Randomized Lempel-Ziv Compression for Anti-Compression Side-Channel Attacks

by

Meng Yang

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2018

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

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.

Acknowledgements

I would like to thank my supervisor Professor Guang Gong for her patience, guidance, and careful supervision of my thesis. Also, I benefited from the weekly seminars she hosted for her graduate students to discuss our research areas and review news on network security. I would also thank my co-supervisor Mark Aagaard for discussions with me on the simple and strong compression side-channel attacks. This project would not have been possible without the cipher server running on the powerful computer that Professor Gongs Comsec group made available for my research. I thank them for being so generous and supportive.

Dedication

I would like to thank my family, my mother and father, who are always supporting me.

Table of Contents

List of Tables

List of Figures

Abbreviations

CSCA Compression Side-Channel Attack 2, 4

SCSCA Strong Compression Side-Channel Attack 4

Chapter 1

Introduction

The internet is growing bigger and bigger everyday. Along with Facebook's new drone project, even more people will have access to the internet [28]. More information will travel on the network, and we want 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.

Internet communications use a protocol called TLS, Transport Layer Security, to compress and secure connections between server and client. TLS provides authentication with certificates, confidentiality with encryption, and integrity with message digest. For compression, TLS also supports DEFLATE [20] and LZS [16].

Currently, servers are using TLS version 1.2, which supports compression. However, combining compression with encrpytion is not secure. Adversaries can use the length of the compression output to retrieve content of the encrypted message. This type of attack is

called side-channel attack, where the attacker does not directly attack the encryption key. A side-channel attack that uses compression length is called Compression Side-Channel Attack (CSCA).

The compression side-channel attack was studied in a paper at the beginning of year 2002 [21]. In the paper, the authors explain how an adversary uses the output length of compressed encrypted text to decipher a secret within the message. In recent years, two compression side-channel attacks, CRIME (Compression Redundancy Infoleak Made Easy) in 2012 [13] and BREACH (Browser Reconnaissance and Exfiltration via Adaptive Compression of Hypertext) in 2013 [17] were demonstrated during computer security conferences. The two presenters, Juliano Rizzo and Thai Duong, successfully decrypted a secret HTTP message. TLS took a big hit and plans to shutdown all compression methods completely in its next version 1.3 [25].

1.1 TLS History

When the internet was first designed, only trusted nodes could connect to it and talk to each other. These nodes were connected to each other by wires, and spying was impossible. Nowadays, internet has gone wiress and any spy can listen to the conversation. Security was needed for the message's confidentiality and integrity.

Netscape designed in 1994 a protocol, SSL 1.0, to add security over the network. HTTP runs on top of this protocol, and it is called HTTPS [24]. The first two versions, SSL 1.0 and SSL 2.0, were implemented but never released due to security flaws. Only the third

version, SSL 3.0, got released in 1995 [15]. Four years later, in 1999, TLS 1.0 was published [9]. It is an upgraded protocol to replace SSL 3.0. Around 2011, Thai Duong and Juliano Rizzo presented BEAST (Browser Exploit Against SSL/TLS) that an adversary can use to retrieve an authentication token. It exploit a vulnerability in Cipher Block Chaining mode that TLS 1.0 is using. TLS 1.1 came out in April 2006 to address BEAST attack [10]. In August 2008, TLS 1.2 was released to implement authenticated encryption [11]. In June 2015, SSL 3.0 is deprecated [3].

Nowadays, most servers are using TLS 1.2. Attacks still appeared, thus forced the upgrade to TLS 1.3. TLS 1.3 is still in its draft phase. But, it has dropped compression among other insecure features [25].

1.2 Contribution

Removing compression will have a big impact on the network traffic due to uncompressed large packets. In this paper, we propose three schemes where compression could be renabled. The schemes modify the inputs and/or outputs of the compression algorithm. They are listed as followed:

(1) Weak Cipher With Lempel-Ziv

(2) Lempel-Ziv With Control

(3) Lempel-Ziv With Control And Feedback

We created and simulated the normal compression side-channel attack on our schemes to analyse the tradeoff between compression ratio and security: the complexity of the compression side-channel attack increased in exchange of a drop in compression ratio.

Additionally, we also simulated another version of CRIME/BREACH attack, called Strong Compression Side-Channel Attack (SCSCA). This attack was designed to break our schemes, but with a higher time complexity.

With the strong compression side-channel attack, our schemes are breached if the adversary was given a large amount of time. The time complexity is high and the attack is not feasible for an attacker to retrieve the secret given a short amount time.

Schemes are not limited only to repeating inputs or outputs around the compression algorithm. Another possible scheme is to fragment the message into smaller pieces then rearrange them before performing compression. This scheme also increases CSCA time complexity with a decrease in compression ratio as trade-off.

The rest of the thesis is organized as follows.

First, in Chapter 2, we summarize three papers that are related to our work. These three papers talk about compression side-channel attack, CRIME, and BREACH attacks.

In Chapter 3, we define the notation used in our work, and we also define terminologies that are substantial.

In Chapter 4, we describe TLS protocol and the Lempel-Ziv compression algorithm on which we run our simulation.

In Chapter 5, we introduce the compression side-channel attack and explain its mechanism. Also, we present our schemes to fight against this attack. Next, we present our

results of our models to show that our randomized Lempel-Ziv models can resist the attack.

In Chapter 6, we test the limit of our models: we extended the compression side-channel attack into a strong compression side-channel attack. After testing, our randomized models fail to resist the strong compression side-channel attack. However, the attack has a lengthy running time and is easily preventable.

In Chapter 7, we introduce a different type of Randomized Lempel-Ziv model that could defer CSCA or even SCSCA. This new model is based on permutation of same-length block of the plaintext before performing compression.

Lastly, we conclude with our findings in Chapter 8 and suggest ideas for future work.

Chapter 2

Literature Survey

The latest version of TLS provides online conversation's privacy by using encryption and decryption. Only the parties who know the correct key can encrypt and decrypt messages. The security level is usually based on how hard it is for an attacker to retrieve this key. But, instead of attacking the key direcly, attackers can use side-channel leakage to get into the conversation.

One of the earliest papers which addressed side-channel information leakage was published in 2002 [21]. Ten years after the publication of that paper, during a hackers conference in 2012, Juliano Rizzo and Thai Duong presented the CRIME attack that uses side channel leakage [13]. Then, a year later, the same two authors published another attack, BREACH, that also exploits HTTPS in a similar manner [17].

2.1 Compression Side-Channel Attacks

Information leakage from combinaton of compression with encryption was first mentioned in the paper titled "Compression and Information Leakage of Plaintext". With this attack, an attacker can extract the entire message under special conditions. Two required conditions must be met: compression must be done before encryption, and the attacker must know a small part of the plaintext. Overall, this paper explained and showed examples of how compression side-channel attack can be executed in many ways using chosen-plaintext attack. It also lists caveats and countermeasures, such as obscuring the output size. In 2012, the concepts from this paper were used to create actual attacks against HTTPS, which forced TLS 1.3 to remove compression. The two attacks are described below.

2.2 CRIME

Two researchers Juliano Rizzo and Thai Duong presented CRIME (Compression Ratio Info-leak Made Easy) at Ekoparty hacker conference in 2012 [13]. This attack uses compression redundancy as side-channel to decrypt cipher-text. The compression algorithm that is being exploited is called DEFLATE [7], which is used in TLS and SPDY protocols. DEFLATE is an efficient compression algorithm by combining LZ77 and Huffman coding [6].

During the conference, the presenters executed their compression side-channel attack on Yahoo Mail and Twitter, and managed to successfully retrieve secret tokens from encrypted messages. The secret tokens are authentication tokens, which an adversary could use to

impersonate another user.

The attack exploits the fact that DEFLATE compression is used before encryption, and also, the same token is being used during one session. The token is located as a cookie within HTTP header.

Normally, this cookie is encrypted and cannot be understood by an attacker, but with CRIME, the attacker can retrieve this cookie in a short amout of time. The typical way for an attacker to recover the cookie is to test all combination of the token, which has $O(256^n)$ complexity where 256 is the number of possible characters and n is the length of the string. With CRIME attack, the attacker can get the string in $O(256 \times n)$ by progressively guessing each character.

For this attack to work, the victim must first visit a malicious website and have his browser infected by CRIME agent. Then this agent will add forged text to the HTTP header, and make many requests to the web server. When these requests are compressed, encrypted then sent, the attacker measures the cipher-text length, and can retrieve the cookie. The conference demo lasted about 30 minutes and the presenters successfully retrieved the cookie from Twitter.

After discovery of this CRIME attack, TLS compression was disabled for most browsers and web servers to mitigate CRIME attack. Compression has been disabled on the transport layer, and caused increase of latency in the network because packets are not compressed. Network performance is reduced, but confidentiality is preserved.

2.3 BREACH

HTTPS was still not safe after CRIME was patched. In 2013, a new attack BREACH (Browser Reconnaissance and Exfiltration via Adaptive Compression of Hypertext) is mentioned during the Black Hat conference [17].

BREACH, presented by Angelo Prado, Neal Harris, and Yoel Gluck, is also a compression side-channel attack. Unlinke CRIME where the attacker examines the HTTP request headers, BREACH exploits compressed HTTP responses [23].

The three researchers launched the BREACH attack on a web server and were able to retrieve the authentication token hidden inside a HTTP header in less than 30 seconds. They also launched the attack on a Microsoft Outlook mail server and, without logging in, they were able to change the language and to add a new mailbox rule to their mailbox in less than 30 seconds as well.

For this attack to work, two conditions must be met. Firstly, HTTP compression must be enabled. Secondly, the HTTP response body must contain the cookie given by the matching HTTP request. Servers that have HTTP compression enabled and reflects the authentication token in its HTTP responses body are vulnerable to this attack. This compression side-channel attack showed that compression before encryption is vulnerable in more layers than just TLS protocol.

Different mitigation steps were suggested in this paper [17]. The easiest mitigation step is to disable HTTP compression for all servers. If HTTP compression is disabled, this would have a major impact on web application performance. Other mitigation steps were suggested as well, such as randomizing the length of the response, making secrets more

dynamic, padding the compressed message, monitoring the traffic, etc. . These methods are not generic: one method of mitigation would work better for a certain application than another. Thus, most web servers would be advised simply to remove HTTP compression.

The two compression side-channel attacks, BREACH and CRIME, show that having compression and encryption is vulnerable. By disabling all compression, network latency is increased even more. In this thesis, we demonstrate that compression side-channel attack can be defered by adding randomness to compression algorithm, however, it comes with an increase in time complexity of the compression, and decrease in compression ratio.

Chapter 3

Preliminaries

In this chapter, we describe the Lempel-Ziv compression algorithm family. We also present the pseudo-random sequence generator, and the encryption algorithms that will be used in our proposed models.

3.1 Compression Algorithms

We use the following notations throughout of the thesis:

$A||B$ 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.

3.1.1 Lempel-Ziv 77 and 78

Lempel-Ziv 77 The Lempel-Ziv compression algorithm, shown in Algorithm 1, was created by Abraham Lempel and Jacob Ziv. They published their first version in 1977, called Lempel-Ziv77, also known as LZ77. This compression algorithm focus on replacing repeated patterns with references to locations where they happened before. The reference is a length-distance tuple, which indicates the location of the pattern and its length. The reference tuple also contains the next character after the pattern.

Algorithm 1 LEMPEL-ZIV77($file$)

```
1: current_position ← start of file
2: outputs_list ← empty list
3: while current_position not reach end of file do
4:     move sliding_window
5:     longest_str ← find longest matching string for current_pos in sliding_window
6:     pos ← relative starting position of longest_str from current_pos
7:     len ← length of longest_str
8:     next_char ← next character after longest_str in sliding_window
9:     append (pos, len, next_char) to outputs_list
10:    current_position ← current_position + j
11: end while
12: return outputs_list
```

LZ77 Example If we run the LZ77 algorithm on "abcabcd", we would get "abc(0,3,'d')" since the second "abc" is repeated, and it gets replaced by "(0,3,'d')", where the first number, 0, is the location, and the second number, 3, is the length. This latter tuple would get converted to two bytes. Thus, for the text "abcabcd", LZ77 would give a compression ratio of $\frac{5}{7}$, since the original text has length of 7 bytes, and LZ77 outputs has length 5 bytes. When decompressing, two numbers, 0 and 3, inside the tuple will be used.

The second number, 3, is the length of the repeated pattern. The tuple "(0,3,'d')" gets replaced by "abcd".

LZ78 In the year that followed, 1978, the same two authors published their second compression algorithm named Lempel-Ziv78, LZ78, which is an extension of LZ77 [31]. 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 with LZ77. This compression algorithm gives an encoding to each character. The most frequently used characters get an encoding of a smaller length, and the least frequently used characters get an encoding of a larger length.

DEFLATE and gzip Both DEFLATE and gzip are popular compression algorithms used by HTTP. When the compression is done using DEFLATE algorithm, the plaintext is first compressed by LZ77, then compressed again by Huffman encoding. The gzip compression algorithm is the same as DEFLATE, but it adds a CRC-32 checksum to the message after compression.

3.2 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 [18].

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 $2^{33.32}$ [14].

Mersenne Twister Mersenne Twister generator was created in year 1997 by Makoto Matsumoto and Takuji Nishimura [22]. 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.

3.3 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 [26].

Chapter 4

Implementation of Components

In this chapter, we explain in detail the ways we implemented TLS/SSL protocol on which we simulated the side-channel. We also implemented Lempel-Ziv77 compression following its algorithm since we are using it to compress messages and Python 2.7 does not provide the library. Additionally, we implemented a hardware version of Lempel-Ziv77 compression algorithm which uses hash tables for quicker reference lookups.

4.1 TLS/SSL Implementation

We implemented in C++ our own TLS/SSL protocol as a target for our compression side-channel attack simulations. This protocol is built on top of the native TCP socket package. So, we first implemented TCP as a C++ class that wraps around the native TCP implementation to simulate a packet sniffer: After a TCP connection, messages that are sent between the server and client are saved to a log file with its length.

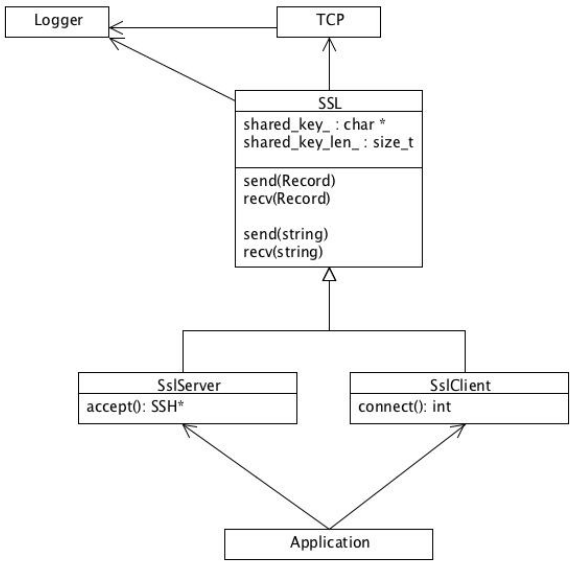


Figure 4.1: Diagram of Lempel-Ziv With Control and Feedback

We built our version of TLS on top of our TCP class. The UML, Figure 4.1, shows the relation between the classes. On top, the TCP class is a packet sniffer: it saves the sent or received messages using the Logger class, which saves those messages inside a log file. The server and client are part of Applications. Both of them are using TLS/SSL class to encrypt their messages before sending through TCP class, and decrypt it on receiving. The AES algorithms are used to encrypt and decrypt the messages. We use CryptoPP library for cryptography functions, such as encryption and decryption. AES uses symmetric keys, which means that both server and client have a copy of the same key. We implemented two key exchange protocol to establish the shared key: Diffie Hellman in ephemeral mode, and Rivest-Shamir-Adleman (RSA).

For the Diffie-Hellman protocol, DHE, the server and client establishes a shared key

together. The server first sends a large prime number p and a base g that is used to compute the shared multiplicative group. It computes its private key s, and public key g^s, and sends the latter to the client. After receiving the information, the client computes the multiplicative group, then computes its private key c, and its public key, g^c, which gets sent to the server. Then, both server and client use the private key and the public key of the other party to compute the shared key, $(g^s)^c$ or $(g^c)^s$.

For the Rivest-Shamir-Adleman protocol, RSA, the client picks shared encryption key. To securely send it to the server, the server sends a public key to the client. The client then uses the public key to encrypt the picked shared key using RSA encryption algorithm. Afterwards, the client sends it to the server, who decrypts it and uses it for secure communications.

Our TLS/SSL implementation code can be found in Appendix A. This implementation allows us to host a server, and have a client connect to it, and perform secure communications. The client sends HTTP headers to the server through this secured tunnel. However, the message length is not hidden, since we can retrieve each length in the log file. By retrieving the lengths with our logger TLS/SSL and by being able to append to the client messages, we can mimic an attacker performing the compression side-channel attack.

4.2 Lempel-Ziv77 Implementation

Our Lempel-Ziv77 implementation is done using Python 2.7. We implement our own version because Python 2.7 does not provide the algorithm, it only provides DEFLATE

which is a combination of Lempel-Ziv and Huffman coding. We implemented software version of the algorithm, Algorithm 1, and hardware version of the algorithm described below. The code for both versions can be found in Appendix B.

4.2.1 Lempel-Ziv77 Original Version

The Lempel-Ziv77 compression algorithm is implemented step-by-step following the Lempel-Ziv77 algorithm, Algorithm 1: for each character and the ones that follows, we try to find the longest match within the sliding window, and if we find one, we replace the matched string by a reference.

4.2.2 Lempel-Ziv Dictionary Version

We also implemented a hardware version of Lempel-Ziv77 compression algorithm [1]. In the software version, for each character and the characters that follows, the algorithm searches for matching strings inside the sliding window. This version uses multiple hash tables, 8 in our case, for quick lookups following these steps below.

1. At the beginning, 8 empty dictionaries or hash tables are created. These hash tables values are strings alongside their location in the file. The key is computed with a hash function explained below. After the initialization is done, the compression can begin.

2. The input file that will be compressed is being read 8 character at a time. These 8 characters are saved inside a working vector, called the shift vector. This shift vector

has length of 16. When 8 characters are read, they are put at the end of the vector. They will shift to the front of the vector when the next 8 are read.

3. From the shift vector, the 8 hashes are computed. Each of the first 8 characters has its own hash values. The hash values are calculated by the ASCII value of the character together with the ASCII values of the three that follows under the hashing function below, Equation (4.1). Those hash values will be used to look up if any matching strings are present in the hashing tables for string starting at each of the 8 positions.

$$\begin{aligned} hash(curr\_char(i)) =& (curr\_char(i) \ll 2) \oplus (curr\_char(i+1) \ll 1) \\ & \oplus curr\_char(i+2) \oplus curr\_char(i+3) \end{aligned} \tag{4.1}$$

4. Once the hashes are computed, each of the 8 characters looks at the value at its hash in each of the 8 dictionaries, and tries to find a matching string. So, for the first 8 characters, the algorithm performs 64 look ups, and returns at most 8 matching strings.

5. After retrieving the values from the dictionaries, the algorithm will update 8 entries, one for each dictionary. For each of the first 8 character, the substring of length 8 starting at that character is saved inside one of the dictionaries. The first substring will be saved in the first dictionary, the second substring will be saved in the second dictionary, and so on.

6. When all the matching strings are gathered, a selection process will choose which matching string to keep. The selection algorithm works almost the same way as a

scheduling algorithm. We first define some terms.

- A reach of a matching string is defined to be the position where a matching string ends.
- A first valid position is defined to be the end of the last matching string of the previous shift instance.

The selection process has six steps, listed below.

(a) Matching strings that are shorter than 3 characters are dropped.

(b) Matching strings that conflict with the first valid position are dropped.

(c) All the matching string's reach are computed.

(d) The last matching string is always kept, so we can compute the next first valid position.

(e) For two matching strings that have the same reach, the smaller one is dropped.

(f) Lastly, all the matching strings that overlaps with the last one are dropped.

This algorithm is not optimal: it drops potential optimal matches, and the dictionary needs better keeping algorithms. For example, if the last matching string is three characters long, but it overlaps with another that is 6 characters, the longer matching string is dropped. Also, if the dictionary contains a longer string, it could be overwritten by a shorter one since the hashing function only takes into account the first four characters of the string. However, this algorithm is fast since it exchanges storage complexity for time complexity: instead of doing a linear scan inside the sliding window, it uses a hash lookup which is constant time.

Chapter 5

Randomized LZ Compression to Resist Compression Side-Channel Attacks

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

5.1 Compression Side-Channel Attack

In this section, we present the algorithm of the compression side-channel attack, along with an example, and we also prove the leakage of combining compression and encryption.

5.1.1 Analysis of Compression Side-Channel Attack

The following algorithm, Algorithm 2, shows the execution steps of a compression side-channel attack.

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** $guess\_secret$

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 concatination 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.

5.1.2 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 alpabets, 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.

5.1.3 Analysis of Lempel-Ziv Against Compression Side-Channel Attack

In this subsection, we 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.

We represent the header together with the forged string as S shown below:

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

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 5.1.1. *$l_{\hat{x} \neq s_t} = l_{\hat{x}=s_t} + 1$, 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 conclude, the length of the compressed file with incorrect guess is indeed longer than the file with correct guess: $l_{\hat{x}\neq s_t} > l_{\hat{x}=s_t}$.

□

5.2 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.

5.2.1 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.

5.2.2 Model Elements

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

Pseudorandom Sequence Generator

Our three schemes below use the same pseudorandom sequence generator, PRSG, which generates a sequence of 0's and 1's. This 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 on Security of Mersene Twister Mersenne twister is not cryptographic secure [22]. However, we use it because of its fast speed. We also do not need it to be cryptographic secure since we are adding randomness to our compression schemes, which is different from encryption.

Weighted Generator We implemented a weighted pseudo-random number generator that generates an uneven number of 0s and 1s. Specifially, 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 varie 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. The weak stream cipher is being used. 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.

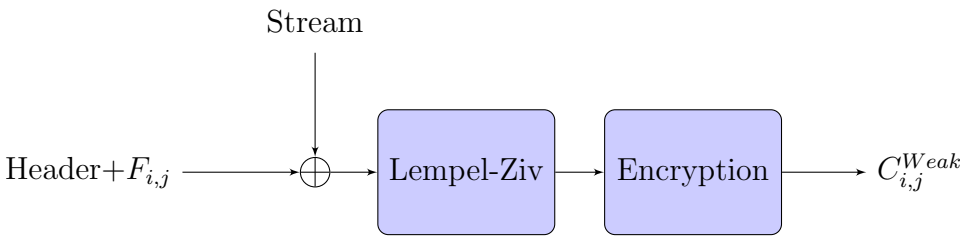


Figure 5.1: Diagram of Weak Cipher with Lempel-Ziv

5.2.3 Randomized Lempel-Ziv Models

By adding a pseudo-random number generator and the Lempel-Ziv77, 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 5.1. 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.

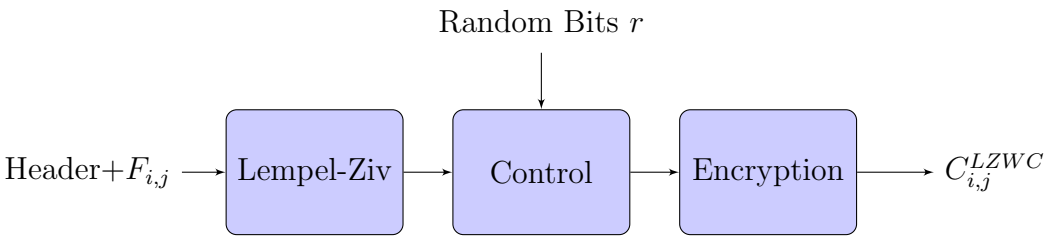


Figure 5.2: Diagram of Lempel-Ziv With Control

Scheme 2: Lempel-Ziv With Control

The idea behind Lempel-Ziv With Control, LZWC, Figure 5.2, 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.

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 5.3. 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.

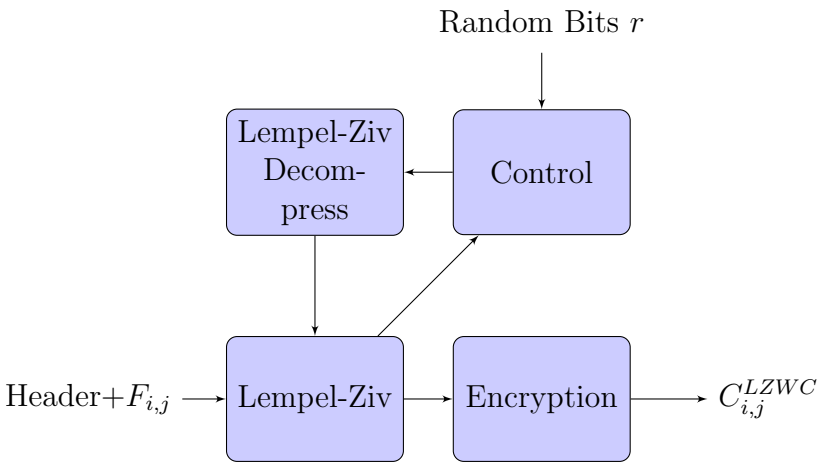


Figure 5.3: Diagram of Lempel-Ziv With Control and Feedback

5.3 Simulation Results for Randomized LZ Models

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

5.3.1 Testing Environment

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

Processors	80
CPU	Intel(R) Xeon(R) CPU 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.

5.3.2 Compression Side-Channel Attack on original LZ77

We ran the compression side-channel attack, i.e. Algorithm 2, on the original compression algorithm LZ77, i.e. Algorithm 1. Just as described above, a forged token with a guessing character was added to the plain-text, which goes through LZ77 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.

5.3.3 Compression Side-Channel Attack on LZ77 with Randomizations

The same attack was tested on plain-texts which went through LZ77 compression algorithm with addition of randomizations. 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 LZ77 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 LZWFC

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.

Chapter 6

Strong Compression Side-Channel Attack

The normal compression side-channel attack that BREACH and CRIME implement can be prevented easily if the victim employs one of the three randomization methods to Lempel-Ziv for compression. The randomness masks the length of the compressed files, which would be deterministic without any randomness involved. This way, the attacker cannot know for certain that the shortest encrypted message has the correct guess. However, the methods cannot provide complete immunity against the attack described in this section.

6.1 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.

Algorithm 3 STRONG_COMPRESSION_SIDE_CHANNEL_ATTACK($http\_request$)

```
1: header ← HTTP header from http_request
2: known_prefix ← secret's prefix from adversary
3: guess_secret ← empty string of length L set by adversary
4: for i = 0 to i = secret_length do
5:    L ← empty list
6:    for each ζ ∈ Σ do
7:       guess_secret[i] ← ζ
8:       g_ζ ← known_prefix||guess_secret
9:       M_ζ ← header||g_ζ
10:      T ← empty list
11:      for j = 0 to j = K do
12:         C_ζ ← Enc(LZ77(M_ζ))
13:         C_ζ gets sent over network
14:         l_ζ ← length(C_ζ)
15:         T.insert(l_ζ)
16:      end for
17:      L.insert(avg(T))
18:   end for
19:   guess_secret[i] ← {ζ|ζ = argmin_ζ{l_ζ ∈ L}}
20: end for
21: return guess_secret
```

The strong 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 guessed character, shown in line 19.

6.2 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.

6.2.1 Strong Compression Side-Channel Attack on Randomized LZ Models

The modified LZ77 with randomizations 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 6.1, 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 6.2, 300 requests were enough to decode one character in most cases.

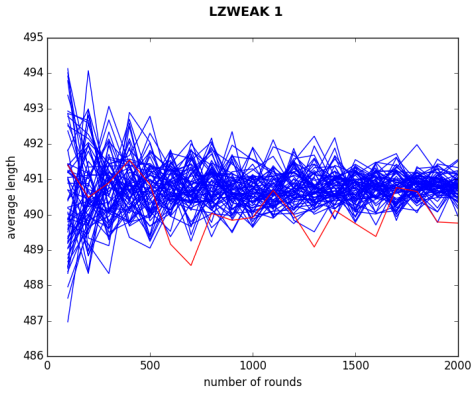
Strong Compression Side-Channel Attack on LZWCF

The testing results of LZWCF are displayed in Figure 6.3. 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.

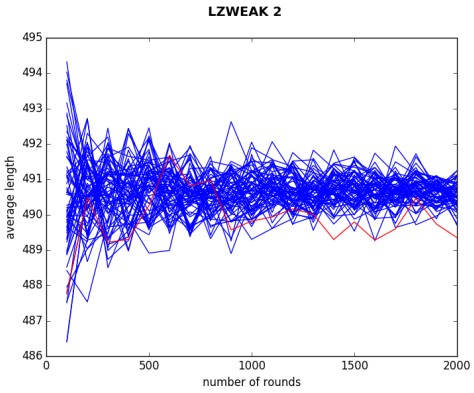
6.2.2 Compression Rate

In Table 6.1, we present the different compression ratio from our randomized LZ models.

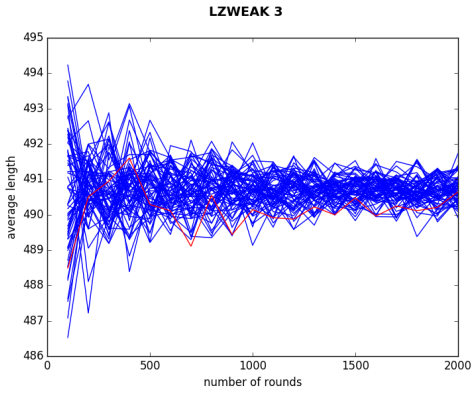
From the table, we see that our schemes have $(.879 - .792)/.792 = 11.0\%$, $(.881 - .792)/.792 = 11.2\%$, $(.943 - .792)/.792 = 19.1\%$ increase in the compressed size.



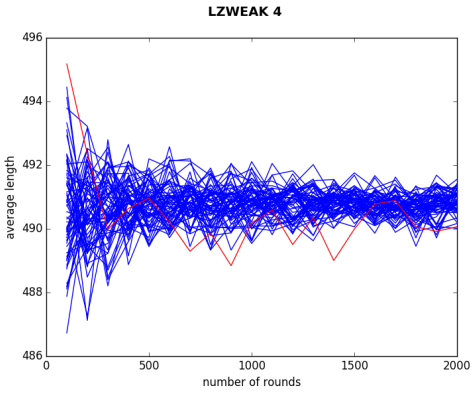
(a) Token 1



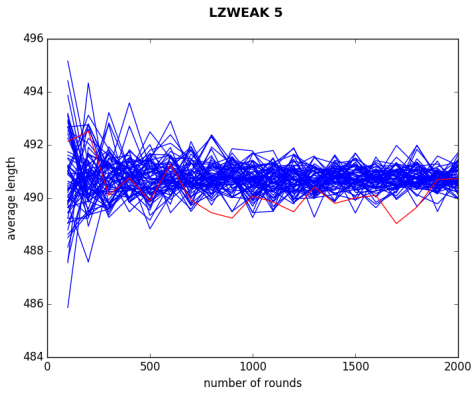
(b) Token 2



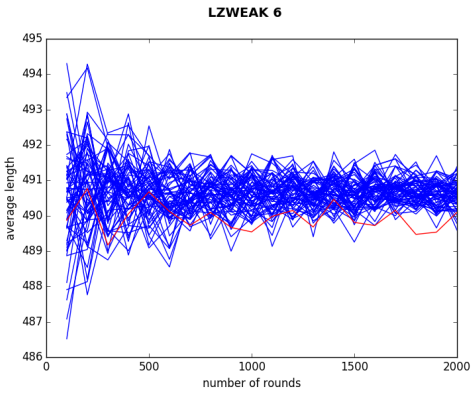
(c) Token 3



(d) Token 4

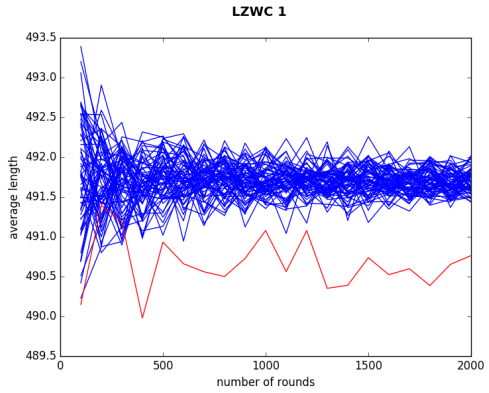


(e) Token 5

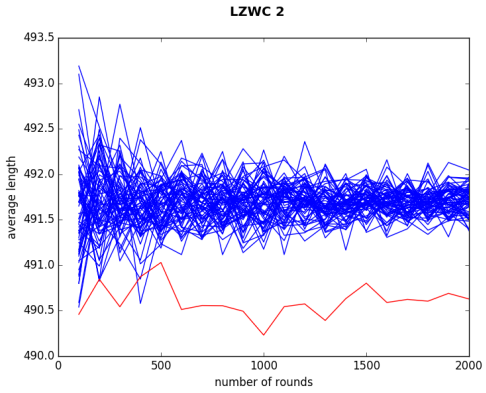


(f) Token 6

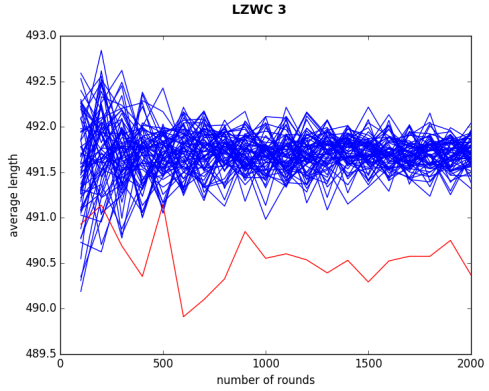
Figure 6.1: Results of SCSCA on LZ with Weak Cipher



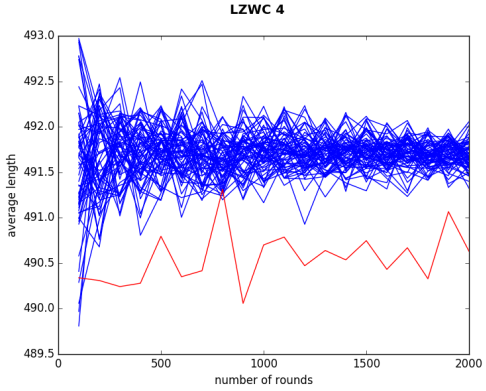
(a) Token 1



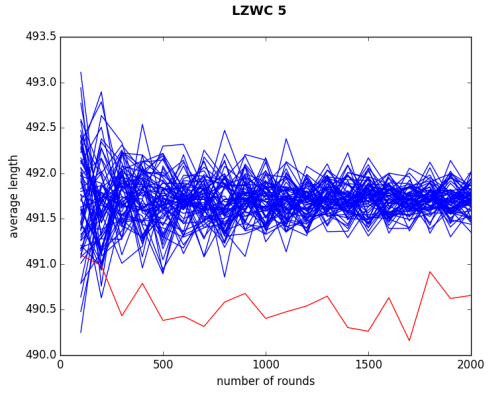
(b) Token 2



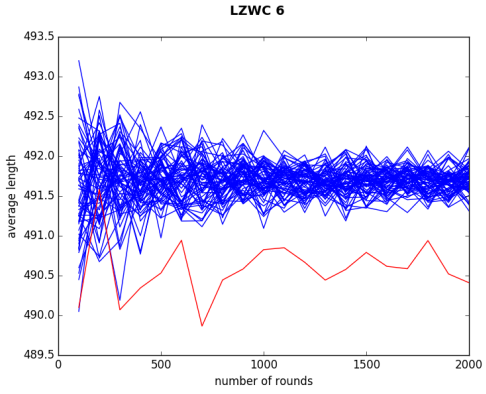
(c) Token 3



(d) Token 4

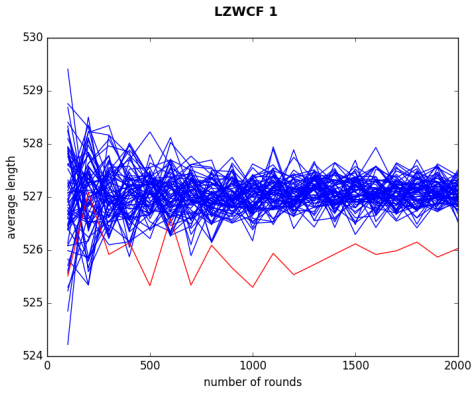


(e) Token 5

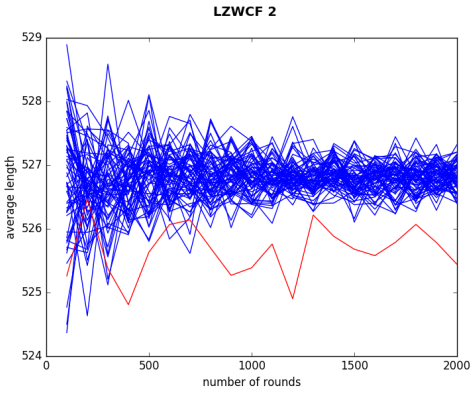


(f) Token 6

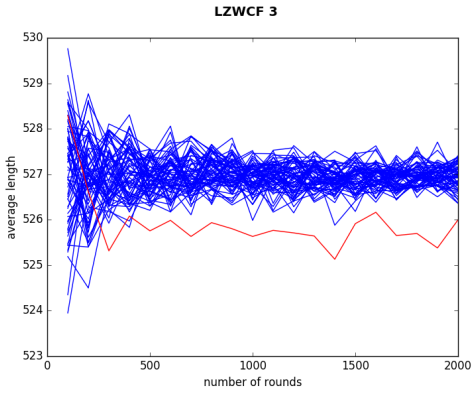
Figure 6.2: Results of SCSCA on LZWC



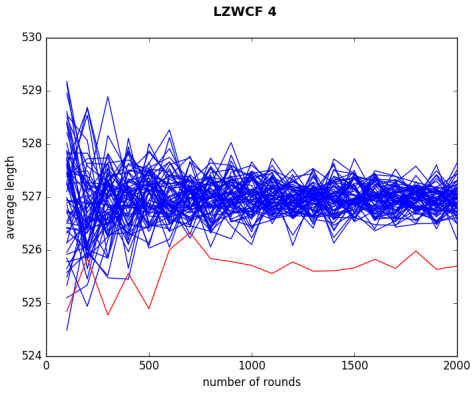
(a) Token 1



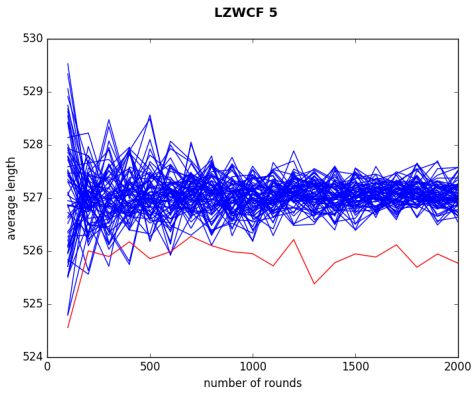
(b) Token 2



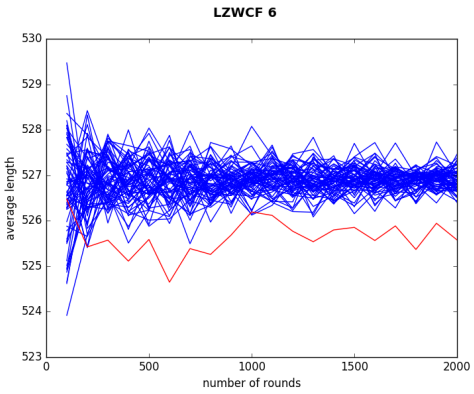
(c) Token 3



(d) Token 4



(e) Token 5



(f) Token 6

Figure 6.3: Results of SCSCA on LZWCF

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

Table 6.1: Compression Ratio from Each Schemes

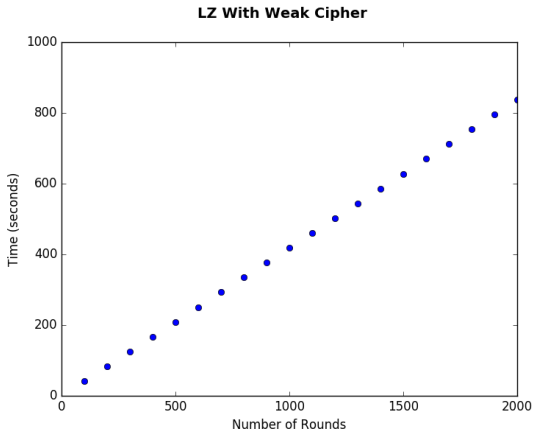
Our three randomized LZ model schemes prevent normal compression side-channel attack, however, the size of the compressed message is increased as the trade-off.

6.2.3 Timing

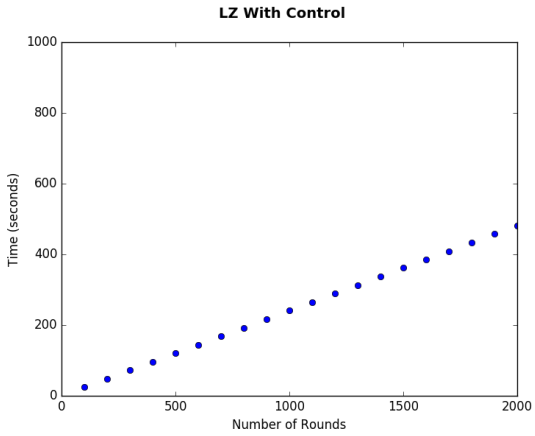
The downside of strong compression side-channel attack is that it requires a long time to execute. In this section, we investigate the time it take to resolve one secret of length 228.

During testing, the time to guess one character using iterations is also recorded. The Figure 6.4 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.

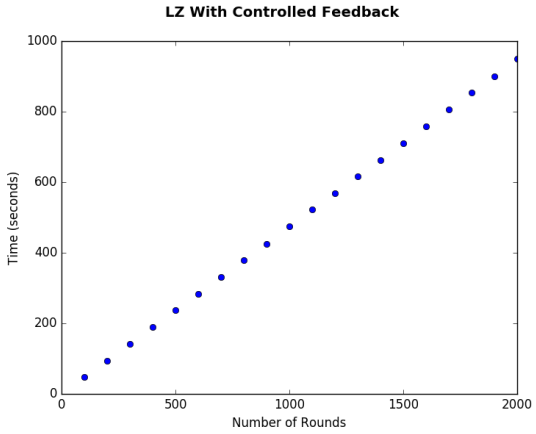
Previously, we have seen that LZWeak requires more than 2000 requests, so to decode one secret, the attack needs $837 \times 228 = 190836\mathsf{seconds} \approx 53\mathsf{hours}$ to decode one secret. For LZWC, 300 iterations take 72 seconds to execute, so the total time it takes is $72 \times 228 = 16416\mathsf{seconds} \approx 5\mathsf{hours}$. For LZWCF, 500 iterations take 237 seconds to execute, so the



(a) LZWeak Average Time



(b) LZWC Average Time



(c) LZWCF Average Time

Figure 6.4: Timing Graphs of SCSCA on Randomized LZ Models

total time taken would be $237 \times 228 = 54036\text{seconds} \approx 15\text{hours}$. These hours are only the time it takes to compress, more hours will be added for encryption and transmission.

6.2.4 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.

Chapter 7

PermuLZ

The principle idea of the three models described previously is to manipulate the output lengths so that they are not deterministic from the compression algorithm. So far, the schemes are only manipulate outputs without changing the inputs. Only Lempel-Ziv with Weak Cipher model changes the input of the file, but it changes the inputs only by a little.

7.1 PermuLZ Description

The main idea of PermuLZ is to cut the input file into many small blocks, move them around, and then pass the file to the compression algorith.

Each run will have a different permutation, giving a different ciphertext length. Having undeterministic ciphertext lengths hinders the adversary performing a compression side-channel attack. The proof below explains how this generates undeterministic lengths.

Algorithm 4 PERMULZ($input\_file, permutation\_key$)

1: $B, P \leftarrow$ empty list of strings
2: $B \leftarrow split(input\_file)$
3: $P \leftarrow permutate(B, permutation\_key)$
4: $C \leftarrow Enc(LZ(P))$
5: **return** C

Proof. Let S be the input file and B be list of equal-sized blocks splits the string S.

$$S = s_0 s_1 s_3 s_4 \cdots s_k$$

$$B = b_1 b_2 \cdots b_l$$

Assume s_i to s_j is the secret, when splitting the block, the cut will happen in one or two places inside the secret. Assuming the cut only happens once at position s_d. Let B_{s1} and B_{s2} be the two blocks that cointains the two parts of secret that get cut.

$$B_{s1} = s_i s_{i+1} \cdots s_d$$

$$B_{s2} = s_{d+1} s_{d+2} \cdots s_j$$

If permutation is equally distributed, B_{s1} and B_{s2} won't be next to each other, since B_{s1} has equal chance of being placed in front of any other blocks. The first letter of other blocks are not necessarily s_{d+1}. When compression side-channel attack guessed s_d, it will try to guess s_{d+1}. But the algorithm will make the wrong guess, since other letters are equally likely to happen because they are from other blocks. Thus, in theory, this can resist strong compression sidechannel attack.

□

Chapter 8

Conclusion and Future Work

In this chapter, we restate the conclusion of our thesis and highlight the potential improvement that is necessary for future growth.

8.1 Conclusion

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

We proposed three models that add randomization around Lempel-Ziv77 to resist to compression side-channel attacks. The three models proposed are: Lempel-Ziv with Weak Cipher, Lempel-Ziv with Control, and Lempel-Ziv with Controlled Feedback. Those models encourage secure communication to re-enable compression inside TLS/SSL, since they

could be implemented in the server and client: hence, an additional key is needed to synchronize the randomizations, and a small amount of time overhead is added compared to Lempel-Ziv77.

The main idea of the models is to confuse the adversary by removing the deterministic property of the lengths from outputs of Lempel-Ziv compression. The first model, Lempel-Ziv with Weak Cipher, uses a weak cipher to change some bytes of the input, then the outputs gets compressed using Lempel-Ziv compression algorithm. The second model, Lempel-Ziv with Control, performs compression first, then it repeats the outputs controlled by a pseudorandom sequence generator. The third model, Lempel-Ziv with Controlled Feedback, repeats outputs just like the second model, but it then uncompress it, and compresses again. The three models deter the compression side-channel attack by modifying the deterministic feature of compression length. When the attacking program is guessing the correct character, it corrolates the compressed-then-encrypted message with the shortest length to the correct guess. By having compression length as undeterministic, adversary could not perform compression side-channel attack, because the correct guess does not correspond to the message with the shortest length.

Using our internal server, we first simulated a compression side-channel attack against a server that uses Lempel-Ziv77 as compression algorithm. We managed to successfully retrieve the secret within a HTTPS Header. Afterwards, we tested the same attack using our three models, and all three models successfully hide the secret, thus preventing compression side-channel attack.

To test the limit of our schemes, we created a better version of the compression side-

channel attack, called strong compression side-channel attack. This attack requires the adversary to make multiple queries for one guess. From those queries, the adversary can collect all the undeterministic lengths and calculates the mean of all the lengths. With the means, the adversary can deduce that the character with lowest mean must correspond to the actual character of the secret.

Our three schemes failed to resist the stronger compression side-channel attack if the attacker is allowed to make a large number of queries. However, they can still be used in practice since the attack takes a long time to decode the secret within the header. 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. Thus, if the secret expired before the given time, then the attack fails because the recovered token is already expired.

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.

8.2 Future Work

8.2.1 Entropy Analysis

We fixed our bit-stream generator to generate 1's ten percent of the time, and 0 ninty 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.

8.2.2 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 repated strings cannot be compressed as well as shorter strings, because they are fragmented, where as these would compress less. Overall, Per-

muLZ provides compression and security. It could be used as a possible solution in the future as a compression algorithm used prior to encryption.

References

[1] Mohamed S Abdelfattah, Andrei Hagiescu, and Deshanand Singh. Gzip on a Chip: High Performance Lossless Data Compression on FPGAs using OpenCL. In *Proceedings of the International Workshop on OpenCL 2013 & 2014*, page 4. ACM, 2014.

[2] Elaine B Barker and John Michael Kelsey. *Recommendation For Random Number Generation Using Deterministic Random Bit Generators (Revised)*. US Department of Commerce, Technology Administration, National Institute of Standards and Technology, Computer Security Division, Information Technology Laboratory, 2007.

[3] R. Barnes, M. Thomson, A. Pironti, and A. Langley. Deprecating Secure Sockets Layer Version 3.0. RFC 7568, RFC Editor, June 2015.

[4] Mike Belshe and Roberto Peon. SPDY Protocol, February 2012. Available at http://tools.ietf.org/html/draft-mbelshe-httpbis-spdy-00.

[5] Lidong Chen and Guang Gong. *Communication System Security*. CRC press, 2012.

[6] Thomas M Cover and Joy A Thomas. *Elements of Information Theory*. John Wiley & Sons, 2012.

[7] P. Deutsch. DEFLATE Compressed Data Format Specification version 1.3. RFC 1951, RFC Editor, May 1996.

[8] P. Deutsch. GZIP File Format Specification Version 4.3. RFC 1952, RFC Editor, May 1996.

[9] T. Dierks and C. Allen. The TLS Protocol Version 1.0. RFC 2246, RFC Editor, January 1999.

[10] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.1. RFC 4346, RFC Editor, April 2006.

[11] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, RFC Editor, August 2008.

[12] Whitfield Diffie and Martin Hellman. New Directions in Cryptography. *IEEE transactions on Information Theory*, 22(6):644–654, 1976.

[13] Thai Duong and Juliano Rizzo. The CRIME Attack. In *Presentation at ekoparty Security Conference*, 2012.

[14] Xinxin Fan, Kalikinkar Mandal, and Guang Gong. Wg-8: A Lightweight Stream Cipher For Resource-Constrained Smart Devices. In *International Conference on Heterogeneous Networking for Quality, Reliability, Security and Robustness*, pages 617–632. Springer, 2013.

[15] A. Freier, P. Karlton, and P. Kocher. The Secure Sockets Layer (SSL) Protocol Version 3.0. RFC 6101, RFC Editor, August 2011.

[16] R. Friend. Transport Layer Security (TLS) Protocol Compression Using Lempel-Ziv-Stac (LZS). RFC 3943, RFC Editor, November 2004.

[17] Yoel Gluck, Neal Harris, and Angelo Prado. BREACH: Reviving the CRIME Attack. *Unpublished manuscript*, 2013.

[18] Solomon W Golomb. *SHIFT REGISTER SEQUENCES: Secure and Limited-Access Code Generators, Efficiency Code Generators, Prescribed Property Generators, Mathematical Models*. World Scientific, 2017.

[19] Guang Gong and Amr M Youssef. Cryptographic Properties of the Welch-Gong Transformation Sequence Generators. *IEEE Transactions on Information Theory*, 48(11):2837–2846, 2002.

[20] S. Hollenbeck. Transport Layer Security Protocol Compression Methods. RFC 3749, RFC Editor, May 2004.

[21] John Kelsey. 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*, pages 263–276. Springer, 2002.

[22] Makoto Matsumoto and Takuji Nishimura. Mersenne Twister: a 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):3–30, 1998.

[23] Angelo Prado, Neal Harris, and Yoel Gluck. SSL, Gone in 30 Seconds. *Breach attack*, 2013.

[24] E. Rescorla. HTTP Over TLS. RFC 2818, RFC Editor, May 2000.

[25] E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3 (Draft). RFC TBD, RFC Editor, July 2017.

[26] Vincent Rijmen and Joan Daemen. Advanced Encryption Standard. *Proceedings of Federal Information Processing Standards Publications, National Institute of Standards and Technology*, pages 19–22, 2001.

[27] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[28] TheGuardian. Facebook Drone That Could Bring Global Internet Access Completes Test Flight., July 2017. Available at https://www.theguardian.com/technology/2017/jul/02/facebook-drone-aquila-internet-test-flight-arizona.

[29] Terry A. Welch. A Technique for High-Performance Data Compression. *Computer*, 6(17):8–19, 1984.

[30] Jacob Ziv and Abraham Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[31] Jacob Ziv and Abraham Lempel. Compression of Individual Sequences Via Variable-Rate Coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.

APPENDICES

Appendix A

C++ Code for TLS/SSL Protocol

A.1 TLS/SSL Client Connection Code

The following function is used by the client to connect to a server and do handshake to establish a shared encryption key.

```
int SslClient::connect(
  const std::string &ip, int port,
  uint16_t cxntype)
{

  // connect
  if ( this->tcp_->socket_connect(ip, port) != 0 ) {
    cerr << "Couldn't connect" << endl;
    return -1;
  }

  // send client hello
```

```
Record client_hello_record;
client_hello_record.hdr.type = REC_HANDSHAKE;
client_hello_record.hdr.version = VER_99;
client_hello_record.hdr.length =
  sizeof(HS_CLIENT_HELLO) + sizeof(cxntype);

char* data = (char*)malloc(client_hello_record.hdr.length*sizeof(char));
memcpy(data, &HS_CLIENT_HELLO, sizeof(HS_CLIENT_HELLO));
memcpy( &(data[sizeof(HS_CLIENT_HELLO)]), &cxntype, sizeof(cxntype));

client_hello_record.data = data;

if ( this->send(client_hello_record) != 0 ) {
  cerr << "Couldn't send client hello" << endl;
  return -1;
}

free(data);

switch ( cxntype ) {
  case KE_DH:
  case KE_DHE: {
    // wait for server hello
    Record server_hello_record;
    if ( this->recv(&server_hello_record) != 0 ) {
      cerr << "Couldn't receive server hello" << endl;
      return -1;
    }

    if ( server_hello_record.hdr.type != REC_HANDSHAKE ) {
      cerr << "Not a handshake message." << endl;
      return -1;
    }

    uint8_t handshake_type;
    memcpy(&handshake_type, server_hello_record.data,
           sizeof(handshake_type));
    if ( handshake_type != HS_SERVER_HELLO ) {
```

```
  cerr << "Not a server hello message."
       << server_hello_record.data << endl;
  return -1;
}

string pqg_gs( &(server_hello_record.data[sizeof(handshake_type)]),
               server_hello_record.hdr.length
               - sizeof(handshake_type));
free(server_hello_record.data);

string p;
string q;
string g;
string gs;

size_t found_p;
found_p = pqg_gs.find("x", 0);
if ( found_p == string::npos ) {
  cerr << "Couldn't find p." << endl;
  return -1;
}
p = pqg_gs.substr(0, found_p);
// cout << "P:" << p << endl;

size_t found_q;
found_q = pqg_gs.find("x", found_p+1);
if ( found_q == string::npos ) {
  cerr << "Couldn't find q." << endl;
  return -1;
}
q = pqg_gs.substr(found_p+1, found_q - found_p - 1);
// cout << "Q:" << q << endl;

size_t found_g;
found_g = pqg_gs.find("x", found_q+1);
if ( found_g == string::npos ) {
  cerr << "Couldn't find g." << endl;
  return -1;
```

```
}
g = pqg_gs.substr(found_q+1, found_g - found_q - 1);
// cout << "G:" << g << endl;

gs = pqg_gs.substr(found_g+1);
// cout << "G^s:" << gs << endl;

CryptoPP::Integer dh_p(p.c_str());
CryptoPP::Integer dh_q(q.c_str());
CryptoPP::Integer dh_g(g.c_str());

CryptoPP::AutoSeededRandomPool rnd;
CryptoPP::DH dh_client;
dh_client.AccessGroupParameters().Initialize(dh_p, dh_q, dh_g);
if ( !dh_client.GetGroupParameters().ValidateGroup(rnd, 3) ) {
  cerr << "Failed to generate dh_client." << endl;
  return -1;
}

// generate client gc
CryptoPP::SecByteBlock priv_client(dh_client.PrivateKeyLength());
CryptoPP::SecByteBlock pub_client(dh_client.PublicKeyLength());
dh_client.GenerateKeyPair(rnd, priv_client, pub_client);

// send client key exchange gc
Record client_ke_record;
client_ke_record.hdr.type = REC_HANDSHAKE;
client_ke_record.hdr.version = VER_99;
client_ke_record.hdr.length = sizeof(HS_CLIENT_KEY_EXCHANGE) +
                              dh_client.PublicKeyLength();

char* ke_data = (char*)malloc(
                client_ke_record.hdr.length*sizeof(char));
memcpy( ke_data,
        &HS_CLIENT_KEY_EXCHANGE, sizeof(HS_CLIENT_KEY_EXCHANGE));
memcpy( &(ke_data[sizeof(HS_CLIENT_KEY_EXCHANGE)]),
        pub_client.BytePtr(), pub_client.SizeInBytes());
```

```
    client_ke_record.data = ke_data;

    if ( this->send(client_ke_record) != 0 ) {
        cerr << "Couldn't send client key exchange g^c." << endl;
        return -1;
    }

    free(ke_data);

    // compute key
    CryptoPP::SecByteBlock pub_server((const unsigned char*)gs.c_str(),
                                      dh_client.PublicKeyLength());
    CryptoPP::SecByteBlock dh_key(dh_client.AgreedValueLength());

    if( !dh_client.Agree(dh_key, priv_client, pub_server) ) {
        cerr << "Client: Failed to compute shared key." << endl;
        return -1;
    }

    // compute SHA 256 of key
    CryptoPP::SecByteBlock shared_key(CryptoPP::SHA256::DIGESTSIZE);
    CryptoPP::SHA256().CalculateDigest(shared_key,
                                       dh_key, dh_key.size());

    // save key inside client connection
    set_shared_key(shared_key.BytePtr(), shared_key.SizeInBytes());

    // log shared keys
    this->logger_->log_raw("DH Key:");
    this->logger_->log_raw((const char*)dh_key.BytePtr(),
                           dh_key.SizeInBytes());
    this->logger_->log_raw("Shared Key:");
    this->logger_->log_raw((char *)this->shared_key_,
                           this->shared_key_len_);

  } break;
  case KE_RSA: {
    // wait for server hello
```

```
Record server_hello_record;
if ( this->recv(&server_hello_record) != 0 ) {
  cerr << "Couldn't receive server hello" << endl;
  return -1;
}

if ( server_hello_record.hdr.type != REC_HANDSHAKE ) {
  cerr << "Not a handshake message." << endl;
  return -1;
}

uint8_t handshake_type;
memcpy(&handshake_type,
       server_hello_record.data, sizeof(handshake_type));
if ( handshake_type != HS_SERVER_HELLO ) {
  cerr << "Not a server hello message."
       << server_hello_record.data << endl;
  return -1;
}

// retrieve n and e, then compute public key
string server_ne(
  &(server_hello_record.data[sizeof(handshake_type)]),
  -sizeof(handshake_type) + server_hello_record.hdr.length );
free(server_hello_record.data);

string server_n_str;
string server_e_str;

size_t found_n;
found_n = server_ne.find("x", 0);
if ( found_n == string::npos ) {
  cerr << "Couldn't find n." << endl;
  return -1;
}
server_n_str = server_ne.substr(0, found_n);
server_e_str = server_ne.substr(found_n+1);
```

```
// cout << "n:" << server_n_str << endl;
// cout << "e:" << server_e_str << endl;

CryptoPP::Integer server_n(server_n_str.c_str());
CryptoPP::Integer server_e(server_e_str.c_str());

CryptoPP::RSA::PublicKey server_pk;
server_pk.Initialize(server_n, server_e);

// encrypt shared key using server public key
CryptoPP::AutoSeededRandomPool rng;
CryptoPP::SecByteBlock shared_key(
  0x00, CryptoPP::AES::DEFAULT_KEYLENGTH);
rng.GenerateBlock(shared_key, shared_key.size());

string shared_key_str((const char*)shared_key.BytePtr(),
                      shared_key.SizeInBytes());
string encrypted_shared_key;
if ( rsa_encrypt(server_pk, &encrypted_shared_key,
                 shared_key_str) != 0 ) {
  cerr << "Couldn't encrypt shared key." << endl;
  return -1;
}

// send shared key
Record client_ke_record;
client_ke_record.hdr.type = REC_HANDSHAKE;
client_ke_record.hdr.version = VER_99;
client_ke_record.hdr.length = sizeof(HS_CLIENT_KEY_EXCHANGE)
                              + encrypted_shared_key.length();

char* ke_data = (char*)malloc(
                  client_ke_record.hdr.length*sizeof(char));
memcpy( ke_data,
        &HS_CLIENT_KEY_EXCHANGE, sizeof(HS_CLIENT_KEY_EXCHANGE));
memcpy( &(ke_data[sizeof(HS_CLIENT_KEY_EXCHANGE)]),
        encrypted_shared_key.c_str(),
        encrypted_shared_key.length());
```

```
      client_ke_record.data = ke_data;

      if ( this->send(client_ke_record) != 0 ) {
          cerr << "Couldn't send client key exchange E(Pk_s, key)."
               << endl;
          return -1;
      }

      free(ke_data);

      // set shared key
      set_shared_key(shared_key.BytePtr(), shared_key.SizeInBytes());

    } break;
    default : {
      cerr << "Unexpected KE type:" << hex << cxntype << endl;
      return -1;
    }
  }

  return 0;
}
```

A.2 TLS/SSL Server Connection Code

The following function is used by the server to connect to a client and do handshake to establish a shared encryption key.

```
SSL* SslServer::accept() {
  if ( this->closed_ ) {
    return NULL;
```

```
}

TCP* cxn = this->tcp_->socket_accept();
if ( cxn == NULL ) {
  cerr << "error when accepting" << endl;
  return NULL;
}

cxn->set_logger(this->logger_);

SSL* new_ssl_cxn = new SSL(cxn);

// wait for CLIENT_HELLO
Record client_hello_record;
if ( new_ssl_cxn->recv(&client_hello_record) != 0 ) {
  cerr << "Didn't receive message after accepting." << endl;
  return NULL;
}

// extract record_type / handshake_type
if ( client_hello_record.hdr.type != REC_HANDSHAKE ) {
  cerr << "Not a handshake message." << endl;
  return NULL;
}

uint8_t handshake_type;
memcpy( &handshake_type,
        client_hello_record.data, sizeof(handshake_type));
if ( handshake_type != HS_CLIENT_HELLO ) {
  cerr << "Not a client hello message." << endl;
  return NULL;
}

uint16_t key_exchange_type = 0;
memcpy( &key_exchange_type,
        &(client_hello_record.data[sizeof(HS_CLIENT_HELLO)]),
        sizeof(key_exchange_type));
```

```
// clean up
free(client_hello_record.data);

// check what kind of key agreement
switch ( key_exchange_type ) {
  case KE_DH: // todo:
  case KE_DHE: {

    // create a new record
    Record server_hello_record;
    server_hello_record.hdr.type = REC_HANDSHAKE;
    server_hello_record.hdr.version = VER_99;

    // send server hello
    string p, q, g, gs;

    ostringstream oss;
    oss << dh_p_;
    p = oss.str();

    oss.str("");
    oss << dh_q_;
    q = oss.str();

    oss.str("");
    oss << dh_g_;
    g = oss.str();

    CryptoPP::AutoSeededRandomPool rnd;
    CryptoPP::DH dh_server;
    dh_server.AccessGroupParameters().Initialize(dh_p_, dh_q_, dh_g_);
    if ( !dh_server.GetGroupParameters().ValidateGroup(rnd, 3) ) {
      cerr << "Failed to generate dh_server." << endl;
      return NULL;
    }

    CryptoPP::SecByteBlock priv_server(dh_server.PrivateKeyLength());
    CryptoPP::SecByteBlock pub_server(dh_server.PublicKeyLength());
```

```
dh_server.GenerateKeyPair(rnd, priv_server, pub_server);

string pqg_data = p + "x" + q + "x" + g + "x";

server_hello_record.hdr.length = sizeof(HS_SERVER_HELLO)
                                 + pqg_data.length()
                                 + pub_server.SizeInBytes();

char* data = (char*)malloc(
  sizeof(char)*( server_hello_record.hdr.length ));
memcpy(data, &HS_SERVER_HELLO, sizeof(HS_SERVER_HELLO));
memcpy( &(data[sizeof(HS_SERVER_HELLO)]),
        pqg_data.c_str(), pqg_data.length());
memcpy( &(data[sizeof(HS_SERVER_HELLO)+pqg_data.length()]),
        pub_server.BytePtr(), pub_server.SizeInBytes());

server_hello_record.data = data;

// send server hello
if ( new_ssl_cxn->send(server_hello_record) != 0 ) {
    cerr << "Couldn't send server hello." << endl;
    return NULL;
}
free(server_hello_record.data);

// wait for client key exchange g^c
Record client_ke_record;
if ( new_ssl_cxn->recv(&client_ke_record) != 0 ) {
    cerr << "Didn't receive g^c." << endl;
    return NULL;
}

if ( client_ke_record.hdr.type != REC_HANDSHAKE ) {
  cerr << "Not a handshake message." << endl;
  return NULL;
}

uint8_t handshake_type;
```

```
memcpy( &handshake_type,
        client_ke_record.data, sizeof(handshake_type));
if ( handshake_type != HS_CLIENT_KEY_EXCHANGE ) {
  cerr << "Not a client key exchange message." << endl;
  return NULL;
}

// extract
CryptoPP::SecByteBlock pub_client(
  (const unsigned char*)
    &(client_ke_record.data[sizeof(HS_CLIENT_KEY_EXCHANGE)]),
  dh_server.PublicKeyLength());
free(client_ke_record.data);

// compute key
CryptoPP::SecByteBlock dh_key(dh_server.AgreedValueLength());

if( !dh_server.Agree(dh_key, priv_server, pub_client) ) {
    cerr << "Server: Failed to compute shared key." << endl;
    return NULL;
}

// compute SHA 256 of key
CryptoPP::SecByteBlock shared_key(CryptoPP::SHA256::DIGESTSIZE);
CryptoPP::SHA256().CalculateDigest(
  shared_key, dh_key, dh_key.size());

// save key inside client connection
new_ssl_cxn->set_shared_key(
  shared_key.BytePtr(), shared_key.SizeInBytes());

// log shared keys
this->logger_->log_raw("DH Key:");
this->logger_->log_raw((const char*)dh_key.BytePtr(),
                       dh_key.SizeInBytes());
this->logger_->log_raw("Shared Key:");
this->logger_->log_raw((char *)this->shared_key_,
                       this->shared_key_len_);
```

```
} break;
case KE_RSA: {
  // prepare server hello
  Record server_hello_record;
  server_hello_record.hdr.type = REC_HANDSHAKE;
  server_hello_record.hdr.version = VER_99;

  string n, e;

  ostringstream oss;
  oss << this->private_key_.GetModulus();
  n = oss.str();

  oss.str("");
  oss << this->private_key_.GetPublicExponent();
  e = oss.str();

  string ne_data = n + "x" + e;

  server_hello_record.hdr.length =
    sizeof(HS_SERVER_HELLO) + ne_data.length();

  char* data = (char*)malloc(
    sizeof(char)*( server_hello_record.hdr.length ));
  memcpy(data, &HS_SERVER_HELLO, sizeof(HS_SERVER_HELLO));
  memcpy( &(data[sizeof(HS_SERVER_HELLO)]),
          ne_data.c_str(), ne_data.length());

  server_hello_record.data = data;

  // send server hello
  if ( new_ssl_cxn->send(server_hello_record) != 0 ) {
      cerr << "Couldn't send server hello." << endl;
      return NULL;
  }
  free(server_hello_record.data);
```

```
// wait for client key exchange shared key
Record client_ke_record;
if ( new_ssl_cxn->recv(&client_ke_record) != 0 ) {
    cerr << "Didn't receive shared key." << endl;
    return NULL;
}

if ( client_ke_record.hdr.type != REC_HANDSHAKE ) {
  cerr << "Not a handshake message." << endl;
  return NULL;
}

uint8_t handshake_type;
memcpy( &handshake_type,
        client_ke_record.data, sizeof(handshake_type));
if ( handshake_type != HS_CLIENT_KEY_EXCHANGE ) {
  cerr << "Not a client key exchange message." << endl;
  return NULL;
}

// extract
string encrypted_shared_key(
  &(client_ke_record.data[sizeof(HS_CLIENT_KEY_EXCHANGE)]),
  -sizeof(HS_CLIENT_KEY_EXCHANGE) + client_ke_record.hdr.length);
free(client_ke_record.data);

// decrypt shared key
string shared_key;
rsa_decrypt(this->private_key_, &shared_key, encrypted_shared_key);

// save key inside client connection
new_ssl_cxn->set_shared_key(
  (const unsigned char*)shared_key.c_str(), shared_key.length());

// log shared keys
this->logger_->log_raw("Shared Key:");
this->logger_->log_raw((char *)this->shared_key_,
                       this->shared_key_len_);
```

```
    } break;
    default:
      cerr << "Unexpected KE type:" << hex << key_exchange_type << endl;
      return NULL;
  }

  this->clients_.push_back(new_ssl_cxn);
  return new_ssl_cxn;
}
```

A.3 TLS/SSL Encryption Code

The following function is used by both the server and client to encrypt and decrypt messages using the shared key.

```
int SSL::send(const std::string &send_str) {
  // make a record
  Record send_record;
  send_record.hdr.type = REC_APP_DATA;
  send_record.hdr.version = VER_99;

  // encrypt
  string cipher_text;

  if ( aes_encrypt(this->shared_key_, this->shared_key_len_,
                   &cipher_text, send_str) != 0 ) {
    cerr << "Couldn't encrypt." << endl;
    return -1;
  }
```

```
  char* data = (char*)malloc(cipher_text.length()*sizeof(char));
  memcpy(data, cipher_text.c_str(), cipher_text.length());
  send_record.data = data;

  // add length to record
  send_record.hdr.length = cipher_text.length();

  // send
  int ret_code;
  ret_code = send(send_record);
  free(send_record.data);

  return ret_code;
}


int SSL::recv(std::string *recv_str) {
  // receive record
  Record recv_record;
  if ( recv(&recv_record) == -1 ) {
    cerr << "Couldn't receive." << endl;
    return -1;
  }

  // check
  if ( recv_record.hdr.type != REC_APP_DATA) {
    cerr << "Not app data." << endl;
    return -1;
  }

  // extract
  string cipher_text(recv_record.data, recv_record.hdr.length);
  free(recv_record.data);

  // decrypt
  if ( aes_decrypt(this->shared_key_, this->shared_key_len_,
                   recv_str, cipher_text) != 0 ) {
    cerr << "Couldn't decrypt." << endl;
```

```
    return -1;
  }

  return 0;
}
```

Appendix B

Python2.7 Code for Lempel-Ziv77 Compression

Below contains the software and hardware implementation of Lempel-Ziv77 compression algorithm.

B.1 Software Version

The software version is written in Python 2.7. It looks for the repeating string by using a scanning window described in the original Lempel-Ziv77 algorithm, Algorithm 1.

```
def compress(self, filename):
    # init
    search_buffer = None
```

```
look_ahead_buffer = None
outputs = []

# open file and process
with open(filename) as fp:

    while True:

        # compute the two buffers
        search_buffer, look_ahead_buffer, preview, end = \
            self._get_buffers(search_buffer, look_ahead_buffer, fp)

        if end:
            break

        # print "search:la \t %s:%s %s" % \
        #     (search_buffer, look_ahead_buffer, preview)

        # find match
        match_pos, match_len = \
            self._find_match(search_buffer, look_ahead_buffer)

        # compute next char
        if match_len == len(look_ahead_buffer):
            next_char = preview
        elif match_len < len(look_ahead_buffer):
            next_char = look_ahead_buffer[match_len:match_len+1]
        else:
            next_char = ""

        # shift the buffers
        for i in xrange(match_len):
            search_buffer, look_ahead_buffer, ignore, end = \
                self._get_buffers(search_buffer,
                                  look_ahead_buffer, fp)

        # add to output
        outputs.append(OutEle(match_pos, match_len, next_char))
```

```
        # end while loop

        # check last element, add EOF if needed
        if len(outputs) > 0:
            if outputs[-1].next_char != '':
                outputs.append(OutEle(None, 0, ''))

    return outputs
```

B.2 Hardware Version

The hardware version of the algorithm is more complex, since it trade for faster run-time.

It uses dictionaries for faster look-ups than linearly scanning the sliding window.

```
def lz_compress(i_file, i_chunk_size,
                i_dict_rows, i_dict_entry_len, i_min_select_len):
    """ Compress a file
    Inputs:
    - i_file (file):
        file to read
    - i_chunk_size (int):
        chunk size to read at each iteration
    - i_dict_entry_len (int):
        max length of each dict entry
    - i_min_select_len (int):
        drops matches that are < than this length
    Output:
    - outputs:
        list of OutEle
    """
    # todo: check inputs
```

```
    # init local variables
    dictionary = init_table(i_chunk_size, i_dict_rows,
                            DictEntry(entry=None, location=-1))

    data_vector = [None] * i_chunk_size * 2
    data_vector_hash = [0] * i_chunk_size
    curr_dict_read = init_table(i_chunk_size, i_chunk_size)
    longest_matches = [None] * i_chunk_size
    curr_pos = 0
    first_valid_pos = 0
    outputs = []

    # compress loop
    datafile = open(i_file, 'r')
    while True:
        debug("new iteration")
        debug2("curr pos:%d" % curr_pos)

        debug("read new data ... ")
        read_len = read_file(data_vector, datafile, i_chunk_size)
        debug2("read len:%d" % read_len)
        if read_len == -1:
            debug("nothing more to read, compression ends ...")
            break
        debug2(data_vector)

        debug("computing hashes ... ")
        get_hashes(data_vector_hash, data_vector)
        debug2(data_vector_hash)

        debug("reading dictionary ... ")
        read_dictionary(curr_dict_read, dictionary, data_vector_hash)

        debug("updating dictionary ... ")
        update_dictionary(dictionary, data_vector, curr_pos,
                          data_vector_hash, i_dict_entry_len)

        debug("find longest match ... ")
```

```
        find_longest_matches(longest_matches, data_vector,
                             curr_dict_read, i_chunk_size)
        debug2("longests=%s" % longest_matches)

        debug("selecting match ... ")
        new_first_valid_pos, chosen_indices = \
            select_match(data_vector, longest_matches,
                         first_valid_pos, i_min_select_len)
        debug2(new_first_valid_pos)
        debug2(chosen_indices)

        debug("generating outputs ... ")
        curr_outputs = \
            gen_outputs(data_vector, curr_pos, first_valid_pos,
                        chosen_indices, longest_matches)

        # add output to list of outputs
        for o in curr_outputs:
            outputs.append(o)

        # set values for next iteration
        curr_pos += i_chunk_size
        first_valid_pos = new_first_valid_pos
    # loop ends

    # close file
    datafile.close()

    print_dictionary(dictionary)

    return outputs
```

Appendix C

Python Code for Making a PDF Plot

We used plotting functions from *matplotlib* packages in Python 2.7 to plot our attack and time data.

C.1 Data Plot

The function, *main*(), takes in the length of all datas from the attack and plots them on a graph, with the red line being the letter corresponding to guess character being correct.

```
def main():

    args = parse_args()
    print args

    # init vars
    in_file = args.file
```

```
out_file = args.output
title = args.title
correct_char = args.char

# read data
dictionary = load_file(in_file)
data = dictionary[DICTKEY_DATA]

# create figure
fig = plt.figure(title)
fig.suptitle(title, fontsize=14, fontweight='bold')
plt.ylabel('False Positives (%)')
plt.xlabel('Number of Rounds')

# plot in data

# count # different keys
all_keys = sorted(data.keys())
total_chars = len(all_keys)

# allocate the 2D array
columns = data[all_keys[0]].keys()
columns = sorted(columns, key=lambda s: int(s))
columns = columns[1:]

dataframe = DataFrame(index=all_keys, columns=columns)
dataframe = dataframe.fillna(method='backfill')

# fill 2D array with data (cols = # of rounds, rows = alphabets)
for k in all_keys:
    for col in columns:
        dataframe[col][k] = data[k][col]

# print dataframe

# make a list of data
y = []
for col in columns:
```

```
        # get all lengths
        all_len = dataframe[col]
        # gather number of false pos
        real_len = all_len[correct_char]
        all_len = all_len.drop(correct_char)
        num_false_pos = 0.0
        for length in all_len:
            if length <= real_len:
                num_false_pos += 1
        # calculate
        y.append(num_false_pos*100/total_chars)

    # set y-axis
    axes = plt.gca()
    axes.set_ylim([0, 100])

    # plot it
    plt.plot(columns, y, 'blue')

    # save fig
    plt.savefig(out_file)
    plt.close()

    return
```

C.2 Time Plot

The function, *main*(), takes in the timing of the attack data for each iterations, and plot them on a graph.

```
def main():
```

```
args = parse_args()
print args

# init vars
in_files = args.files
out_file = args.output
title = args.title

all__times = {}

# read data
for f in in_files:
    dictionary = load_file(f)
    temp_data = dictionary[DICTKEY_TIME]
    for k, v in temp_data.iteritems():
        if k not in all__times.keys():
            all__times[k] = []
        all__times[k].append(int(v))

# print all__times

# calculate average
avg_time = {}
for k, v in all__times.iteritems():
    avg_time[k] = sum(v) / len(v)

# print avg_time

# create figure
fig = plt.figure(title)
fig.suptitle(title, fontsize=14,
    fontweight='bold')
plt.ylabel('Time (seconds)')
plt.xlabel('Number of Rounds')

# sort it by key and retrieve x, y
avg_time = sorted(avg_time.items(),
            key=lambda s: int(s[0]))
```

```
    avg_time = remove_first(avg_time)
    x, y = zip(*avg_time)

    # plot
    plt.plot(x, y, 'bo')

    # add labels
    ax = fig.add_subplot(111)
    for xy in zip(x, y):
        ax.annotate('(%s)' % xy[1],
            xy=xy, textcoords='data')

    # save fig
    plt.savefig(out_file)
    plt.close()

    return
```

Glossary

BEAST Short for Browser Exploit Against SSL/TLS, attack that extracts authentication token by exploiting vulnerability in chain block chaining mode that TLS 1.0 uses 3

BREACH Short for Browser Reconnaissance and Exfiltration via Adaptive Compression of Hypertext, compression side-channel attack that exploits against HTTP compression algorithm 2, 9

CRIME Short for Compression Ratio Info-leak Made Easy, compression side-channel attack that extract secrets from HTTP header 2, 7

DEFLATE Compression algorithm by applying Lempel-Ziv 77 then Huffman coding 1, 7

LZS Lempel-Ziv-Stac, compression algorithm by combining Lempel-Ziv 77 and fixed Huffman coding 1

TLS Transport Layer Security, it provides compression and security for the transport layer of Open Systems Interconnection layers. 1