Lempel-Ziv Compression with Randomized Input-Output for Anti-Compression Side-Channel Attacks under HTTPS/TLS

Meng Yang and Guang Gong

Department of Electrical and Computer Engineering
University of Waterloo,
Waterloo, ON, N2L 3G1, Canada
{m36yang, ggong}@uwaterloo.ca

Abstract. Security experts confront new attacks on TLS/SSL every year. Ever since the compression side-channel attacks CRIME and BREACH were presented during security conferences in 2012 and 2013, online users connecting to HTTP servers that run TLS version 1.2 are susceptible of being impersonated. We set up three Randomized Lempel-Ziv Models, which are built on Lempel-Ziv77, to confront this attack. Our three models change the deterministic characteristic of the compression algorithm: each compression with the same input gives output of different lengths. We implemented SSL/TLS protocol and the Lempel-Ziv77 compression algorithm, and used them as a base for our simulations of compression side-channel attack. After performing the simulations, all three models successfully prevented the attack. However, we demonstrate that our randomized models can still be broken by a stronger version of compression side-channel attack that we created. But this latter attack has a greater time complexity and is easily detectable. Finally, from the results, we conclude that our models couldn't compress as well as Lempel-Ziv77, but they can be used against compression side-channel attacks. ...

Keywords: Lempel-Ziv compression, encryption, compression side-channel attack, randomization, TLS

1 Introduction

The Internet is growing bigger and bigger everyday. Along with Facebook's new drone project that will allow more people to connect online, more information will travel on the network. We would like those information to travel in a fast and secure way. Data compression is used to make the information travel faster, and data encryption is used to make sure the information is confidential.

The TLS communication protocol is used to ensure secure communications between two nodes and to facilitate data exchange with compression. It also provides authentication with certificates, keeps confidentiality with encryption,

and grants integrity with message digest. For compression, TLS uses either DEFLATE algorithm or Lempel-Ziv-Stat algorithm [1].

Currently, servers are using version 1.2 of TLS, which supports both encryption and compression. However, combining these two algorithms has security flaws. Attackers could open the content of the encrypted HTTP header and use the authentication token within the cookie to impersonate an user. To perform this attack, the attacker needs to create multiple forged strings, and append each one to the plaintext. to plaintexts that get compress-then-encrypted, then needs to look for the encrypted text among them with the shortest length to recover a secret within the plaintext one character at a time. The type of attack that does not directly use brute force algorithms to retrieve the content of an encrypted message is called side-channel attack. For our case, the side-channel attack that exploits leakage from output's lengths of compression algorithms is called compression side-channel attack.

The compression side-channel attack was studied in a paper at the beginning of year 2002 [2]. The authors of the paper explain how an adversary could use the length the outputs of the compressed-then-encrypted text to retrieve a secret within the message. In recent years, two compression side-channel attacks were presented to public and demonstrated during computer security conferences: CRIME (Compression Redundancy Infoleak Made Easy) in 2012 [3]; and BREACH (Browser Reconnaissance and Exfiltration via Adaptive Compression of Hypertext) in 2013 [4]. The presenters could successfully recover a secret string hidden inside HTTP message. TLS took a big hit and plans to shutdown all compression methods completely in its next version 1.3 [1].

Removing compression has a big impact on the network traffic due to large packets that are traveling without being compressed. In this paper, we propose three schemes where compression could be re-enabled before encryption without risk of compromising security. The three schemes randomly modify the inputs or randomly repeats the outputs of the compression algorithm:

1. Weak Cipher With Lempel-Ziv: the plaintext passes through a weak cipher before being compressed.
2. Lempel-Ziv With Control: the outputs of the compression are randomly repeated.
3. Lempel-Ziv With Control and Feedback: the outputs of the compression are randomly repeated, then decompressed and compressed again.

To analyse our schemes, we implemented the compression side-channel attack algorithm and performed simulations on them. We examine the results in terms of security, compression ratio, and runtime. Even though the attack was successfully prevented, we noticed the increase in time complexity of the algorithm, and increase in compression ratio.

Since the attack is countered, we would like to test the limit of our schemes. We augmented the attack, named strong compression side-channel attack. This attack builds on top of the compression side-channel attack, and repeatedly sends and collects length of many outputs and uses the average size of the lengths to decrypt the secret within encrypted message.

After running the strong compression side-channel attack simulations, our three schemes resisted to the attack to a certain degree. The attack eventually breached all our schemes given a large amount of time. The time complexity is immense such that the attack is practically not feasible given the lifetime of the secret.

Schemes are not limited only to randomly repeating inputs or outputs around the compression algorithm. We propose another scheme called PermuLZ. In this scheme, the plaintext is fragmented into smaller pieces then rearranged randomly before going through compression. This scheme could counter the attack, but it could have a longer time complexity and a decrease in compression ratio as trade-off.

2 Preliminaries

In this section, we describe the Lempel-Ziv compression algorithm family, component functions used in our schemes, and define the notations used in this paper.

2.1 Compression Algorithms

We use the following notations throughout of the paper:

- $A||B$ concatenates two binary strings A and B
- ζ represents an alphabet or character.
- Σ is a set containing all possible alphabets.
- M is the plaintext.
- C is the ciphertext.
- $argmin_x y$ is a function returning the argument x that associates to the smallest value y.

Lempel-Ziv Family

Lempel-Ziv 77 The Lempel-Ziv compression algorithm, shown in Algorithm 1, was created by Abraham Lempel and Jacob Ziv. They published the first version of the algorithm in 1977, called Lempel-Ziv77. This compression algorithm focus on replacing strings that repeats in the file with references to locations where they appeared first. The reference is a length-distance tuple, which indicates the location of the pattern and its length. The tuple also contains the next character after the pattern.

Lempel-Ziv77 Example When Lempel-Ziv77 algorithm on "abcabcd", we would get "abc(-3,3,'d')" since the second "abc" is happened previously. The second appearance is replaced by "(-3,3,'d')", where the first number, -3, is the relative location, and the second number, 3, is the length. This latter tuple would get converted into two bytes. Thus, for the text "abcabcd", Lempel-Ziv77 would

Algorithm 1 LEMPEL-ZIV77($file$)

```
current_position ← start of file
outputs_list ← empty list
while current_position not reach end of file do
    move sliding_window
    longest_str ← find longest matching string for current_pos in sliding_window
    pos ← relative starting position of longest_str from current_pos
    len ← length of longest_str
    next_char ← next character after longest_str in sliding_window
    append (pos, len, next_char) to outputs_list
    current_position ← current_position + j
end while
return outputs_list
```

give a compression ratio of $\frac{5}{7}$, since the original text has length of 7 bytes, and Lempel-Ziv77 outputs has length 5 bytes. When decompressing "(-3,3,'d')", the two numbers, -3 and 3, inside the tuple will be expanded. First, we move back three characters to point to the letter "a", then copy the next three characters indicated by the second number. Thus, the tuple "(0,3,'d')" gets replaced by "abcd".

Lempel-Ziv78 In the year that followed, 1978, the same authors published their second compression algorithm named Lempel-Ziv78, which is an extension of Lempel-Ziv77 [5]. In this version, the length of the repeated pattern is removed in the reference tuple. The repeated pattern gets replaced only by "(location, next character)" pair. The length is omitted since it can be calculated from start location and position of the next character.

Huffman Coding Huffman Coding is another compression algorithm which is often used together with Lempel-Ziv77. This algorithm gives each character a new encoding. The compression happen when the characters get replaced. The most frequent characters has an encoding of fewer bits, and the least frequently used characters get an encoding of many bits.

DEFLATE and gzip Both DEFLATE and gzip are popular compression algorithms used by HTTP [6]. When the compression is done using DEFLATE algorithm, the plain-text is first compressed by Lempel-Ziv77, then the outputs are compressed again by Huffman encoding [7].

The gzip compression algorithm first compresses the plain-text with DEFLATE algorithm, then adds a header and a footer to the output of the compression. The header contains information about the file, such as the timestamp and compression flags, and the footer is a CRC-32 checksum generated from the compressed message [8].

2.2 Pseudo-Random Sequence Generator

De Bruijn Sequences The sequence is named after the mathematician Nicolaas Govert de Bruijn. Given an order n, the sequence outputs a cyclic sequence of bits where every substring of length n is unique [9].

WG-8 Cipher The WG-8 cipher, designed by Professor Gong with Xinxin Fan and Kalikinkar Mandal, is a light-weight cipher that outputs a sequence with 2-level auto-correlation. The keystream sequences outputs have many randomness properties, such as a period of $2^{160} - 1$, balanced 0's and 1's, two level auto correlation sequence, ideal t-tuple distribution, and large linear span of $2^{33.32}$ [10].

Mersenne Twister Mersenne Twister generator was created in year 1997 by Makoto Matsumoto and Takuji Nishimura [11]. This pseudo-random number generator is based on a Mersenne Prime number, which is a prime number that can be written in this format: $2^n - 1$ where n is also a prime. The period of Mersenne Twister is equal to the prime number. It is a quick pseudo-random number generator and has better randomness properties than other fast generators. Generally, Mersenne Twister picks $2^{19937} - 1$ as its prime. It has good randomness properties such as period of $2^{19937} - 1$, and it passes Diehard tests.

2.3 Encryption Algorithm

AES Cipher AES does block encryption. In HTTP or TLS protocol, compression is used before applying AES encryption. It takes a key of either 128, 196, 256 bits, and an input block of the same length, then outputs encrypted block of that length [12]. Other ciphers are used in HTTPS or TLS [1], but we used AES to demonstrate our results.

3 Compression Side-Channel Attack

In this section, we first analyse the compression side-channel attack (CSCA), then propose three models to deter the attack, and finally present the results of the attack performing simulations against our models.

3.1 Compression Side-Channel Attack

To learn how to fight against the attack, we must first understand the attack. Here below, we present the compression side-channel attack algorithm, along with an example, and we also provide a theoretical proof of the leakage of compression and encryption combination.

Algorithm 2 CSCA($http\_request$)

1: $header \leftarrow$ HTTP header from $http\_request$
2: $known\_prefix \leftarrow$ secret's prefix from adversary
3: $guess\_secret \leftarrow$ empty string of length L set by adversary
4: **for** $i = 0$ **to** $i =$ secret_length **do**
5: $L \leftarrow$ empty list
6: **for** each $\zeta \in \Sigma$ **do**
7: $guess\_secret[i] \leftarrow \zeta$
8: $g_\zeta \leftarrow known\_prefix||guess\_secret[i]$
9: $M_\zeta \leftarrow header||g_\zeta$
10: $C_\zeta \leftarrow Enc(LZ77(M_\zeta))$
11: C_ζ gets sent over network
12: $l_\zeta \leftarrow length(C_\zeta)$
13: $L.insert(l_\zeta)$
14: **end for**
15: $guess\_secret[i] \leftarrow \{\zeta|\zeta = argmin_\zeta\{l_\zeta \in L\}\}$
16: **end for**
17: **return** $guess\_secret$

Analysis of Compression Side-Channel Attack The following algorithm, Algorithm 2, shows the execution steps of a compression side-channel attack.

We assume that the client's computer is already infected by a malware that can only manipulate HTTP requests before they get sent. We are also assuming that the adversary can only inspect and retrieve the lengths of encrypted TCP packages on the network. Additionally, we assume that the victim already made connection to a server, and has established an authentication token that is located inside the header of HTTP request.

Before a victim client sends another HTTP request to the server, the request is first held onto by the malware, and the whole HTTP header is retrieved in line 1.

The adversary has already examined the format of the HTTP header and the cookie within in line 2, and knows the prefix string that prepends the secret. The prefix string has been already given to the malware. The malware first initiates an empty string, *guess_secret*, that will eventually become the real secret at the end of the algorithm on line 3.

The malware uses *guess_secret* to retrieve the real secret token within the HTTP header. It knows in advance that the length of the secret is constant. It constructs a string g as a concatenation of $known\_prefix$ and *guess_secret*, and it will append g to the HTTP header.

The *guess_secret* is constructed in a particular way. The malware tries to guess each character, one at a time, from left to right. For each i-th character in the *guess_secret*, the malware sets it to be one of the possible characters, ζ, from the alphabet set Σ, shown in line 7.

Once a character is set in *guess_secret*, the latter is appended to $known\_prefix$, which becomes g and is then appended to the HTTP *header* in lines 8-9.

This new header, M, is compressed and encrypted, then gets sent to the server by the client in lines 10-11.

Since the message will be going through the network, its length l can be retrieved by sniffer at line 12. The lengths l_ζ for each ζ is then saved in a list L for later analysis, shown in line 13.

After collecting all the lengths, the attacker analyses them and retrieve the message with the shortest length, since the length of the encrypted header with wrong guesses is longer than the length of encrypted header with the correct guess. The forged $guess\_secret[i]$ inside this message corresponds to the correct character at i inside the real secret. In other words, ζ that corresponds to the message with shortest length in L is the correct i character inside the real token, shown in line 15.

Once all the characters are recovered one by one, *guess_secret* will match the real secret token within *header*, and the adversary can use this token, *guess_secret*, to impersonate the victim.

Compression Side-Channel Attack Example The following example explains how we simulated a compression side-channel attack on the following HTTP header. Lempel-Ziv77 was the compression algorithm used. This HTTP header below is the raw HTTP header from accessing the website `https://www.google.com`.

```
Alt-Svc: quic=":443"; ma=2592000; v="34,
33,32,31,30,29,28,27,26,25"
...
Set-Cookie: NID=79=rAJMNHlcYMf6Vg3FxMIPE
kxRcLStbWDVxb7Dng9puqepumjZJ5nsRnOQbiOR0
MILZp8u-jHt2fExUTLMgVgb3MUYwdxbp2V7vb4YP
0LKxhHfx5e8bUekI4_Eo4NupdYpTDvsGqDfhgbG3
kWFw2y_yaNuQAhND4ULU1zCo0Eysyzv1nM6Y6zba
5MOfVj9zhbnltLCVAcoiYl5CeF7opB_DZ5vedm2d
bouqXle;
expires=Thu, 08-Dec-2016 21:59:58 GMT;
path=/; domain=.google.ca; HttpOnly
X-Firefox-Spdy: h2
...
```

We will use the compression side-channel attack to retrieve the authentication token: "`rAJMNHlc ... Xle`" hidden by encryption.

In the first step, we retrieve the header and examine the secret's $known\_prefix$, which is `NID=79=`. Then we create a forge string, g, which has the prefix `NID=79=`, and we add a guessing character from the alphabet, $\Sigma = a, b, \cdots, z, A, \cdots Z$. So, the first forged string g takes the first letter of the alphabet as guess: $g \leftarrow$ `NID=79=a`. Then, we appends g to the end of the header. Before the header gets sent, it gets compressed by Lempel-Ziv77 and encrypted by TLS. The content of the message is hidden, but the lengths of the ciphertext is not. After

collecting the length l, we try the second letter b as guess: $g \leftarrow$ `NID=79=b`. After trying all the alphabets, we can collect the length of each ciphertext containing a different guess. By collecting all the lengths, we could determine that the ciphertext with the shortest length corresponds to the message containing the forged string with guess `r`. Thus, we can deduce that the first letter of the secret is `r`. These steps are repeated for guessing the second letter of the secret, which would be A, and we repeat for all 225 characters. Then, the whole secret token "`rAJMNHlc ... Xle`" is recovered.

Analysis of Lempel-Ziv Against Compression Side-Channel Attack We now analyse compression side-channel attack and prove that an adversary can use it to retrieve any secrets within a encrypted message given that he knows the secret prefix.

Let S be the header concatenated together with the forged string:

$$S = s_0 s_1 \cdots P s_t s_{t+1} \cdots s_{t+T} \cdots G\hat{x}$$

where s_i are characters, P is a string that represents the secret's known prefix, s_t to s_{t+T} represents the real secret of length T, G is the known prefix appended by correct guessed characters, which is added by adversary, and $\hat{x}$ is the next character currently being guessed. We segment the file S into a list of sequence of strings, Q_is.

$$\begin{aligned} S &= s_0 s_1 \cdots P s_t s_{t+1} \cdots s_{t+T} \cdots G\hat{x} \\ &= Q_0 Q_1 \cdots Q_m \end{aligned}$$

After the plaintext goes through compression, some sequence Q_i would be replaced by a Lempel-Ziv77 compression output $r_i = (position, length, next\_char)$ since it happened before.

$$\begin{aligned} LZ77(S) &= LZ77(s_0 s_1 \cdots s_n) \\ &= LZ77(Q_0 Q_1 \cdots Q_m) \\ &= R = r_0 r_1 \cdots r_l \end{aligned}$$

We define length of the compressed file $l = \|R\|$ as the number of Lempel-Ziv77 outputs. For the attack to be successful, the length of the compressed file with incorrect guess must be longer than the file with correct guess: $l_{\hat{x} \neq s_t} > l_{\hat{x}=s_t}$. We will prove this case below.

Property 1. $l_{\hat{x} \neq s_t} = l_{\hat{x}=s_t} + 1$, where l is size of R.

Proof. The prefix, P, and the first character of the secret, s_t, can be segmented in two ways.

1. $Ps_t \in Q_u$: both P and s_t are found in the same segment Q_u.

2. $P \in Q_u$ and $s_t \in Q_{u+1}$: P is found in Q_u and s_p is found in Q_{u+1}.

The prefix and first character will be repeated in the forged string, which then will become an Lempel-Ziv output tuple r_i after compression.

We can eliminate the second case, since because if only the segment that contains P is referenced, then s_t is included in the Lempel-Ziv output as the *next_character*. Thus, we can say that the prefix and the first character are in the same segment: $Ps_t \in Q_u$.

Now we show that the length of the the message with the incorrect guess is larger than one with correct guess.

After compression, the input file S becomes compressed file R.

- If the guess is correct, $\hat{x} = s_t$, then $G\hat{x}$ is compressed to only one output r_{l_1}, that references Q_u. The compressed file R would have this output at the end: $R = r_1 r_2 \cdots r_{l_1}$.
- If the guess is incorrect, $\hat{x} \neq s_t$, then only the G is compressed since it matches the prefix P in Q_u. The output of compressing the prefix P is also r_{l_1} (call this r'_{l_1}) but with a smaller value for *length*, and the guessing character $\hat{x}$ is the *next_character*. However, the Lempel-Ziv77 algorithm also prints a terminal output r_{l_2} that has a new line as the *next_character* to mark the end of the algorithm.

The length of the compression result for the message with the correct guess is $l_{\hat{x_0}=s_t} = \|R_{correct}\|$ where $R_{correct} = r_0 r_1 \cdots r_{l_1}$. The length of the result with incorrect guess is $l_{\hat{x_0}=s_t} = \|R_{incorrect}\|$ where $R_{incorrect} = r_0 r_1 \cdots r'_{l_1} r_{l_2}$.

This clearly shows that the length has increased by one block, thus, $l_{\hat{x_0}\neq s_t} = l_{\hat{x_0}=s_t} + 1$.

To summarize, the length of the compressed file with incorrect guess is indeed longer than the file with correct guess: $l_{\hat{x}\neq s_t} > l_{\hat{x}=s_t}$.

3.2 Adding Randomization to Lempel-Ziv77

In the following, we explain our motivation behind increasing the randomness of Lempel-Ziv compression, then, we propose three methods built around Lempel-Ziv77 compression algorithm before applying encryption to stand against compression side-channel attack. The goal of each of the methods below is to make the compressed message's length not deterministic. With those models, the length of the message containing the forged token with the correct guess is not shorter than length of the message containing the forged token with the incorrect guess.

Length Analysis of Randomized LZ Previously, we showed that an adversary can recover a secret from a cookie inside a web browser. We will show how we can prevent this attack by adding randomness to LZ. Our goal is to prevent the attacker from guessing $\hat{x_0}$. By varying the entropy of the plain-text or by randomly padding the outputs of LZ compression, we can achieve $l_{\hat{x_0}\neq s_t} \geq l_{\hat{x_0}=s_t}$.

With a pseudo-random sequence generator, we can use the output bits as a control to change the entropy of the plaintext and the result of the compression won't be obsolete: the result with the correct guess could be larger than the result with an incorrect guess. Alternatively, we could vary with control the results of the Lempel-Ziv compression. This will provide us the same effect: $l_{\hat{x_0} \neq s_t} \geq l_{\hat{x_0} = s_t}$.

Remark Shannon's entropy is defined to be the number of bits needed to represent the information. By this definition, since compression removed all the redundancy, the entropy of the compressed text must be smaller than the entropy of the original plaintext. The information that could not be compressed is shared between the plaintext and the compressed plaintext, and this information is mutual information. If we add randomization to the compression algorithm, the entropy of the compressed text would not decrease as much, thus, increasing the mutual information, making the length of the compressed text unpredictable. This unpredictability will hinder the attacker from doing a compressed side-channel attack.

Model Elements Before defining our models, we first look at some implementations that are used in the models.

Pseudorandom Sequence Generator (PRSG) Our three schemes below use the same pseudorandom sequence generator, PRSG, which generates a sequence of 0's and 1's. In our simulation, PRSG is constructed based on the random number generator in Python 2.7 that uses the Mersenne Twister algorithm to generate a floating number in the range of 0 to 1 exclusively. The floating numbers has 53-bit precision floats and a period of $2^{19937} - 1$. Note that Mersenne twister is not cryptographic secure [11]. However, in real world applications, it should use de Bruijn Generator or WG-8 (see Section 2.2)

Weighted Generator We implemented a weighted pseudo-random number generator that generates an uneven number of 0s and 1s. Specifically, the PRNG outputs 10% 1's' and 90% 0's. Since the Mersenne Twister random number generator from Python can output a random floating number between 0 and 1, we take this floating number to determine whether to output 0 or 1. If the number is bigger than a given *ratio*, 0.1 for example, then our generator outputs 0, and 1 otherwise.

The ratio must not stay 10% for each compression. It must vary or else the stronger compression side-channel attack works. We modify the ratio before compressing each file. First, our custom PRNG takes the original *ratio* (10%) specified by the user during initialization. Then, the *ratio* would get adjusted between $ratio-0.01$ and $ratio+0.01$, with probability of a Gaussian distribution. For our case with 10% as the base ratio, this will vary in the range of 9% to 11% in a Gaussian fashion. This Gaussian variation is done using the Python `random.gauss` function: another floating number is generated between 0 and 1 with Gaussian distribution with a mean of 0.5 and a standard deviation of

0.25, then its value decreases by 0.5 to make it between -0.5 and 0.5, and then multiplied by 0.02 to make its value between -0.01 and 0.01, finally this number gets added to our original 10% to get a variation of 0.01%.

Weak Stream Cipher We are using a weak stream cipher to increase the entropy of a file without modifying it too much. It uses our custom PRNG as bit-stream input. It then performs XOR operation with the input file. The original message is only transformed by little due to uneven distribution of 0's and 1's.

Lempel-Ziv77 The Lempel-Ziv77 compression algorithm is implemented in Python 2.7. Just as described in Algorithm 1, a sliding window is created before the current character, and for each character, the algorithm would look for the longest sequence that matches within the sliding window.

Randomized Lempel-Ziv Models By adding a pseudo-random number generator, we designed three models to add randomness to a simple Lempel-Ziv77.

Scheme 1: Weak Cipher with Lempel-Ziv77 For this method, we first process the text by passing it through a weak stream cipher before applying compression, shown in Figure 1. Since each message will have a different entropy, they will all compress differently. So the transmitted message that has the shortest length may not contain the forged token with the correct guess since its entropy may be higher than a message with an incorrect guess due to the addition of the weak cipher.

The weak cipher is based on our PRSG. We changed the ratio from 10% to 1.308% because we want 10% of the characters inside the plain-text to be changed. The 1.308% came from $1 - \sqrt[8]{0.9}$, because each character are 8 bits and we want the 8 bits to be all 0's 90% of the time.

Note in [2], the authors also proposed to use weak cipher as one of the countermeasures to the compression side-channel attacks, but neither analysis nor experiments are given.

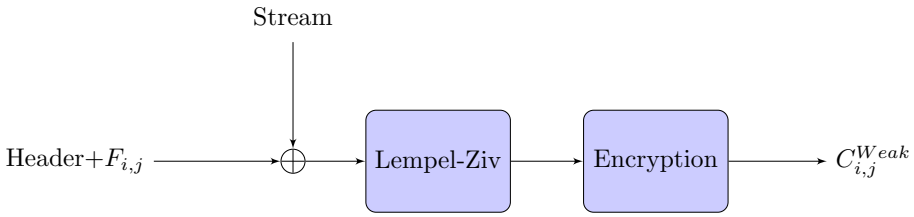


Fig. 1: Diagram of Weak Cipher with Lempel-Ziv

Scheme 2: Lempel-Ziv With Control The idea behind Lempel-Ziv With Control, LZWC, Figure 2, is to repeat some of the output tuples of Lempel-ZIv in order to mask the real length of the compressed output.

A random stream of bits, r, will be needed. Each of the outputs of Lempel-ZIv will be matched with a bit in the stream r: when the bit is 1, then the output will be duplicated, or else nothing happens.

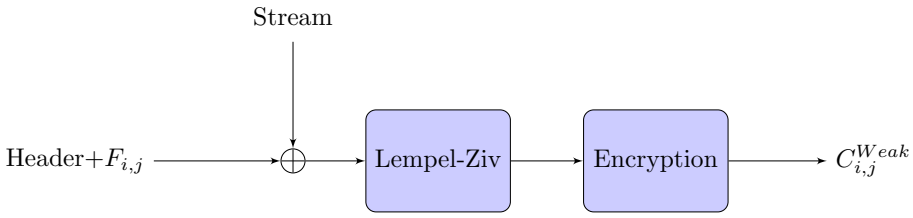


Fig. 2: Diagram of Lempel-Ziv With Control

Scheme 3: Lempel-Ziv With Control and Feedback This method aims to add even more randomness around the Lempel-Ziv compression step to prevent compression side-channel information leakage, Figure 3. First, the plain-text is passed through a Lempel-Ziv77 compression algorithm. Then, some of its results are duplicated the same way as LZWC did. Afterwards, the result would get decompressed, then compressed again with Lempel-Ziv77.

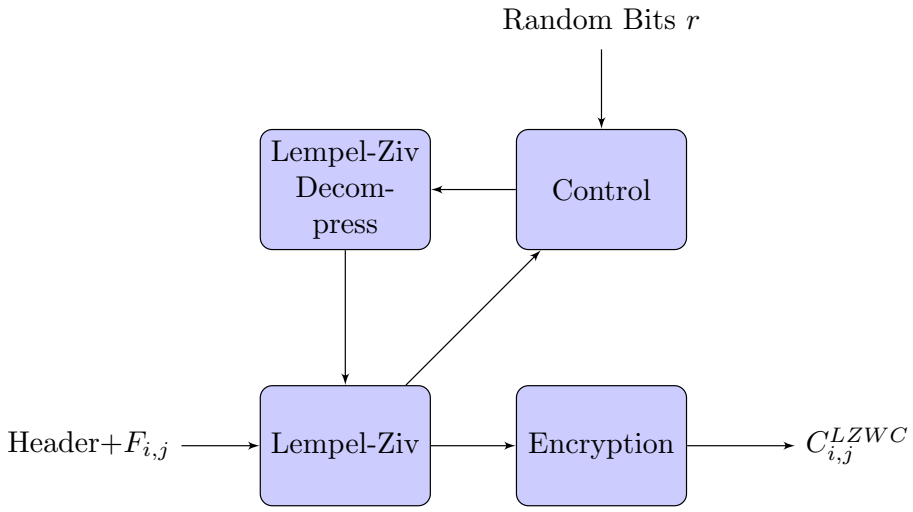


Fig. 3: Diagram of Lempel-Ziv With Control and Feedback

3.3 Simulation Results for Randomized LZ Models

We programmed our three compression schemes in Python2.7 and tested them against compression side-channel attacks.

Testing Environment The simulated compression side-channel attack is run on our server that has the following specifications.

Processors	80
CPU	Intel(R) E7-L8867 @ 2.13GHz
RAM	529105932 kB

The attack alongside the schemes are implemented in Python 2.7. The HTTP header used to do the tests is the same as above, but with different secrets. We ran the attack on 6 different header files, each one with a different secret.

Compression Side-Channel Attack on original Lempel-Ziv77 We ran the compression side-channel attack, i.e. Algorithm 2, on the original compression algorithm Lempel-Ziv77, i.e. Algorithm 1. Just as described above, a forged token with a guessing character was added to the plain-text, which goes through Lempel-Ziv77 compression next; then the attack measures the length of the compressed texts and selects the shortest one to be the correct character. We were able to decode the secret within all the header files in a short amount of time.

Compression Side-Channel Attack on Lempel-Ziv77 with Randomization The same attack was tested on plain-texts which went through Lempel-Ziv77 compression algorithm with addition of randomization. As expected, all three compression schemes prevented the attack: the attacker fails to decode the secret token within the HTTP header.

Compression Side-Channel Attack on Lempel-Ziv With Weak Cipher The weak cipher that the plain-text goes through before being compressed increases the entropy of the message. Since the plain-text changed, some original repetitions did not happen. So the forged token might not match the real token. The length of the compressed text depends on how many repetitions that were kept. Thus, the attack does not succeed since this increase in entropy hides the exact length of the plain-text.

Compression Side-Channel Attack on LZWC By varying the output, the real length of the plain-text is also hidden. After the plain-text has been compressed, some of its output pairs are duplicated. This duplication is random and the plain-text with a correct forged token might have more duplicates than a plain-text with an incorrect forged token; thus, the incorrect guess will result in more outputs after the compression algorithm. This way, the attack will make a wrong guess, and won't be able to decode the secret.

Compression Side-Channel Attack on LZWCF Similar to the previous scheme, this also prevents the compression side-channel attack. This scheme adds more randomization by feeding the output back into the compression algorithm, making the length of the compressed text even more random. Hence, this also prevents the attack.

4 Strong Compression Side-Channel Attack

The normal compression side-channel attack can be easily prevented if the victim employs one of the three schemes mentioned above. The randomness of the algorithm masks the length of the compressed files, which would be deterministic without any randomness involved. This way, the attacker cannot determine for certain that the shortest encrypted message has the correct guess. However, the methods cannot provide complete immunity against this strong compression side-channel attack described in this section.

4.1 Model and Analysis of Strong Compression Side-Channel Attack

A modified compression side-channel attack can still recover the token. We call this Strong Compression Side-Channel Attack, shown in Algorithm 3. The strong

Algorithm 3 STRONG_CSCA(*http_request*)

```
header ← HTTP header from http_request
known_prefix ← secret's prefix from adversary
guess_secret ← empty string of length L set by adversary
for i = 0 to i = secret_length do
    L ← empty list
    for each ζ ∈ Σ do
        guess_secret[i] ← ζ
        g_ζ ← known_prefix||guess_secret
        M_ζ ← header||g_ζ
        T ← empty list
        for j = 0 to j = K do
            C_ζ ← Enc(LZ77(M_ζ))
            C_ζ gets sent over network
            l_ζ ← length(C_ζ)
            T.insert(l_ζ)
        end for
        L.insert(avg(T))
    end for
    guess_secret[i] ← {ζ|ζ = argmin_ζ{l_ζ ∈ L}}
end for
return guess_secret
```

compression side-channel attack is similar to the normal one, Algorithm 2, but with a few differences listed below:

1. After compressing and encrypting the message, instead of sending only one single request for each character ζ_j, the adversary makes the victim send K requests per character, line 11. The adversary then collects the lengths of all K encrypted requests, and saves all of them in a new list T, shown in lines 14-15.
2. Then the adversary computes the average of all K messages that has the same guessing character ζ, and saves that average in the list L, shown in line 17.

The last part is the same as the normal attack: the adversary chooses the character that gives the lowest average to be the guess character, shown in line 19.

4.2 Results from Simulation of Strong Compression Side-Channel Attack on Randomized LZ Models

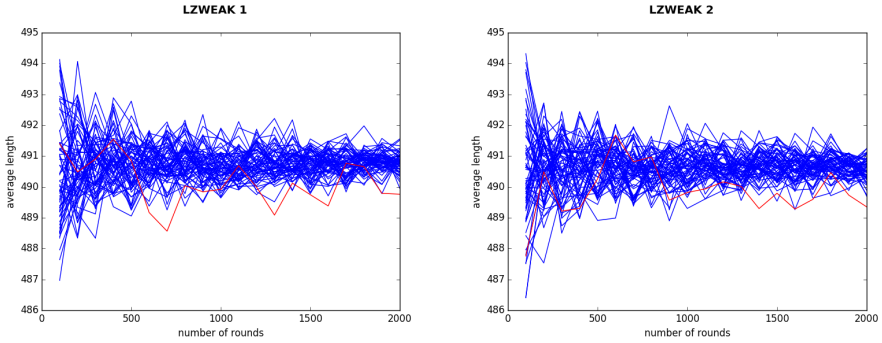
The same testing environment, which was described above, was used to simulate a strong compression side-channel attack on our three randomized LZ models.

Strong Compression Side-Channel Attack on Randomized LZ Models The modified LZ77 with randomization can successfully prevent a normal compression side-channel attack. In this section, we will see how they do against a strong compression side-channel attack.

Strong Compression Side-Channel Attack on LZ77 With Weak Cipher In Figure 4, the red line indicates the length of the compressed plain-text with a forged token with correct guess, and blue lines indicates the ones with incorrect guesses. We see that for most of the cases, the attack can hardly recover one character after 2000 requests. Thus, this scheme is capable of resisting against a strong compression side-channel attack.

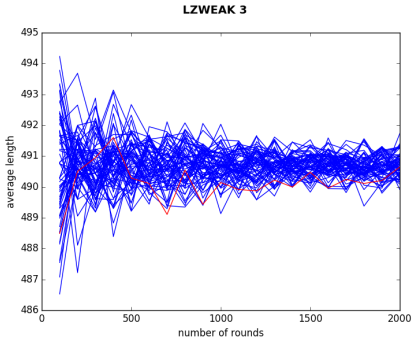
Strong Compression Side-Channel Attack on LZWC Unlike the previous scheme, this one does not resist the strong compression side-channel attack very well. From Figure 5, 300 requests were enough to decode one character in most cases.

Strong Compression Side-Channel Attack on LZWCF The testing results of LZWCF are displayed in Figure 6. As shown in the graphs, LZWCF does better than LZWC, but still cannot prevent the strong compression side-channel attack. To guess one correct character, 500 requests is usually enough.

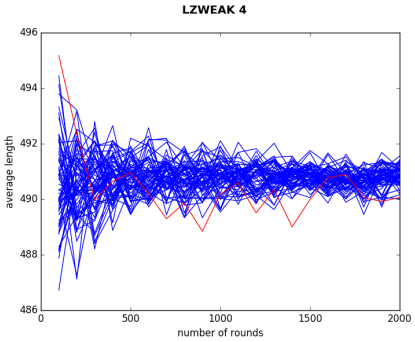


(a) Token 1

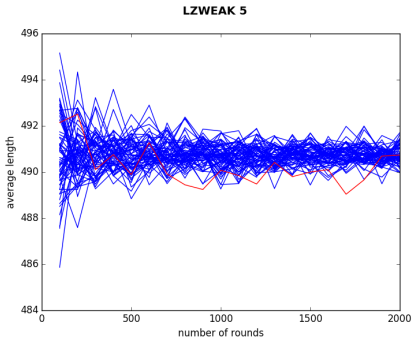
(b) Token 2



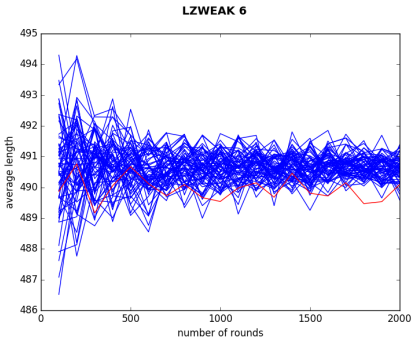
(c) Token 3



(d) Token 4



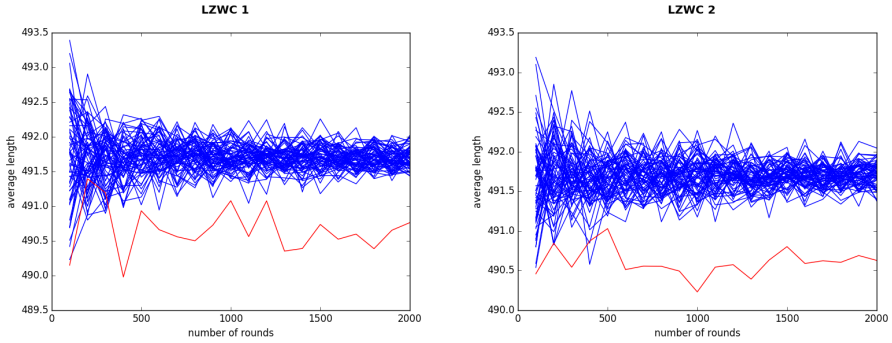
(e) Token 5



(f) Token 6

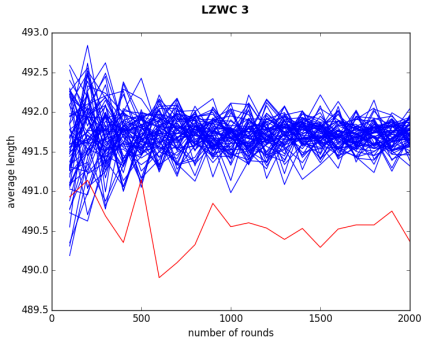
Fig. 4: Results of SCSCA on LZ with Weak Cipher

Compression Rate In Table 1, we present the different compression ratio from our randomized LZ models. From the table, we see that our schemes have $(.879 - .792)/.792 = 11.0\%$, $(.881 - .792)/.792 = 11.2\%$, $(.943 - .792)/.792 = 19.1\%$ increase in the compressed size.

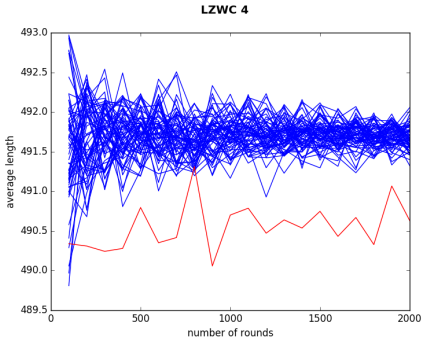


(a) Token 1

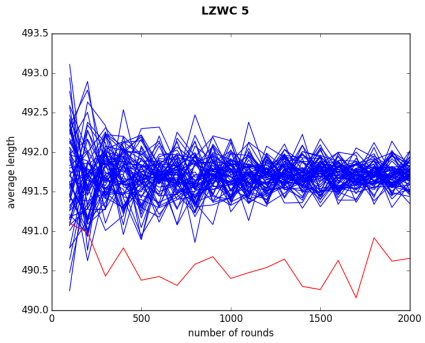
(b) Token 2



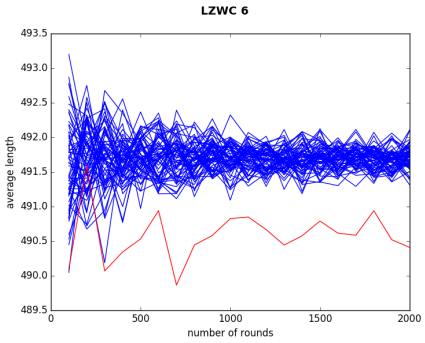
(c) Token 3



(d) Token 4



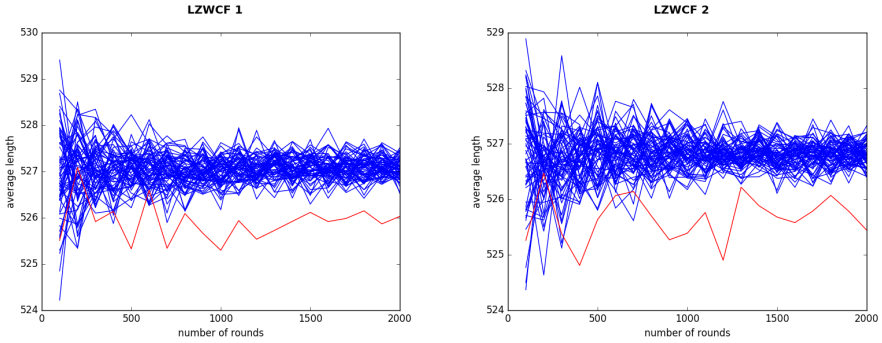
(e) Token 5



(f) Token 6

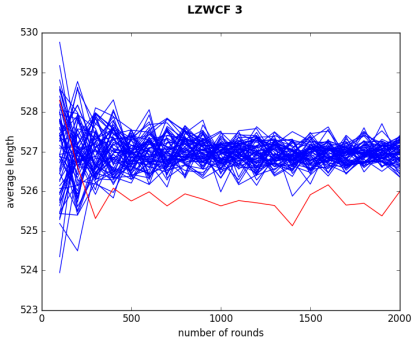
Fig. 5: Results of SCSCA on LZWC

Our three randomized LZ model schemes prevent normal compression side-channel attack, however, the size of the compressed message is increased as the trade-off.

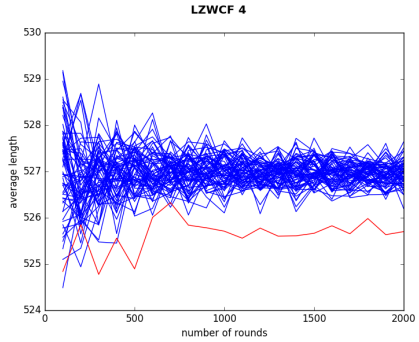


(a) Token 1

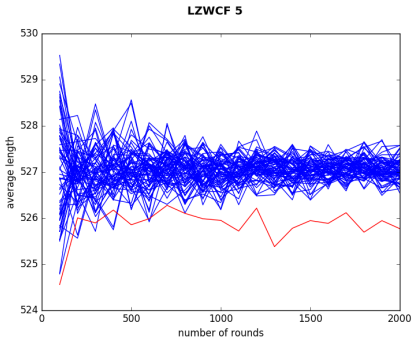
(b) Token 2



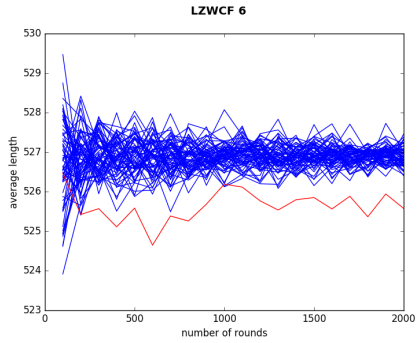
(c) Token 3



(d) Token 4



(e) Token 5



(f) Token 6

Fig. 6: Results of SCSCA on LZWCF

Timing The downside of strong compression side-channel attack is that it requires a long time to execute. In this section, we investigate the time it take to resolve one secret of length 228.

Scheme	Worst (in bytes)	Comp vs Orig
lz	445	0.792
lz with weak	494	0.879
lzwc	495	0.881
lzwcf	530	0.943

Table 1: Compression Ratio from Each Schemes

During testing, the time to guess one character using iterations is also recorded. The Figure 7 shows the average time it takes to append the forged token and to compress. The graphs explain the relative time across all three schemes. The scheme LZWC has the fastest time: 482 seconds for the attack with 2000 iterations. The two other schemes, LZWeak and LZWCF, are twice as slow: 837 and 949 seconds respectively for the same number of iterations.

Previously, we have seen that LZWeak requires more than 2000 requests, so to decode one secret, the attack needs $837 \times 228 = 190836\mathsf{seconds} \approx 53\mathsf{hours}$ to decode one secret. For LZWC, 300 iterations take 72 seconds to execute, so the total time it takes is $72 \times 228 = 16416\mathsf{seconds} \approx 5\mathsf{hours}$. For LZWCF, 500 iterations take 237 seconds to execute, so the total time taken would be $237 \times 228 = 54036\mathsf{seconds} \approx 15\mathsf{hours}$. These hours are only the time it takes to compress, more hours will be added for encryption and transmission.

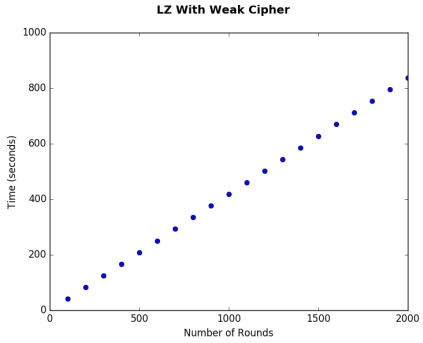
4.3 Number of Requests

When performing the strong compression side-channel attack, we assumed that we can send an unlimited number of HTTP headers to the oracle, the server in this case. However, in practice, sending that many requests is easily detectable by the server. If the server sets a limit on the number of requests a client can make, then it would nullify this strong compression side-channel attack. Thus, strong compression side-channel attack is only possible if the oracle allows an enormous number of requests.

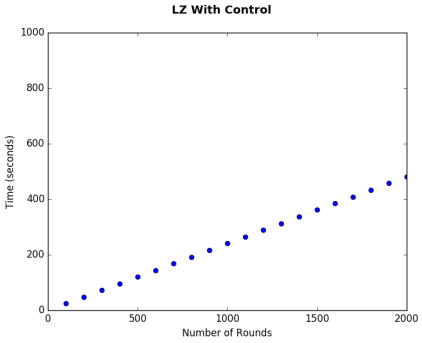
5 Summary, Conclusion and Future Work

An attacker can impersonate a client by using compression side-channel attack to retrieve the client's authentication token that is shared between the victim and the server. THe authentication token is hidden inside the HTTP header, but when the header is compressed then encrypted, the attacker could retrieve this token. To respond to this type of attack, TLS/SSL will disable compression in its most recent version. We analysed the compression side-channel attack that brought the disabling of compression in SSL 1.3 protocol.

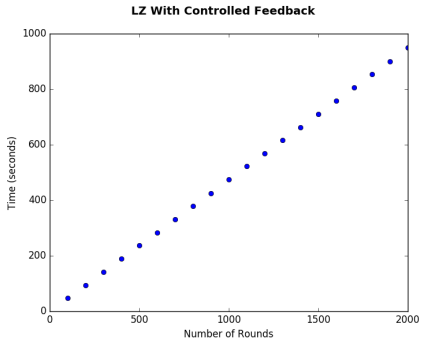
With the implementation of Lempel-Ziv77 compression algorithm and our own TLS/SSL, we simulated compression side-channel attack and successfully retrieved the secret inside an encrypted HTTP header.



(a) LZWeak Time



(b) LZWC Time



(c) LZWCF Time

Fig. 7: Timing Graphs of SCSCA on Randomized LZ Models

We proposed three schemes to counter compression side-channel attack by adding randomization to Lempel-Ziv77 compression algorithm. Each of the three schemes adds randomization in different ways: the first scheme, Lempel-Ziv with weak cipher, changes few characters of the plaintext and then pass it to the Lempel-Ziv77 compression; the second scheme, Lempel-Ziv with Control, repeats some the outputs according to a random control bit stream; and finally, the third scheme, Lempel-Ziv with Control and Feedback, repeats some the outputs like the previous scheme, then decompress the outputs into a file then compressed the file again.

After implementing the three schemes, we tested them against compression side-channel attack using our internal server. All of our schemes were able to resist to the attack, even though our schemes reduce the compression rate.

We also implemented a stronger version, called strong compression side-channel attack, of the attack to test robustness of our schemes. This attack requires the adversary to make multiple queries for one guess. From those queries, the adversary can collect all the indeterministic lengths and calculates the mean of all the lengths. With the average lengths, the adversary can deduce that the

character with lowest average length must correspond to the actual character of the secret.

Our three schemes could resist to the attack until a certain point. Eventually, all our schemes fail to resist to the attack when the amount of requests is high. If an attacker launches this attack, then the Lempel-Ziv with Weak Cipher scheme requires more than 2000 requests to be sent per character, and this would require more than 53 hours. For the Lempel-Ziv with Control, the secret would be safe for 5 hours, and for Lempel-Ziv with Controlled Feedback, the secret couldn't be retrieved within less than 15 hours.

The compression ratios of the three models did not decrease significantly: Lempel-Ziv with Weak Cipher model ratio increased by 11%; Lempel-Ziv with Control model increased by 11.2%; and Lempel-Ziv with Controlled Feedback increased by 19.1%. So, our models provides more security, but with a compression rate loss.

To conclude, TLS has disabled compression to prevent compression side-channel attacks for now. Our three models encourage TLS/SSL to re-enable compression to increase the flow of network traffic. These models can be implemented without much effort in the server and client: only an additional key is required to synchronize the randomization. Also, the required overhead time of our schemes is not as much compared to Lempel-Ziv77.

It is relatively difficult to prevent the attack by preventing users from downloading malwares, but the application of our schemes will ensure that compression side-channel attack malwares remain ineffective.

More focused work could be done in the future, which are summarized as follows.

A. Entropy Analysis We fixed our bit-stream generator to generate 1's ten percent of the time, and 0 ninety percent of the time. We could analyse the ratio of our bit-stream generator in depth with other file types, since all HTTP headers have similar entropy. For instance, XML files have less entropy and could be better compressed than HTTP header by Lempel-Ziv77. By simulating the attack on our models with different file types, we can determine the best ratio for each file type. If we could establish relationships between file entropy and randomness required to prevent compression side-channel attack, then we could maximizing security while minimizing compression loss.

B. PermuLZ Implementation We expect PermuLZ to resist the compression side-channel attack. When the file is being fragmented into blocks, the secret token is fragmented and the secret fragments are rearranged. Even if the adversary could retrieve all the secret fragments, to figure out the correct chronological fragment order has the same time complexity as a permutation of n elements, $O(n!)$. We also expect to reach a better compression ratio as our first scheme, Lempel-Ziv with weak cipher. The entropy of the plaintext may not vary as much. Shorter repeated strings could be found unchanged, but in different fragments, so file would still be compressed. However, longer repeated strings cannot be compressed as well as shorter strings, because they are fragmented, where as

these would compress less. Overall, PermuLZ provides compression and security. It could be used as a possible solution in the future as a compression algorithm used prior to encryption.

6 Acknowledgements

The work is supported by NSERC SPG grants.

References

1. Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.3 (Draft). RFC TBD, RFC Editor (July 2017)
2. Kelsey, J.: Compression and Information Leakage of Plaintext. In: Fast Software Encryption, 9th International Workshop, FSE 2002, Leuven, Belgium, February 4-6, 2002, Revised Papers. Volume 2365 of Lecture Notes in Computer Science., Springer (2002) 263–276
3. Duong, T., Rizzo, J.: The CRIME Attack. In: Presentation at ekoparty Security Conference. (2012)
4. Gluck, Y., Harris, N., Prado, A.: BREACH: Reviving the CRIME Attack. Unpublished manuscript (2013)
5. Ziv, J., Lempel, A.: Compression of Individual Sequences Via Variable-Rate Coding. IEEE Transactions on Information Theory **24**(5) (1978) 530–536
6. Rescorla, E.: HTTP Over TLS. RFC 2818, RFC Editor (May 2000)
7. Deutsch, P.: DEFLATE Compressed Data Format Specification version 1.3. RFC 1951, RFC Editor (May 1996)
8. Deutsch, P.: GZIP File Format Specification Version 4.3. RFC 1952, RFC Editor (May 1996)
9. Golomb, S.W.: SHIFT REGISTER SEQUENCES: Secure and Limited-Access Code Generators, Efficiency Code Generators, Prescribed Property Generators, Mathematical Models. World Scientific (2017)
10. Fan, X., Mandal, K., Gong, G.: Wg-8: A Lightweight Stream Cipher For Resource-Constrained Smart Devices. In: International Conference on Heterogeneous Networking for Quality, Reliability, Security and Robustness, Springer (2013) 617–632
11. Matsumoto, M., Nishimura, T.: Mersenne Twister: a 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator. ACM Transactions on Modeling and Computer Simulation (TOMACS) **8**(1) (1998) 3–30
12. Rijmen, V., Daemen, J.: Advanced Encryption Standard. Proceedings of Federal Information Processing Standards Publications, National Institute of Standards and Technology (2001) 19–22
13. Chen, L., Gong, G.: Communication System Security. CRC press (2012)
14. Gong, G., Youssef, A.M.: Cryptographic Properties of the Welch-Gong Transformation Sequence Generators. IEEE Transactions on Information Theory **48**(11) (2002) 2837–2846
15. Diffie, W., Hellman, M.: New Directions in Cryptography. IEEE transactions on Information Theory **22**(6) (1976) 644–654
16. Cover, T.M., Thomas, J.A.: Elements of Information Theory. John Wiley & Sons (2012)

17. Hollenbeck, S.: Transport Layer Security Protocol Compression Methods. RFC 3749, RFC Editor (May 2004)
18. Ziv, J., Lempel, A.: A Universal Algorithm for Sequential Data Compression. IEEE Transactions on Information Theory **23**(3) (1977) 337–343