Secure Compression and Pattern Matching Based on Burrows-Wheeler Transform

Gongxian Zeng, Meiqi He, Linru Zhang, Jun Zhang, Yuechen Chen, Siu Ming Yiu,
Department of Computer Science, The University of Hong Kong, Hong Kong
Email: {gxzeng, mqhe, lrzhang, jzhang3, ycchen, smyiu}@cs.hku.hk

***Abstract*—Searchable compressed data structures (e.g. Burrows-Wheeler Transform) enable one to create a memory-efficient index for large datasets such as human genomes. On the other hand, storing such an index in a third-party server, e.g. cloud, may have the privacy and confidentiality issues. An open problem in the community is to construct a secure variant of such a data structure. This problem is challenging as most of the existing works were shown to be insecure and none of them is able to perform pattern matching. In this paper, we provide the first solution based on Burrows-Wheeler Transform (BWT) to solve this problem (our scheme can do both compression and pattern matching). A new security definition, called isomophism-restricted IND-CPA security, is proposed. We show that our scheme is secure under this definition and our scheme is practical by experiments.**

I. Introduction

File compression reduces the space required for storing data. There exist two types of compression - lossless compression and lossy compression. As the names imply, the original file can be recovered exactly from a lossless compression but may not from a lossy compression. Compression not only reduces storage space in many file systems (e.g., Microsoft's NTFS, Apple's HFS+ and Oracle's ZFS) but also improves the transmission speed in some protocols (e.g., HTTP and Google's SPDY). For example, a word file can be compressed by as much as 80% or more on average. Thus, it takes about 20% the time to transmit a compressed word file as it would take to transmit the uncompressed file. Large amounts of structured or unstructured data are collected in this big data era, the need for compression is even more pronounced. In this paper, one of the main contributions is about secure compression algorithm, which not only supports lossless compression but also guarantees the security.

At the same time, we also consider secure pattern matching. Pattern matching is a fundamental operation in string processing, which finds all the occurrences of a given pattern in a given text. For a compressed file, an naive idea to perform pattern matching is to decompress the text first and run an ordinary search on the uncompressed text. However, this will be time consuming and may require a lot of memory. A more effective approach is to search the compressed file directly. Many researchers have studied extensively the *compressed pattern matching* problem which aims at finding pattern occurrences in compressed file without decompression. Compressed data indexes are designed for solving the compressed pattern matching problem. But if the indexes are stored in a third-party server which may be untrusted, privacy and confidentiality issues arise.

978-1-5386-7493-2/18/$31.00 © 2018 IEEE

TABLE I
Secure Compressors

Compressor	Version	Known Attacks
WinZip	WinZip 9.0	[1], [2]
WinRAR	WinRAR v3.42	[3]
PKZIP	PKZip 1.10 and 2.04g	[4], [5]

If we just consider security and space of compressed data indexes, a straight-forward solution is to encrypt the compressed data indexes by current secure compressors, such as WinZip with encrypt-then-authenticate primitive, WinRAR with AES and PKZip with a custom stream cipher. These schemes are called encryption-after-compression constructions. This kind of 2-step structure is simple. Besides, adopting stream cipher or authentication would not increase the length of compressed strings too much. However, researchers had found several security vulnerabilities in their security functions or system level functions, as shown in Table I. Taking PKZip 1.10 as an example, Michael Stay [4] found a weakness in the pseudorandom number generator and launched a known plaintext attack to decrypt the cipher.

Another approach for secure compression is explored in academia, which is to integrate compression and encryption together in one step based on existing compressed data structures (e.g., multiple Huffman table (MHT), arithmetic coding (AC), Lempel-Ziv-Welch (LZW) and Burrows-Wheeler transform (BWT)). However, as shown in Table II, most of the compression-encryption combined constructions also have security problems (refer to section I-B). Also, neither current encryption-after-compression schemes nor current compression-encryption combined schemes can preserve the special indexing structures for pattern matching. Thus, we cannot leverage computations on server side based on all these existing schemes. In this paper, we propose a protocol to achieve it.

Our method is based on Burrows-Wheeler transform (BWT) [20]. it is used widely especially for processing the human genomes and is adopted in many compressors, such as bzip2[21]. In 2012, M. Külekci Oğuzhan [18] proposed a secure compression algorithm based on BWT, called scram-

TABLE II
COMPRESSION-ENCRYPTION COMBINED DATA STRUCTURES

Data Structure	Secure Variants	Known Attacks
MHT	[6]	[7], [8]
AC	RAC[9]	[8]
	ISAC[10], [11] SAC[12]	[13], [8]
LZW	[14]	[15]
	[16], [17]	No known attacks
BWT	[18]	[19]

bling BWT (sBWT). However, in their modified Move-to-Front coding (MTF [22]), a statistical attack was found [19]. (see the details in a later section). Besides compression, with the backward pattern matching property [23], BWT can be used to perform searching. To be more specific, with two auxiliary data structures that provide information related to symbol frequency and position, exact substrings that match the searched pattern can be found directly using BWT. However, once the symbol frequency and position are leaked, a malicious attacker can recover the original string easily. To conclude, there still does not exist a secure compression scheme based on BWT, not to mention secure compressed pattern matching scheme.

A. Our contributions

In this paper, we provide a secure compression algorithm and the first provably secure compressed pattern matching scheme. Both of them are based on BWT. We adopt an efficient additive homomorphic encryption (AHE) [24] to protect the compressed data indexes. AHE allows us to leverage the powerful computation abilities of the cloud server. We apply Private Information Retrieval Read (PIR_Read) [25] to retrieve data obliviously. As a result, our protocol not only keeps the searching results confidential, but also hides the searched strings.

We propose a new security definition, called isomophism-restricted IND-CPA security, which is more appropriate to define the security for secure compressed data structures. We show that our scheme is secure under this security definition. To verify the practicality of our scheme, we implement and conduct experiments on real datasets. The results show that our compression performance is comparable to [17], which is now the best unbroken compression-encryption combined schemes. Our pattern matching protocol (the first solution for this problem) is also efficient.

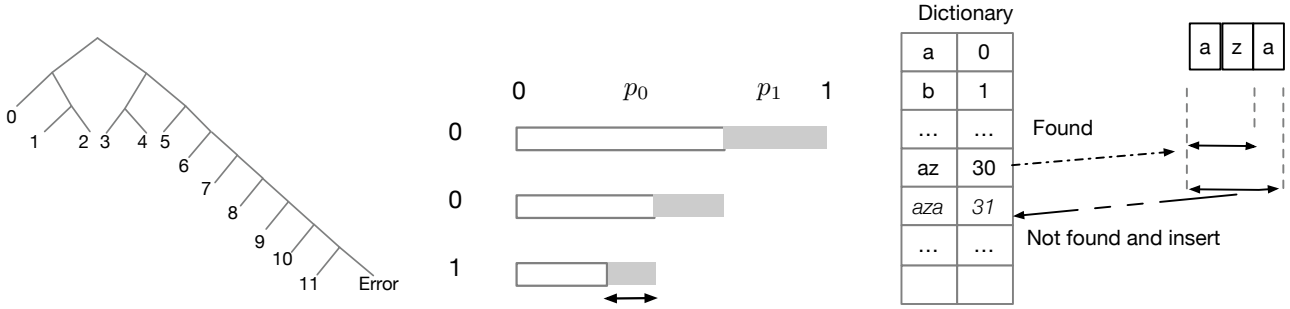
B. Other related work

Existing techniques to construct secure compression schemes are Multiple Huffman Table (MHT), Arithmetic Coding (AC) and Lempel-Ziv-Welch (LZW).

MHT. Huffman table maps each symbol in the alphabet to a variable-length code according to the frequency. Fewer bits are used to encode characters that occur more frequently while more bits are used to encode characters that occur less frequently. Compared to fixed-length encoding (e.g., ASCII character encoding), variable-length encoding requires less space. With knowledge of the Huffman table, we can easily obtain the original message through decoding. Otherwise, it is difficult to recover the original message. Based on this idea, a secure compression scheme was proposed in [6]. They generated multiple Huffman tables and chose a secret one randomly to perform compression. For the same symbol, using different Huffman tables leads to different outputs. However, [7] points out that different Huffman tables provide different security levels. Some tables are easy to guess which is known as weak key problem, reducing the computation complexity of exhaustive search.

AC. Arithmetic coding operates on intervals and subintervals. As shown in Figure 1 (b) to encode string "'001', the interval $(0, 1)$ is first divided into two parts $(0, p_0)$ and $(p_0, 1)$, where p_0 is the proportion of the whole interval for symbol "0". Similarly, p_1 is for the symbol "1" and $p_0 + p_1 = 1$. Since the first character is "0", we choose the subinterval $(0, p_0)$. After that, partition the chosen subinterval $(0, p_0)$ in the same way: $(0, p_0')$ and (p_0', p_0) and choose the subinterval $(0, p_0')$ for the second character "0". Eventually, we get the final subinterval (e.g. the one has an arrow below in Figure 1 (b)) and output a number within the final subinterval. The key to decoding is to know the way of partition (i.e. which subinterval represents which symbol). Thus, many secure variants adopt different methods to partition, such as swapping subintervals randomly (RAC [9]), more than one subintervals representing a symbol (ISAC [10], [11]) and permuting all subintervals together (SAC [12]). However, these schemes are heuristic schemes and not provably secure.

LZW. LZW[26] is a dictionary-based compression algorithm where dictionary is maintained during the coding. As shown in Figure 1 (c), the dictionary is a table that maps strings to the responding codes. Given a string to be compressed (e.g. "aza" in Figure 1 (c)), we scan it sequentially from the first character. In Figure 1 (c), "a" has been in the dictionary so we move to next position. The characters from the first to current scanned position form a substring (e.g. "az"). If current scanned substring occurs in the dictionary, then keep scanning, otherwise output the responding code of last scanned substring in the dictionary and add current substring to be a new entry in the dictionary. In Figure 1 (c), we found "aza" is not in the dictionary, so we output the code "30" for "az" and insert "aza" into the dictionary. To use LZW for secure compression, researchers tried to insert new entries into dictionary randomly [16]. However [16] has some



(a) Huffman table for JPEG dc coefficient coding. Left and right branch are labeled by 0 and 1 respectively.

(b) Arithmetic coding

(c) LZW

Fig. 1. Examples

drawbacks. First, using finite-length indices may cause the dictionary overflow. Second, the code for the same substring is usually fixed so the scheme may turn into a traditional substitution cipher, which is not secure. An improvement to the second issue is to permute the entries [14] or reinsert the coded substring after coding a substring [17]. Researchers construct a provably secure compression scheme - squeeze cipher in [17], which is the first work that presents a formal framework for analyzing secure compression schemes. However, it has two drawbacks. (i) users need to predict the size of the dictionary table. Once the dictionary is full, no new entries are able to be added and the scheme would fail. (ii) if the size is estimated larger than actual need, code length is longer. It will degrade compression performance.

II. Preliminaries

A. The Burrows-Wheeler transform and compression

BWT tends to gather together those characters which are adjacent to the similar text substrings and can be constructed in linear time [27]. Here we just introduce a general transform and it has three steps [23]:

1) Add a special character \$, which is not in the alphabet Σ, to the end of the string T and we assume that \$ is "smaller" than any character in Σ lexicographically. For simplicity, the new string is still called T.

2) Form a matrix M whose rows are the cyclic shifts of T.

3) Sort the rows of M in the lexicographical order and output last column of M, which is the result of BWT.

Table III shows the example for string "mississippi\$", where the last column is the result of BWT. In fact, BWT is often the first step in a compression scheme. [28] tested a lot of coding algorithms based on BWT. One of those well-performed lossless compression algorithms includes the following steps: $Algo\_FM = BWT{+}MTF{+}RLE{+}PC$. Roughly speaking, BWT tries to gather same characters together, MTF would replace those successive same characters with successive zeros, RLE is to encode those zeros with fewer bits and PC is to encode the RLE result in binary form. Details are as follows.

TABLE III
BWT result of string "mississippi\$"

order	first character	medial strings	last character
1	\$	mississipp	i
2	i	\$mississip	p
3	i	ppi\$missis	s
4	i	ssippi\$mis	s
5	i	ississippi\$	m
6	m	ississippi	\$
7	p	i\$mississi	p
8	p	pi\$mississ	i
9	s	ippi\$missi	s
10	s	issippi\$mi	s
11	s	sippi\$miss	i
12	s	sissippi\$m	i

MTF (Move-to-Front coding). Let $T^{'}$ denote the result of BWT, $T^{'} = BWT(T)$. There is a MTF_table containing all characters in the alphabet. The table maps the characters to their relative positions in the current table (e.g. the first character is mapped to "0"). After encoding a character, the coded character would be moved to the first position in the $MTF\ table$ so that this character is mapped to "0" now. Let $Z = MTF(T^{'})$ denote the encoded string after we apply MTF to $T^{'}$. Since the property of BWT, many same characters would be gathered together so that successive zeros would appear after MTF step.

RLE (Run Length Encoding). We can apply run length encoding to each run of zeroes in Z. To be more exact, we replace m successive 0s with the reverse order of the binary form of the number $(m+1)$, discarding the most significant bit. We use another two new symbols ***a*** and ***b*** (a represents "0") to encode the successive zeroes. For example, 5 zeros will be encoded as ***ab***. Therefore, the result is over the alphabet $\{\boldsymbol{a}, \boldsymbol{b}, 1, \ldots, |\Sigma|-1\}$ where $|\Sigma|$ denotes the number of characters in Σ.

PC (variable-length prefix code). We apply a variable-length prefix code to those symbols in alphabet $\{\boldsymbol{a}, \boldsymbol{b}, 1, \ldots, |\Sigma|-1\}$. For ***a*** and ***b***, we encode ***a*** by 10 and ***b*** by 11. For other symbols, e.g. i , we use $1+2*\lfloor log(i+1) \rfloor$ bits: $\lfloor log(i+1) \rfloor$ 0s followed by the binary form of $(i+1)$ in $1+\lfloor log(i+1) \rfloor$ bits, e.g. 00110 for symbol "5". Thus, the final result is

over alphabet $\{0,1\}$. We denote the final result by $BZ = PC(RLE(Z))$.

B. Auxiliary data structures and backward pattern matching

To support backward pattern matching, two auxiliary data structure are often used.

1) $c(e)$: is a table that stores the minimun index of strings strarting with character e in the sorted rotation results M (e.g. $c(s) = 9$ in Table III). We denote the next symbol, which is little "larger" than symbol e in the lexicographical order, by $e{+}1$. Thus $c[e{+}1]{-}1$ returns the ending position of e.

2) $occ(e,h)$: is a table that stores the occurence of character e from the first to some position h in the BWT result (e.g. $occ(s,9) = 3$ in Table III)

Lemma 1. *Given a string $T_1 = t_1 \cdots t_{n-1} t_n$, which is the h^{th} string in the sorted M, then $T_2 = t_n t_1 \cdots t_{n-1}$ is the k^{th} string in the sorted M, where $k = c[t_n] + occ(t_n, h-1)$.*

For example, in Table III we know "sissippi$mis" is in the 10^{th} column, we want to know the position of "ssissippi$mi". Then we get $c(s) + occ(s,9) = 9 + 3 = 12$, so we know that "ssissippi$mi" is in the 12^{th} column.

The correctness is based on the fact, that if string $a_1 a_2 \cdots a_{n-1} t_n$ are sorted before T_1 in the M, then $t_n a_1 a_2 \cdots a_{n-1}$ is also sorted before T_2. Thus, among all the strings starting with t_n in the M, $occ(t_n, h-1)$ can tell us how many strings are sorted before T_2. Then the backwards pattern matching algorithm can be expressed as follows.

Lemma 2. *Given a pattern $P = p_1 \cdots p_n$ and a range $(begin, end)$ in the M where those strings are starting with subpattern $p_{i+1} \cdots p_n$ ($i \geq 1$), then the range of $p_i \cdots p_n$ is $(begin', end')$, where $begin' = c[p_i] + occ[p_i, begin - 1]$ and $end' = c[p_i] + occ[p_i, end] - 1$.*

The range in lemma 2 can be determined by the auxiliary data structures based on lemma 1. However, *occ* is very large and its size can be $O(nlogn)$ where n is the length of the string. In [23], researchers tried to store it hierarchically. Every L characters in the BWT result $T^{'}$ are called a block and every L blocks are called a superblock. $superblock(e,h)$ returns the occurrence of symbol e in the first $\lfloor h/L^2 \rfloor$ superblocks. $block(e,h)$ returns the occurrence of e from the last superblock to $\lfloor h/L \rfloor^{th}$ block. For the occurrence in a block, it is stored in $block\_inner(mtf[i], BZ_i,\ e, h - i * L)$, where BZ_i is the i^{th} ($i = \lfloor h/L \rfloor$) compressed block and $mtf[i]$ stores the state of the MTF table at the beginning of encoding the i^{th} block. Since the same blocks would be compressed into the same when the MTF tables are the same, it saves much space. Although the table mtf also needs to be stored in the server, the whole storage of all auxiliary data structures is only $O(\frac{n \cdot loglog(n)}{log(n)})$ bits when $L = \frac{1}{2} logn / log|\Sigma|$.

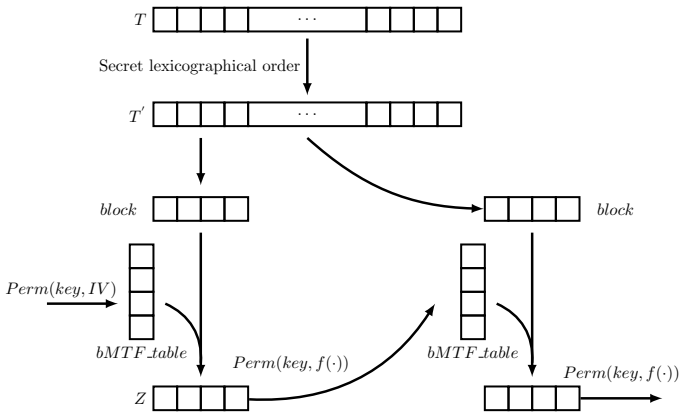


Fig. 2. sBWT + bMTF

III. Our Construction

A. Compression

In [18], M. Külekci Oğuzhan tried to propose a secure compression algorithm by combining sBWT with variant MTF where the initialized MTF table is the same lexicographical order as sBWT. Although the author analyzed that the frequency attack to sBWT is difficult, other kinds of attack, e.g. known plaintext attack, can be feasible. Thus, he modified MTF to strengthen the security so that it can hide the exact output of sBWT. However, MTF with a secret MTF table is something like simple substitution cipher. Although the secret MTF table is unknown, [19] shows that we can guess one randomly and apply it to decoding. By statistical attack, it is easy to find the correct lexicographical order.

To prevent this kind of attack, we change MTF table randomly for every L characters (i.e. a block) so that attackers cannot achieve the same frequency as in [19] since L is not very large. And to reduce the key storage for the pseudo-random function, we use the hash value of last compressed block as one input of the function. Our compression algorithm includes $Algo = sBWT + bMTF + RLE + PC$. The picture 2 shows the two key steps. User chooses a hash function f and a key-based pseudo random permutation function $Perm$. Then choose a random number for permutation function as private key. On inputting a string to be compressed, user processes it by sBWT first and then bMTF.

sBWT(scrambling Burrows-Wheeler Transform). Let T denote the original string and $T^{'} = sBWT(T)$ denote the result of scrambling Burrows-Wheeler transform of T.

- Choose a random number r and then compute $Perm\ (key, r) \rightarrow \xi$, where ξ denotes the secret lexicographical order.
- Add a special character $\$ \notin \Sigma$ to the end of T and we assume that $ is "smaller" than any character $e \in \Sigma$ in the secret lexicographical order ξ. And we still call the new string T for simplicity.
- Form a matrix M whose rows are the cyclic shifts of T.
- Sort the rows of M in the secret lexicographical order ξ and last column of M is $T^{'}$.

bMTF(blocky move-to-front). We group $T^{'}$ by every L characters. For each block, we choose a permutation of the alphabet randomly first and then follow the general MTF steps to encode every character in each block. Here we call the pseudo random permutation function $Perm$ with two parameters: the key and a nonce. The nonce is an initial vector IV for the first block and is the hash value of the last encoded block in other cases.

- Group $T^{'}$ by every L characters.
- For block i, we generate a new MTF_table first:

$$Perm(key, IV \ or \ f(block_{(i-1)})) \rightarrow bMTF\_table_i,$$

  where $bMTF\_table_i$ is a permutation of all characters in the alphabet.
- Follow the general MTF steps.

The step RLE and step PC are reversible and their contributions are to encode symbols by as fewer bits as possible. Considering that BWT and MTF are modified, maybe RLE and PC are not the optimal solution and some other entropy codings may perform better. But it is another research interest so we don't discuss it further here. In fact, after PC step, we will permute all those blocks randomly but the purpose of permutation is to perform secure pattern matching so the detailed discussion would be left in section III-B.

B. Pattern matching

Overview We know that the auxiliary data structures leak the information about lexicographical order and frequency so that the malicious attackers can reconstruct the original string from these structures easily. Here we are going to encrypt these data and propose an interactive protocol to perform pattern matching. For simplicity, here we only focus on counting occurrence protocol. In short, we take three steps: i) permute *occ* tables in order to hide the position information; ii) adopt private information retrieval read to retrieve entries in *occ* tables in order to keep the results confidential; iii) fake some queries to hide the information that "begin" and "end" (referring to Lemma 2) get too close.

Technical details. In aforementioned *occ*, given a position h, we can find the responding superblock and block easily. But the server can also know the occurrence in this way. Therefore, we permute all the entries of the table $superblock$ and $block$ so that they are not ordered any more. For superblock i (similarly in block), we store the information of the i^{th} superblock in the $(g^i\beta)^{th}$ entry of the table $superblock$ where g is a group generator and β is a random number. And client keeps g and β secretly so that $g^i\beta$ is a random number for others. For $block\_inner(mtf[i], BZ_i, e, h - L * i)$, mtf is not feasible since it can be used to decode the bMTF result so we store the hash value of $(i-1)^{th}$ block instead. In our construction, if two blocks have the same hash value of their last blocks, then they would have the same MTF table and their final compression results are the same. And we also permute the entries in the same block of $block\_inner$. Thus, instead of sending h directly to the server, client sends a position vector $pv(h) = (a, b, c)$ where a, b, c are position information for $superblock$, $block$ and $block\_inner$ respectively.

To hide the pattern, every time client requests $occ(e, h)$, it cannot send e directly to the server. An intuitive method to solve this problem is to request $occ(\cdot, h)$(i.e. all data of position h) but it increases message overhead by $|\Sigma|$ and delays the processing time. In [25], Moataz Tarik et al. adopted an oblivious Private Information Retrieval Read (PIR_Read) to retrieve a block. Comparatively speaking, its message overhead is relatively small and we can leverage some computations on the server. Thus we apply a similar operation to retrieve the data obliviously, as shown in Algorithm 1. Firstly, we encrypt occ data with additive homomorphic encryption (AHE) [24] before transmission to the server. And we denote this encryption by AHE_1. When client wants to compute $occ(e, h)$, client sends position information h to server with a special vector $\mathcal{V}$. Supposing the lexicographical order is $e_1e_2 \dots e_{|\Sigma|}$. If $e_i = e$, then $\mathcal{V}_i = AHE_2\_Enc(1)$, otherwise, $\mathcal{V}_i = AHE_2\_Enc(0)$, where AHE_2 is another AHE and we have modules $n_2 > n_1^2$. Then, we perform PIR_Read. For position h, there are $|\Sigma|$ entries $array[1, ..., |\Sigma|]$ for different characters. Server computes $\mathcal{V}_i^{array[i]}$ for each $i \in [1, ..., |\Sigma|]$. After that, the server multiplies those results and returns final result B to the client. Then client computes $AHE_1\_Dec(AHE_2\_Dec(B))$ to retrieve the data.

Algorithm 1 PIR_Read

Require:$array[1, ..., |\Sigma|]$, $\mathcal{V}$

```
procedure PIR_READ(array, V)
    B = 1;
    for i = 1 to |Σ| do
        B = B * V_i^{array[i]};
    end for
    return B
end procedure
```

When performing pattern matching, two variables $begin$ and end would get closer, thus they may be in the same superblock or block. If $begin - 1$ and end are in the same block, then $pv(begin-1)$ and $pv(end)$ have the same first two components so attackers can know that the pattern occurs less than L times. And it usually holds that the longer the pattern is, the smaller the number of occurrence is. In addition, if the attacker knows the length of pattern and that the pattern occurs less than L times, he can obtain a set of possible patterns by statistical attacks. To prevent occurrence and the length of patterns from being leaked, we make some modifications. First, instead of sending pv, we send a special vector set pos (refer to equation 1). This special set contains several vectors and all cases that $pv(begin-1)$ and $pv(end)$ may occur (e.g. in the same block or not). Thus we can hide $pv(begin - 1)$ and $pv(end)$ in pos. As for other vectors, we pad some random numbers into it. Second, we fake some occ queries and client would communicate with server in fixed rounds so that the exact length of pattern would not be leaked.

$$pos = \left\{ \begin{matrix} (a_1, b_1, c_1) \\ (a_1, b_1, c_2) \\ (a_1, b_2, c_3) \\ (a_2, b_3, c_4) \end{matrix} \right\} \quad (1)$$

Protocol. Then the protocol will be conducted as follows:

(1) **Initialization.** The client generates the auxiliary data structures c and occ. occ is generated in the following way.

- $superblock$: choose a prime p_1 that is slightly larger than n/L^2, a nonzero number β_1 and choose a generator g_1 of $Z^*_{p_1}$. Client keeps g_1 and β_1 secret. For superblock i, we store those data in the $(g_1^i\beta_1)^{th}\ mod\ p_1$ (we omit $mod\ p_1$ in the remainder) entry of the table. For those empty entries, pad some random numbers.
- $block$: choose a prime p_2 that is little larger than n/L, a nonzero number β_2 and choose a generator g_2 of $Z^*_{p_2}$. Client keeps g_2 and β_2 secret. For block i, we store those data in $(g_2^i\beta_2)^{th}\ mod\ p_2$ (we omit $mod\ p_2$ in the remainder) entry of the table. For those empty entries, pad some random numbers.
- $Hash\_value$: instead of storing pictures of MTF tables, we store the hash value of $(i-1)^{th}$ block for the i^{th} block. The hash value of $(i-1)^{th}$ block is stored in the $(g_2^i\beta_2)^{th}$ entry in $Hash\_value$ table. For those empty entries, pad some random numbers.
- $block\_inner$: the entry $block\_inner(mtf[i], BZ_i, e, h - L * i)$ is moved to $block\_inner$ $(Hash\_value[g_2^i\beta_2],\ BZ_{g_2^i\beta_2},\ e,\ \tau(key, h - L * i))$, where $i = \lfloor h/L \rfloor$ and $\tau : Z_L \to Z_L$ is a random permutation function.

Thus, to compute $occ(e, h)$, the client needs to send a position vector $pv(h) = (g_1^{\lfloor h/L^2 \rfloor}\beta_1, g_2^{\lfloor h/L \rfloor}\beta_2, \tau(key, h - \lfloor h/L \rfloor * L))$ to the server. So we have $occ(e, h) =$

$$\begin{aligned} &superblock(e, g_1^{\lfloor h/L^2 \rfloor}\beta_1) + block(e, g_2^{\lfloor h/L \rfloor}\beta_2) + \\ &block\_inner(Hash\_value[g_2^{\lfloor h/L \rfloor}\beta_2], BZ_{g_2^{\lfloor h/L \rfloor}\beta_2}, \\ &e, \tau(key, h - \lfloor h/L \rfloor * L)) \end{aligned}$$

Comparing $block\_inner$ here with that introduced in section II-B, we know that $BZ_{\lfloor h/L \rfloor}$ is moved to $BZ_{g_2^{\lfloor h/L \rfloor}\beta_2}$, so we need to permute all blocks to make the computation correct. Therefore, the whole algorithm is $Algo = sBWT + bMTF + RLE + PC + P$ where P is permutation. After encrypting the data in occ with AHE, the client sends them to the server while c is kept by the client. Then choose a suitable number R as the fixed communication round. Let $P[1, \ldots, m]$ be the pattern and initialize $i = m, begin = c[P[m]], end = c[P[m]+1] - 1$.

(2) **Client**: send a special position vector set pos and a reading vector $\mathcal{V}$ according to $P[i-1]$.

(3) **Server**: for all character in the alphabet and the y^{th} vector in pos, compute

$$\begin{aligned} B_y[\zeta] = &superblock(\zeta, pos[y][1]) + \\ &block(\zeta, pos[y][2]) + \\ &block\_inner(Hash\_value(pos[y][2]), \\ &BZ_{pos[y][2]}, \zeta, pos[y][3]) \end{aligned}$$

where $\zeta \in \Sigma$ and $y \in [1, 4]$. Finally, return

$$\{\alpha_y\}_{y\in[1,4]} = \{PIR\_Read(B_y, \mathcal{V})\}_{y\in[1,4]}.$$

(4) **Client**: find the expected α_{i1},$\alpha_{i2} \in \{\alpha_y\}_{y\in[1,4]}$, then compute

$$\begin{aligned} &occ(P[i-1], begin - 1) = AHE_1\_Dec(AHE_2\_Dec(\alpha_{i1})), \\ &occ(P[i-1], end) = AHE_1\_Dec(AHE_2\_Dec(\alpha_{i2})), \\ &begin = c[P[i-1]] + occ(P[i-1], begin - 1), \\ &end = c[P[i-1]] + occ(P[i-1], end) - 1, \end{aligned}$$

and $i = i-1$. If $(begin \le end)$ and $(i \ge 2)$, go to step (1), otherwise, send random pos to the server and no further computations on $begin$ and end. If communication round is R, go to next step.

(5) **Client**: if $end < begin$, pattern is not found, otherwise, occurrence of the pattern is $end - begin + 1$.

We don't need to send real occ queries to the server at the first round and also can extend this protocol to multiple pattern matching easily. We can send occ queries for different patterns alternately in one round. We can fake fewer occ queries in multiple pattern matching than in single and the communication round is fixed. Thus, the protocol can work more efficiently in multiple pattern matching.

IV. Analysis

A. Security for compression

Due to the length of strings with different entropy are distinguishable after compression, [17] modified the semantic security model. Similarly, we are going to introduce a variant security model, called isomorphism-restricted IND-CPA because the property of BWT (or sBWT) to gather same characters together makes adversaries win in the normal game-based security definition. For example, $T_1 = aabbaabb\$$ and $T_2 = abababab\$$ have the same length. Although attackers don't know the exact lexicographical order, attackers can be sure that there is some difference between $sBWT(T_1)$ and $sBWT(T_2)$. Supposing $\$ < a < b$, $sBWT(T_1) = bb\$aabbaa$ and $sBWT(T_2) = bbbb\$aaaa$ so that the more successive zeros the result has after bMTF, the more likely the original string is T_2. Thus attackers can distinguish them from the results of bMTF. Thus, we restrict adversary $\mathcal{A}$ to challenge isomorphic strings.

Definition 1. *Let string $T_1[1, \ldots, n]$ and string $T_2[1, \ldots, n]$ have the same length. If there exists a bijective function $f : \Sigma \to \Sigma$ so that $\forall i, f(T_1[i]) = T_2[i]$, we call T_1 is isomorphic to T_2.*

Then, we define isomorphism-restricted IND-CPA security.

Definition 2. *Let Π_{key} denote a compression scheme, $\mathcal{A}$ denote a PPT adversary and $\mathcal{C}$ denote the challenger. The procedure of $Game^{IR-CPA}_{\Pi,\mathcal{A},\mathcal{C}}(1^\kappa)$ is defined as follow:*

- ***Setup**: the challenger $\mathcal{C}$ runs $key \leftarrow KeyGen(1^\kappa)$ to generate the secret key, where κ is the security parameter.*
- ***Query 1**: the compression oracle $\mathcal{O}$ can be queried by adversary $\mathcal{A}$. On inputting a string $T[1,\ldots,n]$, it outputs the compressed result $BZ \leftarrow \Pi_{key}(T)$. The adversary can perform a polynomially bounded number of queries.*
- ***Challenge**: $\mathcal{A}$ sends two isomorphic strings T_0 and T_1 to the challenger. The challenger picks a random bit b and computes $c \leftarrow \Pi_{key}(T_b)$. The challenger $\mathcal{C}$ sends c to $\mathcal{A}$.*
- ***Query 2**: The adversary $\mathcal{A}$ can perform a polynomially bounded number of queries to compression oracle $\mathcal{O}$ and additional computations.*
- ***Output**: $\mathcal{A}$ returns a guess b' of b.*

$\mathcal{A}$ wins the game if $b' = b$ and the advantage of $\mathcal{A}$ is

$$|Pr[Game^{IR-CPA}_{\Pi,\mathcal{A},\mathcal{C}}(1^\kappa) = b] - \frac{1}{2}|. \tag{2}$$

A compression scheme is isomorphism-restricted IND-CPA secure if for any polynomial time adversary $\mathcal{A}$, it holds $|Pr[Game^{IR-CPA}_{\Pi,\mathcal{A},\mathcal{C}}(1^\kappa) = b] - \frac{1}{2}| \leq negl(\kappa)$ where $negl(\cdot)$ is a negligible function.

Theorem 1. *Let Π_k denote the compression scheme introduced in section III-A where k is the secret key of the secure compression. Then Π_k is isomorphism-restricted IND-CPA secure.*

We introduce a lemma before showing the proof of this theorem.

Lemma 3. *For any two isomorphic strings T_0 and T_1, if $c_0 \leftarrow \Pi_k(T_0)$, then challenger $\mathcal{C}$ can pick up a secret order ξ and a bMTF table t :$c_1 \leftarrow \Pi_k(T_1)$ such that c_0 and c_1 are indistinguishable.*

Proof. For the randomly chosen sub-order ξ_0 for all symbols in T_0, we construct another sub-order ξ_1 for all symbols in T_1 by the following rules: for any symbol $a, b \in T_0$ and $f(a), f(b) \in T_1$, if there is $a < b$ in ξ_0, then we have $f(a) < f(b)$ in ξ_1, where f is the bijective function between T_0 and T_1. Roughly speaking, ξ_1 is isomorphic to ξ_0. Then for characters in Σ but not in T_1, they can be inserted at any positions in ξ_1. After that, we can get that $T_0' \leftarrow sBWT(T_0)$ and $T_1' \leftarrow sBWT(T_1)$ are isomorphic. We can use the similar method to construct t: 1) supposing that the randomly chosen sub-table for the symbols in T_0 is t_0, for any $a \in T_0$ and $f(a) \in T_1$, if there is $t_0(a) = i$, then we have $t_1(f(a)) = i$ in t_1; 2) for characters in Σ but not in T_1, they can be inserted into the empty entries in t_1 and the final bMTF table is t. Similarly, we will get that $Z_0 \leftarrow bMTF(T_0')$ and $Z_1 \leftarrow bMTF(T_1')$ are isomorphic. The results $c_0 \leftarrow \Pi_k(T_0)$ and $c_1 \leftarrow \Pi_k(T_1)$ are in the same structure, because RLE and PC are reversible and have deterministic coding steps. Thus the final results $c_0 \leftarrow \Pi_k(T_0)$ and $c_1 \leftarrow \Pi_k(T_1)$ are indistinguishable. □

Now, comes the proof of Theorem 1.

Proof. Suppose $\mathcal{A}$ can break the scheme Π_k. Then for any Challenger $\mathcal{C}$, $\mathcal{A}$ can find two isomorphic strings T_0 and T_1, such that $c_0 \leftarrow \Pi_k(T_0)$ and $c_1 \leftarrow \Pi_k(T_1)$ are computational distinguishable. So there is no random number r such that challenger $\mathcal{C}$ can get $\xi \leftarrow Perm(key, r)$ or $t \leftarrow Perm(key, r)$ according to Lemma 3. Without losing generality, we assume that $\xi \leftarrow Perm(key, r)$ is non-available. Then we will show that we can construct a Distinguisher $\mathcal{D}$ which can distinguish $Perm(key, r)$ from a random permutation π. Assume that the number of symbols shown in T_0 and T_1 is m totally. Then $Pr[\pi = \xi] = \frac{(|\Sigma|-m)!}{|\Sigma|!}$ and $Pr[Perm(key, r) = \xi] = 0$ for any random number r. So the $\varepsilon = Pr[\pi = \xi] - Pr[Perm(key, r) = \xi]$ is non-negligible, Which is conflict with secure pseudo random permutation function $Perm(key, r)$. Therefore, theorem 1 is valid. □

To simplify the proof, we adopt a very restricted condition. Actually, the condition can be more flexible. For any two strings with same length, if they are isomorphic in each block with size L after the sBWT step, then the final results are also indistinguishable.

B. Analysis for pattern matching

Complexity. Let n denote the length of original string T and γ denote the increasing factor of ciphertext size of AHE. Then the array *superblock* takes $O(\gamma\frac{n}{L^2})$ bits and *block* takes $O(\gamma\frac{n}{L})$ bits. As for *block_inner*, every entry takes $O(\gamma)$ bits and it has $2^\rho \cdot 2^l \cdot |\Sigma| \cdot L$ entries where ρ denotes the constant secure length of hash value and l denotes the length of compressed block. Statistically, l should be smaller than the length of uncompressed block $L * log|\Sigma|$. But if the length of some compressed blocks are too large, a hash table can help to induce them. For simplicity, we have $l < L * log|\Sigma|$ so *block_inner* takes $O(\gamma 2^\rho 2^{L \cdot log|\Sigma|} \cdot |\Sigma| \cdot L)$ bits. If we set $L = \frac{logn}{log|\Sigma|}\eta$ where $\eta < 1$, the whole space complexity would be $O(\gamma\frac{n}{logn})$. As for the communication overhead, the protocol runs $O(R)$ rounds where R is a secure constant parameter, and the maximum message overhead is $O(|\Sigma|\gamma) = O(\gamma)$.

Security. Our security goal is that there is no information leakage through the pattern matching process. Assuming that the server is semi-honest, we prove that the protocol is secure under simulation-based security model and the experiments are defined as follows:

Definition 3. *We say that a Pattern Matching scheme is secure if for any PPT adversary $\mathcal{A}$, there exists PPT simulators $\mathcal{S} = (\mathcal{S}_0, \mathcal{S}_1)$, such that the outputs of the following two experiments in Figure 3 are indistinguishable.*

$\mathcal{O} = (\mathcal{O}_0, \mathcal{O}_1, \mathcal{O}_2)$ are random oracles. $\mathcal{O}_0$ is to generate random patterns while $\mathcal{O}_1$ and $\mathcal{O}_2$ are to generate queries according to the state st. $P[1, ..., t_n]$ has t_n characters.

Experiment $RealExp^{PM,\mathcal{A}}(1^\lambda, 1^n)$:
$(pp, sk) \leftarrow PM.Setup(1^\lambda)$;
$(P_1, ..., P_n) \leftarrow \mathcal{O}_0(1^n)$;
For the i^{th} *occ* query, $(\alpha_i, st) \leftarrow PM.Access(P_j[l_i, .., t_j])$;
When finishing searching the j^{th} pattern, $k_j \leftarrow PM.Search(P_j)$;
Output: $(pp,\ \{\alpha_i\},\ |\{\alpha_i\}|,\ \{op_i\})$.

Experiment $IdealExp_{\mathcal{S}}^{PM,\mathcal{A}}(1^\lambda, 1^n)$:
$(pp, sk) \leftarrow PM.Setup(1^\lambda)$;
For the i^{th} *occ* query, $(\alpha_i, st') \leftarrow \mathcal{S}_0(\mathcal{O}_1(st))$;
When finishing searching the j^{th} pattern, $k_j \leftarrow \mathcal{S}_1(\mathcal{O}_2(st))$;
Output:$(pp, \{\alpha_i\}, |\{\alpha_i\}|, \{op_i\})$.

Fig. 3. Experiments

$PM.Setup$ initials all the parameters, encrypts all data and stores them in the server. $PM.Access$ is a series of access operations that perform accesses to occ tables in the server with additional computations for a subpattern. $PM.Search$ denotes one search for a full pattern. $\{op_i\}$ is a record of a series of access operations to the occ tables in PM.Access step. $\{\alpha_i\}$ is result returned by the server and $|\{\alpha_i\}|$ is the number of all access operations. $\{k_i\}$ is the number of the pattern occurrence, which is known by the client only.

Theorem 2. *Let PM denote the pattern matching scheme introduced in section III-B and $Pattern(P_1, ..., P_n)$ is a random sequence of pattern. Then PM is secure under the security model defined in Definition 3.*

Proof. For any adversary, we construct simulators $\mathcal{S} = (\mathcal{S}_0, \mathcal{S}_1)$ and the random oracles $\mathcal{O} = (\mathcal{O}_0, \mathcal{O}_1, \mathcal{O}_2)$ as follows.On inputting parameter 1^n, $\mathcal{O}_0$ generates n random patterns in real experiment. When PM performs a real Access operation on a sub-string $ss = P_j[l_i, ...t_j]$ for the first time in real experiment, $\mathcal{O}_1$ generates a random Access order op and records it in the ideal experiment. If the same sub-string ss comes again, $\mathcal{O}_1$ will return the same order op. When PM returns the occurrences of pattern $P = P_j$ for the first time in real experiment, $\mathcal{O}_2$ generates a random number r and records it in the ideal experiment. If the same pattern P appears again, $\mathcal{O}_2$ will return the same number r. Simulator $\mathcal{S}_0$ performs the operations $PM(op)$ where op is returned by $\mathcal{O}_1$,then returns result α and the state st. Simulator $\mathcal{S}_1$ returns $k = r$ where r is returned by $\mathcal{O}_2$.
Next, we compare the outputs of real experiment and ideal experiment. It is clear that pp and $|\{\alpha_i\}|$ from different experiments are indistinguishable. Due to there are multiple patterns being matched at the same time, any sub-pattern is possible to be matched at any stage (e.g. at the beginning or towards the end) for each access operation. The distribution of which pattern and which stage is processed in one access operation will follow uniform distribution under the assumption that n is big enough. Therefore, the adversary $\mathcal{A}$ is impossible to find the relationship among the list $\{\alpha_i\}$. What's more, the distributions of $\{\alpha_i\}$ in real experiment and ideal experiment are the same, because we guarantee that the same sub-pattern will contribute same output in the ideal experiment which matches the situation in real experiment. So $\{\alpha_i\}$ from different experiments are also indistinguishable.
As for $\{op_i\}$, they are also indistinguishable in real experiment and ideal experiment. For each op_i, it contains five parts:

$$\begin{cases} (a_1, b_1, c_1) \\ (a_1, b_1, c_2) \\ (a_1, b_2, c_3) \\ (a_2, b_3, c_4) \\ PIR - Read\ Vector\ \mathcal{V} \end{cases}$$

Two of the first four parts are real query and the others are dummy. The adversary $\mathcal{A}$ cannot distinguish them due to unawareness of which stage the operation is. And these four part are in the format $g^a\beta$. Although β could be canceled by division, we can reduce the problem that calculating a from $g^a\beta$ to DDH problem [29], [30]. On the other hands, PIR-Read vectors are guarded by AHE. Therefore, $\{op_i\}$ in the real experiment will not leak any information from the pos or PIR-Read vectors and is computational indistinguishable with those in the ideal experiment.
In summary, both the outputs and the processes from real experiment and ideal experiment are computational indistinguishable for the adversary $\mathcal{A}$. So our PM scheme is $SIM - Secure$. When only single pattern is matched in this scheme, we can construct some dummy patterns and access them randomly. Then the distribution of operations are still the same with the multiple patterns situation. Therefore, our PM scheme is still secure under single pattern circumstances. □

The main idea of the definition 3 is that for any PPT adversary cannot obtain any useful information from the intermediate information and operations between the client and server. When only single pattern is matched in this scheme, we can construct some dummy patterns and access them randomly. Then the distribution of operations are still the same with the multiple patterns situation. Therefore, our PM scheme is still secure under single pattern circumstances.

V. Experimental results

We implemented our scheme and conduct experiments to evaluate the performance of our compression scheme and pattern matching protocol. The experiment is executed on a single server node with Intel(R) E7-2830 CPU. First, we consider the compression ratio, which is defined as the compressed output size divided by the uncompressed input size. We compare our scheme with common compressor gzip[31], bzip2[21]

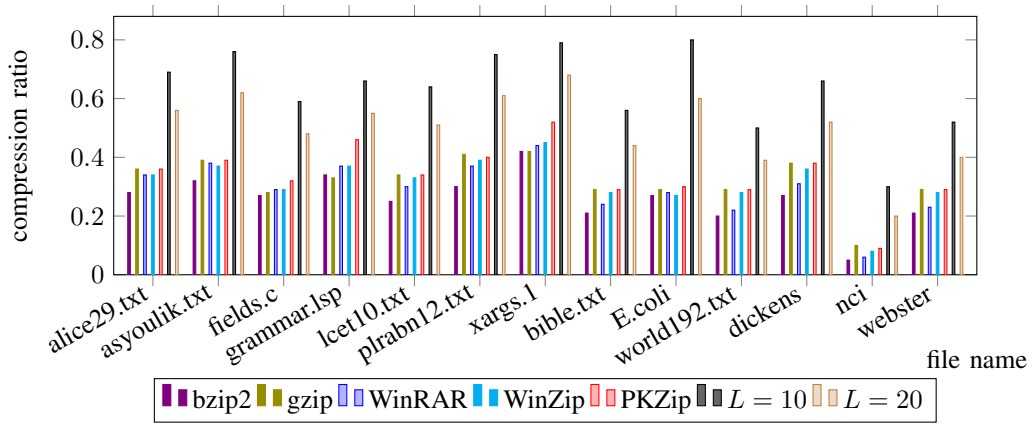


Fig. 4. compression ratio by different compressor

and secure compressor WinRAR, WinZip and PKZip. The tested files used here are chosen from standard corpora of documents for testing lossless compression algorithms: the Canterbury corpus and the Large corpus [32], and Silesia corpus [33]. These files include a segment of DNA, C source file, works of literature and so on. The various files highlight the various performance of our algorithm. For simplicity, the alphabet size in different files are all set 256. The result is displayed in Figure 4, where $L = 10$ and $L = 20$ are our compressor with different parameters. It shows that BWT-based compressor bzip2 outperforms the others and that is why we choose to add security features to BWT-based algorithms. Though the compression ratio of our scheme is not as good as those of encryption-after-compression software, it is close to the result in [17], which now is the best among those unbroken compression-encryption combined schemes. In addition, the figure also shows that when L increases, the secure compression algorithm performs better so if we choose an appropriate L, the performance is acceptable.

Next, we evaluate our patten matching protocol without considering network delay. Let FM_PM denote the pattern matching introduced in [23] and we are going to compare our protocol with FM_PM. To show how much influence encryption and PIR_Read would have on pattern matching directly, we did not implement the faked occ query and fixed R rounds communication. The test files are all DNA files from [34], [35] with small alphabet size 4. Considering the test file size varies from 4MB to 70MB and $L = \frac{1}{2}logn/log|\Sigma|$ where $\eta = \frac{1}{2}$, then $L \approx 6$. As for the additive homomorphic encryption, we set the modular $74390621 = 8623*8627$, so the memory size of each entry of occ data structure in our scheme would be 8 Bytes. The total size after compression with the occ data structure are displayed in Table IV. It shows that when the file size is not very large, the total size is larger than the original file size. It is because the $block\_inner$ occupies a lot of space. However as analyzed in Section IV-B, $block\_inner$ would not increase quickly if the file size gets larger. Figure 5 shows the generating time for different file size and the searching time for different pattern lengths. Comparing with

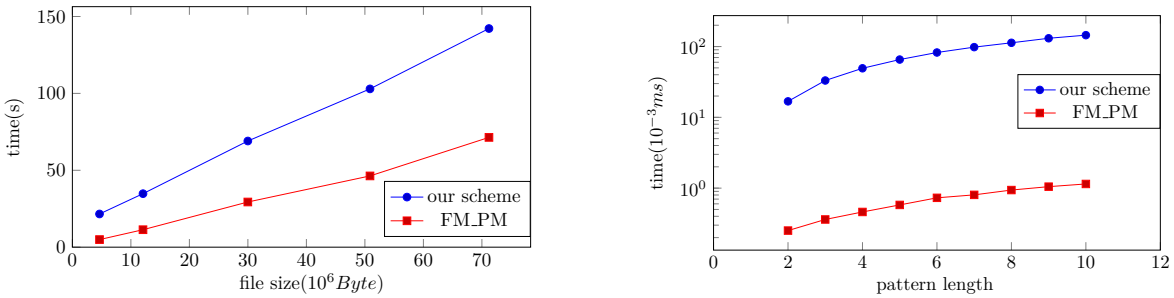
TABLE IV
TOTAL SIZE

file size(MB)	FM_PM(MB)	our scheme(MB)
4.42	26.88	200.04
11.51	31.44	212.90
28.60	42.58	243.98
48.51	53.02	291.52
67.91	66.91	315.05

FM_PM, our scheme is a lot slower but the absolute timing is still reasonable (within a second). Although the generation time is longer (more than 100 seconds for a 60M file), we only need to generate all required data structures once as a preprocessing, this should be acceptable. Overall speaking, our scheme is practical in pattern matching.

VI. Conclusions and Future Work

In this paper, we derive a new compression-encryption combined scheme based on BWT algorithm and a protocol in client-server model to realize pattern matching on our secure compressed index. We show that our scheme is secure under a new security definition, which is proposed for secure compressed indexes. Our solution is the first that can solve the problem. There are several open problems that require the attention of the community. First, a more formal and systematic discussion on the essential relationship between compression and encryption is desirable, since compression is to discard duplications and decrease the entropy while encryption is to increase the entropy. For instance, it is interesting to explore whether the ultimate security we can achieve in secure compression algorithms when certain compression performance is fixed. Another question is how to construct a new protocol for multiple parties to perform pattern matching securely. Also, how to construct a better scheme (in terms of both searching efficiency and memory efficiency) is also important for practical applications.



(a) generating time

(b) searching time

Fig. 5. generating time and searching time

REFERENCES

[1] T. Kohno, "Attacking and repairing the winzip encryption scheme," in *Proceedings of the 11th ACM conference on Computer and communications security*. ACM, 2004, pp. 72–81.

[2] P. H. Phong, P. D. Dung, D. N. Tan, N. H. Duc, and N. T. Thuy, "Password recovery for encrypted zip archives using gpus," in *Proceedings of the 2010 Symposium on Information and Communication Technology*. ACM, 2010, pp. 28–33.

[3] G. S.-W. Yeo and R. C.-W. Phan, "On the security of the winrar encryption feature," *International Journal of Information Security*, vol. 5, no. 2, pp. 115–123, 2006.

[4] M. Stay, "Zip attacks with reduced known plaintext," in *International Workshop on Fast Software Encryption*. Springer, 2001, pp. 125–134.

[5] E. Biham and P. C. Kocher, "A known plaintext attack on the pkzip stream cipher," in *International Workshop on Fast Software Encryption*. Springer, 1994, pp. 144–153.

[6] C.-P. Wu and C.-C. Kuo, "Design of integrated multimedia compression and encryption systems," *IEEE Transactions on Multimedia*, vol. 7, no. 5, pp. 828–839, 2005.

[7] J. Zhou, Z. Liang, Y. Chen, and O. C. Au, "Security analysis of multimedia encryption schemes based on multiple huffman table," *IEEE Signal Processing Letters*, vol. 14, no. 3, pp. 201–204, 2007.

[8] G. Jakimoski and K. Subbalakshmi, "Cryptanalysis of some multimedia encryption schemes," *IEEE Transactions on Multimedia*, vol. 10, no. 3, pp. 330–338, 2008.

[9] M. Grangetto, E. Magli, and G. Olmo, "Multimedia selective encryption by means of randomized arithmetic coding," *IEEE Transactions on Multimedia*, vol. 8, no. 5, pp. 905–917, 2006.

[10] H. Kim, J. D. Villasenor, and J. Wen, "Secure arithmetic coding using interval splitting," in *Conference Record of the Thirty-Ninth Asilomar Conference onSignals, Systems and Computers, 2005.* IEEE, 2005, pp. 1218–1221.

[11] J. Wen, H. Kim, and J. D. Villasenor, "Binary arithmetic coding with key-based interval splitting," *IEEE Signal Processing Letters*, vol. 13, no. 2, pp. 69–72, 2006.

[12] H. Kim, J. Wen, and J. D. Villasenor, "Secure arithmetic coding," *IEEE Transactions on Signal processing*, vol. 55, no. 5, pp. 2263–2272, 2007.

[13] J. Zhou, O. C. Au, and P. H.-W. Wong, "Adaptive chosen-ciphertext attack on secure arithmetic coding," *IEEE Transactions on Signal Processing*, vol. 57, no. 5, pp. 1825–1838, 2009.

[14] J. Zhou, O. C. L. Au, X. Fan, and P. H.-W. Wong, "Secure lempel-ziv-welch (lzw) algorithm with random dictionary insertion and permutation," in *2008 IEEE International Conference on Multimedia and Expo, ICME 2008-Proceedings, Hannover, Germany*, 2008, p. 245.

[15] S. Li, C. Li, and J. C.-C. Kuo, "On the security of a secure lempel-ziv-welch (lzw) algorithm," in *2011 IEEE International Conference on Multimedia and Expo*. IEEE, 2011, pp. 1–5.

[16] D. Xie and C.-C. Kuo, "Secure lempel-ziv compression with embedded encryption," in *Electronic Imaging 2005*. International Society for Optics and Photonics, 2005, pp. 318–327.

[17] J. Kelley and R. Tamassia, "Secure compression: Theory & practice." *IACR Cryptology ePrint Archive*, vol. 2014, p. 113, 2014.

[18] M. O. Külekci, "On scrambling the burrows–wheeler transform to provide privacy in lossless compression," *Computers & Security*, vol. 31, no. 1, pp. 26–32, 2012.

[19] M. Stanek, "Attacking scrambled burrows-wheeler transform." *IACR Cryptology ePrint Archive*, vol. 2012, p. 149, 2012.

[20] M. Burrows and D. J. Wheeler, "A block-sorting lossless data compression algorithm," 1994.

[21] "bzip2," http://www.bzip.org/, 1996-2017, accessed Augest 8, 2017.

[22] J. L. Bentley, D. D. Sleator, R. E. Tarjan, and V. K. Wei, "A locally adaptive data compression scheme," *Communications of the ACM*, vol. 29, no. 4, pp. 320–330, 1986.

[23] P. Ferragina and G. Manzini, "Opportunistic data structures with applications," in *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*. IEEE, 2000, pp. 390–398.

[24] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 1999, pp. 223–238.

[25] T. Moataz, T. Mayberry, and E.-O. Blass, "Constant communication oram with small blocksize," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 862–873.

[26] T. A. Welch, "A technique for high-performance data compression," *Computer*, vol. 17, no. 6, pp. 8–19, 1984.

[27] D. Okanohara and K. Sadakane, "A linear-time burrows-wheeler transform using induced sorting," in *International Symposium on String Processing and Information Retrieval*. Springer, 2009, pp. 90–101.

[28] G. Manzini, "An analysis of the burrows-wheeler transform," *Journal of the ACM (JACM)*, vol. 48, no. 3, pp. 407–430, 2001.

[29] J. Katz and Y. Lindell, *Introduction to modern cryptography*. CRC press, 2007.

[30] D. Boneh, "The decision diffie-hellman problem," in *International Algorithmic Number Theory Symposium*. Springer, 1998, pp. 48–63.

[31] G. Jean-loup and M. Adler, "gzip."

[32] "The canterbury corpus," http://corpus.canterbury.ac.nz/, January 2001, accessed Augest 5, 2016.

[33] "Silesia compression corpus," http://sun.aei.polsl.pl/ sdeor/index.php?page=silesia, accessed Augest 23, 2017.

[34] "Ensembl protists," http://protists.ensembl.org/index.html, 2017, accessed November 11, 2017.

[35] "National center for biotechnology information," https://www.ncbi.nlm.nih.gov/, accessed November 11, 2017.