Detecting Local Machine Data Leakage in Real Time

Jingcheng Liu, Yaping Zhang, Yuze Li, Yongheng Jia, Yao Chen, Jin Cao

Chapter 16

DETECTING LOCAL MACHINE DATA LEAKAGE IN REAL TIME

Jingcheng Liu, Yaping Zhang, Yuze Li, Yongheng Jia, Yao Chen and Jin Cao

Abstract Data privacy leaks are becoming a serious problem. A large percentage of privacy leaks are due to inadvertent user errors. Most data leak detection solutions do not have privacy-preserving functionality. Moreover, due to the third-party delivery of data in the cloud, it is not possible to guarantee real-time leak detection.

This chapter proposes a local-side data leakage detection method that uses a suffix array. The method also employs encryption for data protection. The method is compared with mature data leak detection algorithms to demonstrate its effectiveness in real time and that the additional data protection overhead is acceptable.

Keywords: Real-time data leak detection, suffix array, data privacy

1. Introduction

The mobile Internet has brought great convenience to modern society. However, massive amounts of data are now transmitted over the Internet, which renders the task of securing sensitive private data extremely important.

Detecting and stopping privacy leaks are important components of data security. Private data leaks have three main causes. The first and most common is leakage during transmission, where a user directly transmits unencrypted or weakly encrypted data over the Internet. HTTPS-based encryption is a common solution to data leakage during transmission. The second is data leakage caused by local malware or malicious users. Since the stolen private data is typically encrypted before transmission, the leakage cannot be detected by examining the transmitted data. Therefore, these data leaks are detected by monitoring for abnor-

mal transmission behaviors. The third cause is inadvertent data leakage due to user error or incorrect operations. Examples include erroneously sending internal email to external entities and uploading sensitive data to social networking sites. Inadvertent data leaks are difficult to detect because they are caused by ordinary users who perform normal operations.

Several researchers have focused on detecting and preventing private data leaks by users. If the private data is identified in advance, it is feasible to perform simple plaintext matching or deep packet inspection of outbound traffic. Strict data access policies should be implemented on the host, sensitive data should be watermarked and anomalies in outbound data flows should be detected and investigated. However, these methods consume significant resources and the private data could itself be acquired by malware that compromises the matching process [5].

This research focuses on local-side data leakage detection in real time while protecting the private data. In the proposed data leakage detection workflow, the user first selects the private data and provides it to the detection system. The system compares the outbound data flows from the user against the private data in real time to detect potential leaks. The system immediately alerts the user to the potential leaks and can stop the outbound data flows.

The proposed data leakage method employs algorithms that secure and detect private data in outbound flows while minimizing time and space consumption. In particular, it employs byte stream encryption to secure the private data used for matching as well as the outbound flows themselves. Also, it employs a suffix array technique for local-side leakage detection of encrypted flows in real time. The computational time and memory footprint are optimized for real-time leakage detection. Despite providing data security as an additional feature, the overall performance of the proposed method is comparable with the performance of the classical Knuth-Morris-Pratt and finite automaton string matching algorithms.

2. Related Work

Early data leakage detection was mainly performed on the host side [6, 9]. Several researchers have proposed deep packet inspection of outbound flows to detect data leakage [15, 19]. The approaches leverage efficient string matching algorithms such as finite automaton, heuristic and filtering based algorithms [8, 10, 16]. Finite automaton based algorithms perform matching in linear time, but this comes with large memory requirements in the worst case. The heuristic and filtering based

algorithms use less memory, but they are vulnerable to targeted data attacks that cause surges in the matching time.

Researchers have also proposed methods such as MapReduce, fuzzy fingerprinting and verifiable search for data leakage detection [3, 11, 14, 20]. However, the time and space requirements can be prohibitive and the methods do not protect private data.

The advent of cloud computing enables third parties to provide data leak detection services [1, 13, 17, 18]. However, outsourcing this service can result in secondary leaks of private data from the third parties themselves. It is also important to note that third parties are high-value targets for attackers because they handle private data from numerous clients. When third parties cannot be trusted completely, additional pre-processing and post-processing are required to prevent secondary data leaks. Moreover, it is not possible for third parties to guarantee leakage detection in real time.

3. String Matching with a Suffix Array

This section describes string matching using a suffix array. In the following, a text string submitted by a user is denoted as S and its length is n. The pattern string (private data) to be matched is denoted as T and its length is m. The number of strings involved in multi-pattern matching is k.

The suffix array algorithm has strong stability and versatility properties. In the worst case, it can maintain processing efficiency and consume little computational time due to special circumstances underlying its design. Although it takes a long time to pre-process text strings, the complexity of a single match is only $O(m + log\, n)$ after pre-processing. This is better than string matching, which has a general complexity of $O(m+n)$. The performance can be improved further with multi-pattern matching if the relative relationships between strings to be matched are known. Additionally, the matching of a suffix array using binary search facilitates the use of encryption to maintain data security.

3.1 Suffix Array

A suffix array [12] is commonly used for string processing. At the core of a suffix array is a series of complex pre-processing procedures. By pre-processing a series of target strings and making full use of the relationships between the suffixes of a target string, information about the target string can be obtained. The pre-processing of target strings can be performed in $O(n)$ time and a single search can be performed in $O(m + log\, n)$ time.

Algorithm 1: String matching using a suffix array.

```
Data:
T: Pattern string to be matched.
SA: Suffix array of the text string.
Result:
True or False: Matching result.
a ← 0, b ← n − 1
while b − a > 1 do
    c ← (a + b)/2
    if Compare(SA[c], T > 0) then
        a = c
    else
        b = c
    end
    return Compare(SA[b], T = 0)
end
```

A string of length n has n suffixes of different lengths (i.e., substrings from the i^{th} character of the string to the n^{th} character). The process of string matching is expressed as follows:

$$\alpha = \rho|\beta \tag{1}$$

where α and β are two different suffixes, ρ is the string to be matched and $|$ is a connective operation on strings. After the strings are matched, there must be two suffixes, α and β, that satisfy Equation (1). Furthermore, if the string to be matched ρ is a substring of the text string S, there will be at least one suffix of S that makes ρ become its prefix.

The process of string matching using a suffix array involves examining the applicability of Equation (1) to the text string. Suppose that the n suffixes have been sorted, then a suffix array SA is obtained by storing the starting positions of the i^{th} small suffixes, where i ranges from 1 to n.

Algorithm 1 specifies the process of string matching using a suffix array.

In order to match a string, binary search can be used to identify the suffixes that have been sorted at least once. The range of the search can be halved with each comparison of the matched string and a suffix. Thus, a single search is performed in $O(m \log n)$ time.

The relationships between suffixes can be leveraged to enhance search efficiency. Let $LCP[i]$ denote the longest common prefix of two suffixes beginning with $SA[i-1]$ and $SA[i]$. Then, the following relationship exists between the p^{th} and $p-1^{th}$ suffixes of the input text T:

Algorithm 2: Creating a longest common prefix array.

Data:
S: Text string.
SA: Suffix array of the text string.
Result:
LCP: Longest common prefix array of the text string.

```
j ← 0, k ← 0
for i = 0 → n − 1 do
    Rank[SA[i]] ← i
end
for i = 0 → n − 2 do
    if k > 0 then
        k ← k − 1
    end
    j ← SA[Rank[i] − 1]
    while S[i + k] = S[j + k] do
        k ← k + 1
    end
    LCP[Rank[i]] ← k
end
return LCP
```

$$LCP[p] \geq LCP[p-1] - 1 \tag{2}$$

This property can be used to obtain the LCP array in $O(n)$ time [7].

Algorithm 2 specifies the process of creating the LCP array. In the algorithm, the $Rank$ array and suffix array SA are inverses of each other, i.e., $Rank[i]$ represents the order of the suffix that starts with the i^{th} character in all the suffixes.

Having created the LCP array, it is necessary to revisit the search process with the pattern string. If the comparison is only performed with the i^{th} small suffix, it would be compared with the j^{th} small suffix. Let $LCP(i,j)$ denote the longest common prefix of the two suffixes, then $LCP(i,j)$ corresponds to the minimum value in $LCP[i+1],\ldots,LCP[j]$. If the value of $LCP(i,j)$ could be obtained, then unnecessary matches during the search would be eliminated. Since the LCP has been created, the problem of obtaining $LCP(i,j)$ reduces to a problem with the range of the minimum query, which can be realized in $O(1)$ time after pre-processing, where the pre-processing complexity is $O(n\,log\,n)$. Thus, the time complexity of a single search is reduced to $O(m + log\,n)$.

Algorithm 3 specifies the single search process with the longest common prefix array LCP. Note that the $FastLCP$ function in the algorithm computes the value of $LCP(i,j)$ for the i^{th} and j^{th} small suffixes in $O(1)$ time.

Algorithm 3: Searching with a longest common prefix array.

Data:
T: Pattern string to be matched.
SA: Suffix array of the text string.
LCP: Longest common prefix array of the text string.
Result:
$True$ or $False$: Matching result.

$last \leftarrow 0$
$a \leftarrow 0$, $b \leftarrow n-1$
while $b - a > 1$ **do**
  $c \leftarrow (a+b)/2$
  if *Compare(*$FastLCP(last, c)$, $SA[c]$, $T > 0$*)* **then**
    $a = c$
  else
    $b = c$
  end
  $last \leftarrow c$
end
return *Compare(*$FastLCP(last, c)$, $SA[c]$, $T = 0$*)*

The final task is to obtain the suffix array SA. Since the n suffixes are parts of the original text string, considerable space is required to store the suffixes when employing the regular $O(n \log n)$ sorting method. Therefore, the algorithm proposed by Manber and Myers [12] is employed to quickly sort the suffix array.

Figure 1 illustrates the algorithm for creating the suffix array SA. Assume that the text string is `aabaaaab`. First, the n substrings of length one are sorted. This sort corresponds to the size relationship between the characters, which means that the rank of `a` is one and the rank of `b` is two.

Next, each substring is merged with its subsequent substrings to double its length. The rank of the substring is also equivalent to that of the two substrings, which is an n-ary two-digit number. Note that the last substring cannot be merged with other substrings, which is equivalent to the emergence of an empty string with rank zero. The new rank value is then obtained. Following this, the rank value of each suffix and the suffix array SA are obtained by repeating the process $\log n$ times. Thus, the complexity of the entire algorithm is $O(n \log n)$.

3.2 Multi-Pattern Matching

The following three issues must be considered to implement real-time data leakage detection while ensuring data security:

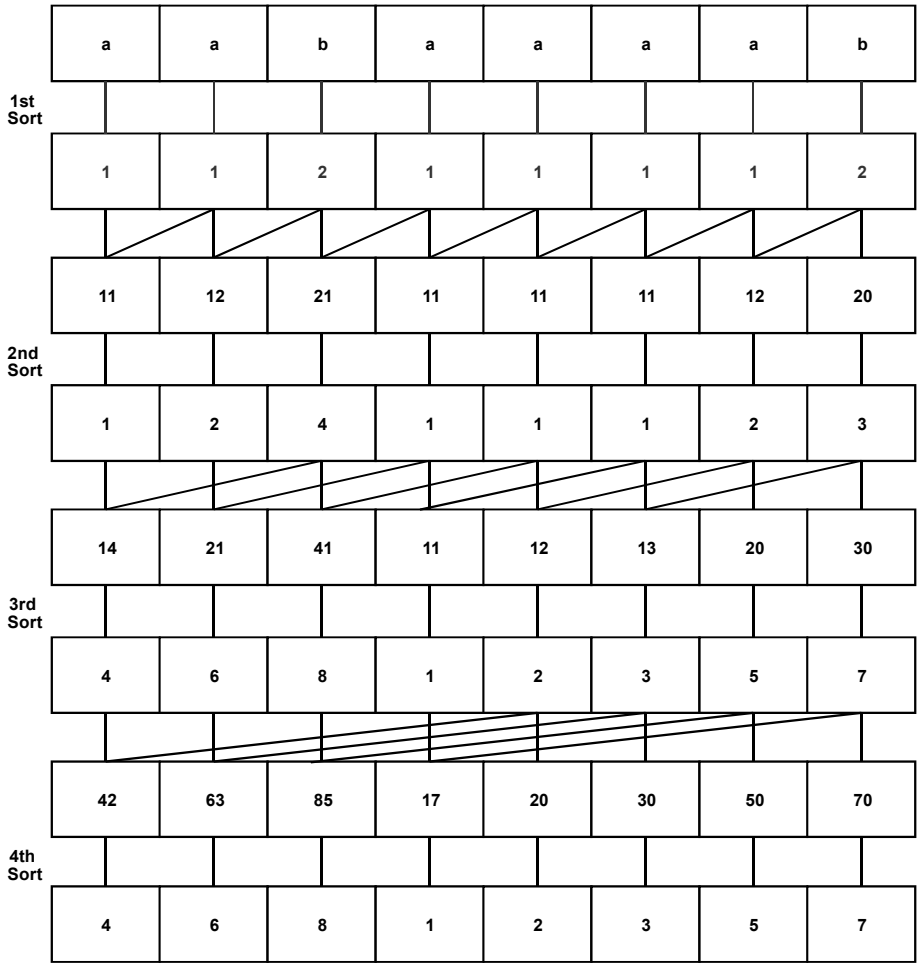


Figure 1. Creating a suffix array.

- The efficiency of the suffix array algorithm should be enhanced when performing multi-pattern matching.
- The data security method that protects private data should not negatively impact data leakage detection.
- Improvements should be incorporated that shorten the search time to ensure good real-time performance.

When a suffix array is used to search for a single string, if the last search succeeds (i.e., the pattern string is matched), then there must be two different suffixes α and β such that Equation (1) holds. However, if

the last search fails (i.e., the pattern string is not matched), then there must be two different suffixes α and β such that the pattern string ρ satisfies $\alpha < \rho < \beta$, and α and β are adjacent.

When k pattern strings are present and these pattern strings have been sorted, then it is possible to start the binary search from the position of the last match (or mismatch) regardless of whether or not the previous match was successful. This enhances the efficiency of multi-pattern matching to some extent.

Private data is protected by transforming the data instead of using its original plaintext version. The transformation should protect the data to a certain extent while guaranteeing real-time performance. The transformation involves the conversion of the text string and all the pattern strings (privacy data) to be matched into byte streams using the XOR operation. Thus, the text content of the transformed data has the same representation as the original data, which does not affect subsequent string matches. Of course, the XORed byte stream containing the string to be matched (detected) should be as long as possible to ensure data protection. The detailed analysis and proof are provided later in this chapter.

Finally, it is necessary to reorder (i.e., resort) private data every time it is transformed to enhance the efficiency of multi-pattern matching. If the $O(n\,log\,n)$ sorting algorithm is repeatedly used to sort the data, there would be an unacceptable negative impact on real-time performance. Fortunately, the data transformation using the XOR operation preserves the original data representation. In other words, the privacy data is still ordered in a relative manner. This order relationship is leveraged to design a resorting algorithm with $O(n)$ complexity.

Algorithm 4 specifies the process for resorting a multi-pattern string. If only the initial characters of all the pattern strings to be matched are considered, then after the XOR operation, the pattern strings with the same initial characters would be concentrated in the same continuous interval after reordering. Since there are only 256 possibilities for a single byte, the starting position of the interval and length of the interval for each possibility must be recorded, and the 256 intervals are reordered so that the initial characters of all the strings are already ordered. Next, the second characters are considered until all the pattern strings are reordered. This process is repeated for each subsequent character of each string. Since there is only one traversal, the total time complexity is $O(n)$.

Algorithm 4: Resorting a multi-pattern string.

Data:
$Multi$: Pattern string to be resorted.
l: Left border of the resorted range.
r: Right border of the resorted range.
pos: Pending position.
$deep$: Deep character to be compared.
Result:
$Resort$: Resorted array.

```
Resort(Multi, l, r, pos, deep)
if l = r then
    Resort[pos] ← Multi[l]
    return
end
for i = 0 → 255 do
    cnt[i] ← 0 pos1[i] ← −1
end
for i = l → r do
    if Multi[i].length = deep then
        Resort[pos] ← Multi[i]
        pos ← pos + 1
        continue
    end
    k ← Multi[i][deep]
    if pos1[k] = −1 then
        pos1[k] ← i
    end
end
cnt[k] ← cnt[k] + 1
for i = 0 → 255 do
    if cnt[i] > 0 then
        Resort(Multi, pos1[i], pos1[i] + cnt[i] − 1, pos, deep + 1)
    end
    pos ← pos + cnt[i]
end
return
```

4. Implementation and Evaluation

The experimental evaluation used a typical laptop computer with four 2.50 GHz CPUs and 12 GB RAM. All the algorithms were implemented in Java.

The performance of each stage of the overall algorithm was assessed by recording its execution time. The Java system method `nanoTime()` was employed to obtain execution times accurately to the nanosecond level. Although the `currentTimeMillis()` method could have been used in theory, it is based on the real time, which means that it does

Table 1. Comparison of execution times.

Text String Length (bytes)	Proposed Method (ns)	KMP (ns)	Finite Automaton (ns)
1,000	161,992	108,457	104,658
2,000	229,878	215,487	174,521
5,000	377,953	522,648	324,758
10,000	750,369	1,054,925	491,358
50,000	4,658,695	5,427,345	2,478,547
10,0000	12,309,142	10,873,483	4,657,857

not provide nanosecond accuracy. The Classmexer instrument agent was used to obtain the memory requirements. These two metrics enable the evaluation of the overall algorithm – whether or not the algorithm meets the real-time standard and whether or not the memory usage is within an acceptable range to meet the real-time standard.

The Enron Email Corpus [2] was employed in the experimental evaluation; email headers as well as email bodies were used in the evaluation. Email is one of the main communication modes and email leaks occur frequently. Therefore, using email data in the evaluation makes for an excellent real-world data leakage scenario. The Chromosome04 gene dataset was also used to evaluate algorithm performance and some extreme cases.

4.1 Comparison with Other Methods

The proposed method incorporates some additional steps to the string matching algorithm in order to implement data protection. Obviously, these steps impact the performance of the overall algorithm.

The first set of experiments was conducted to evaluate if the proposed method meets the real-time standard. The proposed method was evaluated on the local side against two classical string matching algorithms, Knuth-Morris-Pratt (KMP) and finite automaton. All the algorithms were implemented in Java and the Chromosome04 gene dataset was used to evaluate their performance (execution times). Multiple experiments were conducted by selecting text strings of different lengths and fixing the number of pattern strings k and length m for multi-pattern matching to 100 bytes.

Table 1 shows the experimental results. The results reveal that the proposed method has slightly longer execution times than the two traditional algorithms, but this is expected because of the additional steps and

the consequent higher time complexity. Nevertheless, the time overhead is within the acceptable range given that data security is also maintained.

4.2 Enron Email Corpus Experiments

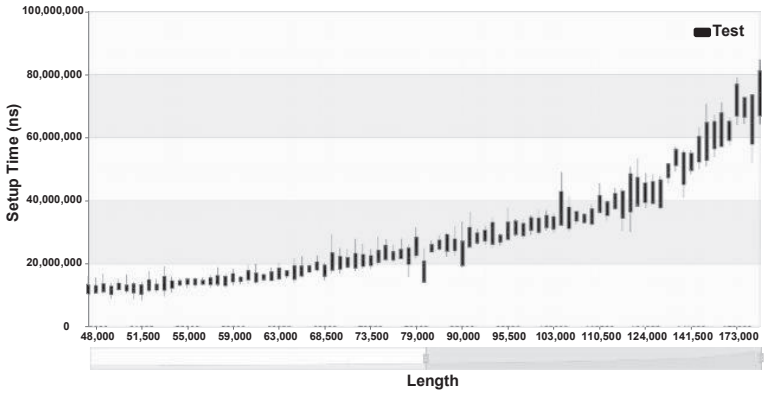
Two sets of experiments were performed using the Enron Email Corpus. One involved single string matching without encryption and the other involved multi-pattern matching with encryption and resorting. Each set of experiments involved matches of all the email in the corpus. In the case of single string matching, the matching string length was set to 10 bytes. In the case of multi-pattern matching, 100 strings of length 10 bytes were matched. All the strings to be matched were random substrings selected from the original strings.

Figure 2 shows the results for single string matching without encryption. Figures 2(a), 2(b) and 2(c) show the setup (pre-processing) times, search times and storage requirements, respectively, for various text string lengths. To simplify the presentation, the experimental results obtained for text string lengths in 500-byte intervals were averaged. In the figures, the maximum, minimum, mean + standard deviation and mean − standard deviation for each interval are displayed in the form of candlestick plots. Intervals with less than five data points were excluded to ensure data validity and eliminate interference by external factors.

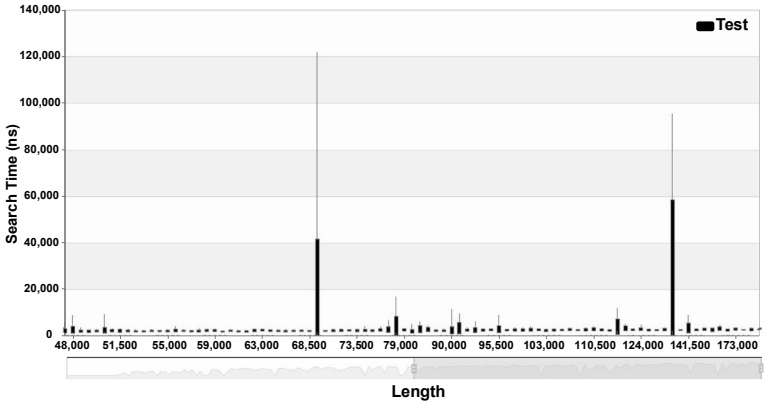
The experimental results reveal that the pre-processing time and storage requirements of the proposed method grow at the rate of $n \log n$ with the length of the string to be matched. With regard to the search times, it should be noted that the lengths of strings to be matched were fixed at 10 bytes and the lengths of the text strings ranged from zero to 200,000 bytes. Thus, the expected time complexity of $m + \log n$ is seen in Figure 2(b). Additionally, when n approaches 200,000 bytes, the pre-processing time plus the search time is still less than 0.1 s. Thus, leak detection of text without encryption meets the real-time standard.

The second set of experiments increased the number of strings to be matched to simulate real-world scenarios and encrypted all the data to ensure security. The efficient resorting algorithm described above was employed after the data was encrypted.

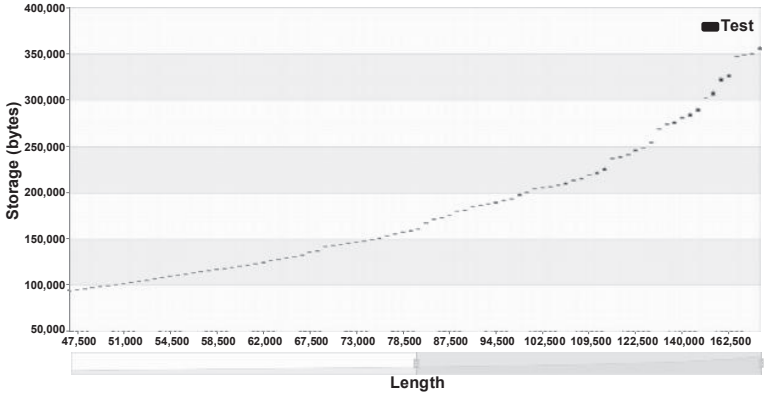
Figures 3 and 4 show the results for multi-pattern matching with encryption and resorting. Figures 3(a), 3(b) and 3(c) show the setup (pre-processing) times, search times and encryption times, respectively, for various text string lengths. Figures 4(a) and 4(b) show the resorting times and storage requirements, respectively, for various text string lengths.



(a) Setup times.

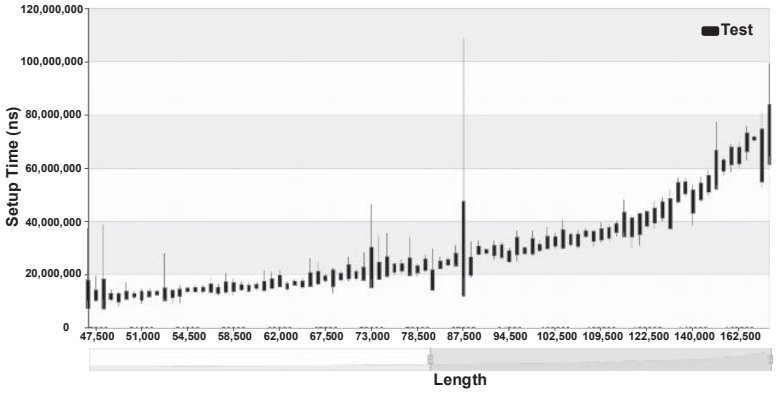


(b) Search times.

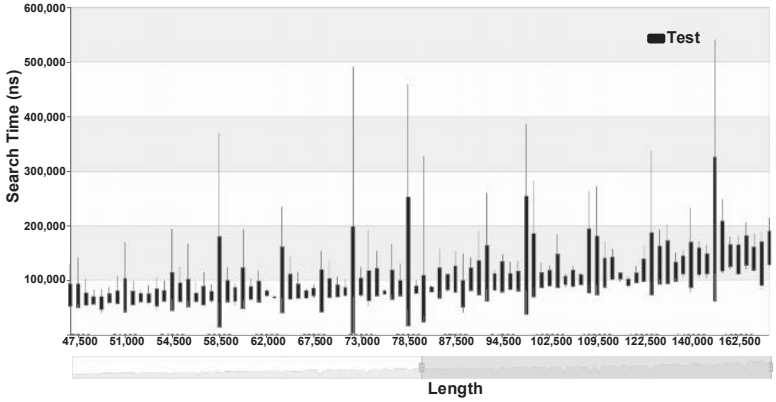


(c) Storage requirements.

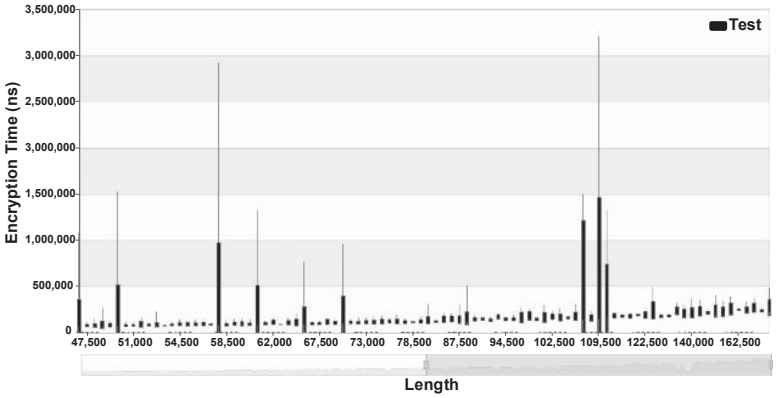
Figure 2. Results for single string matching without encryption.



(a) Setup times.

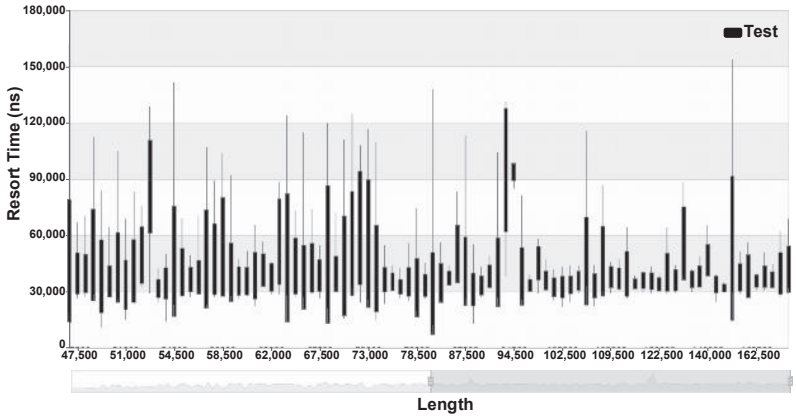


(b) Search times.

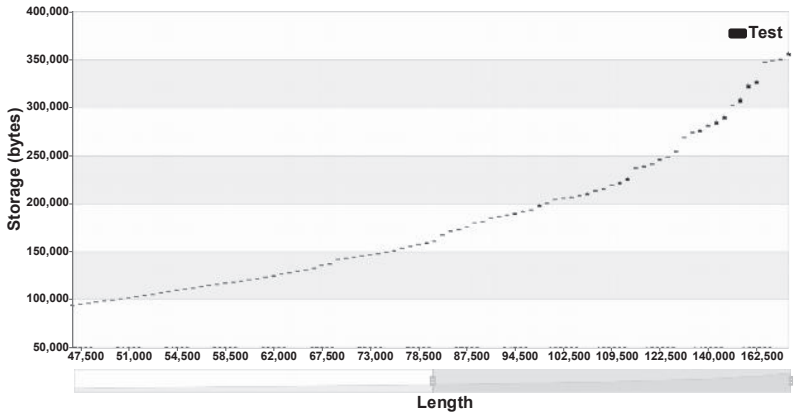


(c) Encryption times.

Figure 3. Results for multi-pattern matching with encryption and resorting.



(a) Resorting times.



(b) Storage requirements.

Figure 4. Multi-pattern matching results with encryption and resorting.

Figure 3(a) demonstrates that the pre-processing times did not change too much compared with the first set of experiments. In fact, the maximum overall time is still within 0.1 s.

The encryption time essentially has a linear relationship with text string length (Figure 3(b)). Since the XOR operation was used to encrypt byte streams, the overall time for encryption is small.

The search time plot in Figure 3(c) shows the largest difference compared with the first set of experiments. This is expected because the number of strings to be searched was increased from just one in the first set of experiments to 100, and the corresponding $O(log\, n)$ time complexity is reflected in the results. However, it is important to note that the overall search time does not grow rapidly, and is still negligible compared with the time required for pre-processing.

Figure 4(a) shows that the resorting time and space requirements are significant. This is expected because resorting is a recursive procedure with large overhead. The resorting time plot does not reflect the theoretical linear relationship with string length. Nevertheless, the maximum time requirement of 10^{-4} s is within the acceptable range.

The results of the two sets of experiments reveal that the time overhead increased as a result of implementing data security. However, the time overhead has little impact on real-time performance. In fact, the overhead is acceptable given the data security requirement.

With regard to the memory overhead in Figure 4(b), it is important to note that only the recursive resorting procedure has significant memory usage. Fortunately, since the maximum resorting time is just 10^{-4} s, the increased memory is required for a miniscule duration and, therefore, does not impact local resources in a significant manner.

5. Discussion and Analysis

Gog and Ohlebusch [4] have demonstrated that it is possible to reduce the time complexity of the suffix array and longest common prefix array computations. However, it was decided not to implement the enhancements in the proposed model for three reasons:

- **Data Security:** Data security is an important goal of the proposed method for outbound data leakage detection. The encryption technique, while providing protection, should ensure that the transformed data can be resorted efficiently. Thus, the proposed method opted to create the suffix array and longest common prefix array as discussed above.

- **Real-Time Performance:** While the time complexity is the principal consideration for real-time data leakage detection, the memory consumption is also an important issue. Recursion, which requires significant runtime memory, is restricted to the resorting stage to minimize the runtime memory consumption and enable users to perform their normal computing tasks while the real-time leakage detection system is operational.

- **Real-World Scenarios:** For individual users, the normal outbound text data throughput is on the order of 100,000 bytes/s. Thus, algorithms with $O(n \log n)$ or $O(n)$ time complexity have little effect on the overall execution time. Therefore, the proposed method opted to use an algorithm with $O(n \log n)$ time complexity to create the longest common prefix array.

Table 2. Time and space complexity of the four stages.

Stage	Time Complexity	Space Complexity
Pre-Processing	$n \log n$	$n \log n$
Searching	$m + \log n$	1
Encryption	n	n
Resorting	n	n

Table 2 shows the time and space complexity of the four stages in the proposed method for real-time leakage detection with data security. Since the byte stream length used in the XOR operation is one byte, there are 256 possibilities for a transformed string. Thus, local malware would be able to obtain the original string after a maximum of 256 attempts. When the byte stream length is increased to two bytes, the maximum number of attempts required would be 65,536. When the strings are long enough, it would be practically impossible for local malware to obtain the original strings. The resorting algorithm also makes it more difficult for local malware to monitor the data transformations. Indeed, the proposed method achieves data security by making it computationally infeasible for local malware to defeat the protection mechanism.

6. Conclusions

Data privacy leaks are a serious problem, especially inadvertent data leaks caused by user error or incorrect operations. Inadvertent data leaks are difficult to detect because they are caused by ordinary users who perform normal operations.

The novel data leakage method presented in this chapter employs algorithms that secure and detect private data in outbound flows while minimizing time and space consumption. It leverages byte stream encryption for data protection and a suffix array technique for local-side leakage detection of encrypted flows in real time. The computational time and memory footprint are optimized for real-time data security and data leakage detection. Despite providing data security as an additional feature, experiments demonstrate that the overall performance of the proposed method is comparable with that of the classical Knuth-Morris-Pratt and finite automaton string matching algorithms.

The proposed method requires users to identify private data in advance, which is undoubtedly a time-consuming task. Moreover, users may not be able to mark all their sensitive data because of a lack of

understanding about their data. Additionally, the volume and types of private data are constantly increasing. Future research will attempt to use machine learning techniques to automate the task of identifying private data.

References

[1] S. Ananthi, M. Sendil and S. Karthik, Privacy preserving keyword search over encrypted cloud data, in *Advances in Computing and Communications*, A. Abraham, J. Lloret Mauri, J. Buford and S. Thampi (Eds.), Springer, Berlin Heidelberg, Germany, pp. 480–487, 2011.

[2] CALO Project, Enron Email Dataset, SRI International, Menlo Park, California (`www.cs.cmu.edu/~./enron`), 2015.

[3] F. Chen, D. Wang, R. Li, J. Chen, Z. Ming, A. Liu, H. Duan, C. Wang and J. Qin, Secure hashing based verifiable pattern matching, *IEEE Transactions on Information Forensics and Security*, vol. 13(11), pp. 2677–2690, 2018.

[4] S. Gog and E. Ohlebusch, Fast and lightweight LCP-array construction algorithms, *Proceedings of the Meeting on Algorithm Engineering and Experiments*, pp. 25–34, 2011.

[5] S. Jha, L. Kruger and V. Shmatikov, Towards practical privacy for genomic computation, *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 216–230, 2008.

[6] C. Kalyan and K. Chandrasekaran, Information leak detection in financial email using mail pattern analysis under partial information, *Proceedings of the Seventh WSEAS International Conference on Applied Informatics and Communications*, vol. 7, pp. 104–109, 2007.

[7] T. Kasai, G. Lee, H. Arimura, S. Arikawa and K. Park, Linear-time longest-common-prefix computation in suffix arrays and its applications, *Proceedings of the Twelfth Annual Symposium on Combinatorial Pattern Matching*, pp. 181–192, 2001.

[8] H. Kim, H. Hong, H. Kim and S. Kang, Memory-efficient parallel string matching for intrusion detection systems, *IEEE Communications Letters*, vol. 13(12), pp. 1004–1006, 2009.

[9] K. Li, Z. Zhong and L. Ramaswamy, Privacy-aware collaborative spam filtering, *IEEE Transactions on Parallel and Distributed Systems*, vol. 20(5), pp. 725–739, 2009.

[10] P. Lin, Y. Lin, Y. Lai and T. Lee, Using string matching for deep packet inspection, *IEEE Computer*, vol. 41(4), pp. 23–28, 2008.

[11] F. Liu, X. Shu, D. Yao and A. Butt, Privacy-preserving scanning of big content for sensitive data exposure with MapReduce, *Proceedings of the Fifth ACM Conference on Data and Application Security and Privacy*, pp. 195–206, 2015.

[12] U. Manber and G. Myers, Suffix arrays: A new method for on-line string searches, *SIAM Journal on Computing*, vol. 22(5), pp. 935–948, 1993.

[13] Y. Shi, Z. Jiang and K. Zhang, Policy-based customized privacy preserving mechanism for SaaS applications, *Proceedings of the Eighth International Conference on Grid and Pervasive Computing and Collocated Workshops*, pp. 491–500, 2013.

[14] X. Shu, D. Yao and E. Bertino, Privacy-preserving detection of sensitive data exposure, *IEEE Transactions on Information Forensics and Security*, vol. 10(5), pp. 1092–1103, 2015.

[15] X. Shu, J. Zhang, D. Yao and W. Feng, Rapid and parallel content screening for detecting transformed data exposure, *Proceedings of the IEEE Conference on Computer Communications Workshops*, pp. 191–196, 2015.

[16] X. Shu, J. Zhang, D. Yao and W. Feng, Fast detection of transformed data leaks, *IEEE Transactions on Information Forensics and Security*, vol. 11(3), pp. 528–542, 2016.

[17] B. Wang, S. Yu, W. Lou and Y. Hou, Privacy-preserving multi-keyword fuzzy search over encrypted data in the cloud, *Proceedings of the IEEE Conference on Computer Communications*, pp. 2112–2120, 2014.

[18] D. Wang, X. Jia, C. Wang, K. Yang, S. Fu and M. Xu, Generalized pattern matching string search on encrypted data in cloud systems, *Proceedings of the IEEE Conference on Computer Communications*, pp. 2101–2109, 2015.

[19] H. Wang, K. Tseng and J. Pan, Deep packet inspection with bit-reduced DFA for cloud systems, *Proceedings of the International Conference on Computing, Measurement, Control and Sensor Networks*, pp. 221–224, 2012.

[20] J. Zhou, Z. Cao and X. Dong, PPOPM: More efficient privacy preserving outsourced pattern matching, *Proceedings of the Twenty-First European Symposium on Research in Computer Security*, part I, pp. 135–153, 2016.