# Implementing a Proven-secure and Cost-effective Countermeasure against the Compression Ratio Info-leak Mass Exploitation (CRIME) Attack

Jayamine Alupotha[†], Sanduni Prasadi[‡], Janaka Alawatugoda and Roshan Ragel
Department of Computer Engineering
Faculty of Engineering
University of Peradeniya
Peradeniya 20400, Sri Lanka
{[†]jayamine.alupotha, [‡]sanduni.prasadi06}@gmail.com

Mohamed Fawsan
Department of Computer Science and Engineering
Faculty of Engineering
University of Moratuwa
Moratuwa 10400, Sri Lanka
fawzanm@cse.mrt.ac.lk

***Abstract*—Header compression is desirable for network applications as it saves bandwidth and reduces latency. However, when data is compressed before being encrypted, the amount of compression leaks information about the amount of redundancy in the plaintext. In web requests, headers contain secret web cookies. Therefore, compression of headers before encryption will reveal the information about the secret web cookies. This side-channel has led to Compression Ratio Info-leak Made Easy (CRIME) attack on web traffic protected by the SSL/TLS protocols. In order to mitigate the CRIME attack, compression is completely disabled at the TLS/SSL layer, which in return increases the bandwidth consumption and latency. In a previous work (Financial Cryptography and Data Security 2015), two countermeasures are presented with formal security proofs, against compression side-channel attacks, namely (1)–separating secret cookies from user inputs and (2)–using a static compression dictionary.**

**In this work we create a test environment to replicate the CRIME attack and verify the attack. Moreover, we implement a proven-secure countermeasure against the CRIME attack, in a real world client/server setup, following the aforementioned two countermeasures. Our implementation achieves better compression ratio (closer to the original TLS/SSL compression), and hence reduces the bandwidth usage and latency significantly (therefore cost-effective). To the best of our knowledge, this is the first proven-secure and cost-effective countermeasure implementation against the CRIME attack.**

***Index Terms*– data compression, CRIME attack, SSL/TLS, cryptography, security**

## I. Introduction

Nowadays, web pages have been grown to need thousands of requests. The headers in these requests consume significant amount of bandwidth, increasing latency. Initially, SPDY [1] addressed this issue by compressing header fields using DEFLATE algorithm [2], which compresses the redundant header fields effectively. But that approach exposed a security risk as demonstrated by the CRIME attack [3].

978-1-5386-1676-5/17/$31.00 ©2017 IEEE

At ekoparty 2012 security conference, Duong and Rizzo announced the CRIME attack; a compression side-channel attack against HTTPS (Secure HyperText Transfer Protocol) traffic [3]. The demonstration given at the ekoparty 2012 showed how to recover the secret web cookies of HTTPS requests. The web cookies are used for web application authentication (after log-in). Therefore, the attacker will be able to hijack the session upon recovering the web cookie. As the obvious fix against the CRIME attack, all the web servers and clients disabled TLS/SSL-level (Transport Layer Security/Secure Socket Layer) compression.

Disabling header compression effects on the performance by increasing the bandwidth usage and latency, that obviously increases the cost. In HTTP/2 this issue is also taken into account and a new header field compression, HPACK [4] is introduced. HPACK is a compression format for efficiently representing HTTP header fields. According to HTTP/2 [5], HPACK does not completely prevent the CRIME attack [3], but it mitigates the risk to some extent.

### *A. Header Compression*

Compression is a mechanism to transmit or store data by reducing its size. Gzip [6] and DEFLATE [2] are considered as the most common compression formats.

The main compression method used in TLS header compression is DEFLATE [2]. DEFLATE compression method is a combination of the LZ77 algorithm and Huffman encoding, which defines a loss-less compression. Huffman coding is used to eliminate the redundancy of repeating symbols. It searches for repeated strings and replaces them with back-references to the last occurrence as (distance, length) and convert the content into a zlib-formatted stream. Gzip is based on the DEFLATE algorithm. Therefore, gzip provides a loss less-compression, similar to DEFLATE compression.

## B. *Compression Side-channel Attacks*

Compression side-channel attacks [7] are not new. Research on compression side-channel attacks were started more than a decade ago. In 2002 Kelsey described a side-channel attack [8] provided by data compression algorithms, yielding information about their inputs by the size of their outputs to reveal information about plaintext. If plaintext is compressed before the encryption, the length of the ciphertext reveals information about the amount of compression, which in turn can reveal information about the plaintext to the attacker. According to Kelsey's attack, if an attacker is allowed to choose inputs $x$ that are combined with a target secret $s$ and the concatenation $x||s$ is compressed and encrypted, observing the length of the outputs can eventually allow the attacker to extract the secret $s$. For example, to determine the first character of $s$, the attacker asks to have the string $x = \texttt{prefix}\star\texttt{prefix}$ combined with $s$, then compressed and encrypted, for every possible character $\star$; in one case, when $\star = s_1$, the amount of redundancy is higher and the ciphertext should be shorter. Once each character of $s$ is found, the attack can be carried out on the next character.

The security research community has had a lot of surprises with SSL/TLS uncovering a few terrific attacks. First, at the ekoparty 2011 Security Conference, Rizzo and Duong uncovered a new attack on Transport Layer Security (TLS), namely BEAST attack [9]. At the ekoparty 2012 security conference, Rizzo and Duong released the details about the CRIME [3] attack, which is a compression side-channel attack against HTTPS traffic to reveal secret web cookies. In 2013 at Black Hat USA, Prado, Harris, and Gluck announced another attack, namely BREACH [10] attack, that targets the vulnerability of HTTP body compression to reveal secrets like anti-CSRF tokens and any other personally identifiable data.

## C. *The CRIME Attack*

Many web applications use cookies in the headers of HTTPS requests for authorization purposes. These requests are subjected to compression and encryption respectively. The primary target of the CRIME attack was the user's cookie in the HTTPS header. If the victim visited an attacker-controlled web page, the attacker could use Javascript to cause the victim to send HTTPS requests to URLs of the attacker's choice on the target server. The attacker could adaptively choose those URLs to include a prefix to carry out Kelsey's attack. Consider a website with basic authentication. If a user with an active session makes the following request:

```
POST /test/demo-form.php HTTP/1.1
Host: example.com
Cookie: JSESSIONID=672CA12B
Allow-Control-Allow-Origin: *
```

With the above request, the attacker sends a string `JSESSIONID=6` as a URL string.

In the above case, `JSESSIONID` which is used to identify the authorized users, is the secret. Note that the string `JSESSIONID=` is repeated in the request. DEFLATE can take advantage of this repeated strings. In fact, if the first character of the guess matches the first character of the actual value of `JSESSIONID` (6 in this case), then DEFLATE will compress the request even more. The request with maximum compression ratio can be considered as the correct guess. Because of that an attacker can exploit to recover the first character of `JSESSIONID`. Then the attacker proceeds to recover the second byte, injecting the recovered first byte and a guessed character for the second byte. By continuing the same process as per the first byte, attacker can recover complete `JSESSIONID` byte-by-byte.

## D. *Our Contribution*

In this work, we replicate the CRIME attack. In order to do that, we setup a client/server environment with TLS/SSL layer compression enabled. We establish the attack setup by injecting a Javascript (using an attacker-controlled website) on the client machine, to inject the URLs (of attacker's guesses on cookie bytes) to be sent to the target server. Moreover, we implement the CRIME attack algorithm to recover the cookie byte-by-byte by measuring the compression length of each request made by the client to the target server. Our setup can be used to test the CRIME attack and countermeasures against it.

Then, we implement a proven-secure countermeasure against the CRIME attack, following the proven-secure concepts of (1)–separating secrets from user inputs and (2)–using a static compression dictionary of Alawatugoda et al. [11]. The security of this countermeasure is theoretically proven in a well-defined security model in the work of Alawatugoda et al. [11]. We implement this countermeasure in a real world setup, using Apache tomcat server. We use a static table to encode header field names, since it will increase the compression ratio. Then, the secret cookies are separated, and after that DEFLATE compression is taken place on the encoded header, except on the separated secret cookies. Finally, the compressed portion and the separated secrets are encrypted, and the HTTPS request is created.

Our countermeasure implementation achieves better compression ratio (closer to the original TLS/SSL compression), because the header is encoded using a suitable static dictionary and only the secrets are kept uncompressed. Therefore, our countermeasure reduces the bandwidth usage and latency significantly. To the best of our knowledge, this is the first proven-secure and cost-effective (by means of bandwidth save and reducing the latency) countermeasure implementation against the CRIME attack.

## II. Related Works

In this section we briefly discuss about currently using countermeasures against the CRIME attack and proposed countermeasures in the literature.

## A. *Disabling Compression*

Disabling compression at the HTTP level completely prevents the side-channel. Therefore it defeats the CRIME attack.

Unfortunately, this solution can have a significant impact on performance. Large number of requests are sent (including handshake requests) to load a page of a web application. These requests take a considerable amount of the network bandwidth and has an impact on network latency. This is the widely used fix against the CRIME attack.

### *B. TLS Extension to Hide The Length*

The risk of CRIME attack can be somewhat mitigated by hiding the length of requests. Because of that attacker has to identify the length first and it will take longer time and the attacker has to send large number of requests. In 2013, Pironti and Mavrogiannopoulos purposed a TLS extension [12] to allow arbitrary amount of padding in any TLS ciphersuite. But this approach does not eliminate the risk completely. The attacker can recover the exact length of the request by sending multiple requests for single guess and averaging the sizes of the requests. This countermeasure delays the attack.

### *C. Static Dictionary Compression*

In the static dictionary compression, the dictionary used for compression does not adapt to the plaintext being compressed, but instead is preselected in advance based on the expected distribution of plaintext messages, for example including common words and phrases to be used in the context (in our case the context of header field names). Therefore, a secret cookie (which is normally a set of random characters), has a negligible chance to be appeared in the static dictionary and being compressed. It is statistically proven that this countermeasure can make the attackers probability of recovering the secret cookie by launching the CRIME attack negligible [11].

### *D. Separating Secrets From User Inputs*

Separating secrets from user inputs [11] has been proven as an effective solution which can completely eliminate the CRIME attack. The sole idea if this countermeasure is to separate the secret cookie from rest of the information that are being compressed. Since the secret cookie is not compressed the origin of the compression leakage is shut. Since the other information is compressed it is possible to achieve significant compression ratio.

## III. Bandwidth Usage of Request/Response Headers

Although disabling header compression completely mitigates the CRIME attack, it has a drastic impact on network latency and bandwidth usage. As an example, average bandwidth consumption by headers in Facebook is 11.8% of the total bandwidth consumption of requests-responses (Figure 1), which is 14.7% in Gmail (Figure 2). We measure this for 2000 exchanges (requests-responses) in each website with 95% confidence level and $\pm 0.7$ confidence interval. This result shows that headers acquire considerable amount of the total bandwidth consumption of a website. Hence, in reduction of network latency and bandwidth consumption, header compression is very useful.

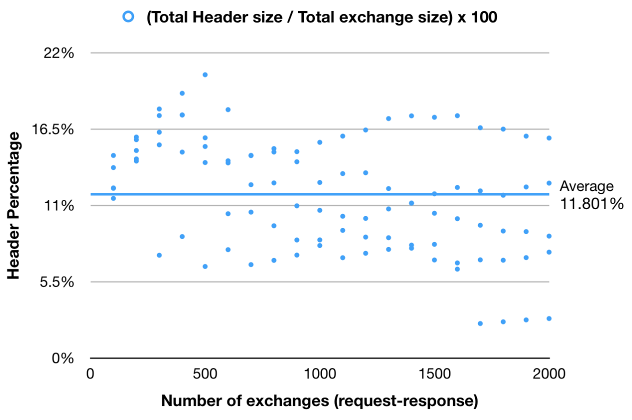


Fig. 1. Request-Response size vs header percentage for Facebook

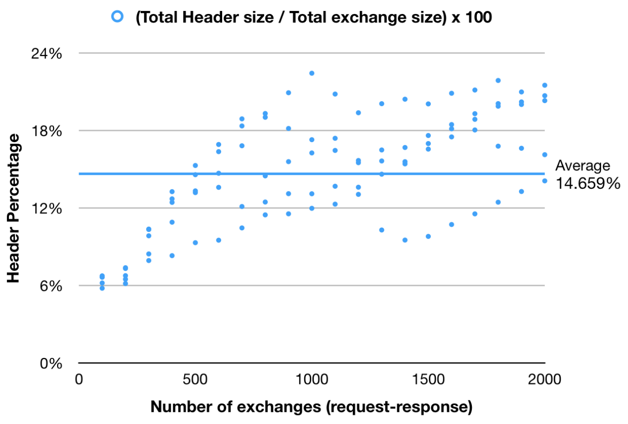


Fig. 2. Request-Response size vs header percentage for Gmail

## IV. Experimental Setup to verify the CRIME Attack

### *A. Prototype*

In order to test the proposed countermeasure, the CRIME attack is remodeled and a test environment is created. For a web application to be vulnerable to this side-channel attack, it must,

- be served from a server that uses HTTP-level header compression,
- sent requests through a browser that supports header compression and
- reflect a secret (such as a cookie) in HTTP request header.

Additionally, while not strictly a requirement, the attack is helped greatly if no noise occurs in the side-channel. This is because the difference in size of the request headers measured by the attacker can be quite small.

The attack is carried out with the assumption that the attacker has the ability to view the victims encrypted traffic. An attacker might accomplish this with a network protocol analyzer like Wireshark or a proxy server. Therefore, the CRIME attacker can be an ISP (Internet Service Provider), network administrator, the government or any party who can eavesdrop encrypted traffic (Man-In-The-Middle). It is also assumed that the attacker has the ability to cause the victim to send HTTP requests to the secure web server. This can be accomplished by coercing the victim to visit an attacker-controlled site (which will contain a JavaScript code that sends requests to the SSL/TLS-protected server with injected values in the request header).

This is an active, online attack. The attack proceeds byte-by-byte. The attacker will coerce the victim to send a small number of requests to guess the first byte of the target secret cookie. The attacker then measures the size of the request header. With that information, the CRIME attack algorithm determines the correct value for the first character of the secret cookie. Since the attack relies on LZ77 loss-less data compression algorithm, the first byte of the target secret must be correctly guessed before the second byte is attempted. Figure 3 shows the CRIME attack setup.

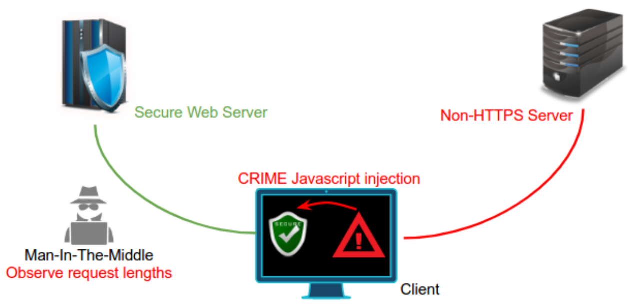


Fig. 3. CRIME attack setup

## B. *CRIME Attack Algorithm*

Huffman coding is a loss-less data compression algorithm [13]. In this algorithm, input characters are assigned with variable-length codes. Frequency of the corresponding character in the content defines the length of the assigned code. The most frequent character gets the smallest character set and the least frequent character gets the largest character set.

When the attacker guesses a character correctly and injects the guess to the request, the resulting string will have a longer repeated character set than in the previous string in the request. This results more compression. Differently, if an attacker guesses a character incorrectly, the length of the compressed request will be *equal* or *less* than the length of compressed request of the correct guess. Having this length equal or less than the length of the correct guess seems problematic. This could be solved using *Two-Tries Method* used by Duong and Rizzo. Instead of sending one request for each guess, two requests are sent as follows:

Let the assumed secret for the first byte is `X`,

- First send a request with <url>/`secret=X` (let the compressed and encrypted request length is L1 in this case)
- Then send a request with <url>/`Xsecret=` (let the compressed and encrypted request length is L2 in this case)

Get the minimum for L1 by considering all the possible characters in place of `X`. If that L1 is less than corresponding L2, then the guessed character is taken as a possible value.

Recovering the secret will act in unexpected manner when there are repeated characters in the secret itself. For example, suppose the target secret is `ABCABXYZ`. Further, suppose the first five characters of the secret: `ABCAB` are recovered. When trying to guess the next character it will likely return `C` and `X` as correct values. This is because `ABCABC` alone is compressible while `ABCABX` matches the actual secret. This is mitigated by checking how well our guess compresses by itself. If a particular guess compresses by itself beyond a certain threshold, it is discarded.

Block ciphers pose and additional challenge to the CRIME attack algorithm. A tipping point is the point at which any additional incompressible character causes the injected string to overflow into an additional block. When the injected string is aligned to a tipping point, it is assumed that only correct guesses are fit within the current block, allowing to use the CRIME attack algorithm the same way as for stream ciphers. In order to get our injected values aligned to a tipping point, a filler string (that is not repeated in the injected string) is added to the request.

Currently, our CRIME attack algorithm has an accuracy of 93% to recover 4 bytes of any length cookie of hexadecimal characters.

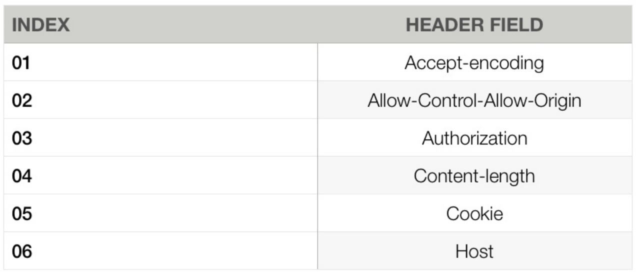
## C. *Pitfalls*

Despite the straightforward nature of the CRIME attack, actually exploiting it to create a real testing environment is challenging. This is mainly because the header compression is completely removed from all servers and browsers. Therefore, an older version of a browser that allows header compression is used. A server is configured to enable HTTP-level header compression using OpenSSL library.

## V. Countermeasure: Static Dictionary Header Encoding and Separating Secret Cookies in Headers

We implement our countermeasure using an Apache tomcat server and tested it with a Java socket client.

When implementing the countermeasure, in order to have a better header compression, static dictionary is adopted for encoding the header field names. In general, static dictionary compression schemes work by advancing through the string `x` and looking to see if the current substring appears in the dictionary `D`: if it does, then an encoding of the index of the substring is recorded, otherwise an encoding of the current substring is recorded. Our countermeasure adopts a static table as the static dictionary, which consists of a predefined static list of header field names. We use that static dictionary for encoding the header fields. Figure 4 shows an example for a static dictionary.



| INDEX | HEADER FIELD |
|---|---|
| 01 | Accept-encoding |
| 02 | Allow-Control-Allow-Origin |
| 03 | Authorization |
| 04 | Content-length |
| 05 | Cookie |
| 06 | Host |

Fig. 4. Example static dictionary

In the client-side, first the header fields of the request are replaced by the relevant encoding taken from the static dictionary. Then the secrets in the request header are separated. The encoded request without secrets is compressed and then encrypted together with the separated secrets. In the server-side (tomcat server), the content is decrypted and then the compressed content is decompressed. Then, the secrets are appended to the uncompressed request, to regenerate the request in its original form. Then the request is decoded using the static dictionary.

Similarly the response header is encoded using the static dictionary at the server-side. If secrets are available, then they are separated out. After that, the response header excluding the secrets is compressed. Finally both the compressed content and the separated secrets are encrypted before sending to the client. At the client-side, the response is decrypted, then the response header without secrets is decompressed. The separated secrets are appended to the response header and finally the response header is decoded to regenerate the original header.

Figure 5 shows the format of a request/response header, that is used in our countermeasure. In order to identify requests/responses with compressed headers, an indicator of value 0 is added to the beginning of the request/response. Next 2 bytes are allocated for the compressed-content's length. After that the compressed request/response header (without the secret cookies) is added. Separated secret cookies are added after ':' character. End of the header is denoted by $\backslash r \backslash n$.

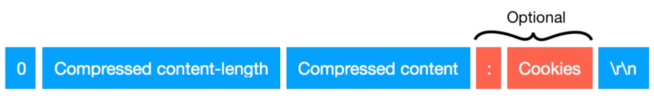


Fig. 5. Format of a request/response header in our countermeasure

The actual request header is rearranged as shown below. Consider the following request:

```
POST /test/demo-form.php HTTP/1.1
Host: example.com
Allow-Control-Allow-Origin: *
Cookie: JSESSIONID=672CA12B
```

This request header is encoded using the static dictionary illustrated in Figure 4:

```
POST /test/demo-form.php HTTP/1.1
06: example.com
02: *
05: JSESSIONID=672CA12B
```

Then the request header is rearranged by separating the secrets:

```
POST /test/demo-form.php HTTP/1.1
06: example.com
02: *
05:
```

Then, it is compressed using DEFLATE compression algorithm, and used as the "compressed-content". The secret cookies `JSESSIONID=672CA12B` are used as the "Cookies". Then the new request header is encrypted. Since the secrets do not compress with the request header (together with the attacker-injected values), the CRIME attack can be completely eliminated. Because we wipe out the origin of the attack.

Figure 6 and 7 illustrates the compression ratios (equation (1)) of the headers for Facebook and Gmail respectively, when using our countermeasure vs the original SSL/TLS-layer compression (vulnerable to the CRIME attack). Although the cookies are separated and kept uncompressed, the headers are significantly compressed with the use of our countermeasure implementation (closer to the compression ratio of original TLS/SSL-layer compression). This is useful in reduction of the bandwidth usage and decreasing the latency.

$$\text{Compression Ratio} = \frac{\text{Compressed Header Size}}{\text{Header Size}} \times 100 \quad (1)$$

## VI. Conclusion and Future Works

In this work we develop a test environment to replicate the CRIME attack and verified the attack. Further, we implement a practical countermeasure, namely static dictionary header encoding and separating secret cookies in request/response headers, against the CRIME attack in a real world client/server setup. To the best of our knowledge, this is the first proven-secure and cost-effective countermeasure implementation against the CRIME attack. We also implement this countermeasure in the server-side, in a way that it works compatibly with the Mozilla Firefox browser.

As a future work we will look at how to improve the compressibility of headers. Particularly, using an adaptive encoding dictionary this can be achieved, given that the secret cookie is excluded when constructing the dictionary. This direction should be further studied. Moreover, we will look at the BREACH attack [10] and work on implementations of

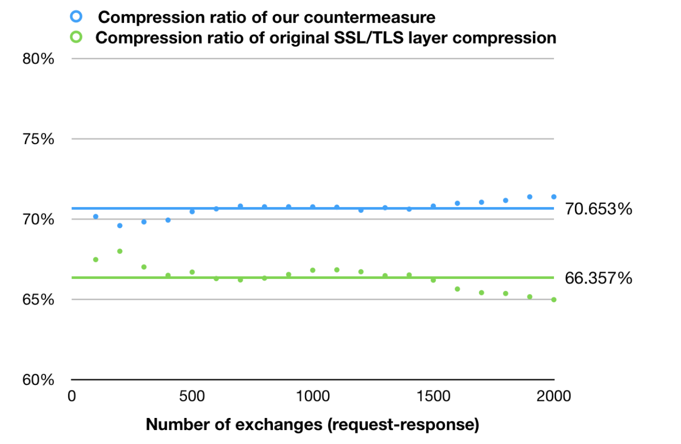


Fig. 6. Compression ratios of headers of Facebook for our countermeasure vs original SSL/TLS-layer compression

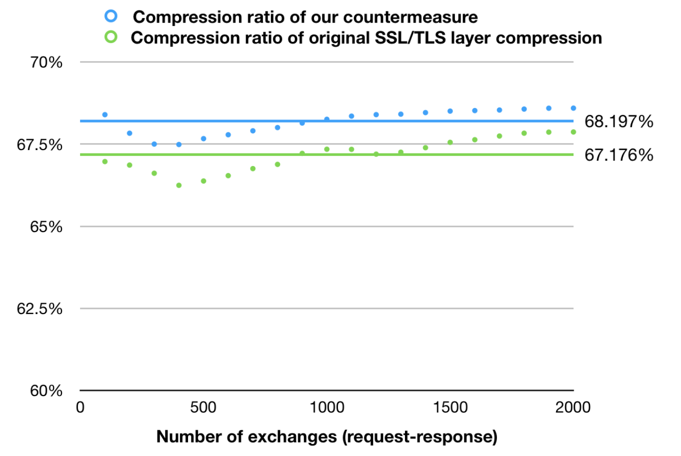


Fig. 7. Compression ratios of headers of Gmail for our countermeasure vs original SSL/TLS-layer compression

proven-secure countermeasures against it. Particularly, it will be possible to adopt the two proven-secure countermeasures of Alawatugoda et al. [11]. Differently, we will have to deal with the HTTP response body.

## Acknowledgement

Janaka Alawatugoda acknowledges the postdoctoral fellowship grant NRC 16-020 of the National Research Council, Sri Lanka.

## References

[1] J. Erman, V. Gopalakrishnan, R. Jana, and K. K. Ramakrishnan, "Towards a spdyier mobile web?," *IEEE/ACM Transactions on Networking*, vol. 23, no. 6, pp. 2010–2023, 2015.

[2] L. P. Deutsch, "Deflate compressed data format specification version 1.3," 1996.

[3] "Crime attack." https://en.wikipedia.org/wiki/CRIME_(security_exploit). (Accessed on 15/03/2017).

[4] R. Peon and H. Ruellan, "RFC 7541 HPACK: Header Compression for HTTP/2," *The effects of brief mindfulness intervention on acute pain experience: An examination of individual difference*, vol. 1, pp. 1–55, 2015.

[5] D. Stenberg, "Http2 explained.," *Computer Communication Review*, vol. 44, no. 3, pp. 120–128, 2014.

[6] L. P. Deutsch, "Gzip file format specification version 4.3," 1996.

[7] S. Chen, R. Wang, X. Wang, and K. Zhang, "Side-channel leaks in web applications: A reality today, a challenge tomorrow," in *Security and Privacy (SP), 2010 IEEE Symposium on*, pp. 191–206, IEEE, 2010.

[8] J. Kelsey, "Compression and information leakage of plaintext," in *International Workshop on Fast Software Encryption*, pp. 263–276, Springer, 2002.

[9] "Transport layer security." https://en.wikipedia.org/wiki/Transport_Layer_Security#BEAST_attack, Mar 2017. (Accessed on 15/03/2017).

[10] Y. Gluck, N. Harris, and A. Prado, "Breach: reviving the crime attack," *Unpublished manuscript*, 2013.

[11] J. Alawatugoda, D. Stebila, and C. Boyd, "Protecting encrypted cookies from compression Side-Channel attacks," vol. 8975, no. Fc 2015, pp. 86–106, 2015.

[12] A. Pironti and N. Mavrogiannopoulos, "Length hiding padding for the transport layer security protocol," tech. rep., Internet-Draft draft-pironti-tls-length-hiding-00, IETF Secretariat, 2013.

[13] P. Deutsch, "RFC 1951: Deflate compressed data format specification." https://dzone.com/articles/tracking-http2-adoption-stagnation, May 1996. (Accessed on 04/06/2017).