

# **SECURITY ANALYSIS OF TOR MULTIPLEXING ALGORITHMS**

BY

**YASIR SULIMAN ALAGL**

A Thesis Presented to the  
DEANSHIP OF GRADUATE STUDIES

**KING FAHD UNIVERSITY OF PETROLEUM & MINERALS**

DHAHRAN, SAUDI ARABIA

1963 ١٣٨٣

In Partial Fulfillment of the  
Requirements for the Degree of

**MASTER OF SCIENCE**

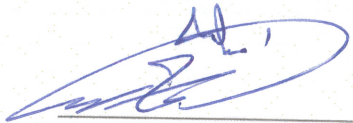
In

**SECURITY & INFORMATION ASSURANCE**

**JANUARY 2018**

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS  
DHAHRAN- 31261, SAUDI ARABIA  
**DEANSHIP OF GRADUATE STUDIES**

This thesis, written by **YASIR SULIMAN ALAGL** under the direction his thesis advisor and approved by his thesis committee, has been presented and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of **MASTER OF SCIENCE IN SECURITY & INFORMATION ASSURANCE**.



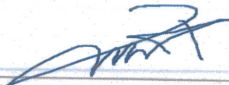
Dr. Khalid Al-Jasser  
Department Chairman



Dr. Sami M. Zhioua  
(Advisor)



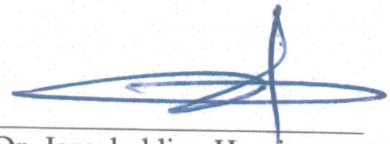
Dr. Salam A. Zummo  
Dean of Graduate Studies



Dr. Ahmad Almulhem  
(Member)

15 / 1 / 2018

Date



Dr. Jameleddine Hassine  
(Member)

© YASIR SULIMAN ALAGL  
2018

## **ACKNOWLEDGMENTS**

To my life-coach, my advisor, and mentor: Dr. Sami because I owe it all to you. Many

Thanks!

To my loving family, father, mother, and my wife...to the endless support I received from you...

To my thesis committee, Dr. Almulhem and Hassine, thank you for your support and continuous feedback.

A very special gratitude goes out to Dr. Al Jasser, our chairman, for his support and hard work. Thank you for the trust.

To the members of The Island, for their cheerleading and support...

To my SOC Family, Thanks for your ideas and discussion...

A thank you, to KFUPM...



# Table of Contents

<b>ACKNOWLEDGMENTS .....</b>	<b>2</b>
<b>LIST OF FIGURES .....</b>	<b>5</b>
<b>LIST OF ABBREVIATIONS .....</b>	<b>6</b>
<b>ABSTRACT .....</b>	<b>7</b>
<b>ملخص الرسالة .....</b>	<b>8</b>
<b>CHAPTER 1 INTRODUCTION .....</b>	<b>9</b>
<b>CHAPTER 2 TOR ANONYMITY PROTOCOL .....</b>	<b>14</b>
2.1 INTRODUCTION TO TOR .....	16
2.2 KEYS AND AUTHENTICATION .....	18
2.3 CELLS.....	20
2.4 CIRCUIT ESTABLISHMENT .....	21
2.5 RELAY CELLS .....	23
2.6 STREAMS.....	25
2.7 HTTP SCENARIO.....	26
<b>CHAPTER 3 LITERATURE REVIEW .....</b>	<b>29</b>
<b>CHAPTER 4 MULTIPLEXING TOR TRAFFIC .....</b>	<b>35</b>
4.1 SINGLE CIRCUIT SCENARIO.....	41
4.2 APPROACH MOTIVATION & PROPOSED ALGORITHMS .....	44
4.2.1 Approach Motivation.....	44
4.2.2 Proposed Algorithms .....	47
<b>CHAPTER 5 EXPERIMENTS AND RESULTS .....</b>	<b>57</b>
5.1 TESTBED .....	57
5.1.1 First Setup.....	57
5.1.2 Second Setup.....	58
5.2 TOR MULTIPLEXING AT ONION PROXY.....	59
5.2.1 First Experiment: Tor General Behavior .....	59
5.2.2 Second Experiment: EWMA Multiplexing Behavior .....	66

5.2.3 <i>The Dictionary Experiment</i> .....	71
5.2.4 <i>The Full Fledge Experiment</i> .....	76
5.3    EXPERIMENTS ON PROPOSED ALGORITHMS.....	83
5.3.1 <i>Ground Truth</i> .....	83
5.3.2 <i>Implementation</i> .....	86
5.3.3 <i>Experiment</i> .....	86
5.3.4 <i>Findings</i> .....	87
<b>CHAPTER 6 CONCLUSION AND FUTURE WORK</b> .....	<b>90</b>
6.1 THREATS TO VALIDITY.....	91
6.2 FUTURE WORK.....	94
<b>REFERENCES</b> .....	<b>95</b>
<b>VITAE</b> .....	<b>99</b>

## List of Figures

Figure 1 Tor Analogy.....	15
Figure 2 Formal process of circuit creation .....	22
Figure 3 HTTP Scenario .....	28
Figure 4 Relationship between Circuit Queues and Output Buffers.....	37
Figure 5 OP Browsing using one circuit.....	42
Figure 6 Illustration of the first approach .....	52
Figure 7 First approach pseudocode .....	52
Figure 8 Illustration of the Second Approach.....	54
Figure 9 Third Approach Queue Example.....	56
Figure 10 First setup illustration.....	57
Figure 11 Illustration of the second setup.....	58
Figure 12 Stream intermixing of two files sized 1 MB each, simultaneously being uploaded using Tor .....	63
Figure 13 Streams intermixing of two files sized 5 MB each, simultaneously being uploaded using Tor .....	64
Figure 14 Streams intermixing of two files sized 100 MB each, simultaneously being uploaded using Tor .....	64
Figure 15 Hashing algorithm used to identify encrypted cells .....	67
Figure 16 The correspondence between file size, TCP packets, and Tor cells.....	75
Figure 17 The intermixing of 3 TCP streams resulting from 3 file uploads where each stream is represented by a different color .....	81
Figure 18 The intermixing of 10 TCP streams resulting from 10 file uploads where each stream is represented by a different color .....	82
Figure 19 Establishing ground truth by comparing the order URL requests to the order of cells leaving Tor with those requests .....	85
Figure 20 Illustration of applying the first approach algorithm to Tor.....	87
Figure 21 HTTP Pipelining Illustration .....	92

## LIST OF ABBREVIATIONS

<b>TOR</b>	:	The Onion Routing
<b>OP</b>	:	Onion Proxy
<b>OR</b>	:	Onion Router
<b>OR</b>	:	Onion Relay
<b>FIFO</b>	:	First-In-First-Out
<b>WF</b>	:	Website Fingerprinting
<b>HTTP</b>	:	Hyper Text Transfer Protocol
<b>SSH</b>	:	Secure Shell
<b>ntor</b>	:	Onion Key
<b>SVM</b>	:	Support Vector Machine



## ABSTRACT

**Full Name** : Yasir Suliman Alagl  
**Thesis Title** : Security Analysis of Tor Protocol Multiplexing Algorithms  
**Major Field** : Information & Security Assurance  
**Date of Degree** : January, 2018

As the Internet grows in size and users, many aspects of our lives start to fold around its center. This calls for more preventive measures that guard against malicious actors and threat cases. Tor strives to cover one aspect by promising its users a low-latency, anonymity protocol that guards against those actors. In recent years, many researches focused on identifying the end servers Tor users intend to reach. Among those, Website Fingerprinting (WF) and Traffic Analysis are considered the most effective due to ease of implementation, lower resources requirements, and promising results in both open and closed world. The efforts to thwart such attacks is solely based on increasing the active circuits by encouraging more users to utilize Tor, and thus increasing the level of multiplexing that occurs at Tor relays. However, these approaches fail dramatically when attackers target lowly utilized relays, or when rigid regimes focus on the sole link between an end user and the entry node. In this research, we aim to unearth the details of Tor multiplexing, and show the extent to which it aids in defending such attacks. We also propose to introduce randomization to Tor by displaying three algorithms to randomize and multiplex Tor streams, right from the initial point, i.e. Tor end-user.

## ملخص الرسالة

الاسم الكامل: ياسر سليمان العقل

عنوان الرسالة: التحليل الأمني لخوارزميات التعدد والتشكيل في بروتوكول تور

التخصص: أمن المعلومات

تاريخ الدرجة العلمية: يناير ٢٠١٨

كلما تكبر شبكة الانترنت، ويزداد عدد مستخدميها، يزداد اعتمادنا عليها، وتصبح محور حياتنا اليومية. لهذا دعت الحاجة إلى ابتكار وإنتاج العديد من وسائل الحماية لمقاومة العدد المتزايد من المستخدمين الخبيثين، والتصدي لحالات التهديد السيبرانية. تور (بروتوكول التوجيه البصلي) يسعى جاهدا لسد هذا الاحتياج عن طريق تقديم بروتوكول سريع الاستجابة، يعد مستخدميه بإخفاء هوياتهم الحقيقية خلف توجيه متعدد لكي يحميهم من هؤلاء المستخدمين الخبيثين. في السنوات الماضية القريبة، ركز الباحثون جهودهم لاستخلاص المواقع التي يقوم مستخدمون تور بزيارتها بطرق متعددة، ليس لمساعدة المستخدمين الخبيثين، بل لابتكار طرق الدفاع أيضا. أحد أشهر هذه الطرق، هو "تبصيم المواقع"، ويعد الأكثر فعالية نظرا لسهولة تنفيذ هذا الهجوم، انخفاض الموارد الحاسوبية المستخدمة لتطبيقه، والنتائج الواعدة في التجارب العملية، وفي الحياة الواقعية السيبرانية. وقد كانت أغلب الجهود ذات الفعالية في صد هذه الهجمات، تتمحور حول زيادة عدد مستخدمي هذا البروتوكول، مما يعني زيادة في استخدام شبكة تور، وبالتالي مضاعفة نسبة خلط البيانات التي تمر بشبكة تور. ولكن هذه الجهود غالبا ما تفشل، عندما يقوم المستخدمون الخبيثون بالتركيز على مقدمي خدمة تور ذو الاستخدام المنخفض نسبيا، وبالتالي تقليل نسبة البيانات المختلطة، أو عندما تقوم بعض الحكومات ذو النظام الصارم بالتركيز على حلقة الوصل الوحيدة التي تربط بين المستخدم النهائي، وشبكة تور. في هذا البحث، نهدف إلى كشف وتدقيق الطرق المستخدمة في تور لخلط بيانات المستخدمين، وتحليل مدى فعاليته في صد الهجمات. أيضا، نقوم بعرض ثلاث خوارزميات تقوم بتحسين عملية خلط البيانات في تور منذ خروجها من المستخدم النهائي، وحتى وصولها إلى الخدمة النهائية المعنية.

# **CHAPTER 1**

## **INTRODUCTION**

As the Internet grows in size and users, many aspects of our lives start to fold around its center. The different services the Internet provides, and the advances in recent technologies, dramatically increase the potential of tasks we are able to exercise on it. From a simple question on a popular search engine, to industrial control systems, different principles of information security are being mandated on daily basis. For some, the assurance that their messages are received as-is without tampering, while others, focus solely on whether their messages were read by some third party during the transmission.

A recent rising demand in today's world is anonymity. No longer does a party requires the confidentiality of their messages only, but the assurance that the other end doesn't identify them, by means of IP, geographical location, or time zone. Also, the inability of an observer to identify the different destinations this party is reaching out to. Hence, the increasing popularity of low-latency anonymity systems is being noticed nowadays in different fields of online-provided services, most notably Onion Routing Protocols, with a dominating percentage utilizing Tor [3]. Tor promises its users with the confidentiality and anonymity they're seeking by introducing a multi-layer, multi nodes, encryption protocol that sparse your communication traffic around the web

before final delivery, and uses middle routing relays as encryption/decryption nodes, where a node is only aware of the previous or upcoming node in terms of metadata [IP, location, etc.], and a complete shadowing of the traffic content.

The unique features Tor promises and provides for its users, derived a new field of use cases for different demographical sets. Users in oppressive regimes, such as China and Iran, can evade censorships and communicate liberally [1]. E-Commerce users can keep their history and shopping preferences private, against online marketing campaigns. Whistleblowers can freely communicate with law enforcements and the press without compromising their identities. Governmental agencies abroad can reach out to their headquarters without notifying host countries. Trade secrets can be securely transferred without alerting possible eavesdropping competition. Hackers and hacktivists can, unfortunately, conduct their destructive actions with an extra layer of confidence, such as the recent attack on The Hacker Group [2].

Despite the promises Tor provides, and the complexity noted by its wide-spread infrastructure, several attacks [4, 5, 6, 7] have successfully de-anonymized some aspects of Tor, thus, compromising the most demanded aspect of it. These attacks differ in complexity, applicability in real world, and the required level of control on network nodes around the globe. While the most successful attacks are those relaying on traffic confirmation, where an adversary monitors traffic on both ends of communications, these attacks fail dramatically outside the environment of a lab. Even



larger-scale entities, such as ISPs and backbone support entities, can't guarantee continuous monitoring of nodes outside their jurisdiction to preclude this kind of attack [8, 9].

However, attacks involving a local observer are considered more realistic and applicable to users of anonymity systems. A local observer is any entity that has control over (or can monitor) the traffic between the user and the first encrypting node of a Tor circuit. In a local coffee shop, this could be a script kiddie capturing all the wireless traffic; an intruder eavesdropping on your home router, an ISP monitoring your entire communication, or a system administrator with the proper tools to capture your traffic. Recently, attacks based on Website Fingerprinting (WF) [10] are considered most effective and realistic on Tor. WF is a special attack derived from traffic analysis that classifies and identifies websites based on certain characteristics these websites exert. The number of resources (files) the homepage of a popular website serves on an ordinary request, the sizes of these resources, the order of which they are served, or the timings between each response, are typical characteristics this class of attack utilizes to draw conclusions about the websites being visited. Recent WF attacks on Tor [8, 9, 11, 12, 13] have shown a surprising 57% of true positive detection rate in a closed-world setting, for a false positive rate below 1%. Some attacks claimed to reach a higher or more accurate results in an open-world environment as well to score an accurate detection rate of 97%.

However, WF is as accurate as the traffic its analyzing. The more noise introduced in a traffic, the difficult it becomes identifying a website. Furthermore, the more websites a user surfs, the harder it is to distinguish different streams. That is why Tor employs the use of algorithms which masquerades the traffic by either dividing them into a fixed width of 514-bytes data chunks, known as cells, or merging the data from multiple packets into a single cell. Additionally, Tor combines multiple TCP streams into a single circuit, and multiplexes those circuits into a single connection, which increases the level of complexity for an observer. Most of the literature work uses the ground truth of the data collection system to decide where to split the stream. Moreover, the algorithms implemented only accept as an input cell sequences that corresponds to a single web resource or page.

In this research, we aim on exploring the internals of Tor multiplexer to truly understand the inner-working and conditions where the multiplexing applies, and fully measure the extent to which multiplexing reach. We aim to unearth a vague point of research that often overlooks the role of different streams composing a circuit. Additionally, we feel the urge to determine whether Tor accounts for streams randomization as it does for different circuits. Furthermore, we propose three different algorithms that can dramatically enhance the overall security of Tor, and provide a strong layer of resistance towards Websites Fingerprinting, and Traffic Analysis attacks in general. Specifically, our contribution is:

- 1- Analyze Tor multiplexing module, explain its interworking, and discuss its security implications.
- 2- Identify the scenarios where Tor applies Multiplexing, or the lack of it.
- 3- Identify the scenarios where Multiplexing is applied but with no additional security value.
- 4- Introduce three different algorithms that enhances on the identified scenarios with lack of multiplexing or multiplexing security added value.
- 5- Conduct empirical experiments that backup our findings in 2 and 3.
- 6- Conduct empirical experiments that backup up our algorithms in 4.

In the remaining of this section, we deliver a brief introduction on Tor, and formally define our problem statement, as well as providing a general overview of Traffic Analysis. In Section 2, we deeply explain the way Tor operates, and provide some technical insights on its processes. In section 3, we provide some recent related work on Tor's fingerprinting in the literature. The familiar reader with the topic can skip to section 4. In section 4, we introduce the incentives and rationality of our proposal, demonstrate the mechanism of Tor circuits and multiplexing, and propose our improvement by presenting three different algorithms of streams multiplexing and randomization. In section 5, we present the empirical work and results of our proposed algorithms.

## **CHAPTER 2 TOR ANONYMITY PROTOCOL**

Tor circuit is a collection of Tor Onion Routers (OR) distributed over the Tor network that aim to provide an encrypted end-to-end channel between a Tor user (client) utilizing an Onion Proxy (OP) and an Internet service, e.g. HTTP server, Tor hidden service, FTP server, or even a VPN server. A circuit is established incrementally by the client (OP), through the use of three Onion Routers (OR), where each two ORs exchange information independently of the rest of the circuit, to establish a one-to-one secure connection between them. While a circuit is intended to be used by a single client (OP) to carry as much services as required by the client, two ORs will only establish a single (TCP) connection among them at any time and will piggyback that single connection with as much circuits as required from different OPs.

The same behavior is also observed between a pair of OP/OR, where a single outer TCP connection is established. Once a full circuit is established, an Onion Proxy can establish as many TCP streams as required on top of that circuit to consume Internet services. In essence, an observer of the traffic between two Tor Onion Routers will see a single TCP connection, that wraps around multi Tor circuits established by many Onion Proxies, where each circuit transmits many TCP streams from only a single Onion Proxy, heading to many Internet services. The analogy used in this section will continue throughout the rest of this paper and is illustrated in Figure 1.



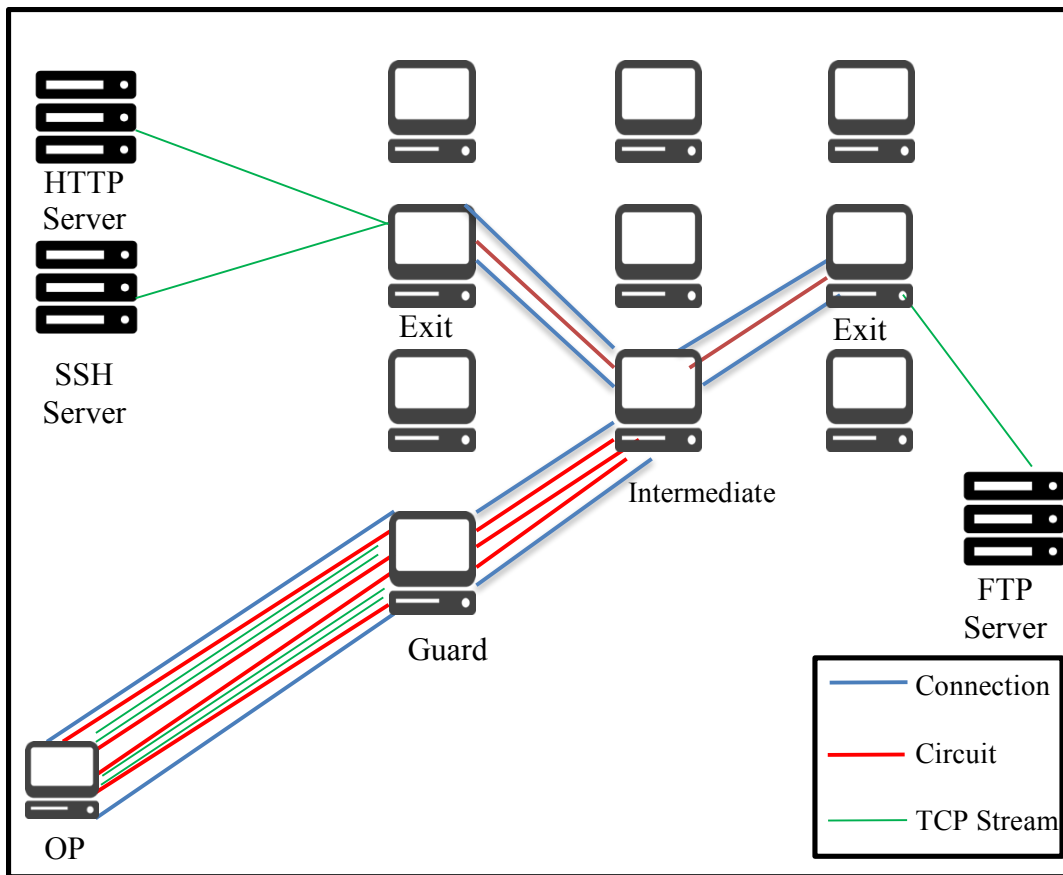


Figure 1 Tor Analogy

## 2.1 Introduction to Tor

Tor is a low-latency anonymity protocol, based on Onion Routing, where a network of volunteers constructs a hard-to-follow route through the Internet, similar to when you want to throw off a tailgater [26]. There are four main components in a Tor network, namely Tor clients: an end user who wishes to establish TCP connections to other entities while keeping their identity anonymous, also referred to as Tor Proxies; Tor nodes: intermediate routing nodes that convey the clients traffic, also known as Tor Relays or Onion Routers; Tor Directory Server: similar in functionality to DNS servers, but uses a different mechanism; Tor Onion Services (previously hidden services): an Internet server that wishes to keep their identity anonymous from end users (Tor clients) and other entities while serving any sort of resources.

Two possible use cases may occur when a Tor client wishes to establish a connection, depending on the end destination. If the client is aiming to reach a normal Internet server that doesn't implement Tor (i.e. not an onion service), the client's Tor software (Tor Onion Proxy) starts by consulting a Tor directory server. The server provides information about currently available Tor Relays (Onion Routers), network topology, and bandwidth information to the client. The client, then, chooses three nodes to build an encrypted circuit incrementally. That is, similar to a VPN server, the client starts by establishing a tunnel connection with the first node in the circuit, utilizing public key cryptography. It, then, uses this portion to extend the connection to the second chosen node, and finally, to the third. During that, the client exchanges symmetric keys with each node in the path (circuit) to carry on session encrypted communication. Once the

circuit is secured, the client can start establishing communication channels with different destination servers by building streams on top of the circuit.

The process starts by encrypting packets in a layered approach where each packet is encrypted three times with the third, second, and first node's key, in that order, resulting in an onion-like shape. Every node, then, peels off its respective layer of encryption to reveal the information about the next relay in the circuit, without compromising the contents of the original packets. Only the first node in the circuit (Guard node) knows the original sender (Tor client), where the last node in the circuit (exit node) can identify the final destination of the packet, and the content of the packet if SSL is not used by the end server. The simplest attack on Tor's anonymity can be implemented by compromising both the guard and exit node, a statistically infeasible approach. In this research, we only focus on this use case of Tor, hence, no much emphasis will be given to the second scenario, and we will explain the general idea for the reader's convenience.

In the second use case, the Tor client wishes to connect to a Tor onion service. In this case, a rendezvous node is selected at random by the client. The client sends the selected rendezvous node through a three-relays circuit to a certain node on Tor network that knows how to reach the onion service. The node forwards the request to the onion service, which by turn establishes a three-relays circuit to that rendezvous point, thus, forming a six-relays Tor circuit. The client and service exchange required services, and the circuit is broken.

## 2.2 Keys and Authentication

Each connection established between two Onion Routers or an Onion Proxy and an Onion Router must use SSLv3/TLS for link authentication and/or encryption; TLS is always preferred. For that purpose, a Tor Onion Router must first prove its identity by the use of Public Key Infrastructure, and must maintain a secure identity utilizing a collection of public/private key pairs. An RSA-1024 key is maintained as an identity key, and is used solely to sign certificates, documents, and the likes, that are issued by that relay. This is a long-term key that is maintained as well by the directory servers of Tor to reference that specific relay.

An alternation of that key, is an ED25519 signing key that is used in later versions of Tor, and is also used to describe the identity of the node, however, is referred to as the master identity key, and is solely used to sign the third key. The third key is a medium-term ED25519 key, that is signed by the master identity key and is utilized for onion skin decryption (discussed below). Two other medium-term keys are used. While the first is an RSA-1024 TAP (onion key) that is used to decrypt onion skins, when accepting circuit extend attempts originating from the Onion Proxy (client); the second key is implanted as an EllipticCurve25519 key, that is used for the same purpose, and is referred to as an “ntor” (onion key). These two keys can be used interchangeably with the first key. Finally, a single short-term RSA-1024 key is used to negotiate point-to-point TLS connections, and is rotated much frequently, minimally once a day.

Furthermore, three ways are available for two Tor relays to authenticate themselves and establish a TLS connection between them. In the first method, namely certificates-up-front, each relay provides its authentication certificate prior establishing a TLS connection, and as part of their initial TLS handshake. In the second approach, namely renegotiation, only the responder relay provides its authentication certificate, allowing the requester to authenticate immediately via a TLS renegotiation. In the third method, namely in-protocol, both relays utilize Tor protocol, after the initial TLS renegotiation, to bootstrap themselves to mutual authentication.

In the first method, the initiator always starts by sending a two-certificates chain consisting of an X.509 certificate utilizing the short-term connection public key, and a second self-signed certificate announcing its identity key. The responder replies with a similar chain. In the second method, the initiator doesn't send a certificate, while the responder provides a single connection certificate. Once the handshake is concluded, the parties renegotiate the handshake with each relay providing its two-certificates chain, as in certificates-up-front approach. In the third approach, a TLS connection is established in a non-conventional way where the parties exchange Tor specific data structures (known as cells) to establish a TLS connection, and to agree on connections properties beforehand, where they engage in relatively longer communication messages.

## 2.3 Cells

Cells are the building blocks of Tor and the smallest unit of communication, as they are utilized in every aspect of Tor establishment of connections, circuits building, and exchange of data. Cells are considered one of the strongest defenses Tor implements against Traffic Analysis attacks as they employ a fixed width of 514 bytes (512 bytes in older versions), that are mostly triple encrypted. A typical Tor cell will have three fields: CircID, a 4-bytes field that identify the corresponding circuit this cell associates to; Command, a 1-byte field having a numeric value ranging from 0 to 127 that describes the purpose of the cell; and Payload, a 509-bytes field that carries the payload of the cell, and comes in a variety of formats and structures, depending on the purpose of the cell, identified by the Command field. It's worth noting that other structures of Tor cells are available, however, they are neglected due to their irrelevancy, restricted use to backward compatibility, and far less common use.

Some common examples of the Command field values are 0 (PADDING), which indicates that the cell is being used for padding purposes; 1 (CREATE) which instructs the receiving relay to create a circuit; 2 (CREATED) which is an acknowledgment cell confirming the creation of a circuit; and 3 (RELAY) which is the most common cell type of Tor that is used for end-to-end data transfers. As mentioned earlier, the interpretation of the payload field is dependent on the Command field. A Command value of 0 (PADDING) indicates that the payload is not used and should be discarded, while a Command value of 1 (CREATE) suggests that the payload contains the handshake challenge. Similarly, a Command value of 2 (CREATED) indicates that the Payload is the handshake response. Of most relevance, a Command value of 3

(RELAY) instructs the receiving relay to further interpret the payload as another data structure, consisting of a Relay Header and a Relay Body. We will further display the internal structure of a Relay cell at a later stage.

## **2.4 Circuit Establishment**

As mentioned earlier, an Onion Proxy (OP) incrementally create a circuit using at least three Tor Onion Routers (OR, relays, nodes, or hops). The process by which this is accomplished is delegate, and requires the use of a multi-spectrum range of cell types. In general, OPs send a CREATE cell to the first node in the path, such that the payload of the cell contains the first half of the handshake challenge. Immediately, that node responds with a CREATED cell that encompasses the second half of the authenticated handshake.

The handshakes mentioned here corresponds to Diffie-Hellman key exchange protocol, and utilizes the different sets of public/private keys illustrated earlier. To extend a circuit past the first relay node, the Onion Proxy sends a Relay cell (explained at a later stage) that has a subtype of EXTEND. This cell instructs the receiving node to send, yet, a freshly crafted CREATE cell to the next node in the path, which in turn responds with a CREATED cell. It's worth noting the second node in the path is not aware of the OP's identity at this point, and is only familiar with the previous OR in this path.

Figure 2 illustrates the process of circuit building in a formal manner, which also accounts for more than three nodes. The steps below are performed by the circuit creator (Onion Proxy).

```

Data: Final destination  $D$ , Pending Stream  $P_s$ , Set of Tor Relays  $\{R\}$ 
Result: Circuit  $C$ 
initialization;
Choose Circuit_ID :=  $y$ ;
Choose  $N$ ,  $N \geq 3$ ;
Choose  $R_N \in \{R\}$ ,  $R_N$  allows connection to  $D$ ;
 $R_N \leftarrow$  Exit Node;
while  $x < N$  do
  | Choose  $R_x \in \{R\}$ ,  $R_x \neq R_{x-1} \neq R_N$ 
end
Open Connection to  $R_1$ ;
Send CREATE CELL;
Wait for CREATED CELL;
while  $i \leq N$ ,  $i \in \{2..N\}$  do
  | Create EXTEND CELL;
  | Set payload to half-handshake, encrypted to  $R_i$  Public Key;
  | Send EXTEND CELL;
  | Wait for EXTENDED CELL;
  | Calculate Share Key;
end
 $C := \{R_1 \dots R_N\}$ ;

```

**Figure 2 Formal process of circuit creation**

This process concludes the mechanism by which an Onion Proxy is able to create a Tor circuit that spans over three or more Onion Routers. By now, the Onion Proxy is maintaining a list of  $N$  routers that constitute the circuit, each of which have negotiated a shared key indirectly with the Onion Proxy (with the exception of the first router  $R_1$ ), and the OP is now able to establish TCP streams to end destinations by crafting and routing special Relay cells across the path.



## 2.5 Relay Cells

Relay cells are the most common cells to traverse the Tor network, that is, they are mainly used to transfer end data (client to server or vice versa) or participate in some special circuit management tasks, e.g. extending circuits beyond the first router, tearing down circuits, etc. Within a circuit, the Relay cell is used to communicate messages between the Onion Proxy and the last Onion Router in the circuit path (i.e. the Exit Node), in addition to being the tunnel that pipelines end servers TCP streams.

Other nodes in the circuit path only route the relay cells to the next hop in the path, without being able derive clues about their contents, given that they are encrypted, and are set to fixed widths. Another observation is that while streams are only initiated by the OP, the exit node is able to initiate commands of its own, wrap it in a Relay cell, and route it towards the OP. The OP is able to perform the same as well. The payload of a Relay cell consists of five headers, in addition to a Data section that is interpreted differently based on the Relay cell headers (not to confuse this with the general cell header discussed earlier).

“Relay Command” is the first of those headers, and is a 1-byte field having a numerical value ranging from 1 to 15. Values observed from 32 to 40 are reserved for Tor hidden services and are of slight value to this research’s scope. The second field is referred to as “Recognized”, and is a two-bytes field that is set to zeros in the plain (unencrypted) Relay cell (further discussion below). “StreamID” is a two-bytes field that hosts a unique end-server stream ID that is arbitrarily generated by the OP to differentiate cells pertaining to different TCP streams. All Relay cells having the same

Stream ID are considered to be belonging to the same end server TCP stream. Stream ID is used in conjunction with Circuit ID to uniquely identify cells. Additionally, no OP is allowed to use a Stream ID of zero, as that ID is reserved to cells that affect the entire circuit rather than the single stream.

“Digest”, is a four-bytes field and is computed as the first four bytes of the running digest of all the bytes that have been destined (or originated) from the hop this Relay cell is intended to (either the exit node or OP). The combination of a correct digest in the Digest field and the Recognized field equaling zero, annotates the respective cell as successfully decrypted. The “Length” field is used to identify the length of the “Data” section, that is, the actual payload of the Relay cell. The remaining of the Data section beyond the identified length should be padded with zeros.

Some common examples of Relay commands values are 1 (RELAY\_BEGIN) which indicates the OP intend to establish a new TCP stream; 2 (RELAY\_DATA) an end-to-end TCP stream data (e.g. HTTP request/response); 3 (RELAY\_END) which communicates the client or server intent to terminate the TCP stream. A common example of the use of RELAY\_END cell is TCP FIN packet which can originate from either the client, server, or an intermediate firewall. Other important commands are RELAY\_EXTEND (6) and RELAY\_EXTENDED (7) which we’ve touched upon in the previous reading, and are used to indicate OP request to extend the circuit, and a successful extension response from the OR, respectively.

## 2.6 Streams

A client intending to open a new anonymized TCP stream, must first choose an established open circuit that has an exit node that is able to connect to the final destination (specified by that node exit policy). The OP must then generate an arbitrary stream ID that is not yet utilized on that circuit, and starts off by constructing a Relay cell of type RELAY\_BEGIN (command is set to 1). The payload of that cell should comply to the expected structure of a RELAY\_BEGIN cell i.e. the exit node will want to interpret an ADDRPORT field, that is null-terminated, in addition to an optional flag of 4-bytes. The ADDRPORT field represents the concatenation of a DNS hostname and a TCP port of the end server (destination) the OP wishes to connect to (e.g. `www.google.com:443`).

The optional flags are not heavily utilized by Tor at the moment, and are currently used to indicate the possible use of IPv6. Upon receiving that cell, the exit node will attempt to resolve that host name to an IP address, and established a new TCP connection to the desired port. It's worth noting the Tor defends against DNS blockage censorship implemented by some organization and countries, in addition to anonymizing DNS requests/responses by relaying that task to the exit node instead of the Onion Proxy (client). A common use of the RELAY\_END cell is when the exit node can't resolve the communicated hostname, or is unable to connect to the desired TCP port, and issues a RELAY\_END cell to the OP. Otherwise, the exit node replies with a RELAY\_CONNECTED cell and awaits further commands from the OP. Note that the RELAY\_CONNECTED cell holds the resolved IP address of the destination in its Data section for the OP to carry further verification should she wishes. Once a

connection is established (as indicated and communicated by the RELAY\_CONNECTED cell) the OP and exit node start tunneling the underlying TCP stream by packaging them in RELAY\_DATA cells.

## **2.7 HTTP Scenario**

Regardless of the end server application protocol, and whether it implies the use of TLS/SSL, the OP/Exit node will treat the application data in a similar manner. For the sake of this writing, we will demonstrate a typical HTTP scenario. The browser starts by creating a typical HTTP GET request, and package it within a transport layer packet (TCP packet). The browser, then, forwards that packet to the Onion Proxy, which in turn strips the content out of the packet and package it in a Tor Relay cell of type RELAY\_DATA. Depending on the content size of the received TCP packet, the OP might choose to further divide the TCP payload among multiple cells, or combine multiple TCP payloads into a single cell, the former being the most probable.

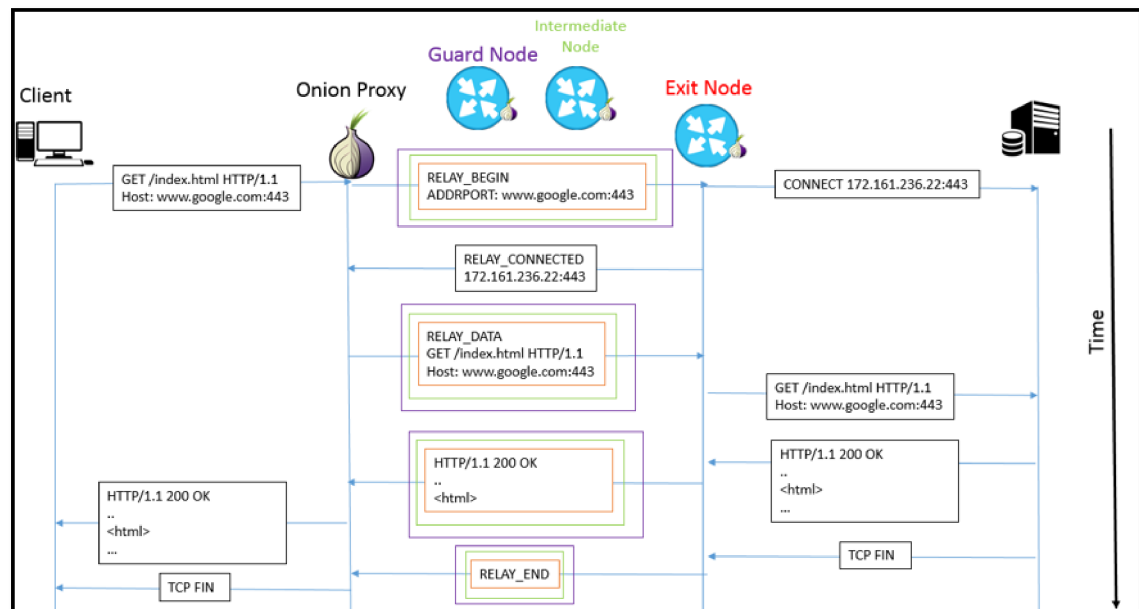
Recalling that an OP maintains the list of the three routers (or more) in the circuit path with their corresponding shared keys, the OP starts encrypting the cell payload (including all relay cell headers and Data section) by the shared key of the exit node (outermost), and work its way with more encryption layers utilizing the keys of the previous hop in the circuit path, ending with the first Onion Router in the path (nearest). At this point, a packed cell of 514-bytes, triple (or more) encrypted is ready to be sent and routed through the circuit. The OP wraps the cell in a TCP packet, which is encrypted a fourth time with the established connection key between itself

and the first node, and send it across the wire. The fourth encryption layer is not totally related to Tor mechanism, but is a result of establishing an SSL/TLS channel between the two nodes.

Upon receiving the cell, the Onion Router perform two decryption operations; the first utilizing the shared key between itself and the previous node as a result of using the shared SSL/TLS channel, while the second using the shared key that resulted from the CREATE/CREATED cells exchange when building the circuit. Again, this OR encrypts the cell with the key pertaining to the established secure channel between itself and the next node in the path, and send it across the wire. The receiving node performs two decryptions as well; the first utilizing the shared key between itself and the previous node, and the second with the shared key between itself and the OP.

Finally, this OR encrypts the resulting cell (which is still encrypted by the exit node key) and pushes it to the exit node through the wire. The exit node, perform similar two-iteration decryption, and perform a sanity check on the resulting cell, by computing the digest and verifying that the “Recognized” field is all zeros. The OP, then, extracts the Data field content from the relay cell payload, verifies it has an open stream corresponding to the stream ID in the cell, packages the payload into a new TCP packet, and pushes it across the established connection with the end server. From the perspective of the end server, it’s the exit node who made that connection entirely without the involvement of the OP. The iterative process of encrypting/decrypting is what gave Tor its name, as the multi-layer encryptions resembles those layers in an

onion, hence the name, The Onion Routing. The HTTP scenario described in the previous paragraph is illustrated in Figure 3.



### Figure 3 HTTP Scenario

## **CHAPTER 3**

### **LITERATURE REVIEW**

The term “Website Fingerprinting” was first coined in the year 2002 by Hintz [10]. He founded the idea of identifying websites by means of utilizing the different resources a certain website serves, while he referred to those resources as objects. In his research, he assumed that browsers will load each individual resource in a different TCP stream, and hence, each stream can be identified from a local observer point of view, through a tuple of source/destinations IP addresses and port numbers. However, Hintz was targeting a specific encrypting web proxy called SafeWeb, and only experimented on 5 websites achieving a detection rate between 45% and 75%.

Sun et al. [14], similarly, used the technique of Hintz in website fingerprinting with respect to object sizes. However, instead of using objects’ sizes identified by TCP stream, they adopted the use of packets counting between subsequent blocks of requests. Each website is identified by a multiset of object lengths, which is then compared to an unknown multiset obtained from a target traffic by applying Jaccard’s similarity. A threshold of similarity value is set, and a result above that threshold is considered as a match. In their work, they constructed a database of 2,000 websites fingerprint and tried to identify them among a test sample of 100,000 websites. They managed to identify 75% of those, by setting a similarity threshold of 0.7, with a false positive rate of 1.5%.

Liberatore et al [15], utilized a different approach to identify visited websites. Their approach takes a step down in the OSI model by analyzing packet sizes and the frequencies they appear at. In their work, they represented the traffic flow as a vector of packets' sizes frequencies, where each visit to a website will produce a histogram of packets' sizes frequency. Additionally, they employed the classification techniques of Jaccard's similarity and Naive Bayes to classify those vectors. In their work, they relayed on the University of Massachusetts's traffic by identifying the top visited 2,000 websites and were able to achieve a detection rate of 73% using Jaccard based classification.

As the need for anonymity systems became more demanding, more researches were focused on migrating the aforementioned Website Fingerprinting (WF) techniques and applying them to implement attacks on anonymity systems, most notably, Tor [3]. Shi et al. [16], for example, combined the techniques discussed in Hintz [10] and Sun et al. [14] and detailed a WF attack on Tor. In their work, they identified an interval as a time period occurring in a traffic capture without a change in flow. They, then, started tracking the number of packets in each interval, and representing the whole traffic trace as vector of intervals.

The vector identifies a website fingerprint by specifying the number of intervals with two packets, the number of intervals with three packets, and so forth. Additionally, they enhance the fingerprint of each website by multiple visits to confirm their findings. Finally, the acquired profiles of fingerprints is compared to a traffic trace of unknown websites, and the similarity is computed using cosine similarity. In their empirical experiment, they managed to identify 20 of the top websites in Japan with an accuracy rate of 50%.



Panchenko et al. [9] generalized their detection mechanism to both Tor and JAP [17], another popular anonymity system. In their approach, they used support vector machines (SVMs) by utilizing multiple traffic trace features. The traffic trace is represented by a sequence of packets lengths, where each flow direction is marked with either positive or negative values. Moreover, and to increase the classification accuracy, they injected additional features to the traffic flow when certain conditions are triggered. For example, the size of the packets in each interval is injected whenever the traffic direction changes and is referred to as “size marker”; the number of packets in each interval is injected whenever an interval ends and is referred to as “number marker”; total transmitted bytes, etc.

In their empirical work, they examined their technique in both open-world and closed-world settings. In the former, they used the same 775 websites that were used in [11] and were able to reach an accuracy rate of 30% using only the basic variant, and an accuracy of 54% when resorting to all features. As for the latter, they conducted the open-world experience on a set of 5,000 websites chosen randomly from Alexa [18] and considered amongst the top 1,000,000 websites in the world, in addition to five censored websites. The censored websites were identified with an accuracy rate falling between 56% and 73%, with a false positive rate of less than 1%.

The previous work discussed so far were all experimented under laboratory conditions. This resulted in more researches that aimed to pinpoint weaknesses when this work is applied in real world scenarios. Juarez et al. [19] work recognized significant differences in the

environment and users' behavior that impose a challenge on these techniques to work in the wild. Specifically, they identified six assumptions that previous work made, that limits the chances of carrying out realistic attacks: 1) Template websites that use similar or identical resources. 2) Closed-world experiments never tested with websites outside the monitored pages. 3) Attacks are vulnerable to stale training. 4) The assumption that users browse the web sequentially and not in parallel. 5) The adversary knows the beginning and end of a web page or a resource. 6) Most work ignores background traffic (OS, session control, browser plugins, etc.).

This led to more work that aims to eliminate these limitations in order to carry the attacks in the wild. For example, several researches [9, 20, 21] discussed tackled the second assumption and came out with an attack that can achieve a true positive rate of 85% in the open-world settings with no limits on web pages' number. Wang et al. [22] also tackled assumptions 3 to 6 by presenting a set of tools that augment current WF attacks to operate under realistic conditions. They defined the full traffic trace of a user as a full sequence, where each web page or resource is expressed as a cell sequence. To tackle the splitting problem (the process of converting a full sequence into cell sequences) they employed the use of two methods: Time-Based Splitting, and Classification-Based Splitting. In time-based splitting, a threshold,  $tgap$ , is defined as the optimal time gap separating two cell sequences (different web pages).

That is, if two cells are separated by a time difference of  $tgap$  or more, they are considered to be belonging to two different cell sequences. The output of time-based splitting is, then, passed to the classification-based splitting to cover any case of two or more pages separated

by a time difference less than  $t_{gap}$  and were mistakenly identified as one cell sequence. They further attempt to reduce background noise by means of classification and counting, and conclude that noise elimination is difficult, however, so as introducing deliberate noise in the traffic due to technical limitations in Tor, especially when using Tor browser. In fact, a small error in noise removal, could lead to a much larger one in page identification.

More recently, Website Fingerprinting is becoming a hot area of research in the past few years [34-38]. Panchenko et al. [35] realized that website index pages, frequently change, especially with those popular websites on Alexa top-X lists, and due to the dynamic nature of the modern Internet. Hence, they built a classifier that utilized more than 50 varying non-index pages of a website to represent its signature. The classifier applies ten iterations of cross validation to determine if a monitored website was visited, and achieved a near 86% accuracy rate. They also employed the use of several tactics to improve on the accuracy and success rate of their classifier, but felt short on subsequent visits to the same website, where the dynamic nature and the use of sessions, dramatically deforms a collected signature.

Herrmann et al. [37] conducted a thorough study of Website Fingerprinting with a wider scope that includes Tor, JAP, OpenVPN, Cisco IPsec-VPN, and OpenSSH. For a single hop system, their naïve bayes classifier outperforms Liberatore's approach and correctly identifies more than 90% of the requests in a closed-world settings of the same 775 sites mentioned in [9]. However, the accuracy dramatically decreases to below 3% when used against multi-hop anonymity system, such as Tor.

Giovanni et al. [39] considered a different approach in circumventing Website Fingerprinting attacks, by implementing the defense controls at the servers' side, which is more appealing to those employing Onion Services in their infrastructure, in addition to introducing a lightweight client side extension, that eliminates the need for mass deployment. They managed to demolish the long standing assumption of WF attack scalability, by only focusing on Onion Services, since they are 1) service size is trivial compared to the Web size, thus easier for an adversary to build a fingerprint database of all available onion services and 2) they usually contain and host more sensitive content where visitors may be subject to more serious consequences.

## CHAPTER 4

### MULTIPLEXING TOR TRAFFIC

Despite the active research field on the topic of Tor and anonymization in general, we find the lack of documentation with respect to the way Tor multiplexes circuits and queue TCP streams concerning. As described previously, Onion Proxies (OP) establishes circuits by randomly selecting three Onion Routers (OR) distributed across Tor network. The selection criteria is made based on the bandwidth of the respective ORs and the associated exit policy. Following, an OP will build a circuit on top of those selected nodes to create a path for cells to be routed. A circuit is always dedicated to a single OP, but can tunnel multiple TCP streams heading to different destinations.

Additionally, each two ORs on the path will establish a single Tor connection between them, and if any two circuits, established by two OPs, happen to use the same two ORs in a row, those circuits will have to share this single Tor connection. According to the latest Tor statistics obtained from Tor metrics [29], the active number of Tor users directly connecting to Tor network peaked 2,000,000 by 2017, while the number of Tor Onion Routers (relays) is little over 7,000. Hence, we can safely infer that connections between ORs will usually be shared between multiple circuits, especially those ORs offering high-bandwidth capabilities, and will likely be a more preferred selection in a path by OPs.

In other words, each OR will establish simultaneous connections to multiple other ORs, while only maintaining a single connection to each individual OR, and piggybacking that connection with as many circuits as required by OPs selecting those ORs in their paths. Additionally, each circuit will usually tunnel multiple TCP streams (of end-applications using Tor) given the design aspects of modern web sites, where a single page is referencing many resources, e.g. HTML, CSS, JavaScript, etc. and each resource requiring its own TCP stream. Also, each cell has a fixed width of 514-bytes, and is structured with a header containing meta-data and a payload. Upon cell arrival to a given OR, the respective OR will decrypt a layer of the cell, and parses the headers to identify the appropriate circuit to route this cell to. Recall that an OR is participating in many circuits and requires identifying cells' associated circuit.

Upon circuit identification, the OR, maintaining a different queue for each circuit that was built on a path containing this OR (circuit queue), will write the cell to the queue corresponding to that cell. Reardon [24] identified this process of cell parsing and writing to the queue as negligible time, cost-wise, and hence presented our research with an opportunity discussed in following sections. The moment a circuit queue is populated with a cell, it is marked as an active circuit, and vice versa, i.e. marked inactive upon de-queueing of all cells. Additionally, an OR will maintain a single output buffer for each established connection with other individual ORs, where data written to that buffer is transmitted to next OR in First-In-First-Out (FIFO) order.

Figure 4 illustrates the relationship between circuit queues and output buffers. In this example, four different OPs selected OR1 as their guard node. Thereafter, two OPs selected OR3 as the following (middle) node, while the other two OPs selected different nodes (OR2 and OR4) for their paths. After circuit establishment, OR1 will end up with creating 4 circuit queues, each representing the different circuits that were built using OR1 as a node in the path. Only cells coming from circuit one, will be populated to the queue designated for circuit one, similarly, circuits two, three, and four will only populate queues designated for their own circuits. However, since OR1 is only connected to 3 other Onion Relays, it will only create 3 output buffers, one for each Tor connection established between itself and the succeeding node in the path.

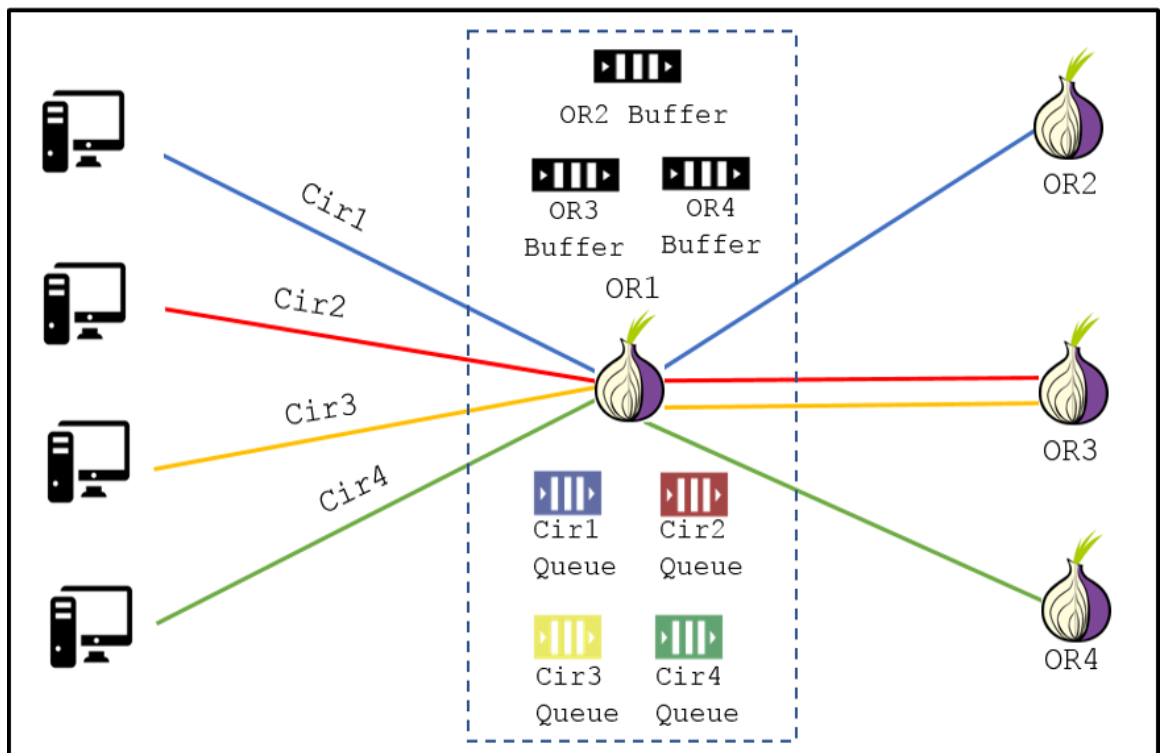


Figure 4 Relationship between Circuit Queues and Output Buffers

The reader can immediately spot the correspondence between the circuits' queues and output buffer. Since an OR can only sustain a single connection to the next OR, but still maintains multiple possible circuits with their queues that share that connection, a need to multiplex those queues to the output buffer arises. When the output buffer is available for more data to be written, the OR will choose one from the available active circuits, and start moving as many cells from its designated queue to the output buffer to fill the available room. The process by which an OR decides which active circuit to move cells from to the output buffer, is referred to as Tor circuit scheduling. In the previous figure, whenever OR3 output buffer has some room available, OR1 must invoke its scheduling algorithm to decide whether to move cells from circuit two or circuit three queue. This results in creating a multiplexed connection between nodes OR1 and OR3, where the single connection carries cells from both circuits.

Earlier versions of Tor employed the use of Round Robin to select from active circuits pool. Tang et al. [25] discussed the performance overhead of utilizing Round Robin fashion on Tor multiplexing and argued that bulky transfers, such as those downloading large files, or connecting to multiple peers (such as BitTorrent) are always prioritized over burst connections, such as those resulting from users surfing the web. In their research, they showed that by employing a different scheduling and selection algorithm (i.e. a multiplexing algorithm), they can insure that burst connection are almost always privileged, and that their cells are moved to the output buffer before those cells in bulky transfer queues.



They proposed an implementation of a more judicious selection criteria based on the Exponentially Weighted Moving Average (EWMA) of circuits. In their approach, they favored those circuit queues with less sent cells over a period of time, by assigning a cell counter to each queue and building a metric of selection. The metric kept readjusting the average value of sent cells, while decaying over a set period of time. When selecting from active circuits, the circuit queue with the lower metric value will be the one to push cells from, thus increasing the overall performance of burst connections, while holding those with bulky transfers (or more active) queues for a little longer.

Indeed, this approach appears to induce randomization to a certain degree, especially when considering the vast amount of Tor users, and the tendency to select nodes with higher bandwidth. However, the motivation behind their approach is purely focused towards performance improvement, hence it lacks the required degree of randomization to disperse traffic analysis attacks. First, the approach heavily relies on the coexistence of multiple circuits on a single connection between two ORs. This exposes those circuits that are relying on two consecutive low-utilized nodes, or those circuits that are uniquely utilizing a Tor relay.

One might come to think that due to the vast amount of users, and their dependency over much lower number of nodes, that this probability is far from occurring. However, consider the link between the Onion Proxy (client) and the first node in the path. As discussed earlier, this link employs a single connection, and is rarely seen to

host more than a circuit. This is the case with every Tor user that wishes to connect to the anonymity network (except in cases where a pool of users share a single Onion Proxy). While Tang et al. approach works well against traffic analysis attacks in the distributed Tor relays, it ceases to provide the same level of protection when the relationship between the output buffer and the circuit queue becomes one-to-one. In fact, it's this link that organizations and oppressive regimes have control over, rather than the distributed network over the globe.

Another point of concern is streams. Recall that each circuit will probably be tunneling multiple TCP streams. Also, recall that an OR will read from the circuit queue as First-In-First-Out. This implies that Tor will be sending out cells to the wild in the same order as they are received from the browser, and the only mechanism of defense would be the existence of multiple circuits thriving to utilize the link. In our empirical work, we show that this assumption is true, and that browsers behavior with respect to stream construction plays a major role in traffic analysis attacks.

Despite that Tor multiplexing wasn't implemented as a security measure, rather an operating requirement, we aim to examine the potential by which it can aid to increase the overall security of Tor. In the following sections, we discuss a scenario where EWMA acts as a pipeline, and the employed scheduling algorithm ceases to work. Additionally, we propose our approaches to enhance this algorithm by introducing, yet, another multiplexing degree on the streams level and discuss its implication from a security perspective.

## 4.1 Single Circuit Scenario

As discussed, Tor multiplexes circuits whenever two circuits or more select the same two consecutive nodes in their paths. This leads to a scenario, where the two circuits compete to utilize the same link between the two nodes, since Tor will only maintain a single link to the next node at a time. Tor resolves this by invoking a scheduling algorithm that fairly decides which circuit will have the priority to utilize that link. However, most Website Fingerprinting attacks are executed by a local observer who possess the ability to monitor the link between the OP and the guard node. Additionally, a typical Tor user will mostly be running a single circuit and initiating as many TCP streams on it, until that circuit expires.

Since a single circuit exists, Tor will still invoke its multiplexing algorithm (circuit scheduling), however, it will only have a single circuit to choose from. Additionally, the cells in this circuit are ordered in the way they are received from the browser. Figure 5 show a typical use case of Tor, where a user browses to a certain website. The browser first loads the HTML pages and parses it for references to other resources (CSS, JS, images, etc.). The browser, then, executes a series of GET requests, each corresponding to a resource on the page, which yields an equivalent number of TCP streams to be opened. Tor arranges those stream (GET requests) in the order they are received from the browser, and packs them in cells. Those cells are considered belonging to the same circuit, and are populated into that circuit's queue. Whenever Tor is ready to send more cells to the guard node (i.e. output buffer of guard node has some space), it will invoke its scheduling algorithm to select among the active circuits.

Since a single active circuit exists, cells will move from the circuit queue to the output buffer in FIFO order, i.e. as received from the browser.

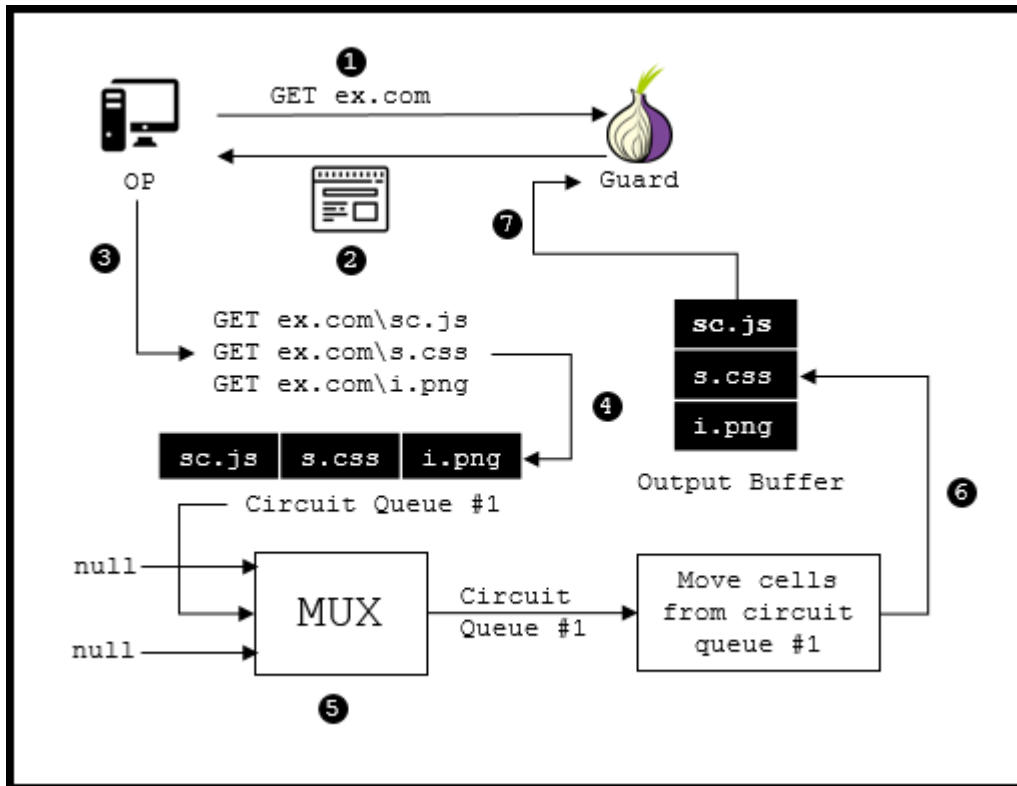


Figure 5 OP Browsing using one circuit

Unless the Onion Proxy (client) established more than a circuit, the previous scenario will always be the case. A single active circuit is available to move cells from, thus eliminating the randomization induced by multiplexing. Additionally, the same behavior will be observed when the guard node responds to client's requests (e.g. HTTP response). The cells packing the requested resources will come ordered as the web server has processed them, and as the exit node received them. The guard node

will have a single circuit queue resulting from the OP establishing the circuit, and a single output buffer corresponding to the single connection established between them.

Not only will this case occur on this link. Consider a lowly utilized relay anywhere on the path. Due to its low utilization, there will be a minimal number of circuits built using that relay, and hence, a higher chance that this relay will also maintain one-to-one relationship between the participating circuits' queues and the output buffer to other relays, thus eliminating randomization of circuits induced by multiplexing.

## **4.2 Approach Motivation & Proposed Algorithms**

In this section, we propose our enhancements on Tor’s algorithms for streams queueing. We state the approach motivation and rationale behind our proposal that lead us to bring about these algorithms. Finally, we describe our multi-approach enhancements over the currently implemented algorithms by suggesting three different implementations that induce additional randomization to Tor, with different performance overhead.

### **4.2.1 Approach Motivation**

So far in this writing, we have deeply investigated, discussed, and displayed the most crucial attacks the literature has identified on Tor’s Traffic Analysis. As discussed, Tor differentiate itself from traditional VPN and Proxy-based approaches in general, by transferring TCP packets and streams from their default behavior and appearance. Tor packages TCP streams into fixed-width cells, and performs massive transformation by either dividing TCP packets contents across multiple cells, or merging TCP packets payload into a single cell, depending on traffic distribution. Yet, Tor also multiplexes circuits that are tunneling different TCP streams into a single connection, thus increasing the challenge for the observer.

Despite that, the rich literature showed that all of those defense mechanisms suffer from a great shortcoming, they are systematic and predictable. If we look at Tor as a black box, and when we supply a certain input to that box, the result is always

predictable with a high certainty in both open-world and closed-world settings. A user browsing only amazon.com, will cause Tor to grab specific resources in a specific order, not only that, but an observer can predict how many cells are going to be exchanged in each direction. Some have gone to further lengths, and identified the exact splitting points that Tor follow.

Additionally, we've discussed Tor's multiplexing algorithms, and how they operate with respect to circuits established by different OPs. Also, recalling how circuits are built, the Onion Proxy will choose an exit node that has an exit policy that allows for connecting to the remote destination. Once a circuit is built, it's very rare the case that an OP is faced with a situation where he has to create extra circuits or rotate his exit node to accommodate a new destination. The OP will keep utilizing the same circuit for new destinations.

Hence, from our observation, we concluded that circuits multiplexing is eliminated at the link connecting the OP with the first node in the circuit path (Entry/Guard node). The reason as stated, is the lack of circuits to multiplex. The OP and the first node in the path will only establish a single connection (which is the norm), but build a single circuit on that connection, resulting in a First-In-First-Out queue, with no multiplexing.

This led us to determining that the only factor missing from this complete system of defenses is randomization. Randomization has always been a critical characteristic

that appeared in all security algorithms, whether in the fields of encryption, hashing, or even security applications. Hence, we sifted through the different design aspects of Tor in attempts to find the perfect venue to implement randomization. In the following sections, we highlight on this effort, and discuss with further details our attempts to introduce randomization to Tor.



### 4.2.2 Proposed Algorithms

To thwart against the most successful traffic analysis attacks, we believe that stream randomization is the key. Recall that Reardon [24] highlighted that cell parsing and circuit identification from cell headers, is a time-negligible process. Also, recall that one of our research objectives is to introduce a randomization characteristic to Tor, with little overhead in performance, or none at all. Hence, we started sifting through the design aspects of Tor, in attempts to identify the perfect location where randomization can be introduced, and seamlessly integrate with Tor while maintaining our design goals inline. Additionally, we wanted that change to cover all possible scenarios of Tor usage, such as those on low utilized links between Onion Routers, and the observable link between an Onion Proxy and the first Onion Router.

Streams were the key. By introducing a stream randomization mechanism in Tor, we will guarantee the alignment of our goals towards the proposed change. From the perspective of an OR, streams are meaningless. Recall that a cell payload is encrypted and is only viewable by the latest node in the circuit path (exit node). Also, recall that Stream ID is part of relay cell Data section, within the cell Payload section. Hence, for an intermediate OR, a cell will be forwarded down the path regardless of its stream. In our solution design, we hypothesis that intermediate relays will have no objection in receiving cells out of order, as ORs only require the knowledge of a Circuit ID. Intermediate relays are all those nodes participating in the circuit path, excluding the exit node, but including the entry (guard) node.

For the above mentioned circuit queues, Tor implements what is called a “Simple queue”, a representation of a basic linked list. The simple queue is only exposing pointers to its head and tail. Additionally, a single implementation of a queue popping algorithm is available, namely `CELL_QUEUE_POP`. The queue expects every element to be of type Tor cell, and offers a single pointer for each element, that is, next element. Additionally, the queue is only traversable by accessing the head, where an iterator can access the next element using the next pointer. Finally, the queue doesn’t allow selective access by index.

As TCP packets arrive from the browser, and in the context of a Relay Data cell (e.g. HTTP GET, a TCP ACK, HTTP response, etc.) Tor strips the content out of the TCP payload and evaluates its size. If the content can fit the boundaries of a cell payload, while accounting for relay cell headers, the content is not split, otherwise the content is split among two cells. On the other hand, if the content is small enough, Tor will examine the following packet in the same stream to test if the two TCP packets can be fit into a single cell. It’s worth mentioning that Tor accounts for different streams, and respects the order they arrive at.

Consequently, Tor packs the resulting payload into a new cell, and populate all required headers, such as Stream ID, Circuit ID, Length, etc. Then, the payload is triple encrypted as detailed earlier, and the new packed cell is pushed to the tail of the queue. At this point, it’s not possible to induce any information about the cell, as all cells in the queue will look similar, due to encryption, and fixed-width. Upon the

invocation of the multiplexing (scheduling) algorithm, and the determination of the active circuit to read cells off, Tor will access the head of the selected queue and invoke the `CELL_QUEUE_POP` method, in its single offered implementation. The pointers are then updated, and the following element becomes the new head of the queue.

In our approach, we suggest three algorithms to manipulate the behavior by which Tor pops cells from the circuit queue. Two of these methods relay on modifying the behavior by which Tor pops cells while maintaining the structure of the queue intact, while the third redesigns the queue structure. Also, it's important to note that we considered other approaches that depend on modifying the behavior of queueing the cells (as opposed to popping them), however, a quick evaluation showed the need to reflect this change on all Tor relays around the globe to accommodate that change.

#### 4.2.2.1 First Approach

This is the easiest approach to implement and induces the highest cost on performance. In this approach, we propose a new popping method that randomly selects from the available streams in the queue, instead of always popping the first element. However, we have to cater for in-stream order, that is, we don't want to break the underlying application's protocol, by popping later cells from the stream, before flushing earlier cells. We can intermix the streams as we wish, but we can't intermix within the same stream.

Also, recall that Tor maintains a stream ID of zero, for those cells that affect the entirety of the circuit. Those cells that belong to stream zero, are order-sensitive, and have to be popped in the exact order they are pushed to the queue. The changes induced by this approach only introduce a newly implemented method to pop cells in a random order. Basically, we are adding a new queue access method with no other changes in Tor. Additionally, this approach is backward compatible. Here is an example of normal scenario that illustrates the algorithm in work.

- 1- When a request is made to pop a cell, we scan the whole queue and identify all streams that are in the queue.
  - a. For the sake of this example, say we identified 5 streams. Note that to get this information we had to traverse (say) 100 cells currently in the queue.
  - b. For each identified stream, we record a pointer to first cell of that stream in the queue and build an index matching each stream with the cell.

- 2- We select a random number between 1 and 5 (number of identified streams). This is the stream number randomly selected.
- 3- By consulting with the created index mapping, we pop the first cell in that stream.
- 4- Reconnect Tor Simple Queue by updating the popped cell's previous item "next pointer", and the popped cell's next item "previous pointer".

Note that the scan and building of the mapping, is being performed every time a cell is to be written to the output buffer, i.e. needs to be popped. However, this approach is mess-free, a randomly selected stream will always provide a cell to be popped. Recall that this approach will scale up when you have many queues (a queue for each circuit) in a Tor Onion Router. Also, the index mapping created above, is locally used in the newly created access method, and can't be reused. This is to avoid further modification of Tor, and eliminate the need for mass re-deployment.

Also, recall stream zero, and the need to push cells from that stream in order. A modification to this approach is made to accommodate for cells belonging to stream zero. When we perform the initial scan of the queue, we stop at the first occurrence of a stream zero, and select randomly from the streams available prior to stream zero. When the first cell of the queue is a cell belonging to stream zero, we pop it immediately without further scanning. Figure 6 illustrates this approach, while Figure 7 demonstrates its implementation.

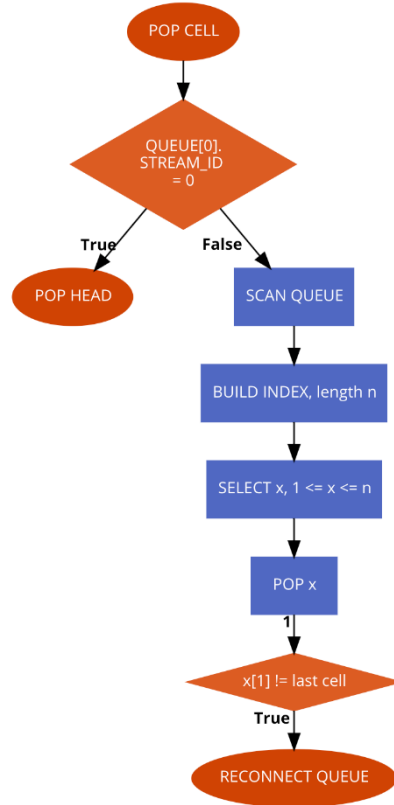


Figure 7 Illustration of the first approach]

```

Data: Circuit Queue  $Q$ , Stream Mapping  $S$  {StreamID, PointerToCell}
Result: Cell  $C_r \leftarrow$  random cell
initialization;
 $N \leftarrow$  length of  $Q$ ;
while  $x < N$  do
   $CID \leftarrow$  Circuit_ID $\{x\}$ ;
  if  $CID := 0$  AND  $x := 0$  then
    POP CELL;
  else if  $CID$  NOT in  $S$  then
    APPEND  $CID$  to  $S[StreamID]$ ;
    APPEND  $Q[x]$  to  $S[PointerToCell]$ 
  else
    next;
  end
end
 $L \leftarrow$  length of  $S$ ;
 $z \leftarrow 1 < z < L$ ;
POP  $S[z] \leftarrow C_r$ ;

```

Figure 6 First approach pseudocode]

#### 4.2.2.2 Second Approach

In the second approach, we implement a two-fold solution to enhance on the performance of the first approach. The first part is performed before every cell push to the queue, while the second part naturally occurs before popping a cell from the queue. On average, this approach costs  $\log_n$ , however, worst case scenario is same as in approach one, that is  $n$ .

PRE PUSH:

- 1- Before a cell is pushed to the queue, we note the Stream ID. We build a mapping of Stream IDs, and the number of cells in the queue with that stream.
  - a. If this is the first time we see the stream, we append it to the mapping.
  - b. Otherwise, we increment it's counter by one.
- 2- We push the cell normally to the queue.

PRE POP:

- 1- Select a random number between  $1..n$  where  $n$  is the length of the created mapping (Stream ID/count). This is the stream number that we will pop the next cell from.
- 2- We insure that the counter of that stream is  $> 0$ . If it's equal to 0, it means no more cells in the queue are available from that stream. This is a mess, we select again.
  - a. This mess is on the scale of our built mapping which corresponds to the number of streams, not the number of cells in the queue.
  - b. That stream ID is removed from the mapping to avoid further mess.
- 3- We start iterating Tor queue and examine the stream ID of each cell.

- 4- We pop the first cell in the queue that has a stream ID of the randomly selected stream and rejoin the queue.
- 5- Decrement the counter in our mapping by one.

This approach will not modify Tor queue, but only the accessing methods. However, it has the potential of scanning the full queue, e.g. we only have one cell of the selected stream towards the end of the queue. Moreover, it also has the potential of being much faster than the first approach on average. This method only adds one queue accessing (and popping) method, in addition to a global mapping to maintain the streams and cell counters. This approach is also backward compatible. Figure 8 illustrates this approach.

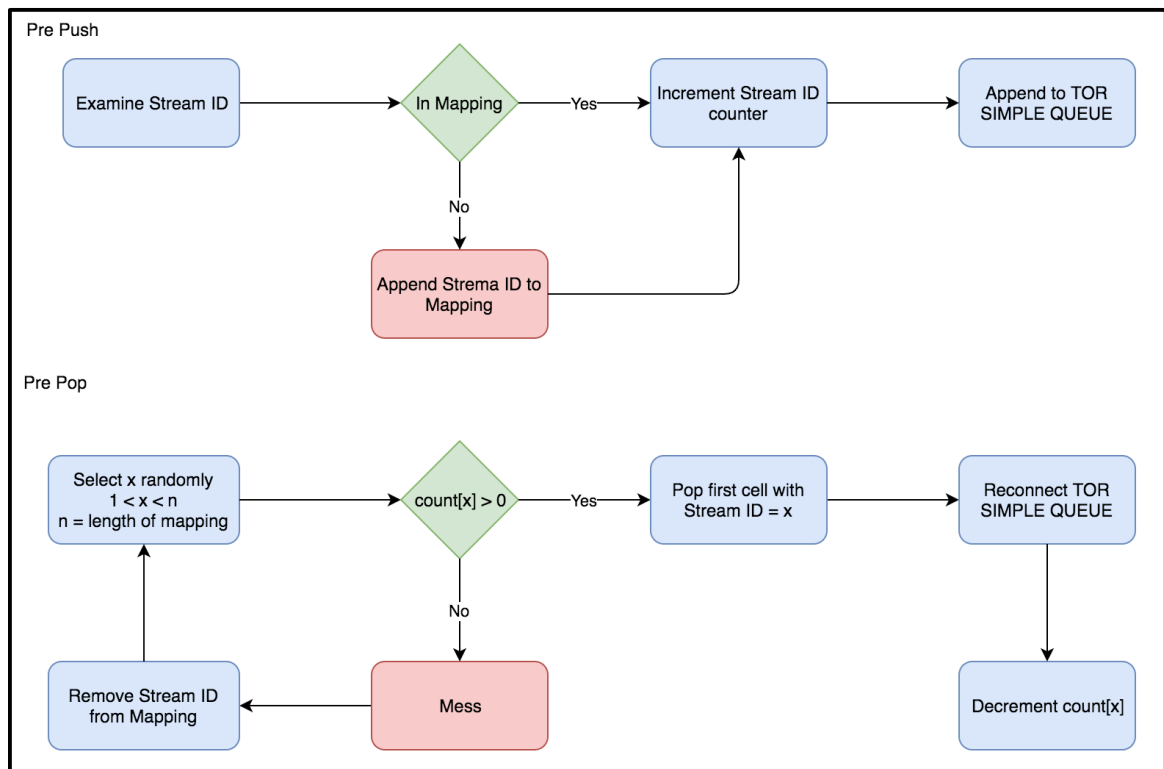


Figure 8 Illustration of the Second Approach



#### 4.2.2.3 Third Approach

This should be the fastest, most efficient, and thus, requiring the most changes. In principle, we abandon the simple queue and create a new one with a much efficient data structure, such as linked list, hashmap, etc. The queue will be two dimensional, where the first dimension is the stream number, and the second dimension is the actual cells belonging to that stream. Before pushing a new cell to the queue, we examine the availability of the stream in our queue. If the stream is available, we append the new cell to that stream second dimension. Otherwise, we create a new entry for that stream in the first dimension, and push the cell to its second dimension. An example queue illustrating the idea of this approach can be seen in Figure 9.

Again, when we want to pop from the queue, we select a random number between  $1..n$ , such that  $n$  is the length of our first dimension. Immediately, we pop the first cell in that queue since they are already in order. Also, this approach involves creating a new data structure, pop/push methods, and other queue auxiliary methods and macros.

<b>Stream 1</b>	Cell 1	Cell 2	Cell 3	Cell 4
<b>Stream 2</b>	Cell 1			
<b>Stream 3</b>	Cell 1	Cell 2	Cell 3	
<b>Stream 4</b>	Cell 1	Cell 2	Cell 3	Cell 4
<b>Stream 5</b>	Cell 1	Cell 2	Cell 3	
<b>New streams</b>				

Figure 9 Third Approach Queue Example

## CHAPTER 5

### EXPERIMENTS AND RESULTS

#### 5.1 Testbed

In this section, we describe the different testbed setup we used throughout the experiment. In general, all machines used in the experiments and testing were virtual machines hosting Linux Ubuntu 16.04.2 (Xenial Xerus). Additionally, the VMs were allocated a single core, each, with 8 Gigabytes of memory. Moreover, we utilized a 200 Megabits Internet connection. Finally, the Tor version installed was alpha 0.2.9.8

##### 5.1.1 First Setup

In this setup we utilized one virtual machine to host Tor, and it was configured to connect directly to the internet, and consequently, to Tor network. The setup is illustrated in Figure 10.

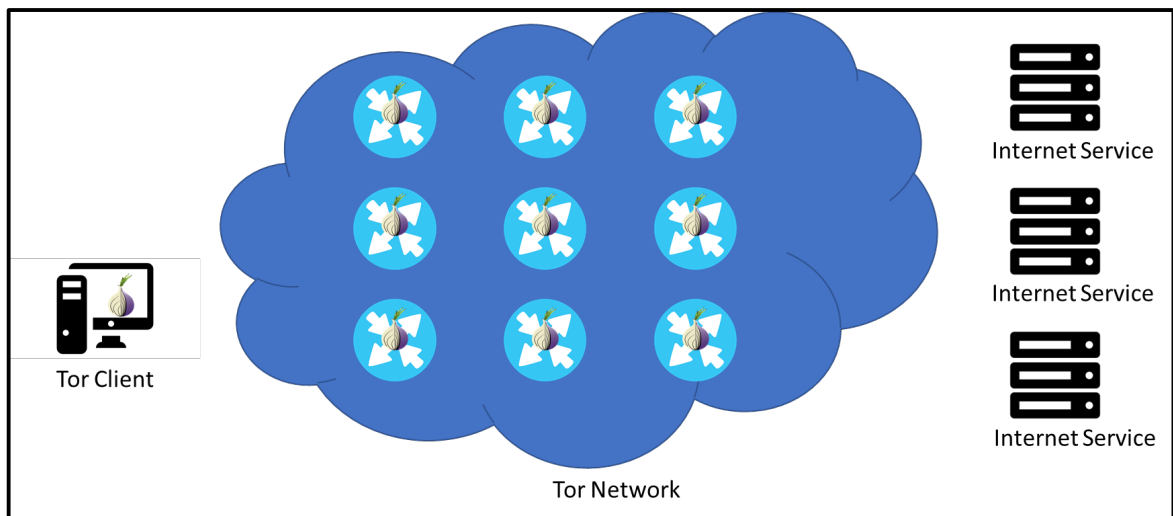


Figure 10 First setup illustration

### 5.1.2 Second Setup

In this setup, we used two virtual machines to establish connections to Tor. The first machine is a regular end-user machine, that has a Firefox version 50 installed, alongside a network sniffing application, namely tcpdump. Additionally, a second virtual machine was setup to host Tor, and exposed its services to the local network. Further details will be displayed within each experiment. The setup is illustrated in Figure 11.

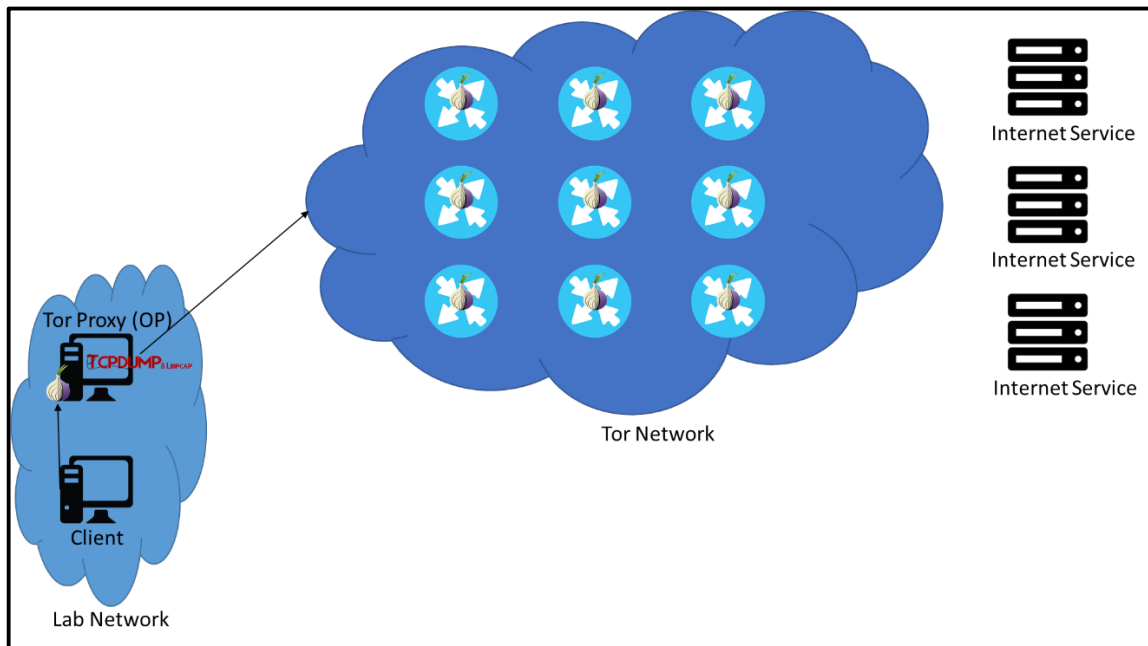


Figure 11 Illustration of the second setup

## 5.2 Tor Multiplexing at Onion Proxy

From Tor design documentation, investigating the source code of Tor, and throughout the literature, we know that Tor is supposed to multiplex cells. However, by invoking many test cases of Tor, we weren't able to reach a concrete evidence that Tor is indeed multiplexing cells. Additionally, we needed to deeply understand how the multiplexing is performed, and to what extent. We needed to find out if Tor pushes streams as they are received from the browser, or does Tor perform some sort of multiplexing on the same circuit, i.e. multiplex streams.

To achieve this, we decided to design a series of different tests, by instrumenting Tor to print and log certain parameters required to establish a fair comparison of circuits and streams. At this point, we only had an intuition about the behavior of Tor, and we needed to establish ground truth and develop a sense of understanding of the multiplexing behavior.

### 5.2.1 First Experiment: Tor General Behavior

Our first experiment was broad and aims to define the general picture of Tor while logging as much information as required to enrich our understanding. To accomplish this, we wanted to graphically interpret the relationship between different streams, and examine if we can detect any kind of similarity or repetition in Tor's behavior. The existence of any repeated behavior would indicate that Tor is systematically

distributing and selecting streams to be written out the wire, while the lack of such pattern could indicate the opposite, among other possibilities.

#### **5.2.1.1 Scenario and Implementation**

To implement our first experiment, we instrumented Tor to printout two parameters for every cell created; i.e. Circuit ID and Stream ID, in the same order as Tor is creating and processing those cells, but before they are pushed to the queue. The output is to be directed to a collection of text files, where each file represents a circuit, and each line of that file documents the stream number of the respective cell, and in the particular order Tor created that cell. Also, recalling that the first cell of a Tor stream is a DNS query relayed to the exit node for resolving, we instrumented Tor to printout the domain name from the first cell of a stream, filtering on the previously explained RELAY\_BEGIN command in the Relay cell headers, and utilizing the ADDRFIELD of its payload.

This will allow us to print the destination of each stream and create a mapping between Stream IDs and their destination, thus enabling us to filter out those streams that are of irrelevance to our experiment, such as circuit control, directory servers' connections, etc. and focusing on streams heading to destinations of our choosing. Consequently, and after each run of Tor, we expect to have multiple text files, each representing a circuit created by the Onion Proxy, and each file containing a variety of streams tunneled through that circuit. Additionally, we will be able to link each stream to its destination by printing out the mapping of stream IDs vs their DNS query.

Recall that we are testing for streams multiplexing from Onion Router perspective, hence, we want to push as much data out from the client to observe the multiplexing behavior. This is as opposed to pulling data towards it which will only show the results of multiplexing from the entry node, i.e. first node in the circuit, as that node will be the one multiplexing the incoming streams into the single connection towards the client. Therefore, uploading relatively large files will naturally allow us to observe that relation in question.

Moving forward, we created a custom web application that allows for a file upload in a single stream. The web application is implemented using NodeJS “http” module to build a simple webserver and an HTTP request/response component, in addition to utilizing “Formidable”, a known NodeJS library for parsing forms, handling file uploads, and allowing for single stream uploading by default. From the client side, we crafted a set of three pairs of files, sizing 1 Megabytes, 5 Megabytes, and 100 Megabytes, respectively. Each file was populated with random characters to reach the required size using a Perl script. Finally, a simple bash script was created to simulate a browser upload of two files utilizing curl (a command line tool to for client-side HTTP protocol utilization), simultaneously, while starting the upload at the exact same time.

### 5.2.1.2 Experiment

Following we launched Tor with our instrumentational logging, and initiated the upload of each pair of files of the same size together. In this experiment, we aimed to create a racing condition between the upload of the two files, thus allowing us to illustrate the behavior Tor will exert on the competing two streams, and hopefully observe a systematic behavior in the output streams ordering. For each pair of files, we repeated the experiment ten times and collected the output files, resulting in 30 runs, where each pair of files of the same size were tested together.

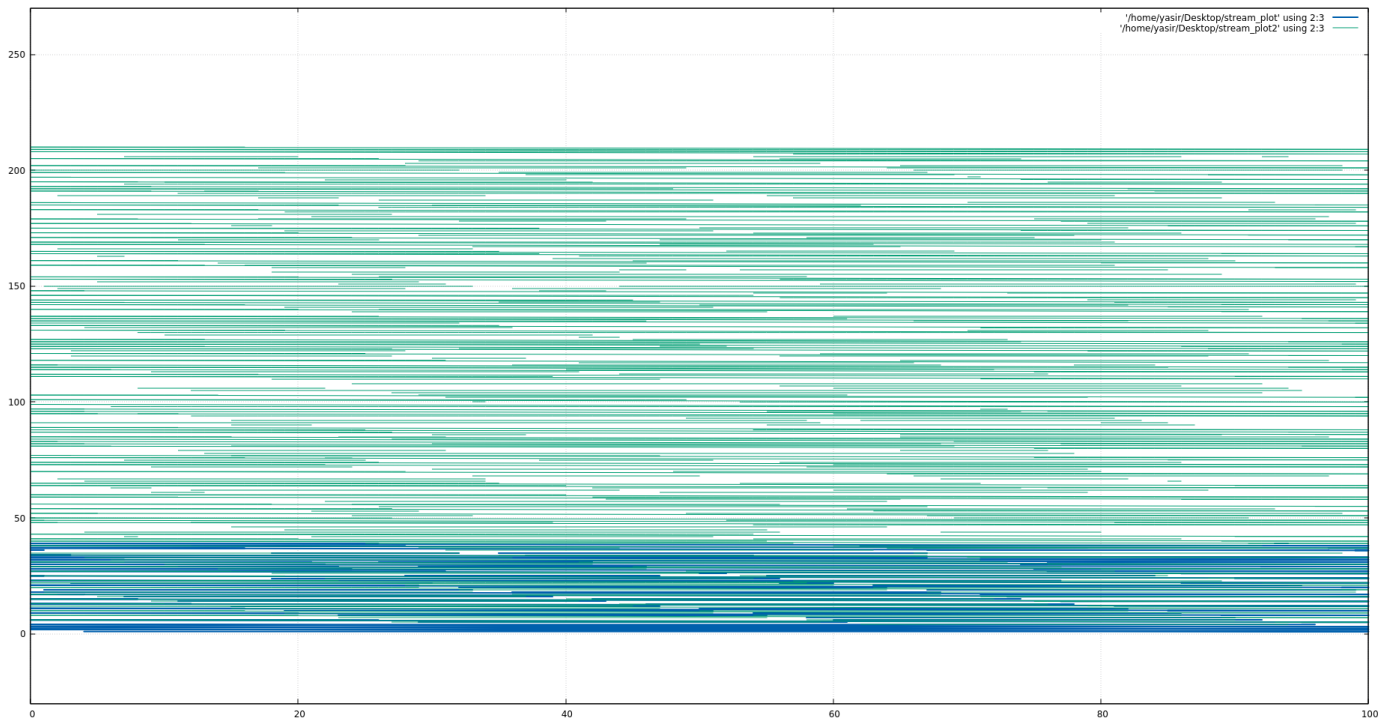
Recalling that we organized the output to print each circuit's streams in an individual file. To our surprise, each run resulted in a single output file. This is the first indication that Tor Onion Proxy (client) only built one circuit, and utilized it for both files uploads, in addition to other Tor activities (circuit building, key negotiation, etc.). By resorting to our created mapping of Domain Names/Stream IDs, we were able to cleanse the output files from all Tor inner communications, and focus solely on traffic heading to our website. This resulted in a data file that has only two streams that are alternating in the order their cells were created in Tor.

To prepare the data for plotting, we used a Python script to transfer the output files from a listing of streams in the order they appeared in Tor, to data points representation to be plotted on a diagram. We set our X-Axis maximum length to 100 (for representational purposes only), then, we start iterating through the streams, and we assign an incremental value for each stream, to represent its location on the X-axis,

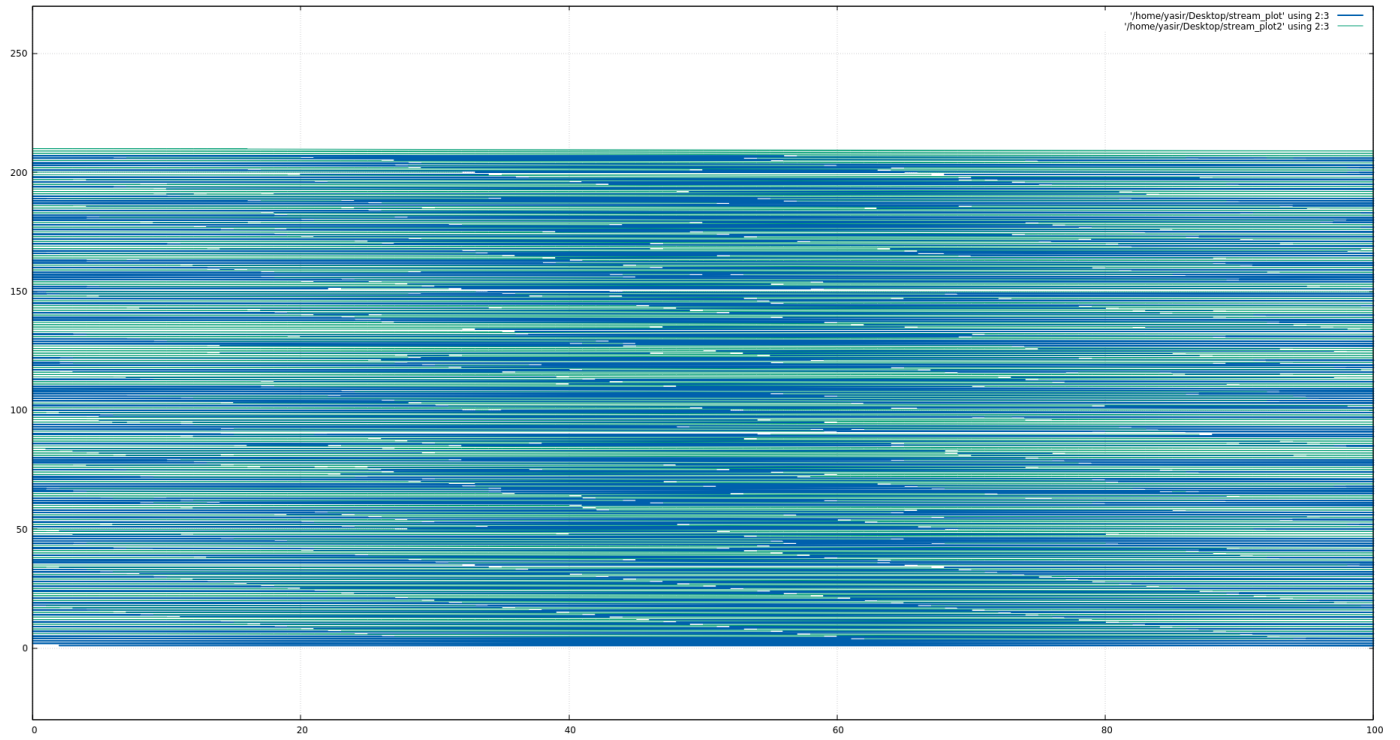


starting by 0. Initially, the stream value on the Y-Axis is 0, and we don't increment it yet. Whenever, we reach a 100 on the X value, we increment the Y value by 1, and reset the X counter. This transformation resulted in a data file where each occurrence of a cell belonging to either streams is given a position on the (X,Y) axis.

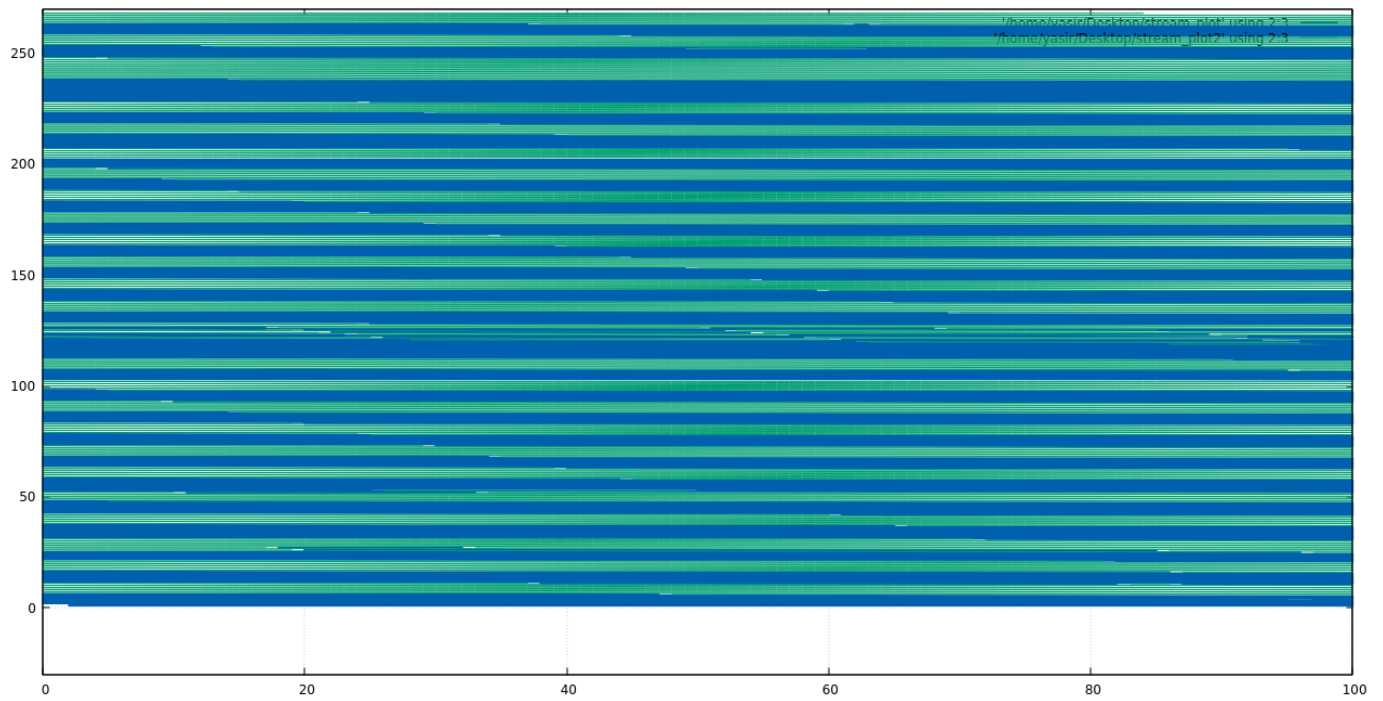
Moreover, we used Gnuplot [28] (a command line graphing tool) to illustrate the intermixing of the streams. As an input, the Gnuplot script will take the Python processed data file, whereas it will output a diagram showing the order and intermixing of streams, where each stream is given a unique color. Furthermore, consecutive points of the same streams are represented by a line connecting them. Figures 12, 13, and 14 show a sample output diagram representing one run of a pair of files of the same size being uploaded.



**Figure 12 Stream intermixing of two files sized 1 MB each, simultaneously being uploaded using Tor**



**Figure 13 Streams intermixing of two files sized 5 MB each, simultaneously being uploaded using Tor**



**Figure 14 Streams intermixing of two files sized 100 MB each, simultaneously being uploaded using Tor**

### 5.2.1.3 Findings

As the reader might have concluded, the three diagrams shows no relation or similarity among them. This was the case with all the 30 runs we have conducted, as different runs of the same pair of files show similar result every time, however, no similarity is observed across runs of different sizes. In Figure 12, where the file sizes are relatively small (1 MB each), we can notice that the first stream (i.e. upload operation) represented in blue concludes much faster than the second stream (in green). However, when we look at Figure 14, where the two files being uploaded are much larger (100 MB each) we can see a pattern of alternation. Each stream is given an equal time share to upload a portion of its content.

This observed behavior of alternation is not induced by Tor. In fact, those alternation and allocation of bandwidth are controlled by the OS, and specifically by the bash process that initiated this upload, where each process is allocated an equal share of resources in quantum-basis. For the first case of 1 MB files, the first file was able to upload most of its content in the quantum given, hence showed no intervention on following quantum. We will continue to show in following experiments that this behavior is induced by the bash process (i.e. the browser). So far, and from the figure illustration, and since we haven't observed any similar pattern when two streams are competing to utilize the link between OP and the first OR, we are assuming that Tor hasn't induced any multiplexing on the two streams before pushing the cells to the queue. We will continue to investigate other aspects and components of Tor further below.

## **5.2.2 Second Experiment: EWMA Multiplexing Behavior**

To further examine the possibility of streams multiplexing, we had to understand the effects induced by EWMA circuit multiplexing algorithm developed by Tang et al. [25].

### **5.2.2.1 Scenario and Implementation**

We wanted to perform a comparison between the order of cells before entering the queue, and as they are leaving the queue (i.e. written to output buffer). We could easily identify cells before entering the queue, as those cells are freshly created and are not encrypted yet, hence their stream IDs are still exposed in plain text. However, recall that cells go through three rotations of encryption as they are populated to the queue, and unless we had access to the Exit node, we couldn't possibly tell which ones were popped first.

Hence, we resorted to identifying the cells through the use of hashing. Initially, we record the stream numbers cells correspond to, in the order we received them from the browser, which also corresponds to the order of cells being pushed to the queue. This is in a similar fashion to the first experiment, however, we don't resort to multiple circuit output files, as we concluded that the OP will only create a single circuit. After a cell is received and its stream is recorded, we let Tor perform the three encryption

rotations it requires, and just before the cell is pushed to the queue, we calculate the hash of the entire cell in its triple-encrypted format, and append it with the respective stream number in the output file. Again, we utilize the first cell of the stream to identify the final destination of the cell, i.e. the domain name.

Additionally, the mapping between each cell and its hash is stored in a hashmap data structure, that will be used to identify cells as they are exiting the queue. Since Tor is a network application and is delay sensitive, and since the need to use hashing is experimental and for identification purposes only, we couldn't compromise to use an expensive hashing algorithm. Hence, we elected to implement a simple, yet effective, stream-rotational hashing algorithm that doesn't induce collisions frequently. The pseudocode of the used hashing algorithm is showed in Figure 15, which is an implementation of the algorithm used in SDBM a public domain implementation of ndbm (the UNIX database), in addition to being used Berkeley DB [33].

```
Parameters: unsigned char str  
Output: hash  
initialization;  
hash: unsigned long int  
c: int  
while c = str++ do  
|   hash = c + (hash << 6) + (hash << 16) - hash;  
end
```

Figure 15 Hashing algorithm used to identify encrypted cells

By the end of this process, we expect to have the order of cells as they are pushed to the queue, in addition to a hash for each cell representing an identification of it in a single output file, as well as having the same information populated to the hashmap data structure.

Thereafter, we instrument Tor to print the order of cells being popped from the queue. Recall that Tor utilized a Simple Queue with a single access method, i.e. POP HEAD. We modified the code of the POP HEAD method to perform the reverse operation, that is, by invoking the same hashing algorithm on every cell being popped from the queue. Upon hash calculation, we consult the built hashmap to identify the stream number each respective cell being popped belongs to utilizing its hash as a lookup, and record the stream number in a dedicated output file. By the end of this process, we expect to have an output files listing all the stream numbers in the order the respective cells belonging to those streams were popped.

#### **5.2.2.2 Experiment**

As in the previous experiment, we prepared three files of three sizes, one, five, and ten Megabytes, and used Perl to populate their contents with random characters. Recall that in this experiment we want to confirm and compare the order of cells before and after the circuit queue, hence, we will only upload a single file at a time. Following, we launched Tor with our instrumentational logging, and initiate the upload of each

single file. We repeated the experiment ten time for every file, resulting in a total of 30 runs. However, we ended up with 60 output files, each two for a single run.

In the first type of output files we observed an entry for each cell processed by Tor, where that entry represents the stream numbers those cells belong to, in addition to the calculated hash of each cell. In the second type of output file, we observed an entry for each cell popped from the queue, represented by each cell corresponding stream number.

#### **5.2.2.3 Findings**

We are now ready to establish a comparison between the order of cells coming to Tor, and the order of cells leaving Tor. Similar to our first experiment, we utilized the Stream ID/Domain name mapping and cleansed our data files from irrelevant stream numbers. Needless to say we didn't have to induce any more processing on the data files, or had the need to illustrate them graphically, as both output files, for each respective run, were identical. This is the solid proof we required to conclude that Tor doesn't induce any stream multiplexing from the EWMA multiplexing algorithm and we are now more convinced of our initial intuition.

#### 5.2.2.4 Discussion

So far, we were able to observe that Tor doesn't induce a notable systematic re-ordering of streams by examining incoming traffic from the browser, also we were able to confirm that Tor EWMA multiplexing algorithm doesn't induce any re-ordering on cells entering and exiting the circuit queue. However, we needed a more comprehensive test that can illustrate the full behavior of Tor. To do this, we needed to record the behavior of the browser at first, then compare it to Tor behavior.

Unfortunately, we are faced with the dilemma of unequal comparison. Recall that Tor will either split TCP packets contents into multiple cells, or will merge smaller ones into a single cell. However, given our test case scenarios of file uploads, we are certain that Tor will only perform the former (i.e. splitting) given that TCP packets will be fully populated with the contents of the file to be uploaded, up to the Maximum Transmission Unit (MTU) the environment allows for. Hence, we had to focus our attention on identifying those cells that correspond to a single TCP packet to be able to establish a fair comparison, that will allow us to illustrate TCP streams versus their corresponding Tor streams.



### **5.2.3 The Dictionary Experiment**

To achieve this, we designed an experiment dubbed “The Dictionary Experiment”, where we aim to explore the relationship between a single TCP packet and the different Tor cells that this packet splits up to.

#### **5.2.3.1 Scenario and Implementation**

In our scenario, we wanted to upload a single file to our developed website, capture both the TCP packets and Tor cells, and establish a method to compare each cell to a TCP packet. By the end of this experiment, we should be able to identify which cells correspond to a single TCP packet, and be able to conclude a factor of distribution between a single TCP packet and the different cells it splits up to.

To implement this, we had to expand our testbed setup to be able to capture plain text packets as they are leaving the browser, in addition to capturing the content of cells in Tor. Moreover, we can’t intercept data from the same machine having both the web browser and Tor Onion Proxy, since any traffic sniffing application will only intercept data as they are passing through the link, i.e. after Tor.

To tackle this, we decided to separate the browser from the Onion Proxy, and implement a standalone Tor Proxy setup. By setting up two virtual machines, we installed and configured Tor on one of them, while allowing Tor to execute in

listening mode. That is, Tor will act as a SOCKS proxy, listening on the virtual machine for connections coming to a dedicated port (i.e. 9100). When connections are established to the SOCKS proxy on that port, Tor will continue to operate normally by tunneling those connections through Tor circuits. The client, on the other hand, will be utilizing another virtual machine where the browser of that machine will be configured to use Tor as a SOCKS proxy.

On the client machine, we intercepted the network traffic coming out of the browser using tcpdump [30], a known command line utility for intercepting traffic on the local machine. We configured tcpdump to sniff and record all outgoing traffic to our Tor proxy server, and store the output to a PCAP file. Additionally, we instrumented Tor to print the contents (payload) of each and every cell in a dedicated output file, alongside the stream number this cell associates with. Similarly, on the client machine we developed a Python script utilizing a library named “python-dcap” [31] that will extract the payload of each TCP packet from a given PCAP, in a dedicated output file. The script will traverse each packet in a PCAP file, filter out any control packets, such as TCP SYN/ACK packets, in addition to any non-TCP traffic out of the result, and finally dump the text content of TCP packets to a file. After each run of the experiment, we expect to have:

- 1- An output file on the client machine containing the payload of each TCP packet in a separate line.
- 2- An output file on Tor Proxy server having the contents of every cell in a separate line.

To prepare for the experiment, we had to bring about a method that will make every TCP cell unique to avoid false-positive rates. Our file to upload, had to have non-duplicate entries, which randomness may or may not guarantee. Hence, we decided to use the English dictionary. In this experiment, we build a set of 26 files, where each file contains words from the English dictionary that starts with a different letter from the English alphabet. The dictionary was obtained from a GitHub project that provides the English words in a text file [32]. When uploading a file to our previously designed website, we expect every TCP packet, and the corresponding Tor cells, to contain a unique payload of words that are not repeated in other packets/cells.

### **5.2.3.2 Experiment**

We conducted 26 runs of the same experiment, each utilizing a different alphabetical file. The experiment resulted in 26 PCAP files on the client machine representing the traffic generated by every file upload, and 26 output file on Tor proxy server, representing the payload of every cell Tor created for that respective upload.

Consequently, we executed our Python script on every PCAP file, which resulted in additional 26 files, each having the contents of every TCP packet in a separate line. In principal, the generated output files from the Python script, had the exact content of the originally uploaded files, separated by TCP boundaries represented by new lines. On the other hand, and on Tor Proxy Server, we had a similar output of 26 files,

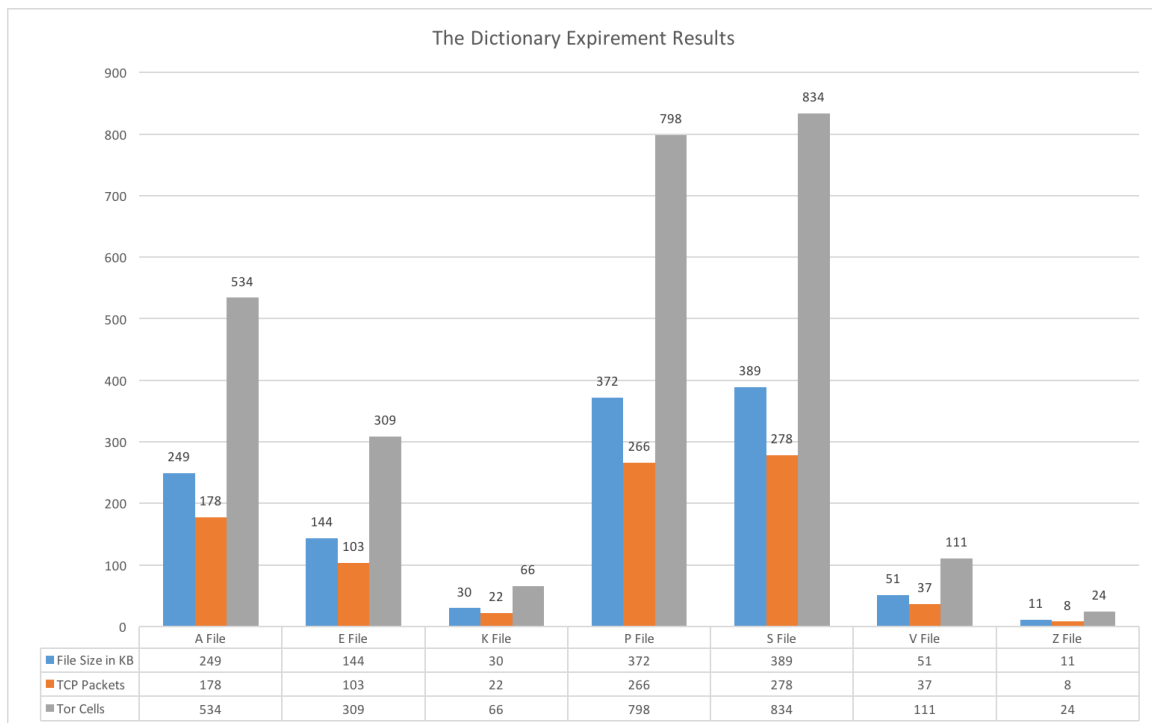
corresponding to each uploaded file, however, since cells accommodate far less data than a TCP packet, we had more lines on the output file representing the boundaries of a Tor typical cell.

Furthermore, we developed an additional python script that takes as an input two different files and initiates a comparison between them. Starting with the first file, the script iterates over every line and stores it in a buffer, and with every iteration, the script starts iterating over every remaining line of the second file, continuing from the previous iteration position. The script attempts to identify the number of lines in the second file that fully appear in the buffer, and exits with the first mess. We, also, maintain a distribution factor average variable, that stores the average number of lines in the second file, that fully appeared in the buffer. Additionally, we maintained two counters from the number of lines in each file.

Finally, we executed the script with every two output files from each run as an input, i.e. the output file representing TCP packets content, and the output files representing Tor cells payloads. By the end of every run, the distribution factor average was always calculated to three, and the number of lines in the TCP output file, was always triple the number of lines in Tor cell files. Additionally, from our observation, we noticed an MTU of 1,500 bytes, where each TCP packet in the experiment is loaded with 1400 bytes of data, and each Tor cell was allowed 500 bytes of data in the payload.

### 5.2.3.3 Findings

This led us to a conclusion, that for every TCP packet that is fully populated under MTU of 1500, Tor generates exactly 3 cells, where the last cell is 80% populated, with 20% of padding, as Tor doesn't include other data from the next TCP packet in the last cell. Figure 16 shows a sample of some files that were uploaded in this experiment. The chart shows three data sets corresponding to the original file size, the number of TCP packets that generated from uploading this file, and finally the number of Tor cells generated from the same file upload operation.



[Figure 16 The correspondence between file size, TCP packets, and Tor cells]

From the graph above, we can witness that a single TCP packets generates three Tor cells, and by increasing the number of TCP packets, Tor doesn't piggyback cells with different TCP packets, given that the packets obey the 1500 bytes MTU, and are fully populated, which is typical in file upload scenarios.

## **5.2.4 The Full Fledge Experiment**

Recalling that our intent is to examine the reordering and multiplexing of streams, and picking up where we left off before the Dictionary experiment, we wanted to observe the full behavior of streams, from the moment they are created in the browser, up until Tor sends them off the wire.

### **5.2.4.1 Scenario and Implementation**

In this experiment, we wanted to upload several files at once, and record the ordering of TCP streams, both at the browser side, and at Tor side. Recall from the EWMA experiment, we already observed that Tor doesn't change the order of packets before and after the circuit queue. Also, recall from the Dictionary Experiment, that Tor will generate 3 cells from every TCP packet. By now, we also know that those cells will align with every corresponding TCP packet, and that additional cell's capacity will be filled with padding.

Our setup in this experiment will be similar to the previous one, i.e. the client and Tor will be hosted on different virtual machines. We will continue to use tcpdump to record the traffic at the client's machine and produce a PCAP file. However, we will be utilizing a mixed setup between the EWMA experiment and the Dictionary experiment at Tor's side. Since we want to compare the intermixing of streams at the

very edges of a file upload (browser and Tor), we will record the order of streams as cells are being moved out from the queue to the output buffer. That is, we will utilize the same hashing algorithm to identify encrypted cells' streams, and record their order in an output file, by emphasizing on the built hashmap from un-encrypted cells entering the queue. Additionally, we will also be printing the contents of every cell in a dedicated file alongside its associated stream.

However, since we already observed the correspondence between TCP packets and cells, we are no longer interested in observing them again in this experiment. Also, recall that we are only interested in the intermixing of streams, i.e. those switching points when the current outgoing packet/cell belongs to a different stream than the previous packet/cell. Hence, and for the sake of this experiment, we will be normalizing the data output, by reducing all packets/cells occurrences between streams' switching to a single TCP packet and the corresponding three Tor cells, i.e. those packets/cells appearing just before the switching. Our comparison will be based on two points:

- 1- Insure that the last TCP packet captured in the client before a stream switch carries the same content as the last three cells before a stream switch in Tor.
- 2- The streams are ordered in the client capture the same way they appear in Tor, by insuring that the TCP packet before switching occurs.

We needed to intercept data coming out from the browser, and label each TCP packet with a unique identifier according to its stream. This is because Tor will also label streams randomly during runtime (recall Stream ID), and we need to establish a mapping between the two labeling. For the identifier, we used TCP source port number, since we know that our application accepts single streaming for each file upload, and every TCP packet will be labeled according to its TCP source port as its stream identifier.

From the client machine, we created a total of 10 files, each sizing to 1 Megabyte, and filled them using a Perl script with random characters. There was no need to use a dictionary approach, since the comparison is to be performed upon every switching and on a single TCP packet, hence, a collision is statistically infeasible. Again, we kept on using our developed web application to upload those files to.

#### **5.2.4.2 Experiment**

We conducted 9 different runs, where in the first run we uploaded two files simultaneously utilizing our previously developed bash script, and we added one more file in each consecutive run. That is, the first run had two files of 1 MB being uploaded at the same time, while in the second run, three files of 1 MB were uploaded at the same time, etc. Each run ensued a PCAP file on the client machine resulting from tcpdump, and a single output file on Tor proxy machine from our instrumentation.



Additionally, we developed a Python script on the client machine utilizing the same Python library (python-dcap) used earlier, to parse the PCAP file. We wanted to traverse each packet in the PCAP file, and output a unique label identifying the stream number each packet associates with, in the same order the browser pushed these packets out, while filtering out irrelevant packets. As mentioned, the label used is the stream source port number. Upon executing, the script will output a single file where each line of the file represents the unique stream identifier the respective packet belongs to, in the order the packet appeared in, alongside the TCP packet content (payload) the respective packet has.

Finally, an additional Python script was developed to conduct the ultimate comparison. In this implementation, we pass the two output files resulting from processing the PCAP, and Tor cells. The script will start traversing both files at the same time but with a different pace, i.e. each packet iteration will also iterate three cells, while noting the unique label identifying each entry, the packet and cell (source port number versus Tor Stream ID). Upon the change of label, i.e. stream switching, the script will buffer the contents of the last TCP packet before the stream switch, and compare it to the previous three cells contents, similar to the Dictionary Experiment. If those match, we know that both streams are aligned at this point, and we output both labels identifying the packet and the cells.

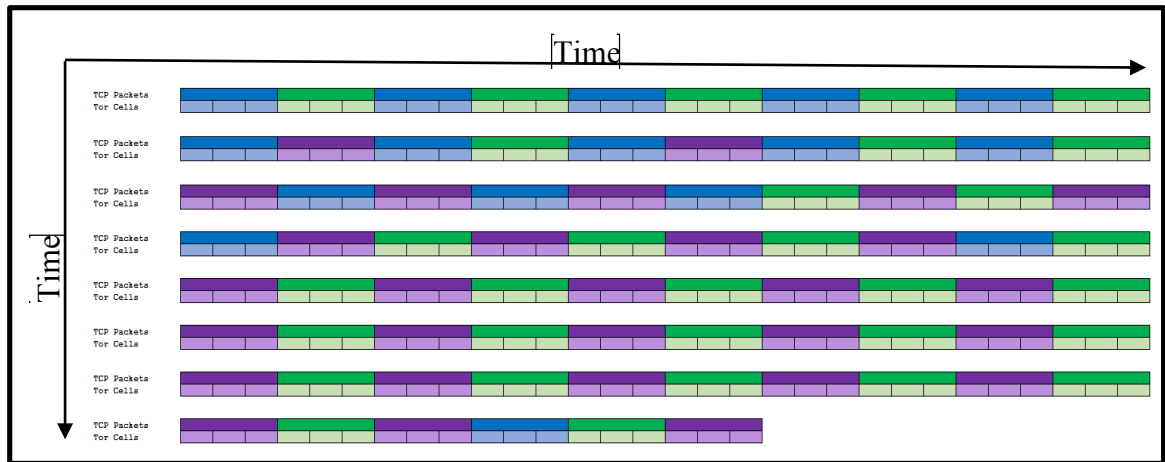
On the other hand, if a single packet didn't match the three cells in content, we immediately break and exit the application marking the two files with a discrepancy, and identifying a possible change of stream ordering. We continue the same comparison through the whole file. At the end of every run, we are presented with a single output that lists the order of the streams in the browser side, next to the corresponding cells stream from Tor's side.

#### **5.2.4.3 Findings**

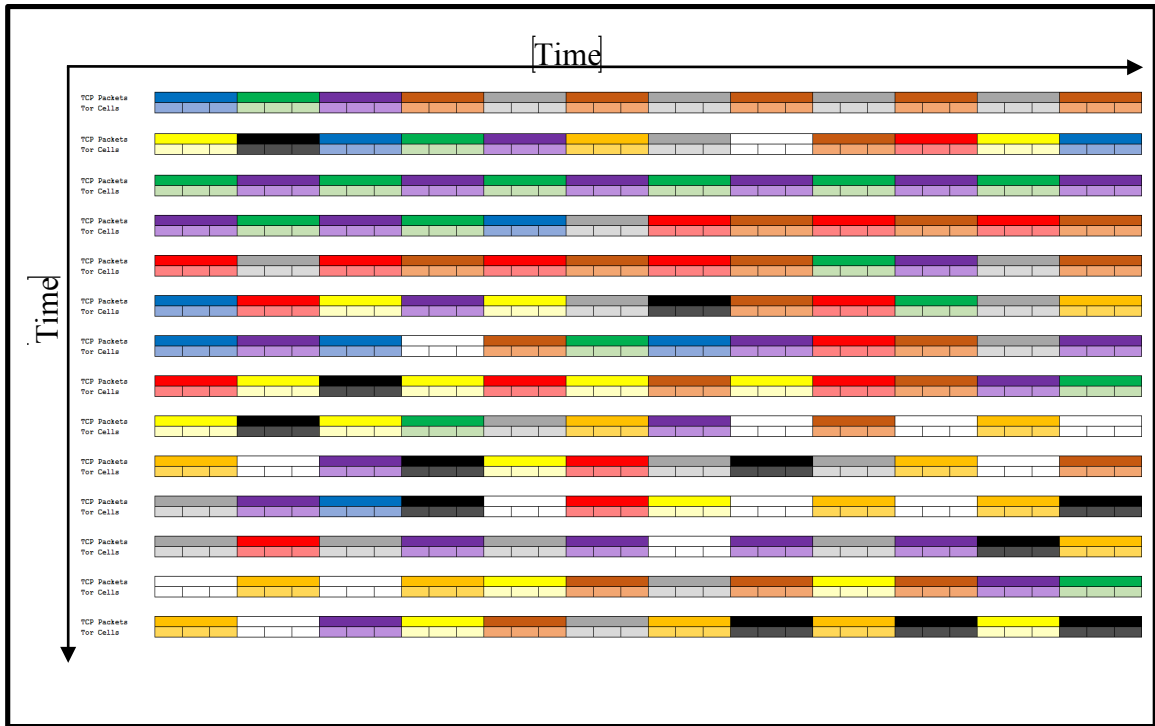
After executing the script on the output results from all nine runs, none of the files caused the script to cease its execution, in an indication that the streams are aligned. The nine runs all resulted in a complete output that can account for all streams' switching, and insure that the switching induce by the browser, is what will continue through the life cycle of a corresponding Tor stream. Also, this gives us the required evidence that it's the browser's behavior (curl in these experiments) that controls the order by which Tor will write cells out the wire, in a First-In-First-Out approach.

The following two figures, Figure 17 and 18, illustrates the behavior observed in this experiment. In Figure 17 we show the output produced by uploading three files simultaneously, and the correspondence between each TCP stream and Tor cells, in addition to illustrating the order those streams appeared in. Note that the chart starts at the top left corner and moves to the right as time progresses, while wrapping around to the next line, when edge of the chart is reached. Each entry labeled "TCP Packet" is a

summarization of all the TCP packets that continuously transferred before a stream switch occurs, whereas every entry labeled “Tor Cell” represents the same with respect to Tor. Figure 18, illustrate the same, but with the last run of 10 simultaneous file uploads. The figure illustrates only a sample of the output due to space constraints.



[Figure 17 The intermixing of 3 TCP streams resulting from 3 file uploads where each stream is represented by a different color]



**[Figure 18 The intermixing of 10 TCP streams resulting from 10 file uploads where each stream is represented by a different color]**

## 5.3 Experiments on Proposed Algorithms

In the previous section, and as the different spectrum of experiments have shown, we now have concrete evidence that Tor doesn't multiplex cells from the perspective of an Onion Proxy (OP), but instead, Tor follows whatever order of streams the browser throws at it. Tor will follow a simple FIFO approach, at the client side to handle the order streams arrive in. Hence, we decided to further invest on the analysis of Tor multiplexing by attempting to introduce streams randomization on the same circuit.

In section 4.2, we displayed three proposed approaches to achieve that, while emphasizing on the benefits of each. Recalling that the first approach was the most expensive in cost (time-wise), yet the simplest to implement, we decided to implement it first. In the following experiments, and as opposed to the previous experiments, we want to test the behavior of Tor multiplexing on a smaller scale, and gradually increase our complexity as we progress with results. Hence, we will be testing the multiplexing algorithms on web browsing (GET requests), instead of file uploading.

### 5.3.1 Ground Truth

To establish our ground truth, we wanted to start off with a controlled environment instead of testing our algorithms in the wild. Hence, we developed a simple website that will allow us to observe the behavior of cells, and record the order of streams before applying our algorithm. The website was developed using "NodeJS", and utilizing on the aforementioned "http" module. The website only serves a simple HTML page, were

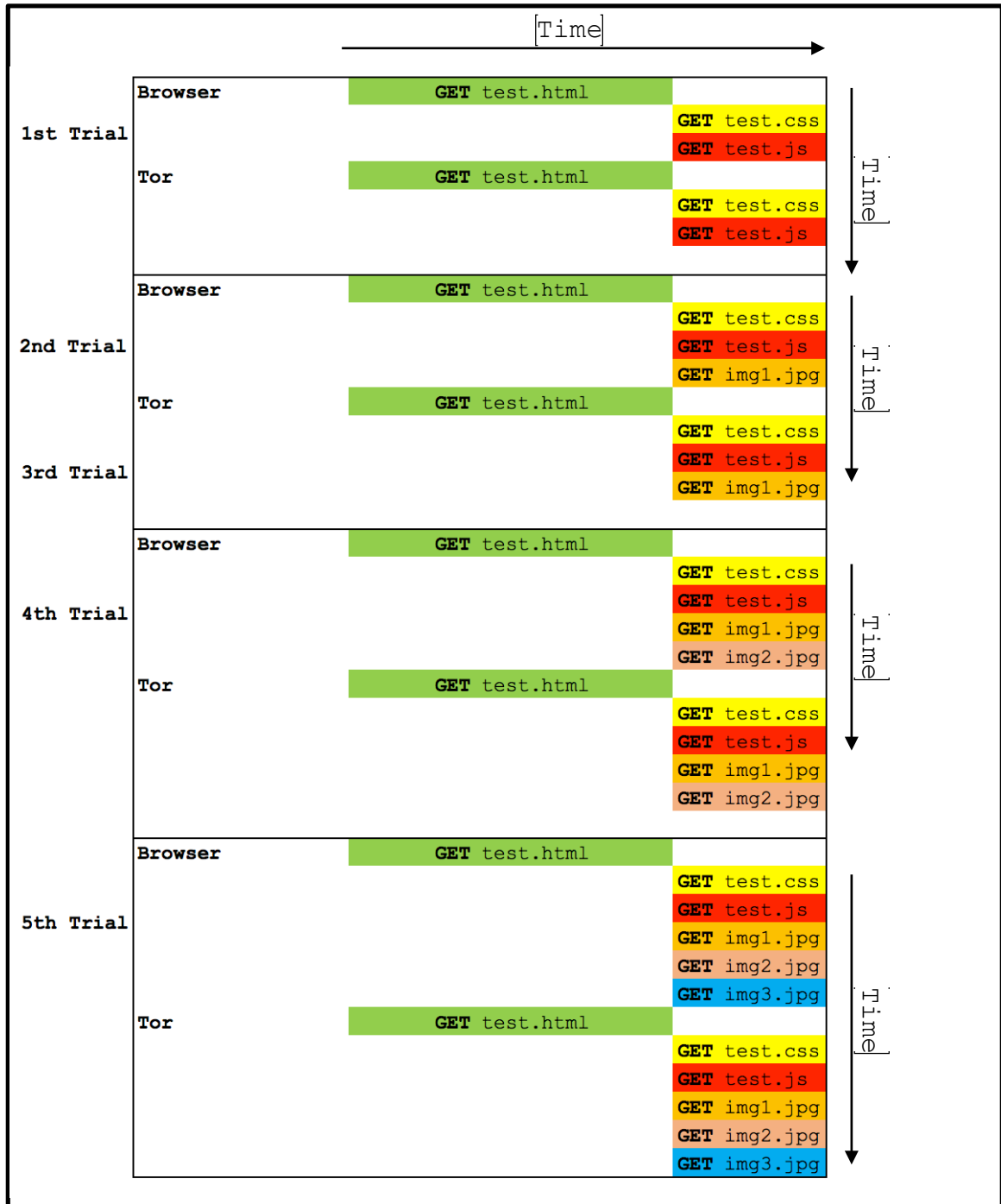
we initially include two external resources, a CSS and a JavaScript files. Recalling the default behavior of HTTP, where the browser will fetch the HTML page, parses it for external resource, and then initiates as many required TCP streams to fetch those resources.

Following, we instrumented Tor to print the URL of the requested resources, in the order Tor received them from the browser. Additionally, and similar to our multiplexing experiments, we hashed the cells that corresponds to those requests and built a hashmap of every cell, and the URL requested by each respective cell. Finally, we modified the `CELL_QUEUE_POP` function to lookup the URL requested by every encrypted cell as they are leaving the queue, and instrumented Tor to print the URL as well.

We started Tor and used Firefox to connect to our developed website multiple times, and observed the order by which resources are requested. After ten runs, we confirmed that the order induced is persistent across all runs, and that the resources are being fetched with the same order in every run. Moreover, we started increasing the complexity of our scenario by including more external resources in the HTML page gradually, and again conducting 10 visits to the website with each resource addition by adding an additional picture (resource) to the HTML page, and observing the order. We kept on repeating the experiment until we had an HTML page with five external resources.

After each run, we observed that Tor is sending out cells, in the same order as URLs are being requested by the browser, which is in direct alignment with our discovery in the

multiplexing experiments. Figure 19 shows the observed order of the streams, and illustrates the order requested by the browser, and the order observed from cells leaving Tor.



[Figure 19 Establishing ground truth by comparing the order URL requests to the order of cells leaving Tor with those requests]

### 5.3.2 Implementation

To examine and experiment with the first approach, we implemented a C function, namely `CELL_QUEUE_POP_RANDOM`, according to the algorithm listed in the first approach in section 4.2. The function was placed in Tor `src/or/Relay.c` class, which implements most of the queue functionality. Additionally, to make effective use of our function, we modified Tor function “`channel_flush_from_first_active_circuit`”, and replaced the call to `CELL_QUEUE_POP` function, with a call to our own implemented function.

The function “`channel_flush_from_first_active_circuit`” is the only function in Tor that calls `CELL_QUEUE_POP`, hence we confirmed that every subsequent need of Tor to move cells from the circuit queue to the output buffer will be utilized using our new method.

### 5.3.3 Experiment

In this experiment, we repeated the same actions taken in the Ground Truth, by utilizing our developed website. Again, in each subsequent trail with a certain number of external resources, we repeated the experiment for ten runs to insure consistency of the results. Figure 20 illustrates the results induced of applying our algorithm, while only illustrating the result of 4 runs due to space constraints.



		1st Run	2nd Run	3rd Run	4th Run
1st Trial	Browser	GET test.html	GET test.html	GET test.html	GET test.html
		GET test.css	GET test.css	GET test.css	GET test.css
		GET test.js	GET test.js	GET test.js	GET test.js
	Tor	GET test.html	GET test.html	GET test.html	GET test.html
2nd Trial	Browser	GET test.html	GET test.html	GET test.html	GET test.html
		GET test.css	GET test.css	GET test.css	GET test.css
		GET test.js	GET test.js	GET test.js	GET test.js
	Tor	GET test.html	GET test.html	GET test.html	GET test.html
3rd Trial	Browser	GET test.html	GET test.html	GET test.html	GET test.html
		GET test.css	GET test.css	GET test.css	GET test.css
		GET test.js	GET test.js	GET test.js	GET test.js
	Tor	GET test.html	GET test.html	GET test.html	GET test.html
4th Trial	Browser	GET test.html	GET test.html	GET test.html	GET test.html
		GET test.css	GET test.css	GET test.css	GET test.css
		GET test.js	GET test.js	GET test.js	GET test.js
	Tor	GET test.html	GET test.html	GET test.html	GET test.html
		Timeout	Timeout	Timeout	Timeout
		Timeout	Timeout	Timeout	Timeout
		Timeout	Timeout	Timeout	Timeout
		Timeout	Timeout	Timeout	Timeout
		Timeout	Timeout	Timeout	Timeout

[Figure 20 Illustration of applying the first approach algorithm to Tor]

### 5.3.4 Findings

As figure 19 illustrates, our algorithm succeeded in inducing a degree of randomization on Tor. By implementing the first approach, we managed to change the behavior of Tor to add an additional level of randomness to streams on the same circuit. For example, in the first trial, we conducted 10 runs (4 are shown), and in every run, we witness an alternating order of streams requesting the resources “test.css” and “test.js”. In the first run, we see that modified Tor sends the cell requesting the “css” file first, where in the

second run, it's requesting the "js" file first, similar to the 3<sup>rd</sup> run, and contrasting with the fourth.

Also, in the third trial, where we increased the level of complexity to host five external resources on the HTML page, we observe that modified Tor induces a severe level of randomization on the cells heading out with URL requests. Modified Tor creates an unpredictable behavior of multiplexing, which increases with the number of streams composing a circuit. Hence, we can observe that the added randomization managed to create a non-systematic, and unpredictable behavior of fetching resources, by alternating the order of which streams are arranged and multiplexed inside a circuit. By doing so, we insured that a classifier will not observe a consistent behavior of traffic patterns to build a fingerprint of a website, specially when the number of resources increases to 10 or more, which produces a factor of magnitude of websites fingerprints.

In our simple website implementation, with 4 resources (3<sup>rd</sup> trial), a classifier would previously build a signature for this website based on the number of cells traversed in each direction, and the boundaries of a cell. However, by introducing a stream-level randomization, we are now forcing the classifier to conduct tens of runs to obtain a total of 24 signatures, each representing a different order of fetching those resources. Furthermore, consider the home page of amazon.com. On average, loading the page for the first time, results in over 250 resources being loaded to your browser. A classifier that previously had to collect a single signature for that website, is forced to create  $3.2 \times 10^{492}$

signatures, for it to consider all possible alternation of streams, a resource consuming process.

## CHAPTER 6

### CONCLUSION AND FUTURE WORK

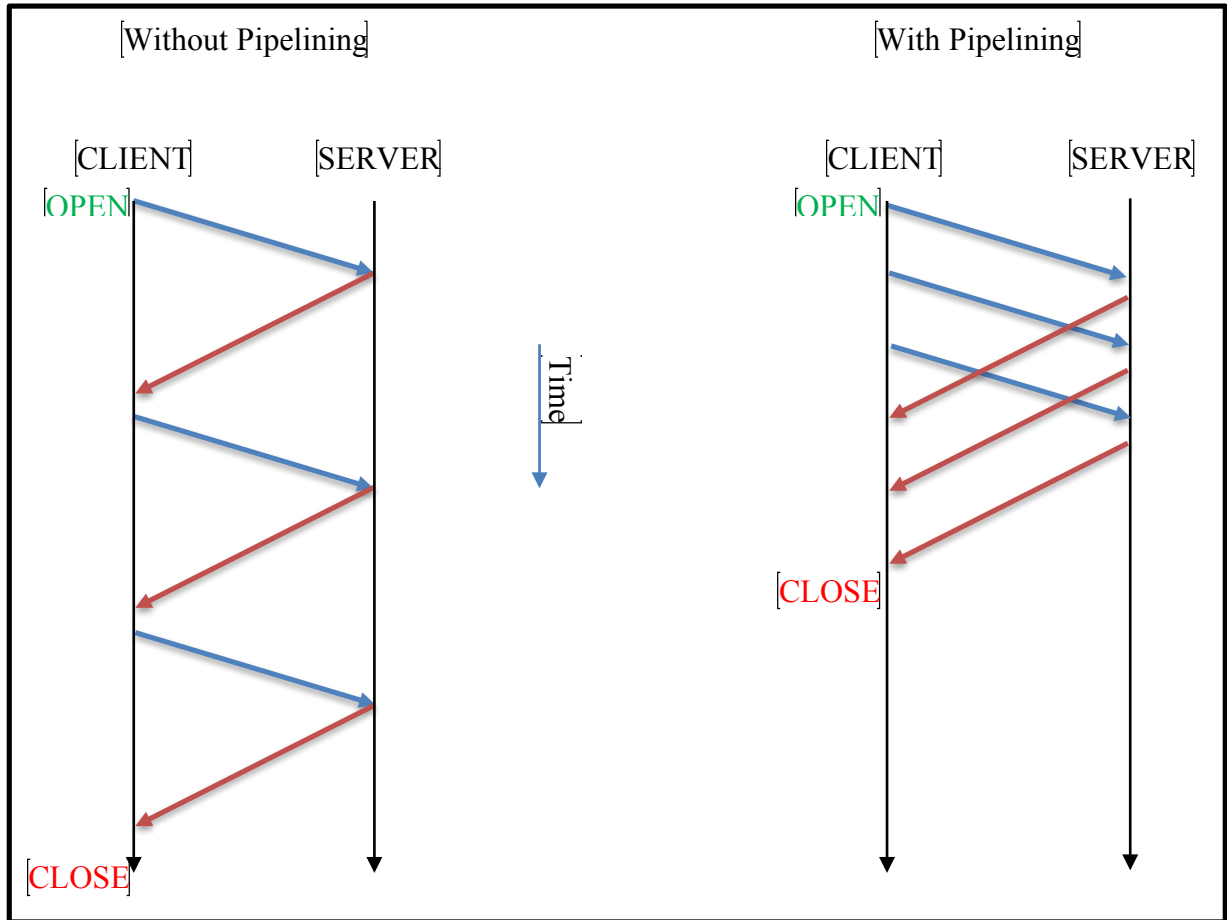
As we witnessed from the results of our experiment, we now confirm that Tor doesn't induce any multiplexing on cells from the Onion Proxy (Client), due to the single establishment of a circuit. Tor will only invoke its randomization (i.e. multiplexing) on streams in cases where a Tor node has more than a circuit. We have shown, by experiments, that Tor's cells will follow the same order as the browser initiating TCP streams and in accordance to the TCP packets forming those streams, and that cells will merely be smaller data containers.

Additionally, we discussed how streams multiplexing on the same circuit induces a high level of unpredictability, a consequence that will persuade much longer observation time from an attacker or a classifier to confirm a website visit. By proposing three different approaches of stream randomization in Tor, we implemented one algorithm and performed various test scenarios, that indicate Tor's ability to multiplex streams to a certain extent. However, stream randomization by manipulating cells in their queue, has shown a limited success to four streams only, and may require more research resources to produce realistic results. We discuss this matter in section 6.1.

## 6.1 Threats to Validity

Unfortunately, our implemented approach ceases to succeed beyond four streams. Once we increase the number of external resources in the HTML page beyond 4 resources, we start to observe multiple timeouts in Tor. After deeply investigating the reasons behind those timeouts, we noticed that the Onion Proxy (OP) starts receiving timeout requests from the exit node. Immediately then, the OP retransmits the requests multiple times, until the exit node refuses to accept further connection, as an internal defense against brute force attacks.

The OP, then, tears down the circuit, and starts building a new one, utilizing a different exit node. Unfortunately, the behavior is repeated with the new exit node, and Tor fails to send the requests. Since we don't have access to the exit node, we tried to investigate the matter from the web server perspective, utilizing some heavy, low-level, debugging messages. We noticed that beyond four streams, the web server enables HTTP pipelining, a technique recently becoming exponentially popular. In HTTP pipelining, the client (browser), doesn't send consecutive HTTP requests, and wait for each response prior sending the next one, instead, the browser sends all requests at one, and waits for the responses to arrive as soon as they are ready. Figure 21 illustrates HTTP Pipelining.



[Figure 21 HTTP Pipelining Illustration]

Since the matter is closely related to HTTP pipelining, we think that further research will sail us away from our original objective, of implementing a stream randomization at the client side. We fear that investing more resources on this research, would induce changes that are not backward compatible, and would require a sudden upgrade of millions of Tor users and relays around the globe. A consequence that doesn't outweigh the benefits sought by our research.

Additionally, since the second and third approach of our proposed algorithms, are merely an enhancement to the performance of the first approach, and doesn't introduce a major change in handling the streams, we decided to not implement them any further.

## 6.2 Future Work

In future work, we recommend seeking other approaches to randomizing streams. One could think of instrumenting Tor to create more circuits at the OP, and forcing different streams of the same session to follow different circuits. Hence, Tor, by default, will multiplex the streams into the single connection linking it with the Guard node. However, such approach requires deep investigation of the performance overhead that might be induced of streams following different routes, and take into consideration the utilization level of intermediate nodes. Another possible approach to induce randomization, is from the browser side. One could think of developing a browser extension that:

- 1- Disables HTTP pipelining.
- 2- Buffers the outgoing requests and randomizing their forward order.

This will cause no further modification to Tor, where the randomization algorithm will be moved to the browser side, and Tor will naturally process streams in the order they are received from the extension.



## References

- [1] D. McCoy, K. Bauer, D. Grunwald, T. Kohno, and D. Sicker. Shining Light in Dark Places: Understanding the Tor Network. In N. Borisov and I. Goldberg, editors, Proceedings of the Eighth International Symposium on Privacy Enhancing Technologies (PETS 2008), pages 63-76, Leuven, Belgium, July 2008. Springer-Verlag.
- [2] Fisher, Phineas. "Hack Back: A DIY Guide." Pastebin. Pastebin, n.d. Web. 27 Sept. 2016.
- [3] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The Second-Generation Onion Router. In Proceedings of the 13th USENIX Security Symposium, pages 303-320. USENIX Association, Aug. 2004.
- [4] G. Danezis. The traffic analysis of continuous-time mixes. In Proceedings of Privacy Enhancing Technologies workshop (PET 2004), volume 3424 of LNCS, pages 35-50, May 2004.
- [5] S. J. Murdoch and G. Danezis. Low-Cost Traffic Analysis of Tor. In Proceedings of the 2005 IEEE Symposium on Security and Privacy, pages 183-195. IEEE Computer Society Press, May 2005.
- [6] S. J. Murdoch and P. Zielinski. Sampled Traffic Analysis by Internet-Exchange-Level Adversaries. In N. Borisov and P. Golle, editors, Proceedings of the Seventh Workshop on Privacy Enhancing Technologies (PET 2007), volume 4776 of Lecture Notes in Computer Science, pages 167-183. Springer-Verlag, June 2007.
- [7] L. Tsvetkov and P. Syverson. Locating hidden servers. In Proceedings of the 2006 IEEE Symposium on Security and Privacy. IEEE CS, May 2006.
- [8] Shi, Yi, and Kanta Matsuura. "Fingerprinting attack on the tor anonymity system." International Conference on Information and Communications Security. Springer Berlin Heidelberg, 2009.
- [9] Panchenko, Andriy, et al. "Website fingerprinting in onion routing based anonymization networks." Proceedings of the 10th annual ACM workshop on Privacy in the electronic society. ACM, 2011.
- [10] A. Hintz. Fingerprinting websites using traffic analysis. In R. Dingledine and P. Syverson, editors, Proceedings of Privacy Enhancing Technologies Workshop (PET 2002), volume 2482 of Lecture Notes in Computer Science, pages 171-178. Springer-Verlag, Apr. 2002.

- [11] D. Herrmann, R. Wendolsky, and H. Federrath. Website Fingerprinting: Attacking Popular Privacy Enhancing Technologies with the Multinomial Naïve-Bayes Classifier. In CCSW '09: Proceedings of the 2009 ACM CCS Workshop on Cloud Computing Security, pages 31–42, Chicago, Illinois, USA, Nov. 2009. ACM Press.
- [12] Wang, Tao, and Ian Goldberg. "Improved website fingerprinting on tor." Proceedings of the 12th ACM workshop on Workshop on privacy in the electronic society. ACM, 2013.
- [13] Carballude González, Pablo. "Fingerprinting Tor." Information Management & Computer Security 21.2 (2013): 73-90.
- [14] Sun, Qixiang, et al. "Statistical identification of encrypted web browsing traffic." Security and Privacy, 2002. Proceedings. 2002 IEEE Symposium on. IEEE, 2002.
- [15] Liberatore, Marc, and Brian Neil Levine. "Inferring the source of encrypted HTTP connections." Proceedings of the 13th ACM conference on Computer and communications security. ACM, 2006.
- [16] Shi, Yi, and Kanta Matsuura. "Fingerprinting attack on the tor anonymity system." International Conference on Information and Communications Security. Springer Berlin Heidelberg, 2009.
- [17] Jap anonymity and privacy. <http://anon.inf.tu-dresden.de>.
- [18] Alexa - Actionable Analytics for the Web. <http://www.alexa.com/>
- [19] Juarez, Marc, et al. "A critical evaluation of website fingerprinting attacks." Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. ACM, 2014.
- [20] Wang, Tao, et al. "Effective attacks and provable defenses for website fingerprinting." 23rd USENIX Security Symposium (USENIX Security 14). 2014.
- [21] Wang, Tao, and Ian Goldberg. "Improved website fingerprinting on Tor." Proceedings of the 12th ACM workshop on Workshop on privacy in the electronic society. ACM, 2013.
- [22] Wang, Tao, and Ian Goldberg. On realistically attacking Tor with website fingerprinting. Technical Report 2015-08, CACR, 2015. <http://cacr.uwaterloo.ca/techreports/2015/cacr2015-08.pdf>, 2015.
- [23] avarro G. A guided tour to approximate string matching. ACM Computing Surveys 1999; 33:2001.

- [24] Joel Reardon. Improving Tor using a TCP-over-DTLS Tunnel. Master's thesis, University of Waterloo, 2008.
- [25] Tang, Can, and Ian Goldberg. "An improved algorithm for Tor circuit scheduling." Proceedings of the 17th ACM conference on Computer and communications security. ACM, 2010.
- [26] "Tor overview" <https://www.torproject.org/about/overview.html.en> Web. 01 May 2017.
- [27] "HTTP Pipelining" [https://en.wikipedia.org/wiki/HTTP\\_pipelining](https://en.wikipedia.org/wiki/HTTP_pipelining) Web. 12 May 2017.
- [28] "Gnuplot" <http://www.gnuplot.info/>
- [29] "Tor Metrics" <https://metrics.torproject.org/>
- [30] "TCPDump" <http://www.tcpdump.org/>
- [31] "Python dpkt" <https://pypi.python.org/pypi/dpkt>
- [32] "English Words" <https://github.com/dwyl/english-words>
- [33] "Oracle Berkeley DB" <https://oss.oracle.com/berkeley-db.html>
- [34] A. Panchenko, L. Niessen, A. Zinnen, and T. Engel, "Website fingerprinting in onion routing based anonymization networks," in Proceedings of the 10th annual ACM workshop on Privacy in the electronic society. ACM, 2011, pp. 103–114.
- [35] A. Panchenko, F. Lanze, A. Zinnen, M. Henze, J. Pennekamp, K. Wehrle, and T. Engel, "Website fingerprinting at internet scale," in Proceedings of the 23rd Internet Society (ISOC) Network and Distributed System Security Symposium (NDSS 2016), 2016.
- [36] T. Wang, X. Cai, R. Nithyanand, R. Johnson, and I. Goldberg, "Effective attacks and provable defenses for website fingerprinting," in 23rd USENIX Security Symposium (USENIX Security 14), 2014, pp. 143–157.
- [37] D. Herrmann, R. Wendolsky, and H. Federrath, "Website fingerprinting: attacking popular privacy enhancing technologies with the multinomial naïve-bayes classifier," in Proceedings of the 2009 ACM workshop on Cloud computing security. ACM, 2009, pp. 31–42.
- [38] H. Jahani and S. Jalili, "A novel passive website fingerprinting attack on tor using fast fourier transform," Computer Communications, 2016.

- [39] Cherubin, Giovanni, Jamie Hayes, and Marc Juarez. "Website fingerprinting defenses at the application layer." *Proceedings on Privacy Enhancing Technologies* 2017.2 (2017): 186-203.

]

## Vitae

Name	: [Yasir Suliman Alagl ]
Nationality	: [Saudi Arabian ]
Date of Birth	: [11/17/1989]
Email	: [alagly@gmail.com]
Address	: [Dhahran]
Academic Background	: [Bachelor of Science in Software Engineering]