledgment field is valid, hence a segment contains an acknowledgment for a segment that is successfully delivered [24]. A TCP connection is established with an end-host (server) by sending a segment with the SYN bit set to one. When the end-host receives the segment, assuming that it actually arrives, it will extract the TCP SYN segment from the datagram, allocate variables and TCP buffers to the connection and hereafter send a segment to the client which grants connection [24]. Within this segment the acknowledgement field of the TCP header is set to $client\_isn + 1$. When the client receives the access-granted segment from the end-host, it too allocate variables and TCP buffers to the connection. The client sends yet another segment, this time setting the flag field of the TCP header to $server\_isn + 1$ and the SYN bit to zero and this segment can potential carry client-to-server data in the payload [24]. This connection procedure is often referred to as a three way handshake [24]. The RST bit is used to close a connection when it is not feasible to perform a three way handshake [29]. Taylor and Alves-Foss used Mahalanobis distance between known attack clusters and normal clusters on FTP, HTTP and SMTP data. Some attacks were easily found whereas others seemed to match some of the generated clustered [22].
Boldt and Carlson took a different approach, here using computer forensic methods to detect privacy invasive software whereas Adware and Spyware are the primary types [22]. The approach consist of creating a clean system, hence a system that is free of privacy invasive software. Then a snapshot of the clean system is considered the baseline for the system in question. When the system baseline is recorded, the system is exposed to privacy invasive software and regular snapshot are recorded. Boldt and Carlson accessed Ad-Ware, the most popular privacy invasive software removal tool at the time and though their technique they found that Ad-Ware found false positives as well as false negatives [22].

Lastly we look at hybrid anomaly-based detection method, which combine static and dynamic anomaly-based detection. A specific malware type referred to as “ghostware” endeavors to stay invisible by deleting itself from the operating system querying utilities. Whenever an user performs a command to list a directory, the malware intercept the command-list of files, and modify it in such a way that it does not figure on the list and

therefore is invisible to the user and cannot be found by Windows Application Programming Interface (API) queries [22]. The stream of API calls are essential equivalent to a program's execution flow, and facilitate user mode processes to services embedded in the kernel of Microsoft [30]. API calls can be made to different functional categories, as registry, memory, sockets, ect. Each API call has an distinctive name, a set of arguments and a return value, where the number and type can vary for dissimilar API calls [30]. Wang et al. [22] used an inside-the-box and an outside-the-box approach. The basic idea is to compare low-level system calls with high-level system calls. A inside-the-box approach performs both a low-level and high-level scan within the same machine. With the outside-the-box approach, a clean host performs a low-level system call without the target host knowing. Then a high-level scan is performed, if there are any differences between the low-level and high-level scan, in either the inside-the-box or the outside-the-box approach, there is a presence of ghostware [22]. The inside-the-box approach did not utilize any false positive, whereas the outside-the-box approach produced some false-positives. Wang et al. [22] used ten ghostware in the experiment [22].

5.5 Signature-based detection

Malware signatures are strings of bytes that are unique for that particular malware program [31]. These signatures are then used to recognize particular malicious software which resign in e.g. executable files, boot records, or memory with a diminutive amount of false positives [31]. In respect to its anomaly-based cousin, signature-based detection are easier to implement and configure [8]. This consequently entail that most commercial systems exploit signature-based detection as part of their implementation [8]. As declared, anomaly-based detection methods have the advantage of being able to detect zero-day attacks, whereas signature-based detection handle these attacks inadequately. However, anomaly-based detection comes with a cost of dealing with a high number of false-positives [8]. Signature-based detection require a repository consisting of information accumulated from known malware signatures [8, 22]. This repository is then searched to assess if a given program have a malicious signature/signatures stored within the repository [22]. Human expertise is exploited in creating malicious signatures and once created, the information is appended to the signature repository [22]. As with anomaly-based detection, signature –based detection employ three different approaches: static, dynamic and hybrid [22].

In static signature-based detection, a program is inspected for sequences of code that could reveal suspicious malicious behavior or intent [22]. Malicious signatures are in general represented by sequences of code and the signature-based detection exploit the knowledge of this, and compare the program in analysis with the information stored in the repository. A programs maliciousness can be precisely determined without execution, which is a major advantage [22].
Static Analysis for Vicious Executable (SAVE) is a method proposed by Sung et al. [22]. Sequences of Windows API calls are represented as a signature for a given virus. The distance (Euclidean distance – a distance between objects in multidimensional space [32, 33]) is calculated between known signatures and the sequence of API calls from the suspicious program [22]. The repository is then searched using three similarity functions, measuring the similarity of the grogram's API calls against the signatures stored in the repository. A

ten percent difference (or less) will result in flagging the suspicious program under inspection as malicious [22]. SAVE was compared against 8 malware detectors: Norton, McAfee Unix Scanner, McAfee, Dr. Web, Panda, Kaspersky, F-Secure, and Anti Ghostbusters. SAVE was the single detector that were able to detect all the variants of the malware used in the study [22].
A diverse approach proposed by Kreibich and Crowcroft [22] is use of honeycomb. Honeycomb is a system that uses honeypots to generate signatures and detect malware from network traffic [22]. Honeypots are automated tools to collect malware and a methodology to lure attackers as automated malware with the intention of studying them [34]. Honeypots can be divided into two general types: low-interaction honeypots and high-interaction honeypots. Low-interaction honeypots offers limited service to a potential attacker and learns about the attack patterns and behavior. Whereas high-interaction honeypots offers a genuine system to interact with, but are more complex to setup and maintain [34]. Furthermore, high-interaction honeypots offers more detailed information about the attacker and an opportunity to learn about proceeding attacks [34]. Kreibich and Crowcroft [22] worked under the assumption that any connection or traffic which were directed to a honeycomb would be suspicious. Odd TCP flags generate a signature for the connection stream, such that the signature is the connection stream that entered a honeycomb modulo the honeycombs response to the incoming connection stream [22]. A horizontal and a vertical detection schemes are used, where incoming stream last n-th message are compared to the n-th message of all streams stored in the honeycomb, and newly arrived streams are aggregated and run though the Longest Common Subsequence algorithm as well with the aggregated form for streams stored in the honeycomb, respectively [22].

Dynamic signature-based detection is characterized by using information gathered during program execution to determine maliciousness. Here behavior patterns reveal maliciousness of the program under investigation [22].
An example of dynamic signature –based detection proposed by Ellis et al. [22], is worm detection using identified malevolent behavior. Four behavioral signatures are identified by means of dataflow monitoring, coming in and out from a solitary node. A signature could be when a server changes into a client, here whenever a worm propagate itself. This occur when a worm compromise a server and acts as an client to infect and spread to other systems (hosts) [22]. Alpha-in and alpha-out is other base signature, which map how worms typically sends similar information across nodes and consequently have same data flow links. Though for some servers, it is not unusually to transmit similar information, here file servers as an example [22]. Ellis et al. [22] analyzed the server-client signature and the alpha-in/alpha-out signature, and found that the server-client signature perfectly detected the worm changes, whereas the alpha-in/alpha-out would be unsupportive with an alpha value of one, with a high false-positive rate [22].

Last but not least, hybrid signature-based detection use both static and dynamic properties to determine maliciousness of a program [22]. Polymorph and self-encrypting viruses are designed to obscure themselves and thus prevaricate pattern patching techniques. A self-encrypting virus encrypt itself, consequently obscuring malicious patterns which would normally be detected by pattern matching detection techniques. Mori et al. [22] propose a method that decrypts a mobile polymorph and self-encrypting virus in an emulator and performs a static analysis of the system calls made by the virus payload [22].

Malicious behaviors are represented by state machines and detection policies are modeled by state machines, which is user-specified. A mobile application is considered malicious whenever a match occur. This technique effectively detected 600 virus/worm samples [22].

6 Implementation - Intrusion Detection

The initial step in the process of implementing our intrusion detection system was to conduct a requirement analysis, to establish key attributes and limitations to our malware intrusion detection system (MIDS). Given that most commercial antimalware systems exploit the signature-based techniques as part of their implementation [8], we adopt the same approach in the implementation of our MIDS. We primarily focus our attention to the static signature-based technique, hence a static data structure of unique malware signatures. First and foremost as it is easier to implement and configure [8]. We found inspiration from a commercial anti malware program, namely Kaspersky Internet Security (version 16.0.1.445(g)). We limited our MIDS to handle internal threats, hence malware which already roams a within a system. With that limitation we can ignore outside threats. Since our main focus is building a MIDS that employ suffix arrays as its main data structure, we defined three primary properties and two secondary properties for our MIDS. Primary 1. Scan a file for maliciousness against our malware signature repository using three different methods: 1. Direct search by means of database queries. The data structure is saved to hard disk as a database file (Microsoft SQL Server Database File.) and loaded when needed. 2. Binary search on a suffix array data structure. Suffix array (including the text) is loaded directly into the memory (RAM). 3. Binary search with LCP values on a suffix array data structure. Suffix array (including the text) is loaded directly into the memory (RAM). 2. Scan a folder for malicious files against our malware signature repository 3. Scan startup files for maliciousness against our malware signature repository Secondary 1. Live background scanning of processes against our malware signature repository. 2. Automatic startup scanner, that scan start up items when system boots up. Primary's should give us the means to compare advantages and disadvantages of using suffix array data structure. Our main goal with the primary attributes is to compare search times for suffix arrays loaded into memory (including the text) for binary search with and without LCP values and a data structure which was saved as a file, and loaded when needed. The first secondary property evaluates and examine the impact of constant scanning a system, whereas the latter examine the impact on system boot time. Our goal is to let the live scanner and automatic start up scanner have as little impact on the system as possible. According to a security report from 2015-2016 by AV-TEST, 85.39% of all identified malware in 2015, was detected in Windows ® [35], of such we have chosen to implement our MIDS on Microsoft's newest Windows® operating system, hence Windows® 10. The programming language came naturally as we needed a language that supported Windows Service application, SQL database and a graphical user interface (GUI). Windows Service applications are executed as the normal lifecycle when Windows boots up and before any program in the startup item list executes, which is ideal for our solution. We decided to create our MIDS in C using Windows forms and Windows service, using Microsoft Visual Studio 2015, as this platform supports all of our needs.

6.1 Data structure - Fixed Length Distinct Strings

Given that most commercial anti-malware systems exploit the signature-based techniques as part of their implementation [8], we adopt the same approach in the implementation of our MIDS. We primarily focus our attention to the static signature-based technique, hence a static data structure of unique malware signatures. First and foremost as it is easier to implement and configure [8].
We were given access to VirusShare.com [virusshare] which is a repository of malware samples. VirusShare.com contains live malware for security researchers, incident responders and forensic analysts, but auxiliary interesting for us, VirusShare.com contains a database of over 27.000.000 MD5 hashes of feasible (not confirmed) malware. MD5 is a message digit algorithm, that takes an input and produce a "fingerprint" of that input [36]. MD5 hashes are weak in the sense of computer security, as its not recommended for digital signatures [**?**] and are breakable [36, 37]. Nevertheless, we can undamaged use it as unique fingerprints of malicious software, given that the chance of randomly finding two different files that produces the same MD5 hash value should be infeasible [38]. Equipped with this knowledge, we are save in assuming that we should be able calculate the MD5 value for any file in a system, and if the calculated MD5 hash value for the given file is represented in the signature repository, we can with (almost) certainty conclude it is malicious. But we cannot guarantee that a given signature roaming the signature repository represents a malicious software, given that we do not know how, or by whom, the malware was acquired. One way of solving this ambiguity, would be to authenticate every MD5 hash from our repository against a commercial repository, say Symantec. Nevertheless, we gave the accumulated repository from VirusShare.com the benefit of the doubt and only checked a handful of random MD5 hashes against the Symantec repository at https://www.symantec.com/security$_r$*esponse/glossary/define.jsp?letter* = *mword* = *md*5 − *hash*, which all checked out.
We decided to build three different data structures for our MIDS: a suffix tree, suffix array and a SQL database structure. The main goal with this approach is to compare construction time, space requirement and maintainability. The suffix tree implementation employed was found at
https://github.com/atillabyte/SuffixTree/tree/master/src/SuffixTree. For the suffix array we used an implementation of the SAIS algorithm based in Ge Nong et al [7] from https://github.com/JoshKeegan/SAIS-CSharp. For our SQL database structure, we used Microsoft SQL Server Database File.
We noticed that we are not dealing with one single text, but a concatenation of numerous distinct strings. To overcome this problem, we adopt the approach by Gusfield [7] to build a generalized suffix tree and suffix array for our set of distinct MD5 values. Then the complete string S consists of fixed length strings concatenated together, where each string $S_0, S_1, S_2, , S_{n-1}$ is terminated with the sentinel (termination) \$, such that $S_0\$, S_1\$, S_2\$, , S_{n-1}\$$.

6.2 Malware Detecting Scanner

6.3 Malware Detection Service

Access to startup items when windows boots up - .exe is executed with the startup, and hnce it too late to detect malware

6.4 Platform and implementation language

According to a security report from 2015-2016 by AV-TEST, 85.39% of all identified malware in 2015, was detected in Windows [?], of such we have chosen to implement our MIDS on Microsoft's newest Windows operating system, hence Windows 10.

7 Experimental Results

8 Discussion

9 Future work

10 Conclusion

11 Appendix

A SAIS Algorithm run

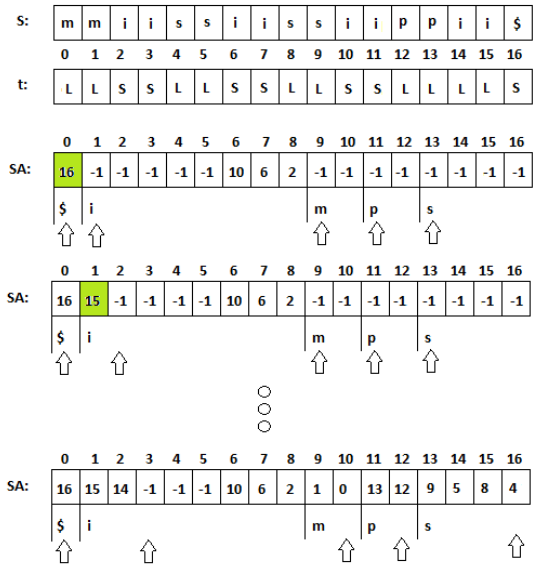


Figure 37: S is scanned from left to right, and indices for each LMS substring is appended to the end of its corresponding bucket in SA. The first LMS substring index is placed at the end of bucket for i, here at position 8 in SA and forwards the bucket end one to the left, hence the bucket end for i now rest at position 7 in SA. This process is repeated until all LMS substring indicies are placed in their buckets.

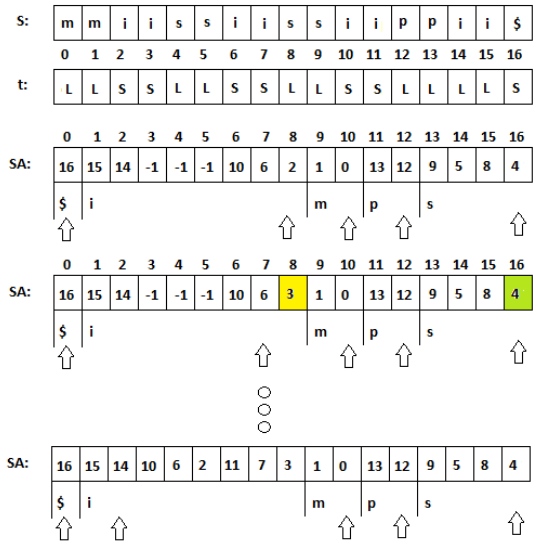


Figure 38: S is scanned from left to right, and indices for each LMS substring is appended to the end of its corresponding bucket in SA. The first LMS substring index is placed at the end of bucket for i, here at position 8 in SA and forwards the bucket end one to the left, hence the bucket end for i now rest at position 7 in SA. This process is repeated until all LMS substring indicies are placed in their buckets.

B SAIS Recurssive step

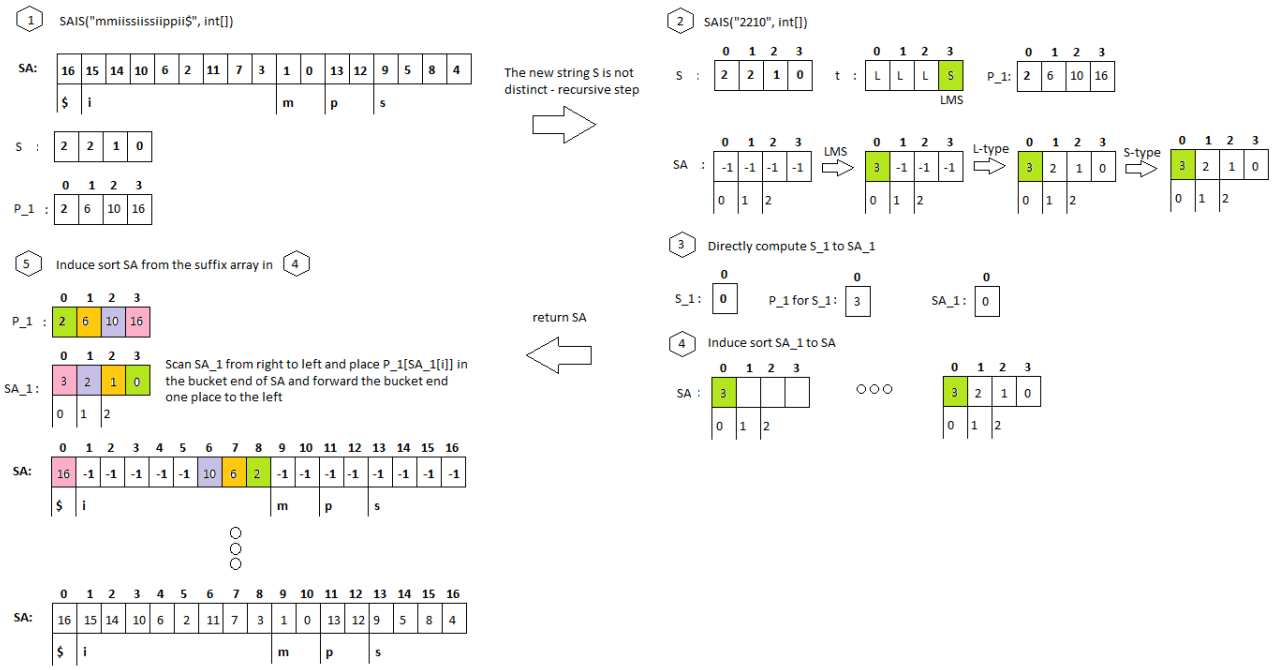


Figure 39: Some description here

C Reversing a compressed string using bwt(S)

References

[1] A. Apostolico, M. Crochemore, M. Farach-Colton, Z. Galil, and S. Muthukrishnan, "40 years of suffix trees," *Communications of the ACM*, vol. 59, no. 4, pp. 66–73, 2016.

[2] D.-N. L. Nguyen Le Dang and V. T. Le, "A new multiple-pattern matching algorithm for the network intrusion detection system," *IACSIT International Journal of Engineering and Technology*, vol. 8, no. 2, pp. 1–7, 2016.

[3] D. Gusfield, *Algorithms on strings, trees, and sequences : computer science and computational biology*. The Pres Syndicate Of The University Of Cambridge, 1 ed., 1997.

[4] R. L. R. . C. S. Thomas H. Cormen, Charles E. Leiserson, *Introduction To Algorithms*. The MIT Pres, Cambridge, Massachusetts, London, England, 3th ed., 2009.

[5] E. O. Mohamed Ibrahim Abouelhoda, Stefan Kurtz, "Replacing suffix trees with enhanced suffix arrays," *Journal of Discrete Algorithms 2*, pp. 53–86, 2004.

[6] U. Baier, "Linear-time suffix sorting – a new approach for suffix array construction," *Institute of Theoretical Computer Science, Ulm University*, pp. 1–10, 2016.

[7] S. Z. Ge Nong and W. H. Chan, "Two efficient algorithms for linear time suffix array construction," *IEEE Transaction on Computers*, pp. 1–20, 2011.

[8] C. Kruegel and T. Toth, "Using decision trees to improve signature-based intrusion detection," in *International Workshop on Recent Advances in Intrusion Detection*, pp. 173–191, Springer, 2003.

[9] "Virusshare." https://virusshare.com/. Lastest Access: 2017-4-15 15:20, Repository download 2016-1-5 20:20.

[10] "Autoruns." https://technet.microsoft.com/en-us/sysinternals/bb963902.aspx. Accessed: 2017-4-15 16:03.

[11] "Strings - advanced data structures." https://www.youtube.com/watch?v=F3nbY3hIDLQl. Accessed: 2016-12-15 08:15.

[12] K. Sadakane, "Compressed suffix trees with full functionality," *2007 Springer Science + Business Media, Inc*, pp. 1–19, 2005.

[13] W. F. S. Simon J. Puglisi and A. Turpin, "The performance of linear time suffix sorting algorithms," *Proceedings of the 2005 Data Compression Conference (DCC'05)*, pp. 1–10, 2005.

[14] G. Manzini, "An analysis of the burrows-wheeler transform," *Journal of the Association for Computing Machinery*, vol. 48, no. 3, pp. 407–430, 2001.

[15] B. Langmead, "Introduction to the burrows-wheeler transform and fm index," pp. 1–12, 2013.

[16] S. A. Toru Kasai, Hiroki Arimura, "Effcient substring traversal with suffx arrays," *DOI Technical Report 185 (2001)*, 2001.

[17] A. L. O. Ferreira, Artur J. and M. A. Figueiredo, "On the suitability of suffix arrays for lempel-ziv data compression.," *International Conference on E-Business and Telecommunications. Springer Berlin Heidelberg*, 2008.

[18] "Fast algorithms for finding nearest common ancestors." `http://epubs.siam.org/doi/pdf/10.1137/0213024`. Accessed: 2016-12-20.

[19] "Fast algorithms for finding nearest common ancestorson finding lowest common ancestors: Simplification and parallelization. siam journal on computing, 1988, vol. 17, no. 6 : pp. 1253-1262." `http://epubs.siam.org/doi/abs/10.1137/0217079`. Accessed: 2016-12-20.

[20] M. Farach, "Optimal suffixx tree construction with large alphabets," *Department of Computer Science, Rutgers University, Piscataway, NJ 08855, USA.*, pp. 1–11, 1997.

[21] A. S. Saeed, Imtithal A. and A. M. Abuagoub, "A survey on malware and malware detection systems," *International Journal of Computer Applications*, no. 67.16, 2013.

[22] N. Idika and A. P. Mathur, "A survey of malware detection techniques," *Purdue University*, vol. 48, 2007.

[23] K. Kosmidis, "Machine learning and images for malware detection and classification," 2017.

[24] J. F. Kurose and K. W. Ross, *Computer networking: a top-down approach*, vol. 5. Addison-Wesley Reading, 2010.

[25] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[26] "Av-test." https://www.av-test.org/en/statistics/malware/. Accessed: 2017-02-01 17:15.

[27] r. D. Taillard, "Heuristic methods for large centroid clustering problems.," *Journal of heuristics*, vol. 9.1, pp. 51–73, 2003.

[28] G. J. McLachlan, "Mahalanobis distance," *Resonance*, vol. 4, no. 6, pp. 20–26, 1999.

[29] L. Deri and S. Suin, "Practical network security: experiences with ntop," *Computer Networks*, vol. 34, no. 6, pp. 873–880, 2000.

[30] F. Ahmed, H. Hameed, M. Z. Shafiq, and M. Farooq, "Using spatio-temporal information in api calls with machine learning algorithms for malware detection," in *Proceedings of the 2nd ACM Workshop on Security and Artificial Intelligence*, pp. 55–62, ACM, 2009.

[31] Y. Ye, D. Wang, T. Li, and D. Ye, "Imds: Intelligent malware detection system," in *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 1043–1047, ACM, 2007.

[32] M. M. Deza and E. Deza, "Encyclopedia of distances," in *Encyclopedia of Distances*, pp. 1–583, Springer, 2009.

[33] "Cluster analysis." http://www.statsoft.com/Textbook/Cluster-Analysis. Accessed: 2017-01-20 22:21.

[34] P. Baecher, M. Koetter, T. Holz, M. Dornseif, and F. Freiling, "The nepenthes platform: An efficient approach to collect malware," in *International Workshop on Recent Advances in Intrusion Detection*, pp. 165–184, Springer, 2006.

[35] "Av-test_security_report_2015-2016." https://www.av-test.org/fileadmin/pdf/security_report/AV-TEST_Security_Report_2015-2016.pdf. Accessed: 2016-04-13 08:15.

[36] S. Turner and L. Chen, "Updated security considerations for the md5 message-digest and the hmac-md5 algorithms," 2011.

[37] X. Wang and H. Yu, "How to break md5 and other hash functions," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 19–35, Springer, 2005.

[38] E. Thompson, "Md5 collisions and the impact on computer forensics," *Digital investigation*, vol. 2, no. 1, pp. 36–40, 2005.