CORE
Metadata, citation and similar papers at core.ac.uk
Provided by University of Minnesota Digital Conservancy

Length Hiding VPN to Mitigate Compression Side-Channel and Traffic Analysis Attacks

A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Ankit Anand Gupta

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

Dr. Peter A.H. Peterson

December 2017

© Ankit Anand Gupta 2017

Acknowledgements

There are many people who have contributed, in their own way towards the completion of this thesis and my journey through graduate school. First among them is my advisor Dr. Peter Peterson, the guiding beacon throughout this journey called research. I am grateful to all my professors for imparting the knowledge that made this work possible.

I would like to express my sincere gratitude to Dr. Neeraj Gupta, Mrs. Rashmi Gupta, Mr. Ravi Gupta and Mrs. Shalini Gupta for being my guardians and making me feel at home in this foreign country. I would like to thank my classmates Brandon Paulsen and Jon Rusert for their valuable inputs and motivating words. I would also like to thank my fellow graduate students with whom I shared many unforgettable and cherished moments and friends in India for being there for me when I needed someone to talk to.

Dedication

Dedicated to my mother, father and brother for being the constants in this world full of variables.

Abstract

Internet traffic is more voluminous than ever before in history. This data transmission over a network involves a trade-off between efficiency and security. On the one hand, compressing data can increase the efficiency if it leads to fewer bytes being sent, but this makes the traffic susceptible to compression side-channel attacks. On the other hand choosing not to compress makes it immune to such attacks but fails to maximize efficiency. CRIME and BREACH are two compression side-channel attacks. These attacks exploiting the property of dictionary compression, where an increase in redundancy in data leads to a better compression. In addition to these, there are indirect attacks that can identify user behavior in spite of it being encrypted. These attacks known as traffic analysis attacks and identify user behavior based on traffic properties such as bandwidth, packet sizes, inter-packet arrival time and total time for data transfer. These aforementioned attacks deter or may deter applications from using compression for data being transferred over the network. Despite it being a safer option, it decreases the efficiency of data transfer, with effects more pronounced in low bandwidth networks. In this work, we try to improve the security-efficiency trade-off in the implementation of a VPN. To boost efficiency, we compress within the VPN so that the data might be available sooner at either end. Following compression, we use a padding scheme for traffic to hide user behavior, which attempts to maintain a fixed throughput irrespective of the compressibility of the data being sent or whether the user is active or idle. The VPN was tested using various data sets. 100 MBs each of Google, Facebook and YouTube data, which represent different degrees of compressibility of data (from most to least compressible). With compression enabled Google, Facebook and YouTube data transferred in 39%, 70.6% and 94.4% of the time it took to send it with compression disabled respectively, while maintaining a consistent throughput of approximately 6.3 megabits/second. These results clearly show that even with a fixed throughput, data transmission is more

efficient with compression enabled. The changes made in order to mitigate TA attacks led to improvement in overall traffic characteristics by hiding more information than before but still reveal some information.

Contents

Abstract iii

List of Figures vii

1 Introduction 1

1.1 Data Compression 1

1.1.1 Adaptive Compression 2

1.2 Internet and Security 4

1.2.1 SSL/TLS 5

1.2.2 VPN 6

1.2.3 Traffic Analysis Attacks 8

1.2.4 Side-channel Attacks 8

1.2.5 Our Work 10

2 Related Work 13

2.1 Compression over the Network 13

2.2 Adaptive Compression Tools 14

2.2.1 ACCENT 14

2.2.2 Datacomp 15

2.3 Traffic Analysis Attacks 16

2.3.1 Liberatore and Levine (LL) Classifier . . . 16
2.3.2 Herrmann, Wendolsky and Federrath (HWF) Classifier . . . 17
2.3.3 Keystroke analysis and Timing attack on SSH . . . 17
2.3.4 Pachenko, Niessen, Zinnen and Engel (PNZE) Classifier . . . 18
2.3.5 Countermeasure: Buffered Fixed-Length Obfuscator (BuFLO) . . . 18
2.4 Compression side-channel Attacks . . . 19
2.4.1 CRIME . . . 20
2.4.2 BREACH . . . 20

3 Implementation **23**
3.1 Hardware Setup . . . 25
3.2 VPN . . . 25
3.3 Packet Buffering . . . 26
3.4 Compression . . . 27
3.5 Padding Scheme . . . 28
3.6 Idle-time filling . . . 30
3.7 Experiments . . . 30

4 Results **32**
4.1 Data transfer times . . . 32
4.2 IO Graphs . . . 36
4.3 Packet size statistics . . . 39
4.4 Inter-Packet Arrival Times . . . 43

5 Conclusions **50**

Bibliography **55**

List of Figures

3.1 The Physical Experimental Setup 24

3.2 The Logical Experimental Setup 25

3.3 The padding Scheme: The figure shows the uncompressed buffer being compressed and padded before going on the wire. Note that the size of the buffer being sent is different from the uncompressed one. 29

4.1 Time taken to transfer the Random Data dataset with compression on vs. off 32

4.2 Time taken to transfer the YouTube dataset with compression on vs. off . . 33

4.3 Time taken to transfer the Fb dataset with compression on vs. off 33

4.4 Time taken to transfer the Wikipedia dataset with compression on vs. off . 33

4.5 Time taken to transfer the Reddit data dataset with compression on vs. off . 34

4.6 Time taken to transfer the Google dataset with compression on vs. off . . . 34

4.7 Time taken to transfer the Zero String dataset with compression on vs. off . 34

4.8 IO Graphs for Random Data dataset in B/s 36

4.9 IO Graphs for YouTube dataset in B/s 36

4.10 IO Graphs for Fb dataset in B/s 36

4.11 IO Graphs for Wikipedia dataset in B/s 37

4.12 IO Graphs for Reddit dataset in B/s 37

4.13 IO Graphs for Google dataset in B/s 37

4.14 IO Graphs for Zero String dataset in B/s 37
4.15 Inter-packet arrival times vs. time for Random Data dataset. 43
4.16 Inter-packet arrival times vs. time for YouTube dataset 44
4.17 Inter-packet arrival times vs. time for Fb dataset 45
4.18 Inter-packet arrival times vs. time for Wikipedia dataset 46
4.19 Inter-packet arrival times vs. time for Reddit dataset 47
4.20 Inter-packet arrival times vs. time for Google dataset 48
4.21 Inter-packet arrival times vs. time for Zero String dataset 49

1 Introduction

1.1 Data Compression

Data compression is the process of reducing the bytes required to transmit or store data. This can be done in one of the two general ways, namely:

1. Lossy Compression: In this method of compression some data is lost and cannot be retrieved after decompressing. The loss is justified as only the less critical data is lost. For example, finer details of an image or audio that might not be significant enough to noticed or perceived by a human, such as a slightly lower resolution over a narrower frequency than the original[1].

2. Lossless Compression: In this method of compression, no data is lost in the process and the data after decompression is identical to the original data that was compressed. An important limitation of lossless compression is that it is not universal, which means that it cannot reduce the size of all input data, only certain inputs. Algorithms leverage redundancy within the data, which they reduce using an encoding scheme. Thus the more random it is (i.e. less redundant) the less it will compress. On the other hand, it will compress a highly redundant file like a dataset with repeating numbers into a very small compressed file. Because our work requires perfect decompression, we use Lossless compression algorithms and hence we will not discuss lossy compression further in this work.

DEFLATE[2] is a lossless compression algorithm that is a combination of the LZ77[3] algorithm and Huffman coding[4]. In this algorithm, duplicate strings in the data are replaced by the "pointers" to their previous occurrences. Pointers are composed of length and offset information; DEFLATE uses Huffman coding to encode them. Thus, this algorithm adds its own metadata while compressing and if the data is not compressible at all, the compressed data might have a larger size than the uncompressed data due to the addition of metadata. Gzip is a popular compression scheme that uses DEFLATE.

Compression is useful for more than just saving space in persistent storage, it is also used when transferring data across networks. Network bandwidths and computational power are higher than ever before. With these resources Internet applications and content are moving towards data centers and the cloud, where people have remote access to these hardware and software resources. If anything, this has made compression more important than ever. Not only is it useful while storing data in the cloud infrastructure but also when data is transferred from the storage to user's device.

1.1.1 Adaptive Compression

As just described, data compression has become a very integral part of network communication. Compression can save time and bandwidth during communication but only when the total cost of compressing and sending data is lower than the total cost of sending it uncompressed. Therefore it is possible that compression degrades the overall performance. No single compression method is the optimal method to compress all types of data. For example, LZO is faster than bzip2 but achieves worse compression. Manually controlled compression choices in real systems may result in suboptimal performance, as they might not take into consideration all factors and not be as responsive to changes. Adaptive Compression (AC) systems make these choices dynamically to obtain optimal results. AC

systems typically use methods to determine or estimate the best compression choice for a given opportunity. For example the size of the input is a factor, as a larger file has a higher chance of having redundancies and hence more compressibility.

Datacomp[5] is one such AC tool. It is a local AC system that makes a decision about compressing the data on the basis of a sampling compressibility estimation algorithm, where the data streams are sampled to decide their compressibility. It uses a process called byte-counting to decide if the data should be compressed or not. Bytecounting heuristically estimates the uniformity of an input's symbols by counting the number of times each symbol appears more than a threshold in a given input. If l is the number of characters in the input and n is the number of possible symbols (256 for 8-bit bytes), the threshold t is l/n, or the number of times the character would appear if all characters appeared equally. The greater number of bytes that are "counted" (i.e., appear more than t times in the input), the more uniform -- and thus less compressible -- is the input. Bytecounting is computationally inexpensive in that it requires only a single pass over the input, only one division operation (to compute t) and can be used on a sample of the input. It has a wide array of compression algorithms to choose from for different types of data, including "No Compression", which can be seen as fastest but poorest compression choice. Datacomp was intended to replace system call functions like read(), write(), send(), and recv(). These functions take an input and its length and return the number of bytes that they wrote / read / etc. The return value can always be less than the input size, and it is the caller's responsibility to re-send/receive the remainder of the data. This behavior doesn't work for a VPN, where the receiving size must get all of an encapsulated packet or message in order to properly respond (e.g., to acknowledge the packet). Thus, Minicomp will always either compress the entire input, or none of the input. But it will never process only part of the input. We wanted a tool that could be used through a function call on a data buffer from within our VPN so that we could buffer data and make appropriate modifications to it before compressing it.

Minicomp is a stripped-down version of Datacomp which simply uses the compressibility estimation to choose between Gzip-1 or no compression. As a result, it has a lower computational overhead than Datacomp and does not have to go through any learning phase.

1.2 Internet and Security

The birth of the Internet as we know it today took place in the form of Advanced Research Projects Agency Network (ARPANET) , originally designed to allow scientists and researchers to collaborate through data sharing and access to remote computing resources. With time this group of collaborators grew and so did the infrastructure. Since the network was limited in its capabilities and was catering to a very limited collaborative and trusting audience, security was not a great concern. Eventually, the data networks originally restricted to researchers and government contractors started to be opened for public use. While the level of trust changed, the technology didn't. Internet users woke up to the notion of security threats when in 1988 The "Morris Worm" exposed the vulnerabilities of the network and how much damage could be caused if these were not fixed.

In the 1990s, with the birth of the World Wide Web and widespread dial-up access Internet use increased greatly amongst the general public. With this increase in the number of users and sensitive information on the web, requirement for security measures was great. One of the initial defenses put up were firewalls, to provide protection to all the systems on a given network at the same time. Other defenses include the various antivirus programs meant to detect malicious code and remove it. With time attacks became more sophisticated and defenses evolved to keep up.

The aforementioned methods did not solve the problem in its entirety as the data on the Internet was still vulnerable to confidentiality and integrity threats due to modification and interception while it was on the untrusted public network. The solution to this problem was

cryptography. Although cryptographic algorithms were known for a long time they were not widely used over the Internet because the threat of interception was seen as being low compared to the CPU cost of encryption. With the development of asymmetric encryption, cryptography became even more of a feasible option in a way. The reason was that even though asymmetric encryption is computationally more expensive than symmetric ones, it led to the birth of Public Key Encryption (PKE) infrastructure which forms the backbone of the security of the Internet today. The integrity and confidentiality needs of the Internet led to the development of protocols that enable encryption for unencrypted communications, such as SSL/TLS. The various older protocols like HTTP were updated using SSL/TLS to incorporate encryption. Encryption was successful in providing data integrity and confidentiality over the network. These advances made the idea of a Virtual Private Network possible. If the data transmitted over the network could be protected through encryption from "end-to-end", a private network could be simulated over a public one.

1.2.1 SSL/TLS

The SSL (Secure Socket Layer) and TLS (Transport Layer Security) protocols provide "end-to-end" encryption; data privacy and integrity between both the parties involved in a communication over the network. They are designed to run over a reliable connection protocol like TCP and provide an abstraction similar to a traditional TCP socket. There are a few differences[6] between the older SSL and the newer TLS namely:

1. The keyed-Hashing for Message Authentication Code (HMAC) algorithm in TLS replaces the SSL Message Authentication Code (MAC) algorithm. HMAC hashes are more secure than the MAC algorithm.

2. In TLS it is not necessary to include certificates all the way back to the root Certificate Authority, instead an intermediary authority can be used.

3. The key derivation differs.

4. The list of cipher suites differ with TLS having a broader suite.

TLS and SSL provide secure HTTP (HTTPS) where the SSL/TLS connection establishment precedes the normal HTTP data exchange. SSL/TLS can also be used for other application level protocols like File Transfer Protocol (FTP), Simple Mail Transfer Protocol (SMTP) and Lightweight Directory Access Protocol (LDAP). They are also a basis for some implementations of Virtual Private Networks (VPN). The fundamental idea behind VPNs is that if the data transmitted over the network could be protected through encryption from "end-to-end", a private network could be simulated over a public one.

1.2.2 VPN

A Virtual Private Network (VPN) is virtual network that is built upon the existing networking infrastructure to simulate a secure, private network over a public network[7]. A VPN works by establishing a virtual Point-to-Point connection making use of dedicated connections, encryption and tunneling. There are two common types of VPNs[8]:

1. Remote Access - Also known as Virtual Private Dial-up Network (VPDN), this is a connection from a user-to-LAN meant to connect to a private network from remote locations.

2. Site-to-Site - These use dedicated equipments and encryption to connect to multiple fixed locations over a public network like the Internet.

In order to protect private information VPNs support different secure protocols, one of which is SSL/TLS.

Tunneling, by itself, does not provide security. If the original packet was merely encapsulated inside another protocol, it would be still be vulnerable to interception through

packet capture if not encrypted. Therefore, the encapsulation process in a VPN typically also includes encryption. VPNs like OpenVPN also have the option of compression, where each individual packet can be compressed before encapsulating and encrypting it. It also includes a rudimentary version of adaptive compression.

The use of compression and encryption over the Internet makes communication more efficient and secure, but these notions are not without their fair share of problems. With both techniques being used widely, there came a wave of attacks that try to subvert and exploit them for gains.

OpenVPN is an open-source SSL/TLS based VPN service. It can provide both point-to-point and site-to site connections. It can run over either User Datagram Protocol (UDP) or Transmission Control Protocol (TCP) in the transport layer. OpenVPN is of interest to us because it also has an option of packet compression in its configuration parameters. This compression can be static or adaptive. The adaptive compression in OpenVPN is a simple one where it periodically samples the data that it sends out to check the compression efficiency. If the data was already compressed, the efficiency would be low and compression would be turned off until the next sampling. This is a suboptimal approach for AC as the check is performed only after a fixed amount of time. This may lead to a poor performance if the traffic compressibility varies quickly as in the case of websites with different content-types. Another important thing to note about the compression in OpenVPN is that it is done on a per-packet basis. So only one packet is compressed at once. This is comparatively a sub-optimal approach compared to compressing multiple packets at a time, as data in it would have a higher probability of being more redundant, leading to a better compression ratio.

1.2.3 Traffic Analysis Attacks

Traffic Analysis (TA) attacks are inference attacks that involve observing data patterns over a network to infer information about communication's content. It is naively assumed that communicating over an encrypted tunnel interface is sufficient to deter any attack on the data, but even going through encryption does not hide the length and number of plaintexts being encrypted and sent[9]. This seemingly harmless information enables TA attacks that can reveal the websites visited and even specific activities over those websites[10, 11, 12, 13].

These attacks leverage varying properties of traffic over the network, and are often classifiers attempting to identify the sites visited among a list of websites. We name a few such classifiers and the properties they leverage.

1. Liberatore and Levine (LL) Classifier
2. Herrmann, Wendolsky and Federrath (HWF) Classifier
3. Keystroke analysis and Timing attack on SSH
4. Pachenko, Niessen, Zinnen and Engel (PNZE) Classifier

1.2.4 Side-channel Attacks

Side-channel attacks exploit information leaked from physical components of a cryptosystem, like power consumption, processor usage, timing patterns. With the development of cryptographic algorithms robust to more traditional forms of cryptanalysis, side-channel attacks have gained popularity. A classic example is the CIA's "TEMPEST"[14] class of attacks in which leaky RF signals broadcast from teletype devices could be translated into keystrokes by an eavesdropper. An attack using the same principle can be traced back to

World War II. This attack can be executed on a target from miles away, making it even more dangerous.

In recent years, a new threat of side-channel attacks have emerged that exploit the software implementation rather than theoretical vulnerabilities in the cryptographic algorithms. One class of such attacks are the attacks on combinations of compression and encryption. Data is always compressed before encryption as encryption randomizes data, which would make it incompressible. Intuitively, using compression along with encryption would seem to have no effect or in some cases enhance security of a communication. But this has been proven wrong through attacks exploiting the problem of information leakage inherent in data compression. A lot can be known about the contents of a sample of data by its compressibility. By exploiting this flaw in compression, attackers are able to figure out the language used for communication over a compressed encrypted Skype session[15]using the varying compressibility amongst languages. A deeper inspection using speech processing algorithms can even reveal the words being used in the communication.

A side-channel exploiting data-compression algorithms making use of the sizes of input and output through the compressor is presented by John Kelsey[16]. These attacks exploit the way in which most compression algorithms work; higher redundancy in data leads to a better compression. A wide spectrum of attacks ranging from passive to chosen plain-text attacks were covered and the results were presented, with the chosen plain-text attack showing a 70% success rate for pseudo-randomly-generated strings compressed using the Zip DEFLATE compression algorithm. More recently, two proof of concepts have made compression side-channel attacks in the real world a possibility, we talk about them as they are the most relevant to our work.

1. CRIME: CRIME is a compression side-channel attack[17] on HTTP requests that leverages information leak through their compression ratios facilitating the attackers

to decrypt these requests. Through this they get access to the user's login cookie which in-turn allows them to hijack user's sessions. Enabling TLS or SPDY compression is necessary for the attack to take place, so, the browsers implementing them were vulnerable to this attack.

2. Browser Reconnaissance and Exfiltration via Adaptive Compression of Hypertext (BREACH): This attack also targets HTTP traffic like CRIME but instead of requests, it targets responses. This attack leverages information leakages through compression ratio of these responses to reveal sensitive information within the responses like the Cross-Site Request Forgery (CSRF) token. This attack works against HTTP-level compression (the most common compression scheme being gzip.), unlike CRIME's attack on TLS compression.

These attacks on compression have led to applications avoiding compression altogether, because developers have started seeing compression as a security liability for network communications. For example, TLS compression has been deprecated in the latest versions of TLS. The problem with this broader approach is that it is not solving the problem but taking an easier way out of it. This comes at a cost; with compression disabled, there is a loss of efficiency of data transfer over the Internet. Instead, through algorithmic tweaking of compression in applications it can be made more secure and our communications can be both efficient and secure.

1.2.5 Our Work

The work we did in this thesis aims at making compression more secure and hence no more a liability. In our work we implemented a TCP based VPN that had adaptive compression built in it for the packets along with a padding scheme. A special feature of this VPN is that it does not perform packet compression on just each packet alone. Our

algorithm can collect multiple packets in a buffer and then compress the buffer to get a better compression ratio, which results in the receiver obtaining the packets earlier than they would otherwise.

In the VPN, the data compression intended to improve the throughput efficiency, while padding was included to add length hiding to the compressed packets. After the completion of these steps, the data would be encrypted to hide the plaintexts and hide the difference between the padding vs actual data. This method brings to light the efficiency-security trade-off as the system's efficiency was not as high as it would be with just compression (without any length hiding) but it was more secure than a compression-only system. One more security feature in the VPN is that we send junk data at the constant connection bandwidth even when there is no actual data being exchanged. This hides if the user is active or idle. Our implementation of hiding traffic characteristics was not perfect, but is a significant improvement.

We collected browsing sessions of very popular websites (Google, Facebook, Reddit, YouTube, Wikipedia) along with some synthetic data types, namely a file with just the character '0' and a file with random data stream to test the extreme cases of compressibility of data.

Our experiments consist of a three system setup.

1. A VPN client (Alice)
2. A VPN server (Bob)
3. A Man in the Middle (Eve)

All the data exchanges took place between Alice and Bob with Eve acting as an invisible network bridge. We transferred data blobs between the server and client using the netcat[18] tool.

We measured time taken for transferring the blobs with compression turned on and off. We also captured the traffic dump at Eve so that we could examine traffic characteristics used in Traffic Analysis attacks.

The next chapter is Related Work, which talks in detail about the works related to our thesis. These include tools that we used in our work, and previous works that we get some of our ideas from and build upon. After that is the Implementation chapter where we discuss in depth about the details of the VPN we implemented. After that we discuss the results of the experiments in the Results chapter. In the end we conclude with the Conclusion chapter by talking about the things we were able to achieve in this work and all the future work that can be done based on this.

2 Related Work

2.1 Compression over the Network

With the increase in the amount of data typically sent and current bandwidth limitations, compression has become an important tool for network communication. Data Compression reduces the size of the data frames to be sent over a network. This reduction of size reduces the time required to send this data and conserves the bandwidth consumed by each user. The compression can take place on different network layers depending on the requirements of the communication. Two such compression frameworks relevant to our work are :

1. TLS compression - TLS included an option to select a lossless data compression method as a part of the TLS Handshake Protocol and this algorithm was then applied as a part of TLS Record Protocol. Compression methods with the ability to maintain state/history information in order to achieve a better compression may also be used as TLS along with the protocols in the lower layers ensure reliable and sequenced packet delivery. Hence both, per packet compression and compression over multiple packets are allowed.

2. HTTP compression - Built into web servers and web clients (as part of the HTTP standard), HTTP compression helps to increase bandwidth utilization. Using it, HTTP responses are compressed before being sent from the server to the client or vice versa. Browsers negotiate the compression method before the compression takes place, and browsers not supporting compression download uncompressed data. The

most common compression methods supported are DEFLATE and gzip (which uses DEFLATE). Compression can take place in two different ways. At a lower level, a Transfer-Encoding header field indicates that the payload of the HTTP message is compressed. At a higher level, a Content-Encoding header field indicates that a resource being transferred is compressed. Obviously, HTTP compression only works for HTTP traffic, however other protocols (such as SSH) include compression.

2.2 Adaptive Compression Tools

The notion of Adaptive Compression (AC) has been introduced in the previous chapter. In the process of our work we researched a few AC tools. These were instrumental in providing important insights that led us to the tool that we designed and finally used in our work. We talk about two such important tools.

2.2.1 ACCENT

Adaptive Cryptography Plugged Compression Network (ACCENT)[19] was designed for SSL/TLS-based cloud services. These cloud services include mobile offices, Web-based storage services, and content delivery services. These run workloads under various device platforms, networks and cloud service providers. ACCENT solves three mismatches pertaining to the emerging cloud services running on top of SSL/TLS. The first being the performance of loosely coupled encryption and communication routines that lead to underutilized computation and communication resources. Second, the conventional SSL/TLS provides no AC, but only a static compression mode. Third, there is a memory allocation overhead due to frequent compression switching in SSL/TLS. To solve these issues, ACCENT uses the following three mechanisms:

1. Tightly-coupled threaded SSL/TLS coding (TTSC): TTSC maximizes the bandwidth and processing power used in the process of SSL/TLS sending and receiving data. Through tight coupling, they avoid various blocking and wake-up operations that would otherwise be encountered leading to a sub-optimal resource utilization.

2. Floating scale-based adaptive compression negotiation (FSCN): FSCN is ACCENT's adaptive compression mechanism. It involves floating scales (dynamic parameters) of the sender and receiver that vary dynamically according to the compression and encryption data rate and the compression ratio.

3. Unified memory allocation for seamless compression switching (UMAC): UMAC minimizes the overhead associated with switching compression algorithms by unifying their memory footprints by automatically transforming memory layouts.

Though ACCENT came pretty close to the adaptive compression algorithm we might have used, we could not use it as it did the compression in the SSL/TLS layer and we needed to perform compression within our VPN which would have come after that. Another reason for not using ACCENT was that it had compression buffer and dictionary ranging from 16KB to 453.3KB. We wanted the compression buffer size to be a configurable parameter in our VPN. There was also no length hiding in ACCENT by default. Length hiding could have been added but that would have changed the TTSC mechanism entirely, which would have degraded ACCENT's performance.

2.2.2 Datacomp

Datacomp monitors four environmental properties for its adaptive compression mechanism that decides whether to compress the data or not and which compression algorithm to utilize: CPU Utilization, CPU frequency, data type and available bandwidth. The number

of different combinations of the values of these factors can be very large; to cope with this Datacomp quantizes these values to reduce the set of combinations to a manageable number. To evaluate the performance of Datacomp, it is compared to the performance results of Comptool, which is an Adaptive Compression Oracle that can empirically determine the best compression strategy for a given scenario. The results show that in the range of 1-100 Mbit/s and a range of realistic data types, Datacomp's strategy produced results statistically similar to the best-case result or extremely close to it. However it fails to break even at one Gbit/s as the cost of computation for computation and adaptation is much higher than sending the data uncompressed over the high bandwidth network.

We didn't use Datacomp proper in part because it had a time consuming learning phase, which was not desired in our VPN. The main factor though, was that Datacomp performed compression within a system call, but as stated above we needed it to be done within our VPN itself. Therefore the adaptive compression method we ended up using was the brand new Minicomp which is a simplified Datacomp with fewer features but less overhead and developer burden.

2.3 Traffic Analysis Attacks

We introduced TA attacks in the previous chapter. These attacks demonstrate the fact that encryption is not the "silver bullet" for privacy. Even with the seemingly invincible tool of encryption, more sophisticated attacks can be mounted that reveal user behavior over the Internet. A few such attacks are described below

2.3.1 Liberatore and Levine (LL) Classifier

This classifier[20] attempted to identify, using a Naive Bayes (NB) classifier with packet lengths and directions as features, which one of a set of web pages was visited. In their

work, Liberatore and Levine built two different systems, a naive Bayes classifier and one based upon Jaccard's coefficient (a similarity metric). They calculated the probabilities of the traffic belonging to a particular website. They had a list of 2,000 websites in total and they used a varying number of websites from the list in the experiments and assigned them labels. The classification was done based on the vectors of the features being used, which in their case were obtained from the count of the packet lengths sent in each direction. The label with the highest probability was selected as the guess of each system.

The naive Bayes classifier had an accuracy of 86.2% and the Jaccard's coefficient classifier had an accuracy of 88.9%. The classifiers worked for the traffic both before and after encryption.

2.3.2 Herrmann, Wendolsky and Federrath (HWF) Classifier

This classifier[21] uses a Multinomial Naive Bayes (MNB) classifier for traffic identification. MNB in this case estimated the probability of the traffic belonging to a website based on the aggregated frequency of the features over all the vectors. The system used normalized counts of the direction and length of the packets. Their list of websites were the 775 most visited domain names between January 2007 and September 2007.

Their classifier had an accuracy of approximately 97%.

2.3.3 Keystroke analysis and Timing attack on SSH

This exploit[22] uses Hidden Markov Models (HMM) constructed upon the keystrokes. In this attack the authors showed how a simple statistical analysis revealed sensitive information like the length of the user's passwords. With slightly more sophisticated analysis on timing information, they were able to learn about what the users were typing in SSH sessions.

The typing patterns were studied by developing a Hidden Markov Model (HMM) through which they could predict key sequences using the inter-keystroke timings. They further developed a system named Herbivore, which reduced the time taken to brute force user passwords by a factor of 50 just by collecting timing information on the network.

2.3.4 Pachenko, Niessen, Zinnen and Engel (PNZE) Classifier

Low-latency anonymization networks like Tor are considered to be one of the safest channels of communications for ordinary citizens to maintain their anonymity over the Internet. The PNZE classifier showed how the security in such networks is not perfect with some glaring flaws as they fail to resist website fingerprinting attacks.

For the classification the authors used a Support Vector Machine (SVM) classifier with features including packet lengths, ordering, total number of packets transmitted, total number of bytes transmitted and proportion of packets in either direction. Their classifier had an accuracy of 55%.

2.3.5 Countermeasure: Buffered Fixed-Length Obfuscator (BuFLO)

In their work[9], Dyer et.al work on a countermeasure to the aforementioned classifiers. It was designed to send fixed-length packets at a constant interval for a fixed amount of time although traffic is allowed to go over that fixed time limit, still with the fixed-length packets and fixed intervals.

BuFlO is governed by three integer parameters d, ρ and τ where:

1. d determines the size of the fixed-length packets.

2. ρ determines the rate (in milliseconds) at which the packets are sent.

3. τ determines the minimum amount of time (in milliseconds) for which the packets are sent.

Therefore it will send a packet of length d every ρ milliseconds until the communication is over or τ seconds have passed, whichever is greater. When there is no data in a buffer, dummy data is sent.

BuFLO was evaluated empirically with parameters in the ranges of $\tau \in 0, 10000$, $\rho \in 20, 40$ and $d \in 1000, 1500$. The least bandwidth-intensive experimental configuration was with $\tau = 0$, $\rho = 40$ and $d = 1000$. In this case the average accuracy of their classifier was 27.3%, compared to a 97.5% average accuracy without any countermeasure. The most bandwidth-intensive experimental configuration was with $\tau = 10000$, $\rho = 20$ and $d = 1500$. The accuracy here was 5.1%.

Through experiments they determined that BuFLO still leaked the total bytes transmitted and the time required to transmit a data set in a couple of cases:

1. The data source continued producing data for more than τ time.

2. The data is not produced beyond τ, but there is still data left in the buffer at time τ.

2.4 Compression side-channel Attacks

As their name suggests, compression side-channel attacks are a special class of side-channel attacks that rely on information leaked though the compression ratios of different input. We talk in detail about the two such the mitigation of which were the main focus of this thesis as we worked to mitigate them.

2.4.1 CRIME

In the year 2012 Rizzo and Duong revealed an attack on HTTPS called CRIME. This attack could be used to recover headers of an HTTP request mainly targeting the secret key in the authentication cookie. CRIME works only when the browser and server both support TLS compression or SPDY, an open networking protocol used by Google and Twitter. In a CRIME attack, the compressed and encrypted message is combined with JavaScript controlled by the attacker to perform a letter by letter brute-force attack on the secret key.

Once the first character is guessed correctly, this process is repeated again on the next character in the key until the remainder of the secret is deduced. In order to mitigate this attack TLS compression and SPDY need to be disabled on the browsers and the servers. Consequently major browsers that previously supported TLS compression and SPDY released patches to disable the vulnerable features.

2.4.2 BREACH

Another attack named BREACH was revealed a year later which was built off the CRIME attack. This attack does not rely on TLS compression but instead leverages common HTTP compression mechanisms such as Gzip and DEFLATE.

BREACH attack targets HTTP responses instead of HTTP requests. It is very common to use Gzip at the HTTP level, even if TLS-level compression is disabled. Secret strings such as Cross-Site Request Forgery (CSRF) tokens are included in the same HTTP response as the user input and are compressed in the same compression context. DEFLATE (the basis for Gzip) shrinks the payload by taking advantage of the repeated strings, thus an attacker can use the URL parameters in the response to guess some token, one character at a time.

Suppose request and response contains a string like 'canary='. In this case the value after canary is a CSRF token. In this attack the guess value is inserted like 'canary=<guess>'

so the 'canary=' string is repeated and if the first character of the guess matches the first character of the actual token the DEFLATE would compress the body even more. In this way the attacker proceeds to recover the secret token byte-by-byte. In the experiment by the developers of BREACH they could recover the CSRF token for Microsoft's Outlook Web Access with an accuracy of approximately 95%, and often within 30 seconds[23]. This attack is facilitated if the response remains mostly the same except for the attacker's guess, as the difference in size of the response is normally very small (on the order of bytes) and hence is susceptible to noise. This attack can work against any cipher suite, but the attack is simpler if a stream cipher is used as the difference in sizes between responses is more granular.

The attacker's ability to view the victim's encrypted traffic requires some kind of Man in the Middle access and could be achieved through ARP spoofing. The attacker must also be able to make the victim send HTTP requests to the vulnerable web server. This is accomplished by making the victim visit a site controlled by the attacker that contains an invisible Iframe pointing to the vulnerable server. This is an active, online attack and so the attacker makes the victim send a small number of requests continuously in order to guess the password byte-by-byte.

A few mitigations for this attack are:

1. Hiding the length by inserting random garbage value as padding, but this only increases the complexity by a factor of $O(\sqrt{n})$[24]. Here n is the number of padding bytes added.

2. Putting secret values in a completely different compression context than that of the user input, but this requires special, uncommon software.

3. Disabling HTTP compression, but this hurts efficiency.

4. Limiting and monitoring the HTTP request-rate.

As it is evident by this section, our work in this thesis combines the network primitives like compression and encryption in a way that strives to be both efficient and secure. Our VPN tries to make the data resistant to compression side-channel and TA attacks by adding length hiding and manipulating the data stream. The resources we mentioned above played a vital role in shaping this work into its current form.

3 Implementation

The experimental setup primarily includes three systems: Alice, Eve and Bob. All the three devices are on the same Local Area Network (LAN) with Internet access. The interface of interest to us is the one between Alice-Eve and Eve-Bob. In this setup Eve acts as an Ethernet bridge between Alice and Bob. This setup aims at simulating a real life scenario where Bob is one of the servers on campus of a company and Alice is the PC client of an off-site employee connected to Bob through a VPN.

Eve is a silent interceptor in this connection, which we use to monitor and analyze the traffic between the server and client. All the VPN traffic flows through this connection. It acts as a Ethernet bridge between the server and the client, which technically is a Man in the Middle. It is used to collect network traffic information between the client and the server as the bridge in this case represents how the data traffic would be on the wire in between the client and the server.

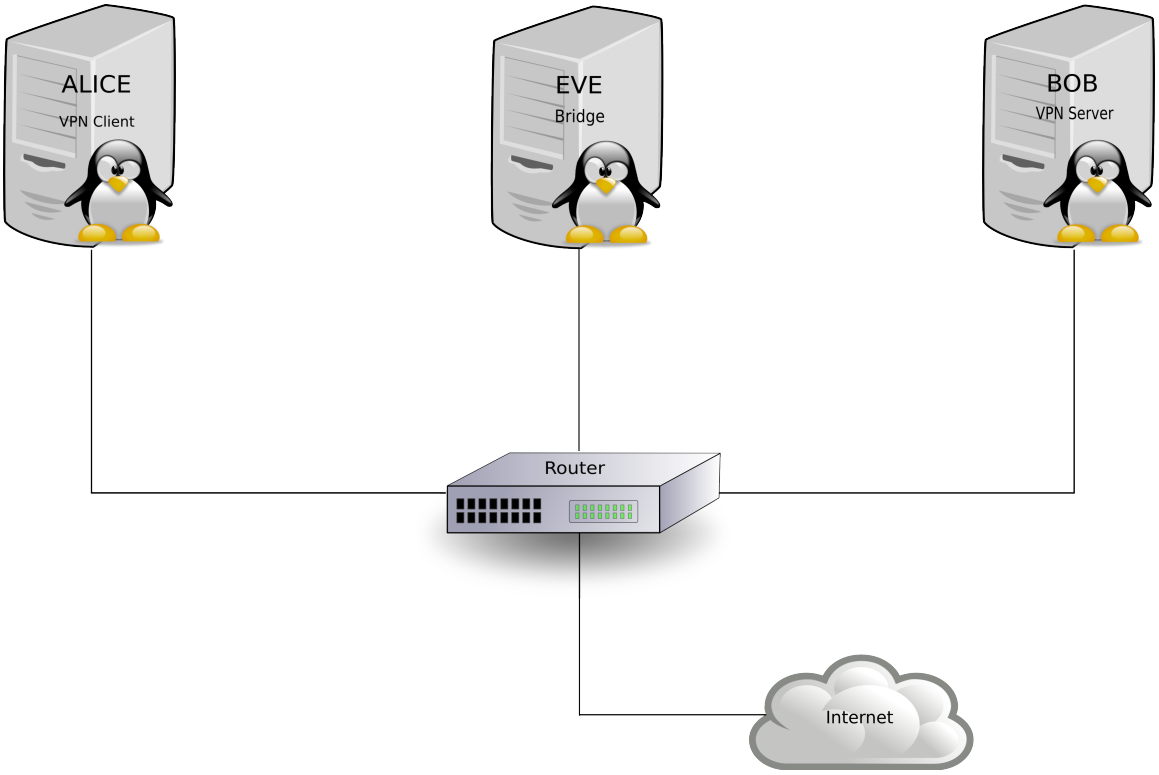

Figure 3.1: The Physical Experimental Setup

3.1 Hardware Setup

Eve is a HP Compaq 6005 pro PC with AMD Phenom II X2 B55 Dual Core Processor. Bob is HP Elite 7100 MT with Intel Core i3 530 Dual Core Processor. The RAM for both these systems is 4 gigabytes and have gigabit network interface cards. Alice is a Dell XPS 13 Ultrabook Intel Core i5-6200U Quad Core Processor with 8 gigabytes of RAM and 867Mbit network interface card. All the systems run Ubuntu 14.04LTS.

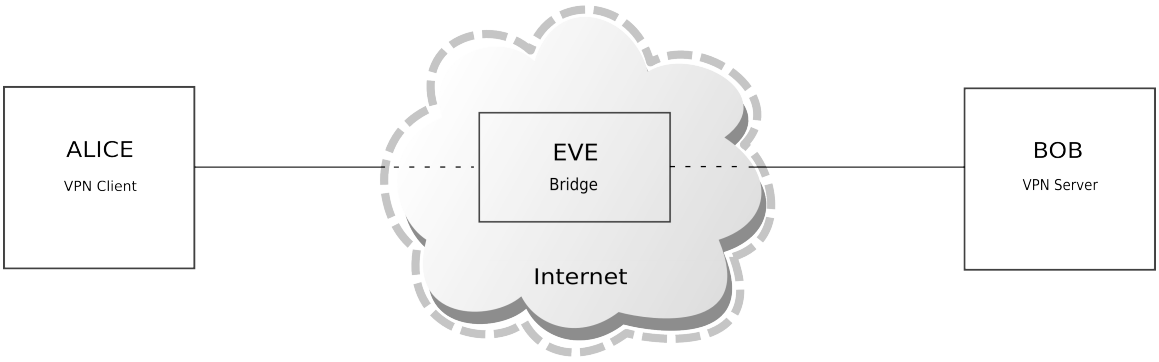

Figure 3.2: The Logical Experimental Setup

3.2 VPN

The first step in the implementation process was to set up a functioning VPN between two systems. The VPN needed to be open source since we intended to modify it to include our own algorithm in it. OpenVPN was considered at the beginning but making changes to it was too complicated, so we decided to go for a simple "bare bones" VPN as it would not affect he value of this research. We found an existing project on GitHub called S-VPN[25] that matched our requirements. S-VPN is a UDP-based VPN. For encryption, S-VPN uses a simple substitution cipher algorithm. It has no built-in compression functionality.

3.3 Packet Buffering

The first step was to add the capability to buffer multiple packets as it would mean more context for compression hence leading to better data compressibility. The size of the buffer was a configurable parameter. In case the number of packets available were not enough to fill the buffer, there was a configurable timeout parameter. This ensured the accumulated packets would be sent out after a fixed time even if the packets did not fully fill the buffer. Before adding each packet to the buffer, a 32-bit header containing the packet length was added to each packet. This information was needed at the receiving ends, as the packets had to be written to the tunnel interface individually. The first packet of the entire buffer had an additional 64-bit header that had the total length of the accumulated packets in the buffer and the padding (which is discussed in a later section). This header information was critical as it gave the receiving end the needed information for reassembling the buffer as the entire buffer might not be sent out as one chunk (it may need to be broken into smaller packets to be sent over the wire).

Since this 'beacon packet' had to be the first packet to be read at the other end before the remaining buffer, it was important for the packets not to be out of order. We could not ensure this with UDP because it is not a reliable protocol. So, we switched to TCP instead, because it is a reliable protocol ensuring that packet order is maintained. This order was also important as once we started using compression, because if any piece of the compressed buffer was out of order during reassembly, the decompression at the other end would have given an inconsistent result.

We used a fixed length buffer of 63,712 bytes. A bigger buffer of one or two megabytes was tested too, but it took too long to accumulate and compress that many packets. Moreover, the buffer wouldn't get full because of the TCP window-size constraint (where TCP only allows a given number of packets to be sent over the wire before getting an acknowl-

edgment for the receipt of a packet from the receiving end). Additionally, we weren't making use of the bandwidth optimally, as there was a longer idle time for any data to be put on the wire. We tried a few buffer sizes and 63,712 bytes gave the best performance out of those. 63,712 seems like an odd number to choose, but it was done because Ethernet allows a maximum segment size (MSS) of 1514 bytes out of which 1448 bytes is the payload and the remaining 66 bytes are the headers. For length hiding, we needed to only send packets of a fixed length. Since the buffer size is a multiple of 1448, this ensured that when it was fragmented into individual packets and sent, they would all be of equal sizes.

3.4 Compression

Once the buffering was working, we added the compression functionality to the VPN. In order to add compression, we used a tool named Minicomp. As mentioned before, this was a watered-down Datacomp. Essentially, works similarly to the memcpy() call in C, but instead of simply copying a buffer from one location to another, it adaptively compresses the source buffer and copied it to the destination buffer.

Minicomp added its own 32-bit header to the buffer. The header had the following information in it:

1. a magic number
2. Whether the buffer was compressed
3. The uncompressed size of the buffer
4. The compressed size of the buffer.

Minicomp chose between two options, either to compress the buffer using Gzip-1 or not to compress at all.

3.5 Padding Scheme

After compression, we added a padding scheme to our VPN. For padding, we filled the buffer with the '0' bit. Any other character or group of characters could have been used without making a difference to our implementation as the padding was simply ignored on the receiving side. We tried a few different padding schemes to go along with the buffering and compression algorithms:

1. Initially, we tried filling the buffer with packets, compressing them, filling the remaining length of the 63,712-byte buffer with padding and sending it to the other end. Unfortunately, this didn't give our VPN much of a performance boost compared to the uncompressed buffered packets, as even though compression reduced the size of the buffered packets (as a result of which the valid data reached the other end faster), a lot of padding was being sent along with it that had to be ignored. Hence, this approach was not using the bandwidth efficiently.

2. Next, we tried going through multiple iterations of filling the buffer with packets and compressing. In this approach, after compressing the filled buffer once, we filled the remaining length of the buffer with packets again and compressing them. This was done until there were no more packets to be read from the application and we had a timeout, or the buffer was full. In case of a timeout, the remaining empty buffer was filled with padding. This approach also gave a sub-optimal performance, as the multiple iterations of reading packets from the application and filling them in the buffer was too costly. The TCP window-size only allowed approximately 80 kilobytes of packets to be read, which was too little to fill the buffer after being compressed. This caused the VPN to sit idle waiting for the timeout to occur. With this extra time spent idle waiting for a timeout and the time spent compressing, this version performed worse than when uncompressed data was being sent out.

3. In the scheme that currently use, the buffer is filled with packets and compressed once. Then if the new size of the compressed buffer was not a multiple of 1448, we just added minimum amount of padding to make it a multiple. Due to this, the header format for the beacon packet was 64 bits long, where the the first 32 bits were the length of the compressed valid data and the other 32 bits were the length of the padding added. This information was needed as the total buffer size that was fragmented and sent was not constant. This method was chosen as it gave a much better performance when compared to uncompressed data while still keeping the packet sizes constant.

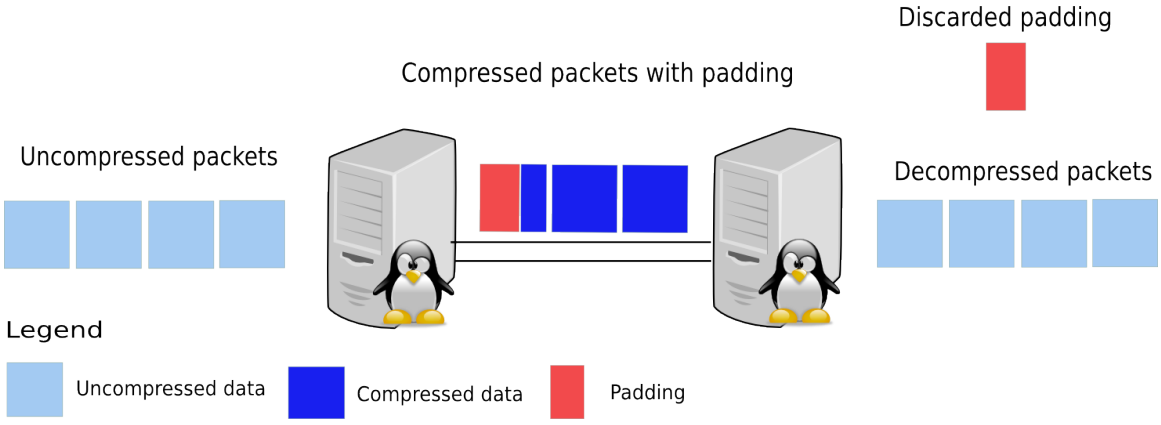

Figure 3.3: The padding Scheme: The figure shows the uncompressed buffer being compressed and padded before going on the wire. Note that the size of the buffer being sent is different from the uncompressed one.

It should be noted that the experimental setup had a high bandwidth of about 200Mbits/sec. At this bandwidth, compression starts losing it's advantage of being faster than uncompressed data because of the compression throughput of the CPU. Since our VPN is more appropriate for lower bandwidth connections (like mobile networks and Internet in developing nations), we artificially reduced our bandwidth to 6Mbits/sec. All the experiments were performed at this bandwidth.

3.6 Idle-time filling

An important feature that we added to our VPN was idle-time filling; we sent out junk data when the network would have otherwise been idle. The intention of this was to have a constant bandwidth of traffic over the network so as to hide Alice's usage pattern from any eavesdropper. We used 2,896 bytes of buffers filled with the character '0' as the junk data. This buffer was sent out whenever there was nothing to read on the interface from the application or from the other side. The size of junk buffer was smaller when compared to the buffer for valid data, as we didn't want the VPN to wait for sending out a large buffer if there was actual data ready to be read from an interface in the middle of the process of sending out junk data. The junk data was appropriately timed so as to imitate the bandwidth behavior of the valid data.

3.7 Experiments

For testing our VPN, we used seven different datasets:

1. a file with a string of zeros in it (Zero String) from /dev/zero/
2. a file with a string of random characters in it (Random Data) from /dev/urandom
3. a trace file with data from browsing Google
4. a trace file with data from browsing YouTube
5. a trace file with data from browsing Reddit
6. a trace file with data from browsing Wikipedia
7. a trace file with data from browsing Facebook

All the traces were 100 megabytes each and were created by capturing browsing sessions on the websites mentioned above. This was done through the Network Monitor feature in Mozilla Firefox[26] browser, which allows capturing uncompressed and unencrypted browsing sessions. Another significance of these files is the different degrees of compressibility of each. The table below highlights these differences.

File Name	Compressed Size (MB)	Compression ratio
Random Data	100.01	1.000
YouTube	93.68	0.937
Facebook	68.07	0.680
Wikipedia	48.05	0.480
Reddit	47.31	0.473
Google	36.38	0.364
Zero String	0.43	0.004

Table 3.1: Compressed file sizes and compression ratios for the different datasets when compressed using Gzip-1

For the experiments these files were transferred from the client to server using Netcat[27] in the TCP mode. This was repeated for every file 15 times with adaptive compression on and 15 times with all compression off. Each file transfer was preceded and followed by 20 seconds of idle-time filling traffic. The network traffic was captured at the bridge interface (Eve) between the client and the server using Tcpdump[28].

In the next section we present the various traffic characteristics from these experiments.

4 Results

4.1 Data transfer times

One of the main goals of this thesis was to incorporate compression in the VPN to achieve lower data transfer times when compared with a VPN without compression. As mentioned in the previous section we ran 15 tests each with compression on and off for all seven datasets. The comparisons between total time taken for each dataset to transfer through the VPN, (with compression on and off) are presented below.

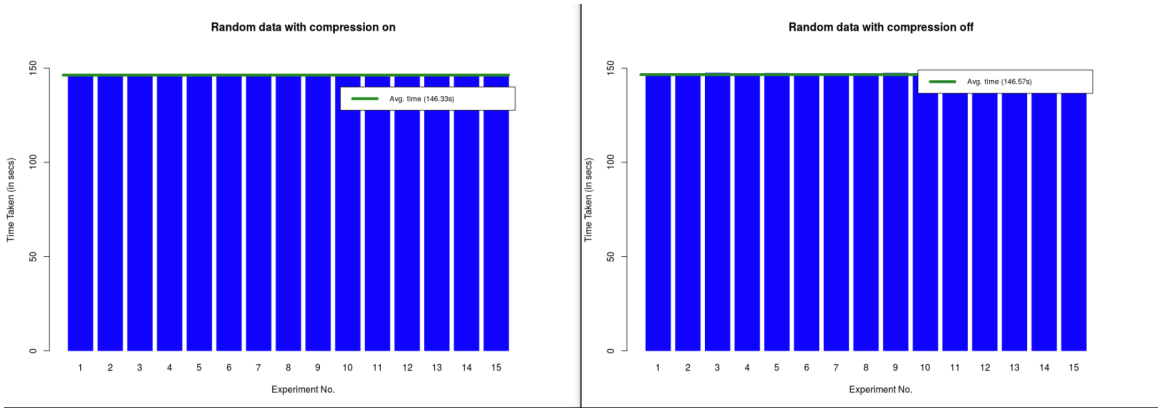

Figure 4.1: Time taken to transfer the Random Data dataset with compression on vs. off

We notice that the transfer times with both compression on and off were almost equal for the Random Data dataset and was very close in case of YouTube. The reason for this is the incompressibility of the datasets. In these cases Minicomp correctly chose to not compress the data. Facebook, although not the most compressible datasets, did benefit from compression with an average transfer time of 103.98s compared to 147.01s without

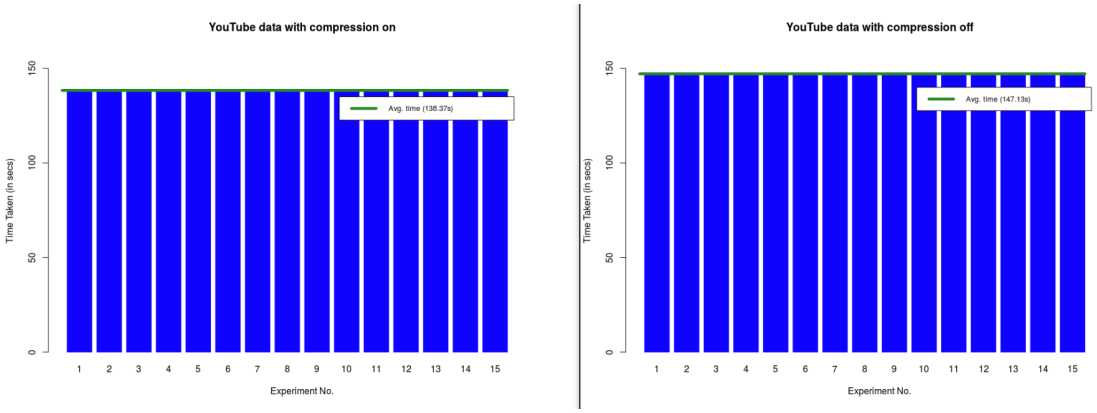

Figure 4.2: Time taken to transfer the YouTube dataset with compression on vs. off

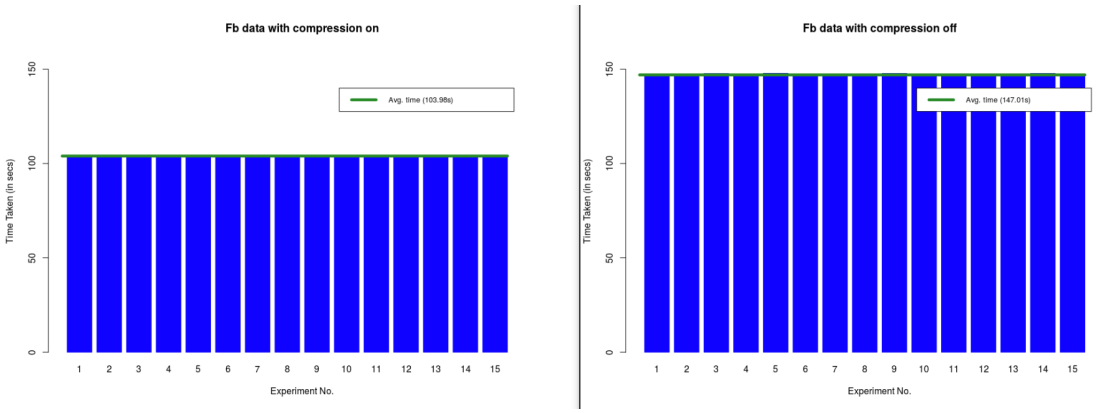

Figure 4.3: Time taken to transfer the Fb dataset with compression on vs. off

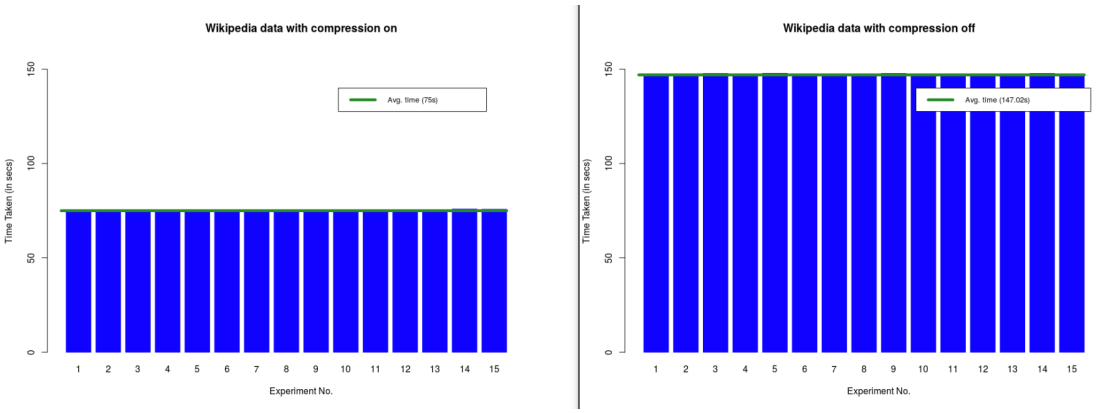

Figure 4.4: Time taken to transfer the Wikipedia dataset with compression on vs. off

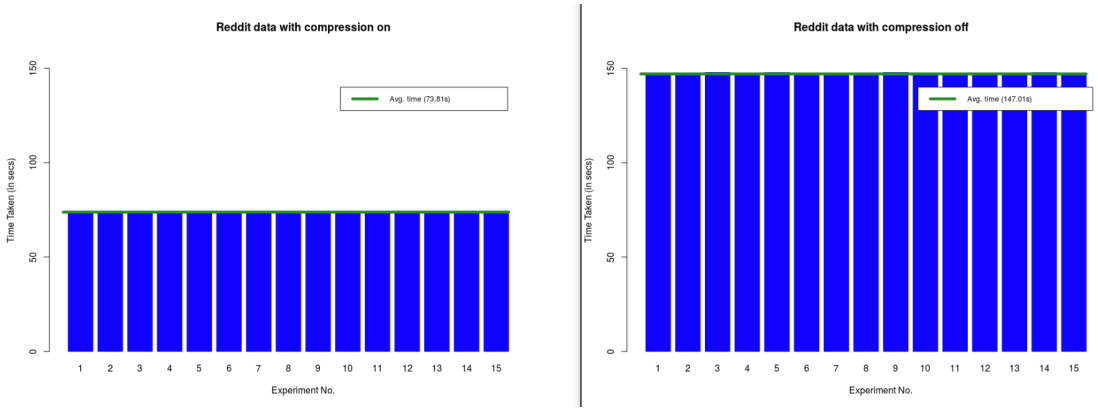

Figure 4.5: Time taken to transfer the Reddit data dataset with compression on vs. off

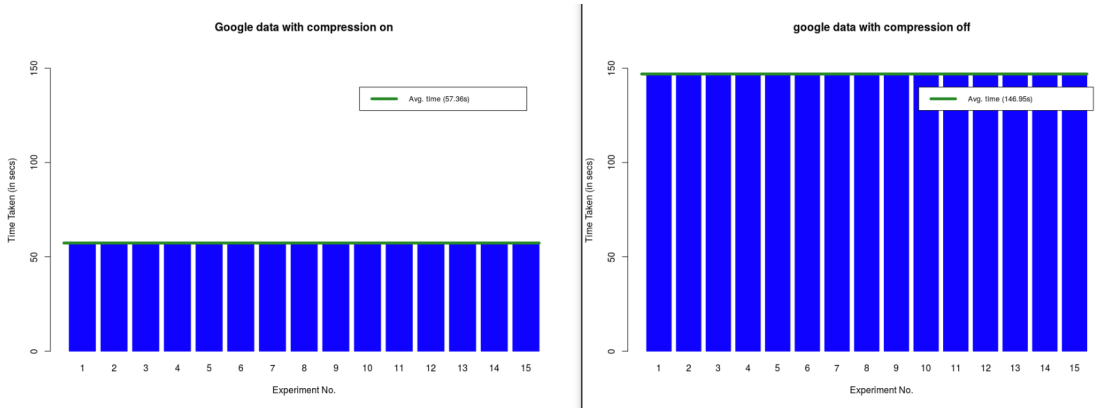

Figure 4.6: Time taken to transfer the Google dataset with compression on vs. off

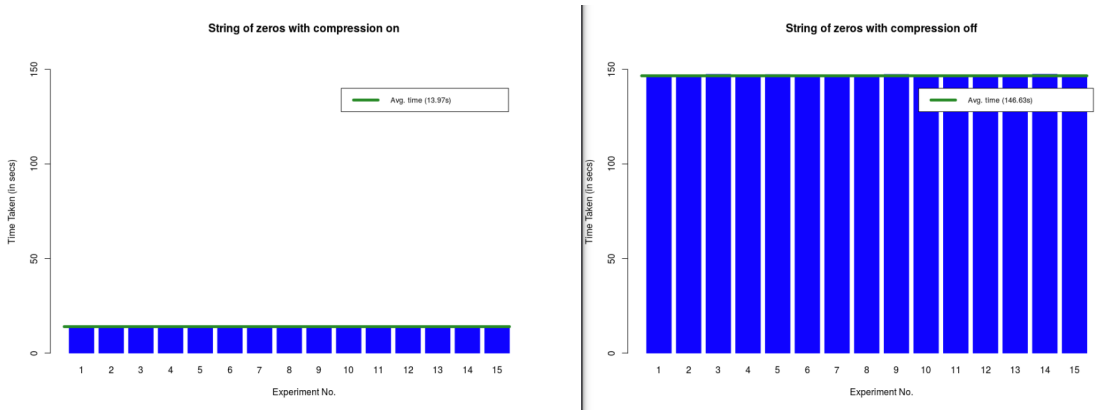

Figure 4.7: Time taken to transfer the Zero String dataset with compression on vs. off

File Name	Without compression (s)	With compression (s)	Absolute time difference (s)	Percentage time difference
Random Data	146.57	146.33	0.24	0.16
YouTube	147.13	138.37	8.76	5.95
Facebook	147.01	103.98	29.27	19.91
Wikipedia	147.02	75	72.02	48.98
Reddit	147.01	73.81	73.2	49.79
Google	146.95	57.36	89.59	60.96
Zero String	146.63	13.97	132.66	90.47

Table 4.1: Transfer time for all datasets with compression vs. without compression.

compression, an improvement of 19.91% . Wikipedia with and Reddit datasets took almost half the average time, with compression enabled; Wikipedia and Reddit took on an average of 75s and 73.81s with compression on and, 147.02s and 147.01s with compression off respectively, an improvement of 48.98% and 49.79% respectively.

The Google dataset is even more compressible had an even better performance with an average time of 57.36s with compression on and 146.95s with it off, an improvement of 60.96%. Obviously, the best performance was with the Zero String data with an average time of 13.97s with compression on and 146.63s with it off, an improvement of 90.47% — an improvement of more than ten times in the transfer time.

The remaining results presented in this section are only for the VPN with compression and idle-time filling features, since these results can be evaluated independently, and don't actually need a comparison with traffic characteristics of the VPN without these additional features (as hiding traffic characteristics is not expected of it).

4.2 IO Graphs

In order to hide the traffic characteristics in our VPN, we used idle-time filling; the idea was to maintain a constant data flow rate in case of both valid data and idle-time filling. Presented below are IO graphs from one representative experiment for each dataset (averages over multiple experiments are not helpful because they would normalize subtle patterns in the graphs, which would depict a lesser information leak than actually occurs).

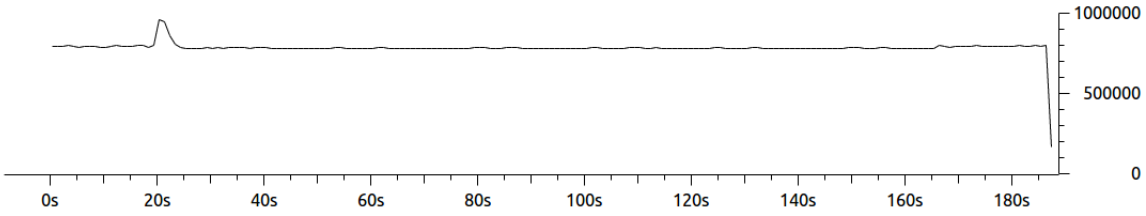

Figure 4.8: IO Graphs for Random Data dataset in B/s

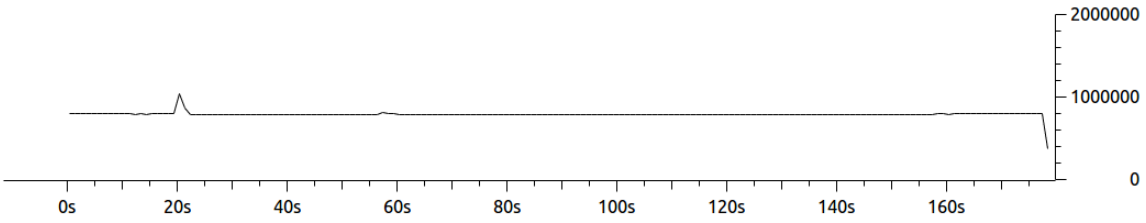

Figure 4.9: IO Graphs for YouTube dataset in B/s

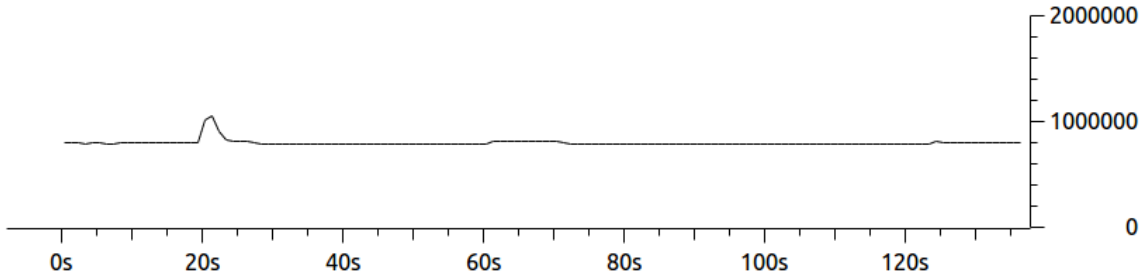

Figure 4.10: IO Graphs for Fb dataset in B/s

The IO graphs were generated using Wireshark[29] on packets captured by Tcpdump. In this capture process, the first and last 20 seconds in each capture is the only idle-time

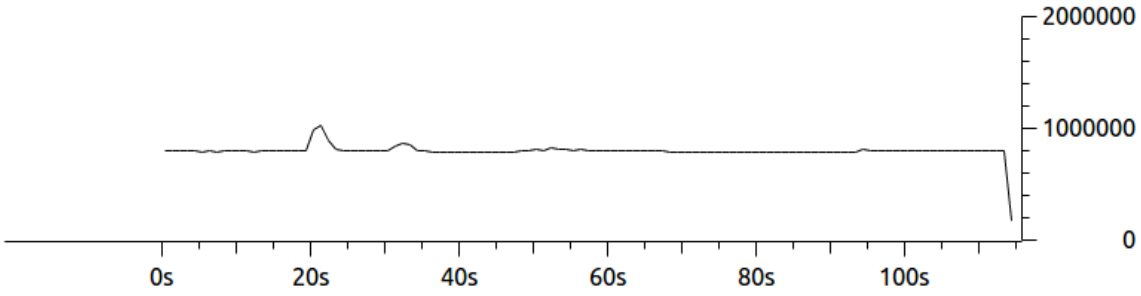

Figure 4.11: IO Graphs for Wikipedia dataset in B/s

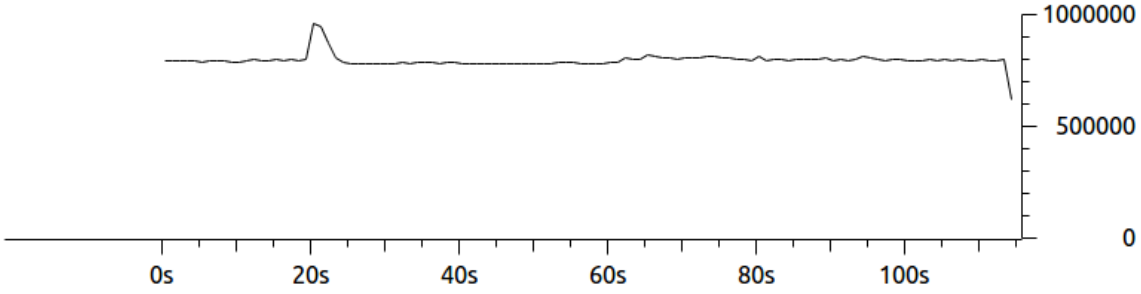

Figure 4.12: IO Graphs for Reddit dataset in B/s

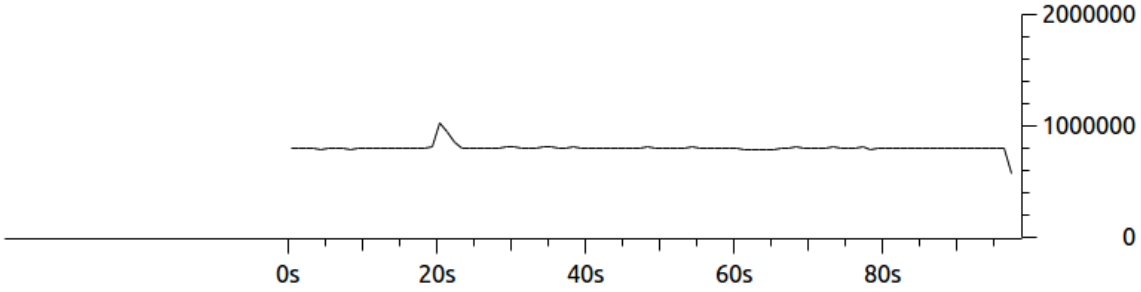

Figure 4.13: IO Graphs for Google dataset in B/s

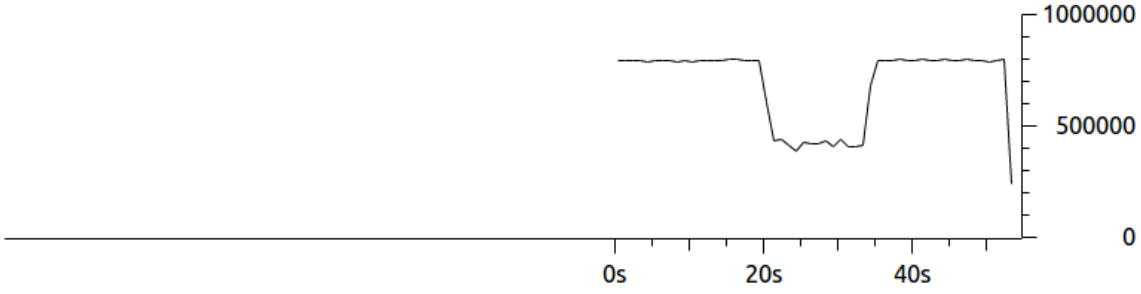

Figure 4.14: IO Graphs for Zero String dataset in B/s

filling and the duration in between is the datasets being transferred. This was done in order to compare the traffic characteristics of idle-time filling to actual dataset transfer.

For all datasets except Zero String, the IO graph seems to be nearly identical for both idle-time filling and dataset transfer. Although there is a momentary spike seen at the beginning of dataset transfer, it is an artifact of our bandwidth limitation and not because of the VPN. The reason for it is that we artificially create a network bandwidth of 6Mbits/s by limiting it in the tunnel, but in case of a sudden surge in data traffic it takes a few seconds to adjust to it and bring the bandwidth back to the set value. This would not be a concern in a network with an actual bandwidth limit.

The IO graph for Zero String dataset is an exception. In this case, the data transfer rate actually drops below the data transfer rate of the time filling traffic. The reason for this is that even though the time spent buffering and compressing the data stays the same as in the case of the other datasets, the data compresses to a negligible size. So, in the same time window, a very small amount of data is actually being transferred, thus reducing the bytes transferred per second. As a result, traffic hiding isn't effective for this dataset. We will discuss a potential approach for solving this problem in the conclusion.

4.3 Packet size statistics

The variation in packet sizes within a data stream reveals information about the data being transferred. It is used as a feature by many of the traffic analysis attacks we discussed to classify network traffic into specific websites visited. As mentioned in the previous section, our VPN uses a padding scheme that keeps the packet sizes constant, with a value of 1514 bytes. The other packets in the stream are the TCP acknowledgment packets and TCP window size update packets. We present the packet size statistics from one of the representative experiments for each dataset.

Packet Lengths (B)	Count	Average	Min. value	Max. value	Percent
0-19	0	-	-	-	0.00
20-39	0	-	-	-	0.00
40-79	44179	66.01	66	78	31.75
80-159	0	-	-	-	0.00
160-319	0	-	-	-	0.00
320-639	0	-	-	-	0.00
640-1279	0	-	-	-	0.00
1280-2559	94973	1514	1514	1514	68.25
2560-5119	0	-	-	-	0.00

Table 4.2: Packet Length Statistics for Random Data dataset

Packet Lengths (B)	Count	Average	Min. value	Max. value	Percent
0-19	0	-	-	-	0.00
20-39	0	-	-	-	0.00
40-79	41927	66.01	66	78	31.65
80-159	0	-	-	-	0.00
160-319	0	-	-	-	0.00
320-639	0	-	-	-	0.00
640-1279	0	-	-	-	0.00
1280-2559	90526	1514	1514	1514	68.35
2560-5119	0	-	-	-	0.00

Table 4.3: Packet Length Statistics for YouTube dataset

Packet Lengths (B)	Count	Average	Min. value	Max. value	Percent
0-19	0	-	-	-	0.00
20-39	0	-	-	-	0.00
40-79	32539	66.02	66	78	30.59
80-159	0	-	-	-	0.00
160-319	0	-	-	-	0.00
320-639	0	-	-	-	0.00
640-1279	0	-	-	-	0.00
1280-2559	73827	1514	1514	1514	69.41
2560-5119	0	-	-	-	0.00

Table 4.4: Packet Length Statistics for Fb dataset

Packet Lengths (B)	Count	Average	Min. value	Max. value	Percent
0-19	0	-	-	-	0.00
20-39	0	-	-	-	0.00
40-79	25668	66.02	66	78	30.30
80-159	0	-	-	-	0.00
160-319	0	-	-	-	0.00
320-639	0	-	-	-	0.00
640-1279	0	-	-	-	0.00
1280-2559	59051	1514	1514	1514	69.70
2560-5119	0	-	-	-	0.00

Table 4.5: Packet Length Statistics for Wikipedia dataset

It is apparent from the tables that there are packets in only two of the size ranges, namely 40-79 and 1280-2559 bytes. The minimum packet size for the former range is 66 bytes, which represents the TCP acknowledgment packets that, as their name suggests acknowledge the receipt of a packet. These packets dominate their size range as suggested by the average packet size for this range. The maximum packet size and the only other packet size in this range is 78 bytes. These packets are TCP window update packets that indicate that the sender's TCP receive buffer size has increased.

The second and the only other packet range with a non-zero packet count i.e. the 1280-2559 bytes have just a single packet size as the minimum and maximum packet sizes for

Packet Lengths (B)	Count	Average	Min. value	Max. value	Percent
0-19	0	-	-	-	0.00
20-39	0	-	-	-	0.00
40-79	25498	66.02	66	78	30.24
80-159	0	-	-	-	0.00
160-319	0	-	-	-	0.00
320-639	0	-	-	-	0.00
640-1279	0	-	-	-	0.00
1280-2559	58828	1514	1514	1514	69.76
2560-5119	0	-	-	-	0.00

Table 4.6: Packet Length Statistics for Reddit dataset

Packet Lengths (B)	Count	Average	Min. value	Max. value	Percent
0-19	0	-	-	-	0.00
20-39	0	-	-	-	0.00
40-79	20812	66.03	66	78	29.15
80-159	0	-	-	-	0.00
160-319	0	-	-	-	0.00
320-639	0	-	-	-	0.00
640-1279	0	-	-	-	0.00
1280-2559	50582	1514	1514	1514	70.85
2560-5119	0	-	-	-	0.00

Table 4.7: Packet Length Statistics for Google dataset

this range is the same. These packets are the packets containing the data sent by the VPN, including both the valid and idle-time filling data. Another thing to notice from these tables is how the compressibility of data in a file is related to the number of packets sent in each experiment. The total number of packets in an experiment are maximum for the Random Data dataset and minimum for Zero String dataset.

Packet Lengths (B)	Count	Average	Min. value	Max. value	Percent
0-19	0	-	-	-	0.00
20-39	0	-	-	-	0.00
40-79	9055	66.22	66	78	27.38
80-159	0	-	-	-	0.00
160-319	0	-	-	-	0.00
320-639	0	-	-	-	0.00
640-1279	0	-	-	-	0.00
1280-2559	24014	1514	1514	1514	72.62
2560-5119	0	-	-	-	0.00

Table 4.8: Packet Length Statistics for Zero String dataset

4.4 Inter-Packet Arrival Times

Inter-packet arrival time is the time elapsed between arrival of two consecutive packets. This information can be used in a TA attack as it reveals user activity and possibly the type of website being visited. We present the graphs of the inter-packet arrival times vs time elapsed from one of the representative experiments for each dataset.

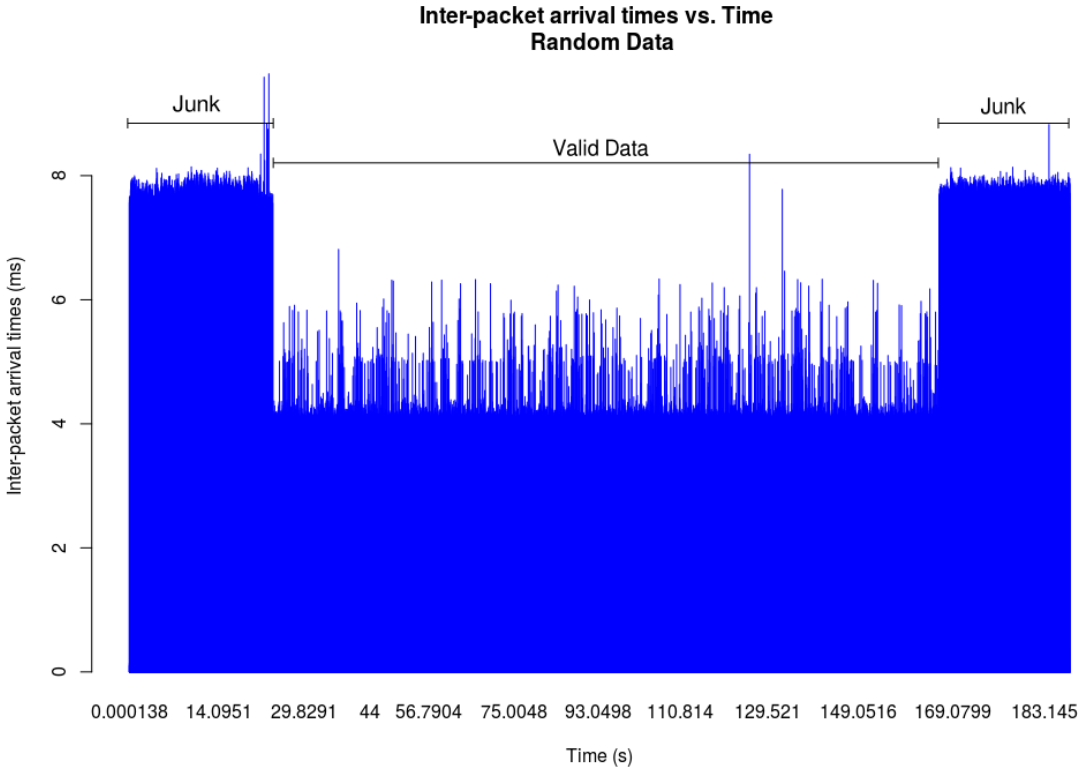

Figure 4.15: Inter-packet arrival times vs. time for Random Data dataset.

In these graphs, idle-time filling at the beginning and the end are clearly distinguishable from the valid data transfer in the middle. The VPN wasn't successful in hiding this information. The reason for this is that in order to keep a constant flow rate of data, the timing of idle-time filling data was altered to imitate the behavior of valid data. This approach did give a constant flow rate, but leaked information through the inter-packet arrival times.

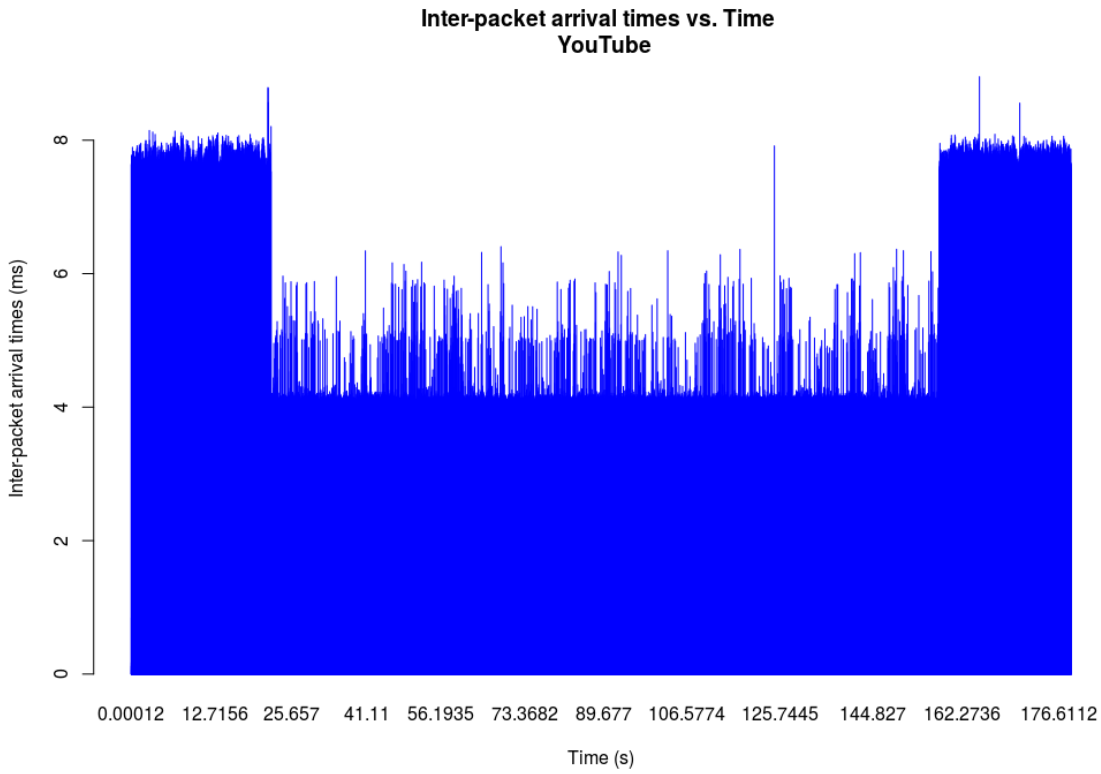

Figure 4.16: Inter-packet arrival times vs. time for YouTube dataset

Another important thing to notice in these graphs is how the Zero String dataset behaves differently than all the other six datasets. The reason for this is that since the Zero String dataset was highly compressible, the entire buffer of packets compressed into a a size less than that of a single packet. Thus with the time taken for buffering and compressing the packets staying almost the same, there was just one packet being sent over the network in this case, unlike the other datasets where there was a stream of multiple packets.

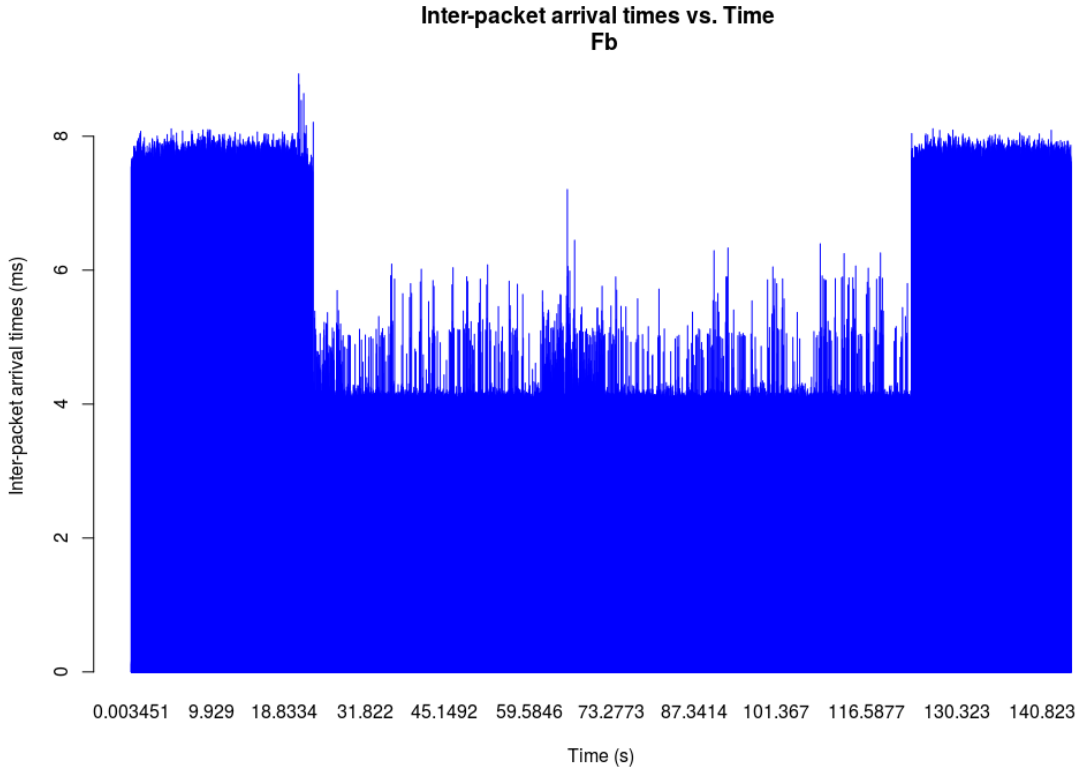

Figure 4.17: Inter-packet arrival times vs. time for Fb dataset

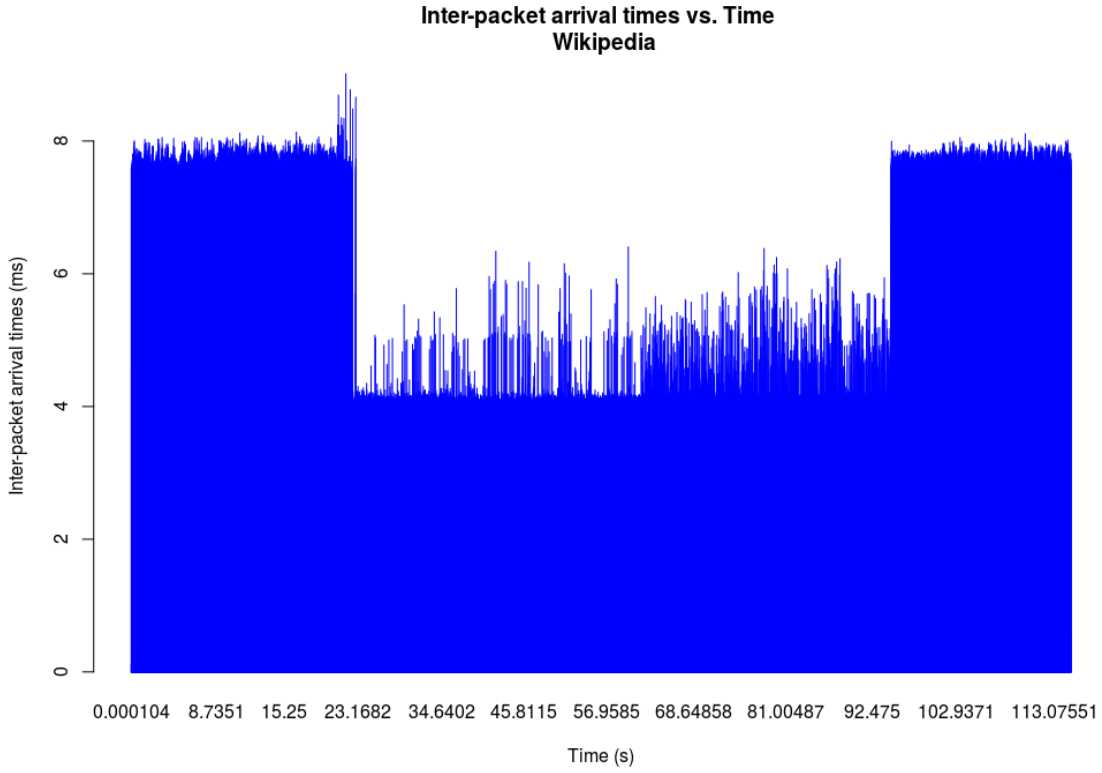

Figure 4.18: Inter-packet arrival times vs. time for Wikipedia dataset

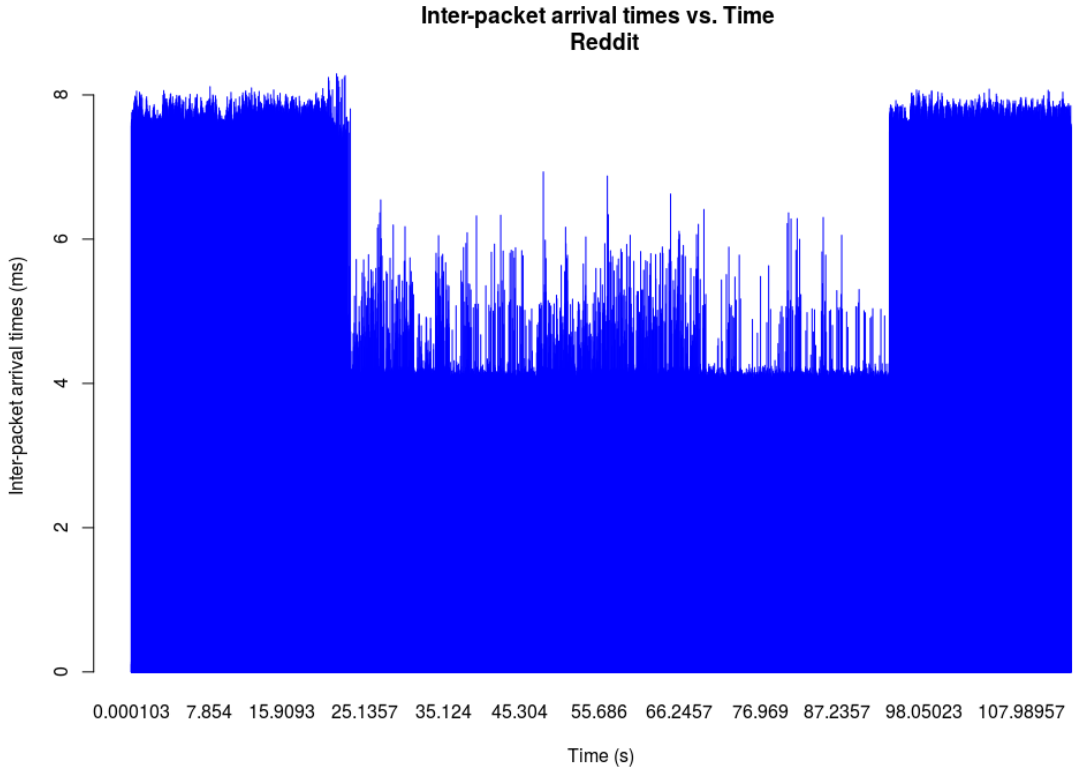

Figure 4.19: Inter-packet arrival times vs. time for Reddit dataset

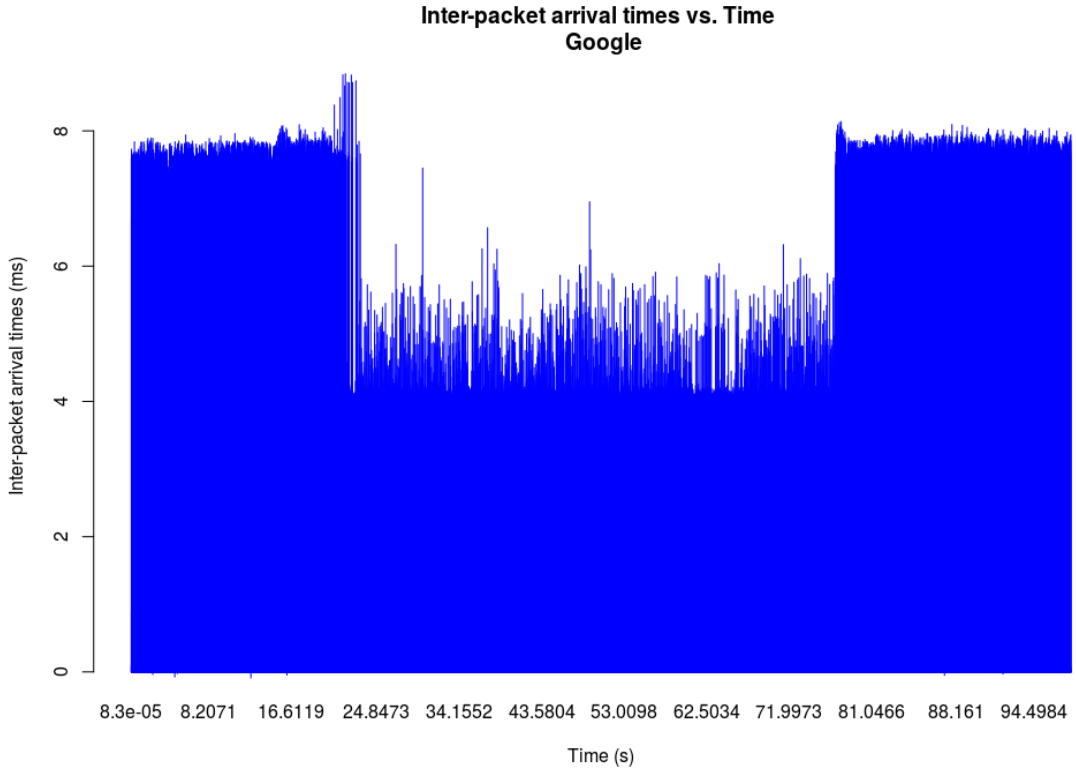

Figure 4.20: Inter-packet arrival times vs. time for Google dataset

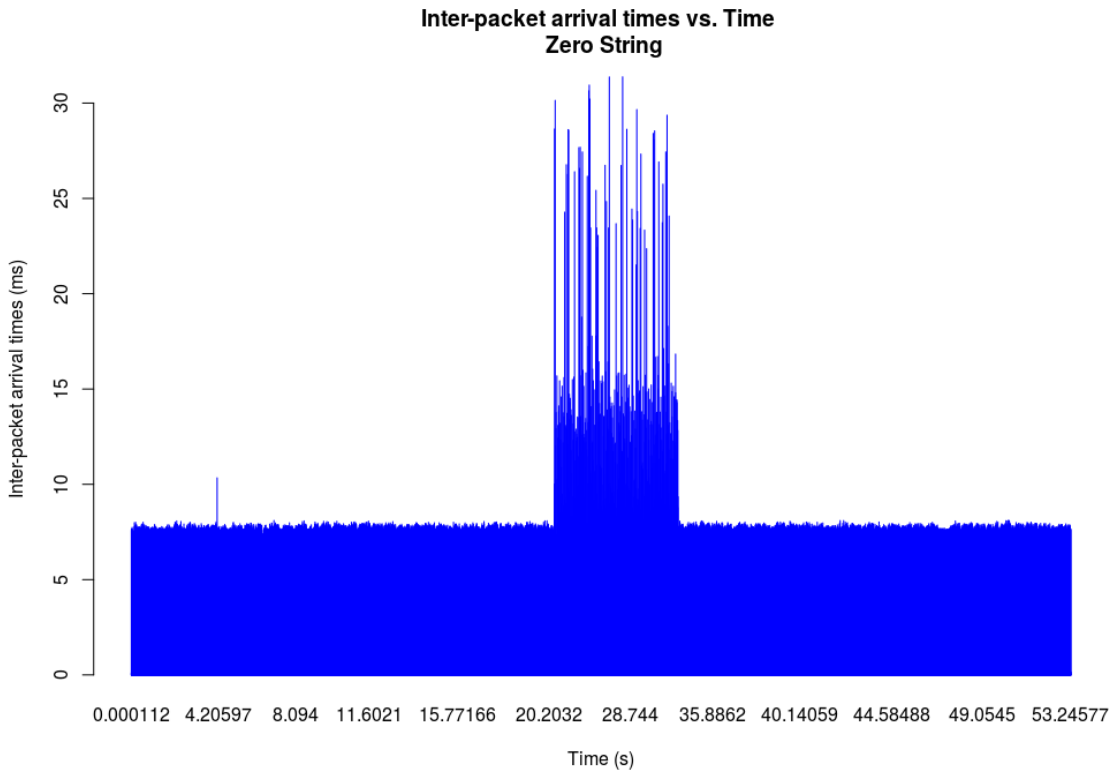

Figure 4.21: Inter-packet arrival times vs. time for Zero String dataset

5 Conclusions

Data Compression has a great potential for increasing efficiency of communication over the Internet. Its importance increases significantly for networks with a lower bandwidth. In the earlier days, this compression process was a static one; where the compressing application would try to compress all the data going through it, regardless of the fact that it was compressible or not. Then the concept of adaptive compression was introduced, which dynamically made the decision of compressing the data or not, based on its compressibility. This made the whole process of compression even more efficient, compared to just static compression.

Compression comes with its fair share of challenges though, as information leakage through compression ratio was employed by cyber attacks called compression side-channel attacks. BREACH is one such attack that targeted HTTPS responses by leveraging the information leakage through compression ratio of encrypted responses to reveal sensitive information within them, such as the Cross-Site Request Forgery (CSRF) token. This attack works against HTTP-level compression. Even the encryption provided through SSL/TLS didn't make the traffic immune to such attacks. As a result, compression is often seen as a security liability, and is being avoided in applications despite its advantages. In addition to compression side-channel attacks, there exist another form of attacks called Traffic Analysis (TA) attacks that, based on the traffic properties like packet sizes, inter-packet arrival times and bandwidth usage, can identify insights about the websites visited, type of data being accessed on the website and the user usage patterns.

In our work, we developed a TCP based Virtual Private Network (VPN) application

that uses compression to enhance its efficiency while also trying to mitigate the compression side-channel and TA attacks. The VPN has a unique way of compressing data where it does not perform per-packet compression but rather buffers packets and then compresses them. Compressing a buffer of packets achieves a higher compression ratio than per-packet compression as the buffer provides more compression context thus increasing the compressibility. We use fixed buffer with a length of 63,712 bytes to buffer the packets. Each packet in the buffer is preceded by a 4 byte header containing its length information.

Once the buffer is full of packets, we compress the packets using a lightweight adaptive compression tool called Minicomp. Minicomp makes the decision to compress the data or not, based on a process called bytecounting. If the decision is to compress, the data is compressed using Gzip-1, and a 32-bit header is added to the data. The API call to Minicomp was similar to memcpy(), where the difference was that the data was also compressed in addition to being copied to a new location.

In addition to compression, a padding scheme is also used in this VPN. This padding scheme hides the packet size information by padding the buffer after compression in such a way that when the buffer is fragmented into individual packets and sent over the network, all the packets are of the same size. This size in our case is the maximum segment size of packet payload over Ethernet, which is 1448 bytes. When padding is added to the buffer, there is a 8-bit header added at the front of the buffer that contains the length of the valid data and the padding in the buffer.

An additional capability of idle-time filling is also included in the VPN. Since bandwidth usage analysis is one of the techniques employed by TA attacks, we tried to hide bandwidth utilization by sending junk data when there was no actual data being transferred. This junk data was timed to imitate the behavior of actual data being transferred.

Experiments were conducted between a server and client, with a bridge interface in the middle to gather information about the traffic between them. We used seven different

datasets, each 100 megabytes in size, that were transferred between the server and client through the VPN using netcat. Out of the seven datasets, five were data dumps from browsing popular websites namely, Google, Facebook, Reddit, Wikipedia and YouTube. The remaining two datasets were a text file full of random data (Random Data) and a string of zeros (Zero String). All these datasets had varying compressibilities, with the random data dataset being the least compressible and the zero string dataset being the most compressible. The network bandwidth was artificially lowered to 6Mbits/s to simulate low bandwidth connections. Twenty seconds before a file transfer was initiated and twenty seconds after the file transfer concluded, idle-time filling traffic was sent, in order to compare its behavior to the valid data transfer.

The first set of results we presented was the time taken to transfer each dataset with compression on and off. The minimum difference in transfer times with adaptive compression on and off was for the Random Data dataset and the maximum difference was for Zero String dataset. For the Random Data dataset, it took an average time of 146.33 seconds to transfer with compression on and 146.57 seconds with compression off, which should be considered to be equal. The minuscule difference in this case is not an improvement, but rather, noise introduced due to the random nature of network communications. For the Zero String dataset, it took an average time of 13.97 seconds to transfer with compression on and 146.63 seconds with compression off, an improvement of 90.47%. The time differences for the other datasets were within this range. Wikipedia had the median time difference, with 75 seconds with compression on and 147.02 seconds with compression off, an improvement of 48.98%. Thus, we can see there is a clear benefit of using compression; for the file with the median time difference, data transfer with compression on was twice as fast as transfer with compression off.

The second set of results were the IO graphs for each dataset. These graphs were meant to demonstrate that the idle-time filling was indiscernible from the actual data transfer. For

all the datasets except Zero String, idle-time filling was almost indiscernible from the actual data transfer, except for a momentary spike due to the latency in the artificial bandwidth reduction. For the Zero String data, however, the bandwidth decreased significantly while the actual data transfer as the entire packet buffer was compressed to a negligible size and therefore only a small amount of data was sent over the network, though the time taken for buffer compression stayed almost the same as in case of the other datasets.

The third set of results were the packet size statistics for each dataset. Here we find that there were packets with only three unique sizes in the traffic of all the datasets.

1. 1514 bytes for the actual data and the idle-time filling data
2. 66 bytes for TCP acknowledgment packets
3. 78 bytes for TCP window size update packets.

Hence the VPN successfully maintained constant sized packets.

The fourth set of results were the inter-packet arrival times. From these graphs of inter-packet arrival times vs. time, a clear distinction could be seen between the actual and the idle-time filling data transfer. The reason for this was that, in order to imitate the bandwidth usage behavior of the actual data, the idle-time filling data was delayed for a certain amount of time before being sent over the network (so as to emulate the compression and buffering process of the actual data). Therefore, our VPN fails at hiding the difference between actual and idle-time filling data in the context of inter-packet arrival times.

As we can see our VPN successfully achieved a faster data transfer rate through adaptive compression, hiding traffic characteristics like bandwidth usage differences between when the user is active vs. idle, and packet sizes, but it failed in hiding the information leakage through inter-packet arrival times. While our project is a success, this leaves a scope of improvement for the VPN.

In order to improve the VPN, we could use the concept of multithreading to our advantage. In our implementation, since there is only a single thread, maintaining a fixed bandwidth usage for the idle-time filling and valid data becomes challenging. In order to maintain that rate, the inter-packet arrival times got modified in a way that traffic characteristics were leaked. This VPN could be further modified to have two threads running in parallel, where one thread handles sending a fixed chunk of data at a fixed interval, and the other handles all the processing of the valid data. Whenever there would be valid data to be sent, it would be copied into the buffer about to be sent out by the thread responsible for sending data along with the necessary padding. In case of no data being there in that buffer before the time interval ends, idle-time filling can be sent out instead. This method could potentially lead to a constant bandwidth without leaking any other critical traffic information. Also, the system could be tested against TA classifiers to demonstrate the effectiveness empirically, instead of just statistical analysis.

Since this VPN compresses data, it most probably is more energy efficient than a VPN without compression, as it leads to lesser packets being sent over the network. We could measure the energy differences between VPN with and without compression. Since the idle-time filling data would increase the energy consumption, we can add an option to turn it off, if the user prefers the extra energy efficiency over the security. Another option would be to explore having the VPN installed and running on both ends of a dedicated network connection with multiple clients simultaneously connecting to it. In this way, individual clients might have more energy savings because they are not responsible for generating idle data, etc.

Another addition to the VPN could be to send the data with variable bandwidth, chosen according to the bandwidth of the network. An investigation could also be performed to check if having variable bandwidth could also lead to a potential side-channel attack inferring traffic characteristics through the variations in bandwidth.

Bibliography

[1] M. Mahoney. ``Data compression explained". In: *mattmahoney. net, updated May* 7 (2012) (cit. on p. 1).

[2] P. Katz. *String searcher, and compressor using same*. US Patent 5,051,745. Sept. 1991. URL: https://www.google.com/patents/US5051745 (cit. on p. 2).

[3] J. Ziv and A. Lempel. ``A universal algorithm for sequential data compression". In: *IEEE Transactions on information theory* 23.3 (1977), pp. 337–343 (cit. on p. 2).

[4] D. A. Huffman. ``A Method for the Construction of Minimum-Redundancy Codes". In: *Proceedings of the IRE* 40.9 (Sept. 1952), pp. 1098–1101. ISSN: 0096-8390. DOI: 10.1109/JRPROC.1952.273898 (cit. on p. 2).

[5] P. A. H. Peterson. ``Datacomp: Locally-independent Adaptive Compression for Real-World Systems". In: (2013) (cit. on p. 3).

[6] E. Rescorla. *SSL and TLS: Designing and Building Secure Systems*. Addison-Wesley, 2001. ISBN: 9780201615982. URL: https://books.google.com/books?id=zBhrQgAACAAJ (cit. on p. 5).

[7] A. Mason. *Cisco Secure Virtual Private Network*. Cisco Press, 2002 (cit. on p. 6).

[8] V. O. Click. *VPN Types and Types of VPN Protocols*. https://www.vpnoneclick.com/types-of-vpn-and-types-of-vpn-protocols/. [Online; accessed 1-November-2016]. 2016 (cit. on p. 6).

[9] K. P. Dyer, S. E. Coull, T. Ristenpart, and T. Shrimpton. ``Peek-a-Boo, I Still See You: Why Efficient Traffic Analysis Countermeasures Fail". In: *2012 IEEE Symposium on Security and Privacy*. May 2012, pp. 332–346. DOI: 10.1109/SP.2012.28 (cit. on pp. 8, 18).

[10] A. Hintz. ``Fingerprinting Websites Using Traffic Analysis". In: *Proceedings of the 2Nd International Conference on Privacy Enhancing Technologies*. PET'02. San Francisco, CA, USA: Springer-Verlag, 2003, pp. 171–178. ISBN: 3-540-00565-X. URL: http://dl.acm.org/citation.cfm?id=1765299.1765312 (cit. on p. 8).

[11] G. D. Bissias, M. Liberatore, D. Jensen, and B. N. Levine. ``Privacy Vulnerabilities in Encrypted HTTP Streams". In: *Proceedings of the 5th International Conference on Privacy Enhancing Technologies*. PET'05. Cavtat, Croatia: Springer-Verlag, 2006, pp. 1–11. ISBN: 3-540-34745-3, 978-3-540-34745-3. DOI: 10.1007/11767831_1. URL: http://dx.doi.org/10.1007/11767831_1 (cit. on p. 8).

[12] Q. Sun, D. R. Simon, Y.-M. Wang, W. Russell, V. N. Padmanabhan, and L. Qiu. ``Statistical identification of encrypted web browsing traffic". In: *Security and Privacy, 2002. Proceedings. 2002 IEEE Symposium on*. IEEE. 2002, pp. 19–30 (cit. on p. 8).

[13] C. V. Wright, L. Ballard, S. E. Coull, F. Monrose, and G. M. Masson. ``Spot me if you can: Uncovering spoken phrases in encrypted VoIP conversations". In: *Security and Privacy, 2008. SP 2008. IEEE Symposium on*. IEEE. 2008, pp. 35–49 (cit. on p. 8).

[14] R. Singel. *Declassified NSA Document Reveals the Secret History of TEMPEST*. http://www.wired.com/2008/04/nsa-releases-se/. [Online; accessed 10-December-2015]. 2008 (cit. on p. 8).

[15] B. Dupasquier, S. Burschka, K. McLaughlin, and S. Sezer. ``Analysis of information leakage from encrypted Skype conversations". English. In: *International Journal of Information Security* 9.5 (2010), pp. 313–325. ISSN: 1615-5262. DOI: 10.1007/s10207-010-0111-4. URL: http://dx.doi.org/10.1007/s10207-010-0111-4 (cit. on p. 9).

[16] J. Kelsey. ``Compression and Information Leakage of Plaintext". English. In: *Fast Software Encryption*. Ed. by J. Daemen and V. Rijmen. Vol. 2365. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2002, pp. 263–276. ISBN: 978-3-540-44009-3. DOI: 10.1007/3-540-45661-9_21. URL: http://dx.doi.org/10.1007/3-540-45661-9_21 (cit. on p. 9).

[17] D. Goodin. *Crack in Internet's foundation of trust allows HTTPS session hijacking*. http://arstechnica.com/security/2012/09/crime-hijacks-https-sessions/. [Online; accessed 1-November-2015]. 2012 (cit. on p. 9).

[18] G. Giacobbi. ``The GNU netcat project". In: *URL http://netcat. sourceforge. net* (2013) (cit. on p. 11).

[19] K.-W. Park and K. H. Park. ``ACCENT: Cognitive cryptography plugged compression for SSL/TLS-based cloud computing services". In: *ACM Transactions on Internet Technology (TOIT)* 11.2 (2011), p. 7 (cit. on p. 14).

[20] M. Liberatore and B. N. Levine. ``Inferring the Source of Encrypted HTTP Connections". In: *Proceedings of the 13th ACM Conference on Computer and Communications Security*. CCS '06. Alexandria, Virginia, USA: ACM, 2006, pp. 255–263. ISBN: 1-59593-518-5. DOI: 10.1145/1180405.1180437. URL: http://doi.acm.org/10.1145/1180405.1180437 (cit. on p. 16).

[21] D. Herrmann, R. Wendolsky, and H. Federrath. ``Website Fingerprinting: Attacking Popular Privacy Enhancing Technologies with the Multinomial NaïVe-bayes Clas-

sifier". In: *Proceedings of the 2009 ACM Workshop on Cloud Computing Security*. CCSW '09. Chicago, Illinois, USA: ACM, 2009, pp. 31–42. ISBN: 978-1-60558-784-4. DOI: 10.1145/1655008.1655013. URL: http://doi.acm.org/10.1145/1655008.1655013 (cit. on p. 17).

[22] D. X. Song, D. Wagner, and X. Tian. ``Timing Analysis of Keystrokes and Timing Attacks on SSH." In: *USENIX Security Symposium*. Vol. 2001. 2001 (cit. on p. 17).

[23] Y. G. Angelo Prado Neal Harris. *BREACH - SSL, gone in 30 seconds*. Tech. rep. 2013 (cit. on p. 21).

[24] D. Karakostas and D. Zindros. ``Practical New Developments on BREACH". In: () (cit. on p. 21).

[25] geraint0923. *S-VPN*. https://github.com/geraint0923/S-VPN. 2014 (cit. on p. 25).

[26] M. Foundation. In: *Version 56.0.2* (2017). URL: https://www.mozilla.org/en-US/firefox/56.0.2/releasenotes/ (cit. on p. 31).

[27] M. Frysinger, a3alex, and anatoly techtonik. *Netcat 1.10*. https://sourceforge.net/projects/nc110/. 2006 (cit. on p. 31).

[28] V. Jacobson, S. Floyd, V. Paxson, and S. McCanne. *tcpdump*. http://www.tcpdump.org/tcpdump_man.html. 1988 (cit. on p. 31).

[29] G. Combs et al. ``Wireshark-network protocol analyzer". In: *Version 0.99* 5 (2008) (cit. on p. 36).