

Thwarting Traffic Confirmation Attacks in Tor with Traffic Modulation and Cover Traffic

Afonso Pedro Antunes da Mota Gomes

Thesis to obtain the Master of Science Degree in
Computer Science and Engineering

Supervisor(s): Prof. Nuno Miguel Carvalho dos Santos
Prof. Kevin Christopher Gallagher

Examination Committee

Chairperson: Prof. João António Madeiras Pereira
Supervisor: Prof. Nuno Miguel Carvalho dos Santos
Member of the Committee: Prof. Luís David Figueiredo Mascarenhas Moreira Pedrosa

November 2023

Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

First, I would like to thank my supervisors, Prof. Nuno Santos and Prof. Kevin Gallagher, for taking me on as their student and for all the assistance and guidance they have provided me with throughout this challenging year.

I am also immensely grateful to Afonso Carvalho and Prof. Diogo Barradas for all the insightful discussions, feedback, and crucial support. In addition, I wish to express my gratitude to Francisco Silva for the technical support provided, without which the evaluation of our system would not have been possible.

Finally, I am extremely grateful for the company and support provided by my friends and family not only throughout the duration of this work but also throughout all my university years.

This work was supported by national funds through IAPMEI via the SmartRetail project (ref. C66322 06063-00466847).

Resumo

O Tor é uma rede de anonimidade de baixa latência muito utilizada que permite aos utilizadores navegar na Internet de forma segura. Infelizmente, é conhecido que a rede Tor é vulnerável a ataques de confirmação de tráfego, isto é, ataques em que um adversário pode observar o fluxo de tráfego tanto para dentro quanto para fora da rede Tor. Estes ataques funcionam ao contar o tempo entre pacotes e ao analisar os seus tamanhos, para conseguir correlacionar um fluxo que entra na rede Tor com um fluxo que sai da mesma. Assim, o adversário pode comprometer a anonimidade do tráfego entre um cliente e um servidor. Abordagens recentes para solucionar este problema sugerem técnicas como mistura de tráfego real com tráfego falso, e atraso de pacotes por períodos de tempo escolhidos a partir de uma distribuição estatística específica, a fim de combater os ataques de confirmação de tráfego. No entanto, a maioria das soluções propostas requerem a implementação de novos sistemas de anonimidade, o que pode ser desafiante, tendo em conta o tamanho e adoção da rede Tor. Neste projeto, estudamos a viabilidade de integrar a modulação de tráfego e a mistura de tráfego de cobertura em loop no Tor. Desenvolvemos também um sistema, chamado Shaffler, e realizamos uma análise extensiva para determinar se essas técnicas ajudam a defender contra ataques de confirmação de tráfego conhecidos e, se sim, quais as penalizações de desempenho envolvidas.

Palavras-chave: Tor, Anonimidade, Ataques de correlação de tráfego, Mistura de tráfego, Modulação de tráfego

Abstract

Tor is a popular low-latency anonymity network that allows users to surf the Internet privately. Unfortunately, the Tor network is known to be vulnerable to traffic confirmation attacks, i.e. attacks where an adversary can observe a traffic flow both into and out of the Tor network. These attacks work by counting the times between packets and looking at their sizes to correlate one flow entering the Tor network with a flow exiting the Tor network. Thus, the adversary can compromise the anonymity of traffic between a client and a server. Recent approaches to anonymity have suggested techniques such as mixing real traffic with covert traffic and delaying packets for an amount of time chosen from a specific statistical distribution, in order to frustrate traffic confirmation attacks. However, most of the proposed solutions to this problem require the deployment of new anonymity systems, which may be a challenge, considering the current size and adoption of the Tor network. In this project, we study the feasibility of integrating traffic modulation and loop cover traffic mixing into Tor. We develop a system, called Shaffler, and perform an extensive analysis of it to determine whether these techniques help defend against known traffic confirmation attacks, and if so, what performance penalties are incurred.

Keywords: Tor, Anonymity, Traffic correlation attacks, Traffic mixing, Traffic modulation

Contents

Acknowledgments	v
Resumo	vii
Abstract	ix
List of Tables	xiii
List of Figures	xv
Glossary	xvii
1 Introduction	1
1.1 Motivation	1
1.2 Objectives	2
1.3 Contributions	4
1.4 Thesis Outline	4
2 Background and Related Work	5
2.1 Background on the Tor Anonymity Network	5
2.2 Attacks against the Tor Network	6
2.3 Defenses in the Tor Network	9
2.4 Defenses Used in Other Anonymity Systems	14
3 Design	21
3.1 Threat Model	21
3.2 Architecture	22
3.3 Traffic Modulation	23
3.4 Generation of Cover Traffic	26
4 Implementation	29
4.1 Exploratory Prototype	29
4.2 Final Prototype	32
5 Evaluation	43
5.1 Methodology	43
5.2 Traffic Modulation Results	47
5.3 Cover Traffic Results	55

5.4 Full System Results	59
6 Conclusions and Future Work	65
6.1 Conclusions	65
6.2 Achievements	66
6.3 Future Work	66
Bibliography	69

List of Tables

2.1	Comparison of various anonymity systems.	18
3.1	Table showing the modulation functions that are provided by the Shaffler system.	24
5.1	Maximum F1-score and P4 values obtained for each of the four tested configurations of the Normal distribution modulation.	50
5.2	Maximum F1-score and P4 values obtained for increasing window sizes and overlaps of the DeepCoFFEA attack, when using the Normal distribution modulation.	53
5.3	Maximum F1-score and P4 values obtained for modulation functions configured to result in a mean of delays equal to 20 milliseconds.	55
5.4	Maximum F1-score and P4 values obtained for three different configurations of cover traffic with different amounts of generated data per request.	57
5.5	Maximum F1-score and P4 values obtained for three different configurations of cover traffic with different periods of requests.	58
5.6	Maximum F1-score and P4 values obtained for each technique independently and when using both techniques simultaneously.	60
5.7	Maximum F1-score and P4 values obtained for decreasing levels of protection provided by each technique.	62

List of Figures

1.1	High-level overview of the Shaffler system showcasing its central techniques: (i) traffic modulation is implemented by the middle node, and (ii) cover traffic generation by the Tor client via a traffic loop between the client and a co-located onion service.	3
2.1	Example of a Tor circuit used by a client to access a Web server anonymously.	6
2.2	Overview of Vuvuzela's [34] conversation protocol.	15
2.3	Overview of Loopix's [36] architecture.	16
3.1	Shaffler system architecture.	23
3.2	Diagram showcasing the potential issues with relying on the entry node or the exit node to modulate traffic. Our threat model considers that an adversary may be able to monitor traffic at both the entry and the exit nodes, but not at the middle node.	25
3.3	Shaffler's cover traffic generation design.	27
4.1	Diagram showing the additions made to the circuit creation protocol to allow the communication of delay policies.	32
4.2	Diagram showing the process for delaying Tor cells. Includes the usage of a delay queue that stores cells ordered by ready time ($rt_n \leq rt_{n+1}$) and a single timer that waits for the first cell's ready time.	36
5.1	Simulation configuration used for the dataset collection.	47
5.2	Graph showing the results of the DeepCoFFEA attack for varying similarity thresholds, when using the "Vanilla" dataset.	48
5.3	Graphs showing the results of the DeepCoFFEA attack for varying similarity thresholds, when using the normal distribution with different parameters.	49
5.4	Performance results obtained by <code>tornettools</code> for all four tested configurations of the Normal distribution.	51
5.5	Graphs showing the effects on accuracy of increasing the window size of the DeepCoFFEA attack to 7 seconds, for the "Vanilla" dataset and the $Normal(\mu = 30, \sigma = 10)$ dataset.	52
5.6	Graph showing the effects on accuracy of increasing the window size of the DeepCoFFEA attack even further to 9 seconds, for the $Normal(\mu = 30, \sigma = 10)$ dataset.	53

5.7	Graphs showing the results of the DeepCoFFEA attack for varying similarity thresholds, when using each modulation function with parameters configured to result in a mean of delays equal to 20 milliseconds.	54
5.8	Performance results obtained by <code>tornettools</code> for each modulation function configured to have a mean of delays equal to 20 milliseconds.	55
5.9	Graph showing the results of the DeepCoFFEA attack for varying similarity thresholds, when using a “Vanilla” dataset generated by simulating 200 clients.	56
5.10	Graphs showing the results of the DeepCoFFEA attack for varying similarity thresholds, when using three different amounts of cover traffic.	57
5.11	Performance results obtained by <code>tornettools</code> for three different configurations of cover traffic with different amounts of generated data per request.	58
5.12	Graphs showing the results of the DeepCoFFEA attack for varying similarity thresholds, when using three different periods of requests.	59
5.13	Graph showing the results of the DeepCoFFEA attack for varying similarity thresholds, when using 2 threads to generate cover traffic at a rate of 500KB every 15 seconds each.	60
5.14	Graphs showing the results of the DeepCoFFEA attack for varying similarity thresholds, when using each technique independently and when using both techniques simultaneously.	61
5.15	Graphs showing the results of the DeepCoFFEA attack for varying similarity thresholds, when using both techniques simultaneously, with different parameters.	61
5.16	Performance results obtained by <code>tornettools</code> for three different datasets obtained by using both techniques simultaneously, with different parameters.	62

Chapter 1

Introduction

1.1 Motivation

Although the Internet has become an integral aspect of people's lives, it has also posed a significant challenge for Internet users concerning their privacy and anonymity. This state of affairs is due to the traditional network protocols for Internet communications that rely on strong traffic-identifying metadata, such as source and destination IP addresses. IP addresses are essential to enable communication parties to establish TCP/IP connections and respond to each other's messages. As a result, they constitute strong identifiers that enable network observers, such as Internet Service Providers (ISPs), to trace communications to a sender and a receiver, making it difficult to remain anonymous on the Internet.

Thankfully, there are systems capable of providing users with better anonymity. One of those systems, and the most prominent one, is the Tor network [1]. When using Tor, a sender's traffic passes through at least three "onion routers" before reaching the recipient. Traffic created by the sender is encrypted sequentially, with all secret keys negotiated between the sender and each node of the circuit. As this traffic passes through the circuit, each node decrypts it with its key, revealing the address of the next node. With this mechanism, only the sender is aware of both its own and the recipient's IP addresses.

However, while the Tor onion routing protocols provide strong anonymity properties, there remain some more complex methods to identify the parties involved in a communication. Traffic correlation attacks [1] are one of these methods, which consists of attempting to correlate patterns observed in traffic that is sent by a user to the Tor network with patterns of traffic observed leaving the Tor network and arriving at a remote host. Dingledine et al. [1] describe these attacks as *end-to-end timing correlation*, when the attack is performed through observation of timing patterns, and as *end-to-end volume correlation* when an attacker attempts to correlate flows through the observed volume of packets.

For a long time, launching these attacks in practice has been considered to be difficult given the geographical dispersion of Tor circuits' routes and the subsequent difficulty for a single ISP to intercept the traffic at both communication endpoints of a Tor circuit. Today, however, traffic correlation attacks are becoming increasingly realistic threats to the anonymity of Tor users, as they can be launched at a large scale by a coalition of ISPs approximating state-level adversaries. Nithyanand et al. [2] presented

an empirical study where they discovered that up to 40% of all Tor traffic is vulnerable to attacks by traffic correlation from network-level attackers; 42% from Autonomous Systems (ASes) that may collude with each other and 85% from state-level adversaries. Users in some countries (China and Iran) are particularly affected, since 95% of all possible circuits are vulnerable to traffic correlation. A second study by Johnson et al. [3], shows that 80% of all users can be deanonymized when faced with a relay-level attacker. Surprisingly, most users can be successfully deanonymized within three months by an AS-level attacker. Colluding ASes can correctly correlate users in $90\times$ less time. When targeting a specific web user, colluding ASes can effectively deanonymize them in a single day.

Understanding how these risks are becoming increasingly realistic, the research community has been working on new defenses to protect Tor users against such attacks [2, 4–8]. However, strong defenses for low-latency anonymity networks cannot be achieved without a cost. In a highly cited paper, Das et al. [9] describe this problem as the *anonymity trilemma*, stating that it is not possible to simultaneously achieve strong anonymity, low latency, and low resource usage. As such, to provide strong protection against traffic correlation attacks, one must sacrifice either latency or resource usage. To this end, constrained by the fundamental limitations formalized by Das et al. [9], this work aims to explore the effectiveness of two defensive techniques in enhancing Tor anonymity properties: (i) *traffic modulation*, which mainly sacrifices latency, and (ii) *cover traffic generation*, which mainly sacrifices resource usage.

1.2 Objectives

Our goal is to develop a system, that can significantly decrease the effectiveness of traffic correlation attacks performed to the Tor anonymity network. It is our objective to do this in a way that preserves compatibility with vanilla Tor, by ensuring standard Tor functionality between entities running our version of Tor and entities running vanilla Tor. Regarding defenses, our design will focus on two principles: (i) the flows in the network should be as indistinguishable from each other as possible in terms of timing patterns, and (ii) the higher the number of flows passing through an extremity of the circuit, the better. Both of these principles serve the common purpose of making it difficult for an attacker to identify which flow at one end of the circuit corresponds to another flow observed at the other end. By making flows more indistinguishable from each other, we increase the adversary’s uncertainty when identifying pairs of flows, since the differences between the various candidates become smaller. Similarly, by increasing the number of flows at one end of the circuit, we also increase the uncertainty of the adversary, as the number of possibilities seen by them increases.

Regarding the first principle, our approach is to modulate traffic in the middle of a circuit, so that the patterns of traffic are modified between the two observation points used by an adversary to perform traffic correlation, i.e. the ingress and egress of the circuit. In this scenario, traffic modulation consists of using a statistical distribution to select and apply delays to traffic, to modify its timing patterns. However, it is not trivial to determine which distribution is the best for our purposes, as we have to evaluate not only the effectiveness in thwarting attacks, but also the performance impact introduced by each distribution. Therefore, it is also our objective to test multiple distributions and learn which one provides the most

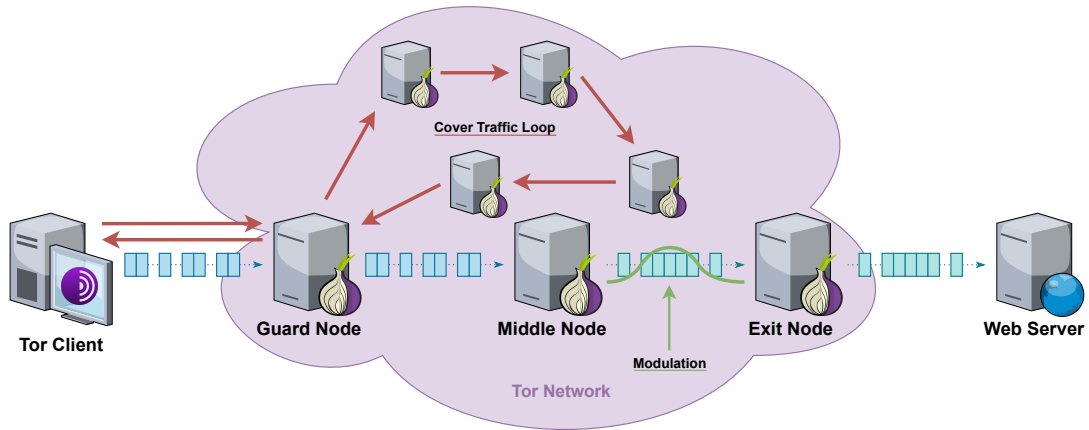


Figure 1.1: High-level overview of the Shaffler system showcasing its central techniques: (i) traffic modulation is implemented by the middle node, and (ii) cover traffic generation by the Tor client via a traffic loop between the client and a co-located onion service.

indistinguishability between flows and the least performance impact.

Regarding the second principle, we will study the feasibility and effectiveness of the cover traffic generation technique. This technique consists in having a client send cover traffic to an Onion Service (OS) hosted and managed by itself, to increase the number of flows passing through a specific node of the network. Our idea is to use the OS connection protocol to allow a client to connect to itself through the Tor network without requiring Network Address Translation (NAT) to be configured. With such a connection set up, it is possible to send cover traffic through the same Tor process as real traffic, increasing the amount of traffic going through the guard node, thus increasing confusion. Additionally, by using a self-hosted service, we are able to avoid affecting the performance of real services with our fake traffic.

Figure 1.1 presents an overview of the functioning of both these techniques, traffic modulation, and cover traffic generation. We can see, in green, the middle node of the circuit modulating the traffic, which alters the timing patterns of the traffic, represented by the shifted positions of the blue rectangles that symbolize packets. The path of the cover traffic created by the Tor client is highlighted in red. It goes through the guard node, is forwarded by multiple Onion Relays (ORs), and then returns to the guard node, forming what we call the *cover traffic loop*. With both of these mechanisms in place, we address both types of correlation that an attacker can perform at the endpoints of the circuit. This includes timing correlation, which is made more difficult by the traffic patterns being modified in the middle of the circuit, and volume correlation, which is made more challenging by the increased volume of traffic passing through the entry point of the circuit. These techniques will be implemented in the system we will develop, called Shaffler.

1.3 Contributions

This thesis presents the design, implementation, and evaluation of Shaffler, a system aimed at thwarting traffic correlation attacks on Tor by utilizing traffic modulation and cover traffic generation techniques. Specifically, this thesis makes the following contributions:

- Design of a mechanism that allows traffic modulation to be configured by a client and performed by a middle node in, that is fully compatible with the current Tor infrastructure.
- Design of a method for clients to generate cover traffic to add noise to the ingress of a circuit with minimal impact on the network, using an OS hosted and managed by the client itself.
- Implementation of a Tor version that supports our traffic modulation design, without requiring any large changes to be made, and thus allowing for a gradual deployment.
- Implementation of our cover traffic generation design in a way that can be easily used by clients by running a single component, that can be configured through various options provided.
- Evaluates the trade-offs of using different statistical distributions, with different parameters, for traffic modulation, in terms of the level of indistinguishability between flows and the performance impact on the network.
- Evaluates the effectiveness of the cover traffic generation technique, using different configurations, in terms of the level of noise added to the ingress of a circuit and the performance impact on the network.

1.4 Thesis Outline

The remainder of this document is organized as follows. Chapter 2 provides an introduction to the Tor network and its mechanisms, as well as a description of different types of attack targeting anonymity networks and defenses for such attacks, with a special focus on traffic correlation attacks targeting Tor. Chapter 2 also describes other anonymity networks and their mechanisms of defense against such attacks. Chapter 3 describes the design of Shaffler. Chapter 4 describes the details of implementation of the Shaffler prototype. Chapter 5 presents the results of the experimental evaluation. Chapter 6 concludes the document, highlighting the main findings and pointing possible directions for future work.

Chapter 2

Background and Related Work

In this chapter, we start by providing an overview of Tor, explaining how it provides anonymity and how that anonymity can be broken by known attacks. We then cover related work, starting by providing an overview of the known techniques used in traffic correlation attacks, as well as various defenses studied. Finally, we describe and discuss various defensive mechanisms used by other anonymity systems that are resistant to traffic correlation.

2.1 Background on the Tor Anonymity Network

Tor is an anonymity system that routes users' traffic through a series of routers, based on a technique known as onion encryption which provides users with the ability to connect to a server without allowing any entity to learn of their connection.

When a client wants to connect to a server through the Tor network, a circuit is created (see Figure 2.1). This circuit is usually made up of three nodes: the entry node that connects directly to the client, the middle node, and the exit node that connects directly to the server.

All data sent through that circuit is then encrypted three times, with each shared key previously negotiated between the client and each node, and in reverse order of the circuit. In this way, by having each node remove one layer of encryption, it is ensured that any data sent through the circuit changes form at each node, making it hard for an observer to trivially identify its path. Additionally, it ensures that only the exit node is able to learn the address of the server being connected to, so that it can forward the data to it.

Onion services (previously known as hidden services) are a way for users to publish web services that are only accessible through the Tor network as their addresses are not publicized on Domain Name System (DNS) servers. Instead, during setup, an OS nominates a selection of Tor nodes as its introduction points and creates a Tor circuit to each of them. It then creates a "service descriptor", which includes the public key of the service for authentication purposes, as well as the list of all the introduction points of that service. This descriptor is published in a distributed database on the Tor network called OS directories, along with the service address, which will serve as the key to its entry in the database.

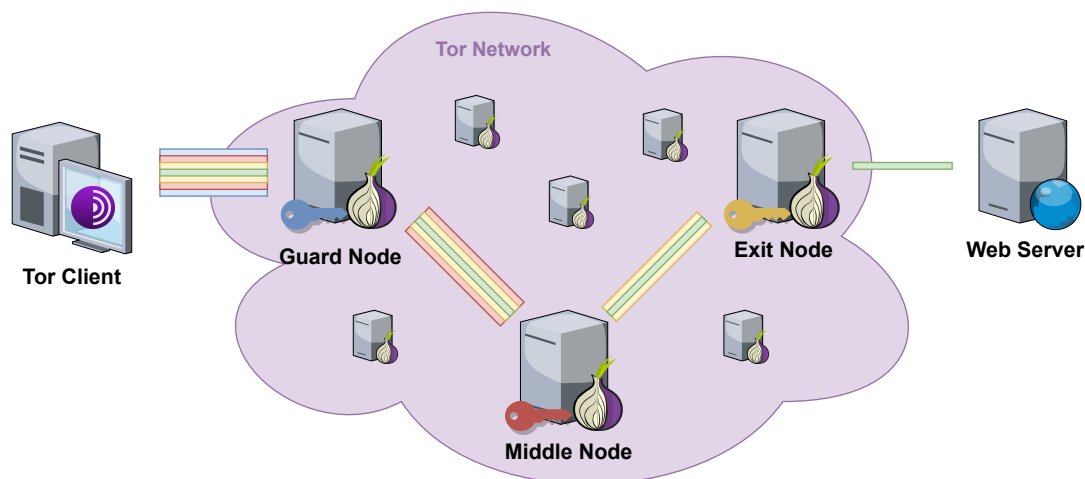


Figure 2.1: Example of a Tor circuit used by a client to access a Web server anonymously.

These service addresses are special addresses that end in “.onion”, and are usually long and difficult to memorize. These addresses are then manually shared with clients through a variety of means.

When a client wants to connect to an OS, he must first select a node as the Rendezvous Point (RP), establish a circuit to it, and send a request to connect to a specific address. The RP must then query the OS directories, select an introduction point from the received list, and send an introduction request to it. Upon receiving this request, the service connects to the RP through another Tor circuit, and from there onward all communications between the client and the service pass through that RP.

2.2 Attacks against the Tor Network

One of the most powerful classes of attacks aimed at deanonymizing Tor circuits are *correlation attacks*. A correlation attack is an end-to-end attack in which an adversary searches for a correlation between two flows captured at the ingress and egress of a circuit. Since an entry relay connects directly to a client and an exit relay connects directly to a server, an attacker can conclude that a client is accessing a certain server if they find that two observed flows are correlated. State-level adversaries are particularly well positioned to launch such attacks due to their privileged control over a country’s network.

We briefly describe some of the most effective correlation attacks and supporting techniques known today, as well as defensive mechanisms, studied and proposed by several authors that could possibly be integrated into Tor. However, many of the known defense mechanisms suffer from causing some kind of impact on the performance of the system. This concern for the user experience should not be considered to be unrelated to security, as anonymity systems rely on having large anonymity sets to hinder an attacker’s ability to identify the senders of observed traffic [10]. If the usability of an anonymity system is not satisfactory to users, it will result in a low adoption, which in turn will impact the privacy that the system can provide.

2.2.1 Timing attacks

These are end-to-end active attacks in which an attacker attempts to determine the endpoints of a circuit by correlating the time it takes for an identifiable flow generated by the attacker to travel from the entry to the exit relay. Chakravarty et al. [11] employ single-end bandwidth estimation to deanonymize the source of a given Tor connection. The attacker places probe servers alongside the egress and ingress routers at the boundary of ASes. Assuming that the client will connect to a server colluding with the adversary, the server will then send traffic along with a pattern that can be detected by the attacker's probing nodes. In this way, the attacker can discover the AS of the client and may further escalate the attack to pinpoint the actual location of the client.

Pries et al. [12] presented a different timing attack based on the disruption of the AES-CTR counter used to encrypt Tor cells. This attack requires that the entry and exit nodes are controlled by an attacker and works by duplicating a cell in the entry node, which causes a decryption error at the exit node. An attacker can correlate the time of cell transmission with the time of the decryption error to ensure that the error was caused by the cell duplication and ultimately identify the source of the connection.

Murdoch and Danezis [4] propose some defensive strategies to mitigate timing attacks. One strategy designated by *perfect interference* state that the output streams should all have the same shape, e.g., using threshold mix batching. However, Tor does not employ threshold mix batching, which relies on waiting for several streams and then flushing them all at once. Such a technique would increase the latency, defeating the low-latency characteristic of Tor. To prevent timing attacks, Tor relays send cells from different streams in a round-robin fashion.

2.2.2 Sybil attacks

These attacks serve as support for other types of attacks launched against the Tor network. In particular, Sybil attacks consist of the deployment of multiple malicious relays controlled by an attacker, while creating the illusion of belonging to different identities. The objective is to obtain a large disproportional influence in the network [13], to increase the likelihood that user-created circuits include malicious relays. In fact, an abundance of attacks on Tor, such as correlation attacks or timing attacks, depend on the amount of network traffic the attacker can control and on the ability to trick users into connecting through malicious relays [14].

Defenses against Sybil attacks are not trivial since a central authority would be required to assure a correspondence between a relay and an identity [15]. However, using a central authority contradicts Tor's goal to eliminate single points of control. Said authority would increase barriers on new relays' deployment. Another approach proposed by Bauer et al. [16] consists in limiting the number of new accepted relays at directory authorities based on some kind of identification. The idea is to limit the number of relays by IP address subnet. An attacker would need to have access to multiple sub-nets to conduct the attack. Neither solution directly stops sybil attacks but instead increase the attacker's cost for this kind of attack.

2.2.3 Predecessor attacks

Repeated communication between two endpoints of a Tor circuit may open the door to vulnerabilities that can be exploited by a traffic analysis capable adversary [17]. To perform a predecessor attack, an attacker must compromise an entry and an exit relay. This setup can arguably be made simpler through a Sybil attack, where an attacker can reduce the time to achieve control of more nodes. The goal of this attack is to learn the identity of a single or multiple senders when they are connected to a destination over time. The attacker maintains a shared counter (one per honest node) in each malicious node, initially zero. When a malicious node is selected to be a part of an anonymous circuit with a given destination, the counter of its predecessor is incremented. This counter represents the number of times a honest node is predecessor of a malicious node on circuits with the same destination. Figueiredo et al. [17] characterize the capabilities of the attack and the sufficient and necessary conditions for the attacker to succeed. They consider two situations regarding initiators: a single and multi-initiators, both continuously sending traffic to the same destination. The attacker can use the counter values to discover the set of initiators of a given circuit towards a specific destination.

Wright et al. [6] present a defense against predecessor attacks assuming a static network model, that is, nodes do not leave the network. In this model, they assumed a set of N nodes from which $C < N$ are controlled by an attacker. All N nodes communicate with the next node, the *responder*, which does not belong to N . The defense acts in rounds. In each round, each node computes a path containing all other nodes between them and the responder. To defend against predecessor attacks, the authors study the concept of fixing nodes in certain positions. There are variants of these defenses according to the selected positions (e.g. first, last, or first and last). Considering the case of selecting the first position on the path, unless the fixed node belongs to the set of nodes controlled by the attacker, users are protected since the initiator of user communications is indeed the node. Tor uses the same approach by imposing guard rotation restrictions [18].

2.2.4 Circuit fingerprinting attacks

Sun et al. [19] present an approach based on asymmetric traffic analysis allowing an adversary to correlate and deanonymize a circuit's endpoints, even if the attacker has access to different directions of the flow, e.g., from client to entry and server to exit. The rationale of this technique is based on the fact that the Internet relies on asymmetric connections, i.e. the path from the client to the server may differ from the same server back to the client, and that an adversary is still able to perform traffic correlation by observing different directions of a given flow. To augment the possibility of an adversary to observe a flow, the authors also propose a BGP intercept attack that can be launched by a malicious AS in order to divert traffic, enable traffic analysis, and forward traffic to the original destination.

2.2.5 Correlation-based analysis

One of the first correlation attacks based on timing analysis was proposed by Shmatikov and Wang [5]. Inter-packet timing information is usually not carefully protected in mix networks since it would require to

delay packets to hide timing patterns. An attacker can exploit this timing property by correlating the inter-packet time on both endpoint links, concluding that those links belong to the same circuit, which would tie a source to the corresponding destination. They analyzed the resilience of low-latency anonymous systems regarding correlation attacks based on packet interval time. They conduct several experiments using HTTP traces. The attacker starts by observing a set of entry and exit relays in order to link entries to exit nodes. Using an observation time of 60 seconds, the attacker divides the time into a fixed size window. An attacker counts, during each time window, the number of observed packets. The correlation between any two sequences is then computed using a cross-correlation metric.

A very influential work for performing flow correlation is DeepCorr [20]. DeepCorr uses advanced machine learning algorithms, instead of statistical metrics to conduct accurate flow correlation on Tor. Contrary to previous attacks, DeepCorr learns a correlation function which is able to link flow samples regardless of their destination, while accounting for the unpredictability of the Tor network. Another influential and current state-of-the-art work described in the literature is DeepCoFFEA [21], which is more effective than the previous state-of-the-art (DeepCorr), while also introducing a significant speedup.

In their work, Shmatikov and Wang [5] propose an approach consisting of using intermediate relays to inject dummy packets to normalize statistical information, named *adaptive padding*. This padding would reduce the ability of an adversary to fingerprint packets on a circuit. To protect against traffic correlation attacks based on machine learning techniques, as in DeepCorr, Nasr et al. [20] propose that Tor should enforce the use of pluggable transports across all relays, instead of just on bridges as in vanilla Tor. However, while the use of pluggable transports enables the obfuscation of both traffic patterns and content, the deployment of such a solution translates in significant performance reductions and is thus disregarded as a feasible defense against correlation attacks to be implemented in Tor. Other recent and promising countermeasures against traffic correlation attacks rely on employing AS-aware relay selection mechanisms [2, 7, 8] that effectively decrease the probability that an adversary is in a position necessary to observe traffic and carry out a correlation attack.

2.3 Defenses in the Tor Network

To mitigate traffic correlation attacks targeting the Tor network, researchers have proposed multiple defensive approaches with various strengths and weaknesses. Next, we survey the most relevant ones, organizing them into three main categories as described in each of the following sections.

2.3.1 Avoiding Unsafe Relay Nodes

As discussed above, by controlling the entry and exit nodes of existing Tor circuits, an adversary may be able to deanonymize them, i.e., identify the IP addresses of the corresponding sender and receiver, through multiple techniques. To mitigate adversary's attempts to launch such attacks, clients may attempt to avoid unsafe relay nodes by employing several strategies:

Run a co-located trusted relay node: Instead of connecting directly to an entry node that could be

controlled by an adversary, the user may run a local (trusted) relay node alongside the Tor client, and use that relay as entry node to the Tor circuits created by the user. As a result, the downstream relay nodes will not be able to determine whether the traffic forwarded by the trusted relay node was originally produced by the local user or by another user that may be using that same relay node for building its own circuits. However, requiring every single user to run a relay node cannot be broadly implemented, partly due to the hardships in configuring the system, but mostly because many Tor clients are located behind restrictive firewalls where they cannot relay traffic [22].

Scan and flag bad relay nodes: Another approach is to decrease the possibility that clients select malicious relays, by detecting and reporting bad relay nodes. Generally, a relay is considered to be bad if it is malicious, misconfigured, or unreliable. To mark relays, Tor uses a set of labels designated as *flags* [23]. Directory authorities vote to apply those flags by measuring bandwidth, uptime, and reliability. This allows the client to select relays according to their position on the circuit. For example, an entry relay should be long-lived. Otherwise, an attacker can setup a malicious relay and start discovering client identities right away. Tor maintainers run a service aimed at verifying the reports of possibly unsafe relays [24]. They will then attempt to reproduce the problem and possibly try to get in touch with the relay operator. If the problem cannot be solved, Tor maintainers will assign a flag – e.g., *BadExit* – to the reported relay, thereby instructing the clients to not use it any further in the future as exit node. Tor maintainers scan the network for bad relays, especially bad exit nodes, using a tool named *exitmap* [25]. Additional tools can be used for that purpose by the community in general, such as *torscanner*¹, and *tortunnel*². Some of these tools use decoy traffic to detect bad relays [26]. Tor also imposes that relays' bandwidths should be continuously monitored by directory authorities in order to prevent malicious relays from lying about their bandwidth (for load-balancing purposes, clients choose relays proportionally to their measured bandwidth capacity). Directory authorities compute weights associated with each relay class (entry, middle, exit) based on the current bandwidth of relays. This prevents a malicious relay from advertising a (fake) 100 Gbps bandwidth, which would make it eligible to be selected by many clients.

Restrict the set of entry nodes used by a client: Suppose that an adversary controls or observes C relays. Assume that the total number of relays on the Tor network is N . If every time a client uses the network, it selects a new entry and exit relays, this means that the adversary can correlate the traffic sent by the client with a probability of about $(C/N)^2$, i.e., (C/N) chance of connecting to the first relay and $(C - 1/N - 1)$ chance for the last relay. Thus, selecting many random entry and exit nodes will leave the user in a situation where his communications could be profiled by such an adversary. As a defense mechanism against such attacks, Tor employs *entry guards* [18]: each client chooses a (guard) relay from a list of three relays when making the first hop of circuits, and uses only those relays as entry nodes. If the adversary does not control (or cannot observe) the guard, then the user is secure. Otherwise, the adversary can indeed see a larger fraction of the user's traffic, but the chance of avoiding profiling drops significantly to a probability of about $(N - C)/N$.

In the past, clients chose an entry (guard) relay from a list of three relays, and each guard was

¹<https://code.google.com/archive/p/torscanner/> Accessed: 2023-01-05

²<https://github.com/moxie0/tortunnel> Accessed: 2023-01-05

discarded after a 30-60 day period. This rotation of guards helps prevent known attacks and allow to distribute client's load across multiple guards. In particular, if a client was unlucky and selected a malicious guard, he had the chance to regain anonymity when its current guard changed. However, such parameters were not strong enough to hold a large AS-level adversary; this is described as Guard Rotation Weakness [18]. Elahi et al. [27] empirically demonstrated that Tor's time-based guard rotation criteria led to clients switching guard relays more often than they should, increasing the possibility of profiling attacks. The authors mention two main threats: when a guard replaces another guard due to unavailability, and when the guard rotation occurs after a period of 30-60 days. In the first scenario, selecting a malicious relay to replace an unavailable one does not present an immediate threat. The malicious relay is placed at the end of a list composed of three guards that can be selected by the client. If the unavailable guard becomes available again, then the malicious relay is discarded and replaced with the previous guard. Selecting a malicious relay in the second scenario is a more critical threat, as the malicious guard relay will be used multiple times by the client. To achieve a better trade-off between anonymity and load balancing, clients are currently recommended to keep the same guard relay for a period of nine months. To be suitable to be elected as a guard, a relay must meet a minimal bandwidth threshold, its uptime must be greater than the median over all relays, and the node must have been present in the network consensus for at least two weeks.

2.3.2 Avoiding Unsafe Autonomous Systems

Unfortunately, the techniques presented above may not be effective against an adversary that can observe large fractions of the network. Such an adversary may not even need to control any specific relay node to deanonymize Tor traffic, but only to possess the ability to eavesdrop on the inter-relay traffic that crosses the network controlled by the adversary [3, 28].

To cope with this problem, several authors have proposed new defensive approaches against AS-level adversaries, i.e., those with the ability to access the network infrastructure of an entire AS. Typically, such approaches attempt to help Tor clients choose paths away from the prying eyes of malicious ASes by leveraging the analysis of the Internet topology boundaries and inter-relay latencies [29].

AS-level monitoring and tuning: As a way to prevent attacks based on traffic interception through BGP hijacking, Sun et al. [19] have proposed a monitoring framework for detecting BGP changes. The use of such a framework allows Tor to inform vulnerable clients, which in turn can opt to suspend Tor communications or use another relay. The BGP monitoring framework uses two main heuristics. First, if some AS announces a path to a prefix that it does not own frequently, the framework would alert for a possible hijacking attack. Second, if the prefix is advertised only for a short period of time, then it might be a routing attack. To have a robust solution, it is necessary to accurately know which ASes are traversed by a given circuit path. To this end, the proposed framework computes the traceroute of every Tor relay daily. Based on these results, it is possible to observe AS-level path changes and detect suspicious ASes. The same authors also propose several techniques to reduce the probability of AS-level attackers to perform BGP hijacking and interception attacks. As a matter of fact, most Tor

relays' IP addresses (90%) have a prefix shorter than $/24$. Therefore, authors suggest that Tor relays should not operate with a prefix shorter than the $/24$ prefix. However, even with a $/24$ prefix, an attacker can advertise another equal prefix. To cope with this situation, clients should prefer guard relays whose AS-level path is the shortest. Given that a smaller number of ASes on the path translates into a lower chance that one of them will advertise incorrect prefixes, this measure reduces the chance that malicious AS falsely advertise routing prefixes.

AS-aware path-prediction: In order to mitigate correlation attacks, several authors have proposed AS-aware path selection algorithms to decrease the chance of an AS-level attacker being able to observe traffic flowing between both endpoints of a Tor circuit [2, 7, 8, 19].

Edman and Syverson [8] have experimented with adding two different requirements to Tor's path selection algorithm. The first requirement mandates that each node in a circuit must be located in a different country, while the second dictates that, instead of requiring unique countries, each node should be located in a different AS. Although the two approaches decreased the probability that an AS would be able to eavesdrop at both ends of a connection, they did not sufficiently mitigate the possibility for a malicious AS to perform traffic correlation. Thus, the authors have proposed a more effective heuristic for safe AS path-aware selection. First, the client finds all the shortest forward and reverse AS paths from the client's AS to the entry node and from the exit node to the destination. The entry and exit paths are sorted according to the cumulative frequency values of each edge in the path. The shortest n paths with the greatest cumulative edge frequency values are then assumed to be the n most likely AS-level paths from a source AS to the destination. If the same AS appears in any of the n entry paths and any of the n exit paths, the chosen entry-exit node pair is discarded and a new pair is selected.

Akhoondi et al. [7] propose LASTor, an AS-aware Tor client that selects safe paths between a client and a destination. The key insight of this technique is in the prediction of the set of ASes through which traffic may be routed between a pair of IP addresses, instead of performing a prediction for a precise route between these addresses. The potential for the existence of a malicious AS able to perform traffic correlation is determined by checking if the intersection between the AS sets for the paths between the client and the entry relay and between the exit relay and the destination is non-empty.

Sun et al. [19] approach consists in monitoring and storing paths between the client and the guard relay, and also between the exit relay and the destinations. The idea is that each relay publishes, as part of the Tor consensus document, the list of ASes it uses to reach a given relay or destination. Clients can then use this information along their own measurements when constructing circuits. This allows clients to select relays in a way that one AS will not appear on both the entry and the exit relay.

Nithyanand et al. [2] further developed an AS-aware variant of Tor, called Astoria. Astoria avoids vulnerable circuits while employing an efficient network management by load balancing circuits across secure paths. First, Astoria takes advantage of Internet topology maps to predict which ASes are placed in critical locations for performing traffic correlation. Second, it identifies which ASes are more likely to collude with each other and increase the chance of malicious ASes to launch a successful attack. Astoria leverages a probabilistic model to select circuits on paths less liable to be correlated by an adversary, when no completely safe circuits are found. This model minimizes the amount of traffic that

is observable by an attacker over time. When there is more than one safe path available Astoria causes a load balancing technique to prevent relay overload. By leveraging these techniques, Astoria is able to reduce the probability of selecting a vulnerable circuit from 40% to 3%. The path selection algorithm of Astoria is, however, arguably incomplete. Due to the existence of active BGP interception attacks [19], Astoria's Internet topology maps may become outdated for short periods of time, allowing an attacker to still launch a correlation attack.

2.3.3 Avoiding Unsafe Geographical Regions

In addition to avoiding specific ASes, related literature focuses on avoiding entire geographic regions altogether, typically at the country-level granularity. The motivation is oftentimes the need to evade censorship policies against Tor traffic implemented by repressive governments.

Tor allows users to select a set of countries to exclude from circuit selection [22] i.e., regions to which it should not forward client's traffic. DeTor [30] presents techniques to prove that a Tor circuit did not travel within excluded regions. To provide the so-called *provable geographic locations*, DeTor authors borrow the idea of *alibi routing* [31] into Tor. Alibi Routing (AR) uses the packets' Round Trip Time (RTT) and the speed of the light as a constant to prove that a given packet did not travel within forbidden regions. AR uses a single relay located outside the forbidden region to confirm that traffic is going from that relay to the destination. This confirmation is based on a message authentication code issued by the relay itself, attesting that the traffic was forwarded by that relay. If the RTT from the relay to the destination is less than the smallest RTT that also includes a forbidden host, then it is possible to conclude that the packet could not travel from within the forbidden region. While AR uses a single relay, DeTor [30] generalizes this approach to three relays.

However, there are a few limitations regarding DeTor which make it an unconvincing solution for providing provable geographical avoidance. A first limitation concerns the fact that DeTor obtains the list of relays from the Tor's public database which contains several bits of information, such as: IP address, port, public key and country. If the information about the country is unavailable, DeTor uses IP geolocation services to find the exact location of Tor relays, but without any further confirmation. This may hamper DeTor's ability to perform accurate measurements [32, 33]. A second limitation involves handling links with high latency, where it is not possible to measure the packets' travel times accurately solely relying on RTT measurements. Another limitation is that DeTor assumes a symmetric routing nature, i.e. assumes that the request and reply will traverse the same geographical locations, something that may be unlikely to happen in practice.

A more recent piece of work by Kohls et al. [33] introduces the concept of *empirical avoidance* and proposes new improvements to overcome DeTor's main limitations. In their work, the authors propose TrilateraTor, a system introducing a new measurement technique that derives a circuit end-to-end timing directly from the handshake in Tor's circuit establishment procedure. To prevent the use of fake GeoIP information in its measurements, TrilateraTor leverages a distributed measurement infrastructure to perform trilateration and obtain accurate estimates of the physical location of Tor nodes.

2.4 Defenses Used in Other Anonymity Systems

In addition to the defensive mechanisms studied for Tor, various alternative anonymity systems have emerged that incorporate a wide range of defenses against traffic analysis attacks. In the following sections, we provide an overview of the key mechanisms commonly employed across these systems, starting with the most prevalent strategies, namely *traffic mixing* and *distributed mixnets*.

2.4.1 Traffic Mixing and Distributed Mixnets

Mixnets are used to obfuscate the correlation between a user's traffic at both ends of a communication. These mixnets consist of multiple servers or intermediaries through which traffic is routed, effectively shuffling the packets' order and timing.

Systems such as Vuvuzela [34], Karaoke [35], Loopix [36], Nym [37], and Atom [38] all leverage variations of this traffic mixing technique. Generally, mixnets use three different mixing methods that, when combined, provide a strong defense against traffic correlation attacks:

- **Permutation mixing:** The order of the arriving packets is shuffled before being sent to the next node of the mixnet.
- **Chaff mixing:** Dummy packets are created and mixed with real packets to ensure that the real traffic is always mixed with some other traffic, even when there is no other real traffic to mix with.
- **Path mixing:** Traffic from different clients is routed through different paths, making it difficult for an adversary to even determine the endpoints to observe.

Although these three techniques in combination provide the strongest defense against traffic correlation attacks, they also introduce a significant increase in latency. As such, some systems may opt to use only a subset of these techniques, trading anonymity for better latency.

However, for these techniques to be as effective as possible, mixnets must follow a round-based approach. This consists in forcing senders in the network to wait for a round of communications to send their data. These rounds may be defined in different ways, but the two most common approaches are by being globally scheduled or by being triggered by a certain amount of traffic ready to be sent surpassing a threshold. Vuvuzela [34], Karaoke [35] and Atom [38] are all examples of systems that operate in scheduled rounds. By operating in predefined rounds of communication, these systems can simplify the timings of all communications, making it difficult for adversaries to find and correlate timing patterns to break anonymity. Additionally, by accumulating traffic over time and sending it at a predefined time, the effectiveness of the mixing performed by the mixnet also increases. However, this approach also introduces a significant increase in latency, since clients must always wait for the next round to transmit data. And, depending on the system, even the mix nodes may also have to wait for the next round to transmit data, as is the case in Atom [38]. This makes these systems unsuitable for applications that require low latency, such as web browsing. As such, it is used only by systems designed for applications where low latency is not required, such as messaging systems.

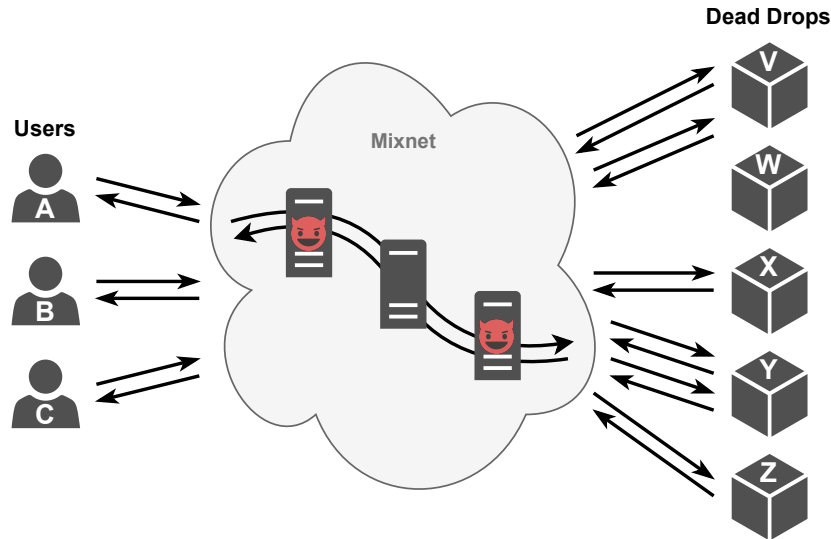


Figure 2.2: Overview of Vuvuzela's [34] conversation protocol.

Triggered rounds are another approach commonly used in mixnets to balance the trade-off between anonymity and latency. In triggered rounds, the start of a round is not pre-scheduled, but instead depends on certain conditions being met, typically related to the amount of traffic waiting to be sent. This is commonly used by mix nodes of a mixnet, which benefit strongly by accumulating received traffic and relaying it to the next node only after enough traffic was accumulated to be able to perform the mixing techniques effectively.

2.4.2 Dead Drops and Message Relays

The concept of dead drops, or intermediary storage points for messages, is another commonality. In systems like Vuvuzela [34] and Karaoke [35], messages sent by users are temporarily stored on these dead drops before recipients retrieve them. This separation in the communication process introduces uncertainty, as adversaries cannot directly link sender to receiver. However, this approach often necessitates the use of a rounds-based system, described further in this section, which increases latency in the communication process.

Figure 2.2 shows an overview of the conversation protocol of Vuvuzela [34], where two clients, to communicate with each other, must connect to the same dead drop through a mixnet each. For example, if users A and C want to communicate, they must agree to use, for instance, dead drop X, and connect to it both through their own mixnet circuit each. Additionally, Vuvuzela tolerates that two of the three servers used in each of those circuits may be compromised by an adversary, which is represented in the figure by the two red faces.

Loopix [36] uses something similar to dead drops, service providers, which can be seen in Figure 2.3 at the edges of the mixnet. In Loopix, these service providers serve as intermediaries and mailboxes for clients. When a client wishes to communicate with a certain entity, it sends all traffic through its service provider. Its service provider will then forward it through the mixnet to the destination and, upon

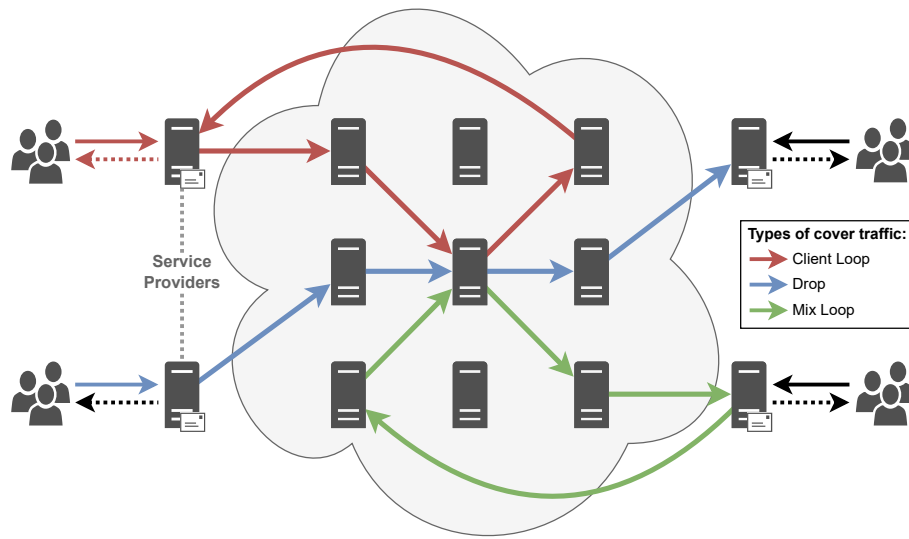


Figure 2.3: Overview of Loopix's [36] architecture.

receiving the destination's response, will store it until the client requests it. The usage of these service providers improves receiver anonymity by ensuring that recipients only receive messages when explicitly requested, rather than as soon as possible. This makes it harder for an attacker to link a sender to a receiver through observation of timings, as the time the receiver requests its messages from the service provider may be completely unrelated to the time they are sent.

2.4.3 Constant-Rate Traffic and Padding

A prevalent countermeasure against traffic correlation attacks is the injection of noise into the communication process. Both Vuvuzela [34] and Karaoke [35] introduce noise to obscure information about communications that an observer may learn. These systems mainly do this by forcing all clients to always send a message each round, even if they have nothing to send and no communication to perform. In the case of Vuvuzela [34], the mixnet servers add noise to thwart attackers' efforts to correlate user actions with traffic patterns. This noise serves to introduce uncertainty about the authenticity of observed traffic and the presence of actual communication.

Another anonymity system, Loopix [36], employs a unique approach to noise injection by incorporating three types of cover traffic. Of these three types of cover traffic, two are sent through loops, allowing the senders of traffic to avoid overloading other destinations and allowing them to detect possible issues or attacks being performed in the network. The three types of cover traffic used in Loopix are the following.

- **Drop Cover Traffic:** Generated by a client and sent to a randomly selected service provider, instructing it to drop the traffic.
- **Client Loop Cover Traffic:** Traffic generated by a client and sent back to itself, passing through

the mixnet and arriving back at the client.

- **Mix Loop Cover Traffic:** Traffic generated by a service provider and sent back to itself, passing through the mixnet and arriving back at the service provider.

These types of cover traffic, shown in Figure 2.3, introduce uncertainty about the authenticity of observed traffic and the presence of actual communication. By blending real messages with cover traffic, Loopix [36] improves user privacy and makes it challenging for adversaries to distinguish real communication from noise. The design of our system, Shaffler, takes a lot of inspiration from these techniques used by Loopix.

Systems such as TARANET [39], an anonymity system based on HORNET [40], that aims to thwart traffic analysis, employ the concept of constant-rate traffic, achieved through the use of traffic padding. A sender in the TARANET [39] system divides its traffic into *flowlets* that transmit packets at a constant rate defined globally. Since all senders transmit at the same rate, their traffic becomes indistinguishable, making it theoretically impossible to correlate traffic entering the network with traffic exiting it.

To achieve this constant rate of traffic, as well as to allow a client to transmit data even when it is not enough to fill a flowlet, TARANET [39] uses traffic padding. This padding is added to the client's traffic to ensure that it is always transmitted at the constant rate. However, determining the optimal constant rate remains crucial. Setting it too high risks network overload, while setting it too low may limit the rate at which users can transmit real traffic. The challenge is to strike the right balance to ensure strong anonymity and low latency.

Other systems, such as ditto [41], a traffic obfuscation system for Wide Area Networks (WANs), employ similar techniques that, instead of aiming for a constant rate of traffic, aim for a repeating pattern of traffic that is as similar as possible to real traffic. These systems do this by adding dummy packets or chaff traffic in a controlled manner, to achieve that predefined traffic pattern. This pattern is then repeated over time, making it difficult for an adversary to know when real traffic is being sent and the amount of real traffic being sent.

2.4.4 Traffic Modeling and Delaying

Traffic modeling involves representing traffic as a stochastic or probabilistic process, typically following a known statistical distribution or modulation function. One possible way to perform this traffic modeling is by applying to each packet individually a delay selected through a certain function based on probability. In Loopix [36], for instance, the Poisson distribution is used for this purpose due to its mathematical properties and suitability for modeling random independent events over time.

Essentially, each client models its traffic as a composition of four independent Poisson processes, representing real traffic and each of the three previously mentioned types of cover traffic. An interesting property of this design is that the sum of these Poisson-distributed processes results in a composite Poisson-distributed traffic pattern. As a result, the cover traffic blends seamlessly with the real traffic in a way that preserves the statistical properties of the original Poisson process, making it challenging for adversaries to distinguish real communication from the noise introduced by the cover traffic.

	Low Latency	Resistant to TCA	Traffic Modulation	Cover Traffic	Compatible with the Tor Network
Tor [1]	✓	✗	✗	✗	✓
Vuvuzela [34]	✗	✓	✗	✓	✗
Atom [38]	✗	✓	✓	✗	✗
Loopix [36]	✓	✓	✓	✓	✗
Nym [37]	✓	✓	✓	✓	✗
Shaffler	✓	✓	✓	✓	✓

Table 2.1: Comparison of various anonymity systems.

Additionally, Loopix [36] introduces a delay mechanism to further obfuscate the correlation between the traffic of a user at both ends of a communication. This delay is chosen by the client independently for each hop in the path, from an exponential distribution, and sent inside its corresponding layer of encryption, so that each corresponding mix node can retrieve its delay and apply it. By using an exponential probabilistic distribution, not only are the delays likely to be small, but the traffic’s modeling as a Poisson process is also preserved, thanks to the memoryless property of the exponential distribution. Furthermore, a study published by Danezis [42] concludes that the optimal distribution from which to select delays for continuous-time mixing is the exponential distribution, in terms of the resulting anonymity and latency. Another system that uses the exponential distribution to select delays is the Nym network [37].

Although these probabilistic distributions are chosen specifically to minimize as much as possible the latency introduced to the system, they still introduce significant communication delays in exchange for stronger anonymity.

2.4.5 Discussion

As discussed in Sections 2.2 and 2.3, Tor is vulnerable to traffic correlation attacks, where an adversary attempts to correlate traffic observed at both endpoints of a circuit through the analysis of various patterns to link the source of a communication to its destination.

One way to reduce the risk of such attacks is to avoid unsafe relay nodes. This can be done, for example, by deploying trusted relays on the client endpoints, continuously scanning and announcing bad relay nodes, or by imposing the usage of reliable guards by limiting the period of rotation.

However, even if we manage to avoid using compromised or malicious nodes, it is still possible for a larger AS-level adversary to perform these attacks simply by observing the traffic in the network, without the need to control ORs. While this threat can also be mitigated by attempting to identify and avoid unsafe ASes and geographical regions, other defenses have been studied that tolerate the fact that an adversary may have the ability to observe traffic at both endpoints of a circuit.

Several anonymity systems have been proposed that employ various techniques to deal with these threats, such as generation of cover traffic and traffic modulation, both of which we aim to implement in our modified version of Tor. Table 2.1 presents a summary of some characteristics provided by Shaffler, and compares it with various other anonymity systems. Although these techniques that we aim to use are already present in many anonymity systems, Shaffler aims to bring these techniques to the Tor

network, providing a version of Tor that not only supports them, but also maintains compatibility with the existing Tor infrastructure.

Summary

This chapter discussed the concept of the Tor anonymity network and its components, such as OSeS. We noted the various types of attacks studied against the Tor network, which can be divided into two categories: those that involve malicious Tor entities and those that involve traffic analysis. The chapter also discussed the main defenses studied against these attacks. Entering in more detail about the main approaches to defend against traffic analysis attacks, the chapter discussed preventing users from selecting unsafe relays or autonomous systems entirely. Finally, the chapter presented an overview of the main techniques used by other anonymity systems to defend against traffic correlation attacks, discussing their advantages and disadvantages. The next chapter presents the design of Shaffler, our proposed solution to enhance Tor against traffic correlation attacks.

Chapter 3

Design

In this chapter we present the design of Shaffler, our proposal to protect Tor against traffic confirmation attacks. First, we start by presenting our threat model (Section 3.1). Then, in Section 3.2, we present the architecture of Shaffler. We start by providing an overview of both our system and its novel defensive strategy to prevent said attacks. Then, we further explain how we propose to overcome several technical challenges of integrating Shaffler with Tor covering independently the two central defensive techniques of our system: traffic modulation (Section 3.3 and covert traffic generation (Section 3.4).

3.1 Threat Model

Our threat model extends Tor’s original threat model by adding focus to the threat of traffic correlation, which is listed as a non-goal of Tor in the original paper [1]. Specifically, we consider our main adversary to be an attacker that can passively observe the *ingress* and *egress* flows of a circuit and may attempt to correlate them. Such an adversary can be impersonated, for instance, by an ISP or multiple colluding ISPs with the ability to monitor the guard nodes and exit nodes of Tor circuits. ISPs could be mandated by governments, law enforcement agencies, or other organizations with the resources and capabilities to conduct targeted or mass surveillance operations.

Although this threat model does not consider global passive adversaries, which would assume that the adversary could observe the communications of all the Tor nodes, it does assume that adversaries may have the ability to monitor network traffic, either by tapping into the network at various points or by controlling routers used in a circuit. However, we realistically limit the strength of the attacker to being able to control at most two of the three ORs used in a circuit. Considering that the attacker’s objective is to perform traffic correlation, the most threatening of the resulting combinations are the entry and exit nodes. As such, we assume that the middle node is trusted.

Similarly to Tor’s original threat model, we consider that the exit relay of a circuit may be compromised, meaning that the attacker may have access to its internal state. However, we do not consider any active attacks.

Attackers may also have access to advanced computing resources that, while not enough to break

cryptographic primitives, can be used to deanonymize users. Adversaries may also possess sophisticated traffic analysis techniques, such as machine learning algorithms.

3.2 Architecture

Shaffler aims to protect Tor clients against traffic confirmation attacks. To do this, Shaffler proposes a new traffic mixing strategy for Tor. Considering that we want to protect the circuit of a specific Tor client (a *target circuit*), this approach is based on two main ideas. First, we want to ensure that the entry node, the exit node, or both receive enough concurrent covert traffic so that it can disguise the packets tunneled through the target circuit. Otherwise, in the extreme case where the target circuit is the only circuit being relayed through the entry and exit nodes, the adversary can trivially infer that a single source is transmitting packets through both nodes and perform timing and volumetric analysis to deanonymize the client.

To prevent this problem, Shaffler generates client-controlled covert traffic directed toward the entry node of the circuit. This is done by running a dedicated web server behind an OS in the client's own machine and initializing a *cover client*, which creates *covert sessions* with that OS. By using the same Tor process for the real user traffic, the cover client traffic, as well as the cover OS traffic, we not only ensure a better mixing of all types of traffic, but also ensures the usage of the same guard node for all traffic.

Secondly, Shaffler will further perturb the timings of the packets tunneled through both the target circuit and the covert OS sessions, making timing analysis harder for an adversary to perform. This perturbation is achieved by carefully delaying packets at the middle node of a circuit so that the timing patterns observed at the entry of the circuit suffer modifications before reaching the exit of the circuit, where an adversary would expect to observe them again. By making modifications to the timing patterns of traffic in the middle of the circuit, we make it harder for an adversary to accurately identify correlations between timing patterns observed at both edges of the circuit. Additionally, when used in combination with the generation of cover traffic, it may increase the probability that flows observed at other exit points of the network are identified as being more similar to a flow observed at the entry than the truly related exit flow. However, packet delaying must be achieved without causing visible alterations in the typical packet time distributions of regular Tor circuits and without introducing significant overheads to the end-to-end circuit latency. Furthermore, modulating packet timings should be performed without the need to rely on the correct or informed behavior of the exit nodes, which could be controlled by the adversary. To satisfy these requirements, our idea is to implement packet timing modulation controlled by the client and with the cooperation of the middle nodes, employing specific modulation functions that need to be carefully studied.

With all this in mind, we designed Shaffler following the architecture depicted in Figure 3.1. We propose to integrate four custom components for modulating traffic: a *modulation instructions decoder*, a *modulation instructions encoder*, a *modulation function*, and a *cell delayer*. These components can be observed in the Tor software stack, in Figure 3.1, at the client endpoint and at the mix nodes, colored

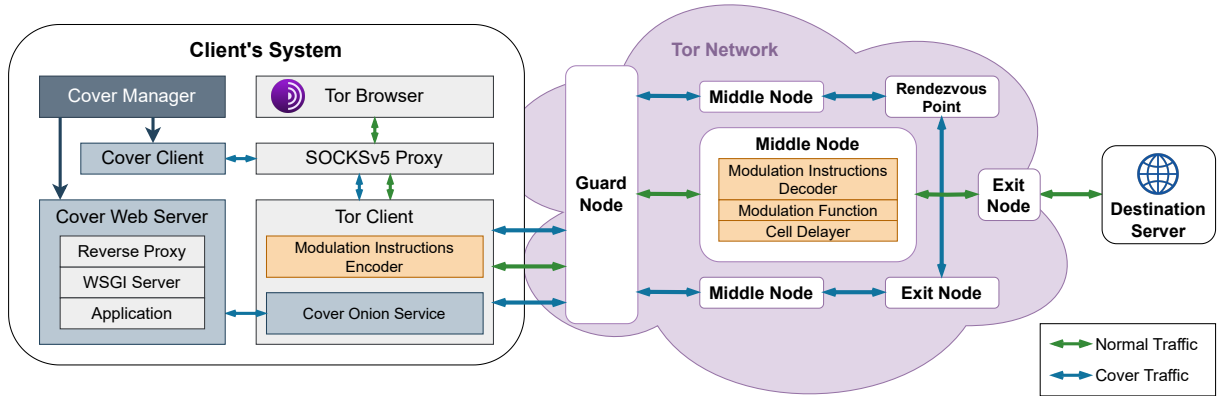


Figure 3.1: Shaffler system architecture.

orange. In addition to these components, the client will include an additional component named *cover manager* which will be responsible for setting up and maintaining covert OS sessions, by deploying the necessary processes. This component and its spawned processes can be seen represented in the Client's system in Figure 3.1, colored dark gray. Together, these components generate cover traffic and delay the transmitted data to attempt to prevent attacks carried out by an adversary that can use machine learning algorithms to correlate the traffic observed at both endpoints of a circuit. Additionally, Shaffler is designed to be compatible with the vanilla version of Tor, so that if a user running Shaffler attempts to form a circuit through non-Shaffler nodes, the standard functionality of Tor is guaranteed, even if the features of Shaffler are unavailable. To use Shaffler, a user must simply install the modified version of Tor and launch both that version of Tor and the *cover manager*.

3.3 Traffic Modulation

One of the techniques used by Shaffler is traffic modulation, which consists of delaying traffic through the use of a modulation function that specifies how delays are chosen. These modulation functions can be based on multiple approaches, ranging from state machines to statistical distributions.

Modulation functions: Although our main objective in this thesis is to provide and study modulation functions based on statistical distributions, it is also our focus to design our system in such a way that other researchers can easily implement and test their own modulation functions. Table 3.1 shows the modulation functions we include in our prototype, as well as their parameters and a visual representation of their probability curve. Additionally, it includes a brief description of the characteristics of each function with respect to the distribution of probability across a range of values.

Although these characteristics may provide us with intuitions about the effectiveness and impact on performance of each function, it is important to verify those intuitions with experimental results. For this purpose, we must perform an experimental analysis comparing all functions, which we will present in Section 5.2.

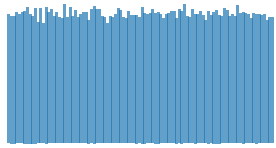
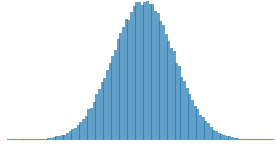
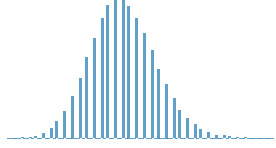

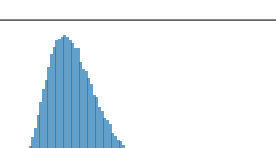
Name	Parameters	Characteristics	Curve
Uniform	$min : minimum$ $max : maximum$	All values between the min and the max have the same probability of being selected.	
Normal	$\mu : mean$ $\sigma : std. deviation$	Values closer to the mean have a higher probability of being selected, and both sides of the mean have an equal distribution of probability.	
Poisson	$\lambda : mean$	Values closer to the mean have a higher probability of being selected, and the distribution becomes more concentrated as λ increases.	
Exponential	$\lambda : rate$	Values closer to zero have a higher probability of being selected and the distribution becomes more concentrated as λ increases. High values have a low probability of being selected but are possible.	
Lognormal	$\mu : location$ $\sigma : shape$	Higher concentration of values around the mean, with a long tail of high values, which are possible but have a low probability of being selected.	

Table 3.1: Table showing the modulation functions that are provided by the Shaffler system.

Who modulates: It is also important to decide which entity in the circuit should be responsible for applying the modulation functions. Although having as many entities as possible modulating traffic and applying delays would possibly lead to stronger protection, it would most likely have an unacceptable impact on the latency of any communication. An exception to this would be if we were able to achieve a fine enough time resolution when delaying to allow modulation to be distributed through various entities. However, achieving such a fine resolution may not be feasible and even if possible, may result in requiring better computational resources to run Shaffler. As such, we decided on a single entity to assign the role of traffic modulator. Considering our threat model, which states that the entry and exit nodes of a circuit may be observed by an adversary, choosing either the entry or exit node for this role would result in the technique being either defeated or weakened. Figure 3.2 shows how either of these nodes, the entry and the exit, would be able to easily defeat the effects of the traffic modulation technique, in the case of the entry node, by linking the client with the traffic patterns observed after modulation, and in the case of the exit node, by linking the patterns observed before modulation with the web server that is being accessed.

Another option would be to have the client modulate its own traffic, which would likely help normalize the traffic timing patterns of all flows. However, by modulating the traffic before it even reaches the entry node, we are unable to modify the timing patterns so that these patterns are not seen again at the exit

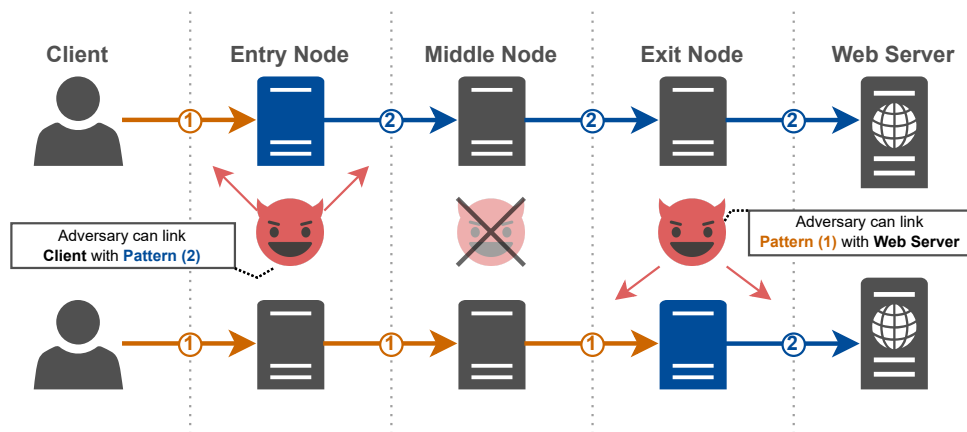


Figure 3.2: Diagram showcasing the potential issues with relying on the entry node or the exit node to modulate traffic. Our threat model considers that an adversary may be able to monitor traffic at both the entry and the exit nodes, but not at the middle node.

node. Taking into account these limitations, we decided to assign this responsibility to the middle node, which our threat model assumes to be trusted and which is in the best position in the circuit to be able to modify the traffic patterns between the two observation points of the path followed by the user's traffic.

Who decides how to modulate: It is also important to decide which entity in Shaffler should have the responsibility of choosing how to modulate traffic. While having the delaying node decide which delays to add may lead to a simpler solution, it does not provide users of the Tor network with the ability to choose and tweak their own priorities. This may be important as different clients may have different priorities, and while one user may be more concerned with ensuring the best anonymity possible, another user may prefer sacrificing some protection to traffic correlation attacks for better performance. Providing this option to clients is especially important, given that Tor is already known to be slow, and delaying traffic will worsen that even more. Furthermore, by making this a client's decision, even if the technique's performance results leave a lot to be desired and are unacceptable for regular use, it may still be used as an *ultra protection* mode that may be activated only when a user truly feels it is needed.

Another option would be to have the client alone decide how their traffic is to be modulated. For this, a mechanism would be required that allows the client to inform the middle node of the exact delay they should apply to a packet. This delay would have to be encoded in some way in every Tor cell, so that it may be decoded and applied. However, this approach may raise some problems, as it is likely to require adding an overhead to each Tor cell, and it may be hard to maintain compatibility with the existing Tor network. As such, a hybrid of these two approaches was also considered. In a hybrid approach, both the client and the delaying node have roles to play. The client is able and expected to provide, during the creation of the circuit, instructions to the delaying node on how traffic should be modulated. This information must then be remembered by the delaying node so that it may follow those instructions when deciding on a delay to apply to a cell passing through that circuit.

How traffic is delayed: To avoid changing the order of cells, delays must not be applied in relation to the arrival of a cell but instead in relation to the time the previous cell was sent. The only exception to this is if the previous cell has been sent too long ago, which would lead to no delay being applied. In this case, the delay must exceptionally be applied in relation to the arrival time of the cell.

3.4 Generation of Cover Traffic

As mentioned in Section 3.2, our solution resorts to generating cover traffic to complement the traffic modulation technique explained above and carried out at the circuits' middle nodes. The goal is to strengthen the anonymity guarantees provided by Shaffler by further hindering the correlation of flows captured near the client and the ones captured near the webserver being accessed.

Shaffler does this by generating artificial Tor traffic and directing it towards the legitimate circuit's guard node. Instead of accessing a different publicly available webserver, our solution spawns one in the client's machine, making it accessible through an OS, and looping the traffic in an approach similar to the one used in Loopix [36]. A possible alternative to using an OS would be to use a simple web server, also hosted on the client's machine. However, although initially it might give the idea of reducing the load of the technique on the network, that does not seem to be the case. To compare the unnecessary load on the network created by these alternatives, we can use the *Precision* metric (given by Equation 3.1), where $\#TP$ is the number of cover traffic flows that pass through the target node, and $\#FP$ is the number of cover traffic flows that pass through other nodes, as a side effect.

$$Precision = \frac{\#TP}{\#TP + \#FP} \quad (3.1)$$

For the non-OS approach, we have $\#TP = 1$ as the flows of cover traffic pass once through our target, the guard node, and $\#FP = 2$ as the flows of cover traffic pass through two other nodes, the middle and the exit. For the OS approach, we instead have $\#TP = 2$ since each flow of cover traffic passes twice through the guard node and $\#FP = 4$ since these flows will pass through four other nodes, as shown in Figure 3.3. By calculating this metric for both approaches, we obtain the same efficacy value of $1/3$, which means that even though the OS approach requires more non-target nodes to forward traffic, the increase in “useful” load increases proportionally with this increase in side effects.

However, there are some specific differences, such as that by using an OS instead of a simple web server, we can avoid forwarding cover traffic through exit nodes, which are less common in the Tor network than other types of nodes. Additionally, by using an OS we avoid requiring that users configure NAT on their local network to make the web server accessible from outside the network. For these reasons, we decided to use an OS.

In addition to spawning an OS to receive and respond to loop traffic, Shaffler also runs a dedicated process responsible for generating it. This component, called *cover client*, sends requests to the client's OS, making sure to do so via the same guard node as the legitimate traffic. This *cover client* is customizable, allowing us to control the frequency of requests and adjust the desired response by the OS.

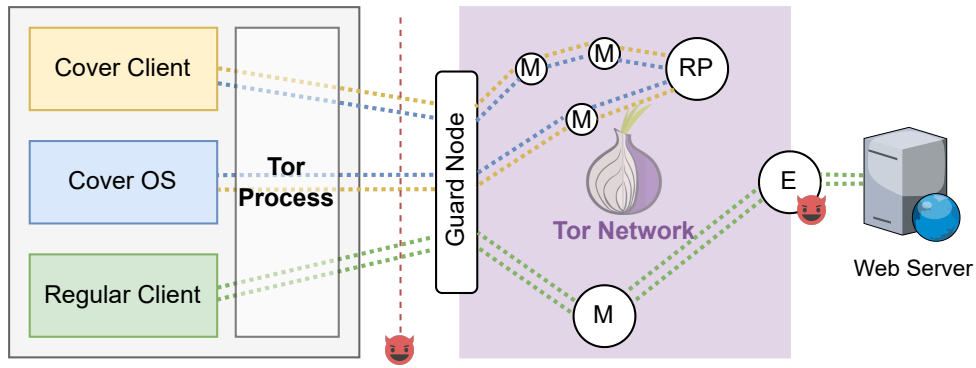


Figure 3.3: Shaffler's cover traffic generation design.

The highly customizable nature of both *cover OS* and *cover client* enables Shaffler to tailor the generation of cover traffic to the characteristics of the legitimate process. We intend to study how different patterns of artificial traffic impact the system in both efficacy and network overhead, seeking to strike a balance between a solution that indeed defends against correlation attacks while not causing significant performance degradation for the user or excessive load on the Tor network.

Figure 3.3 depicts Shaffler's cover traffic generation design. In it we can see a user, on the left, using the system to mask its access to a web server, on the right, through the Tor network. In particular, we can see how, in addition to the user's "regular client" (the one used to access the web server), there are two additional components involved: the *cover OS* and the *cover client* mentioned above.

Shaffler's design requires all 3 components to send traffic through the same Tor process running on the user's machine. This makes it so we can ensure all packets sent to the Tor Network do so via the same Guard Node with minor configuration required. The fact that all Tor traffic sent from the user's machine enters the network through the same Guard hinders an attacker's ability to correlate the flows captured at both the Guard and Exit nodes. In fact, even if the packets captured near the web server concern only the user's access to it (green colored traffic), the packets captured near the Guard now consist of a mix of web server traffic (in green) and cover traffic (in yellow and blue). Both OS requests and responses can be modulated as needed, allowing for the introduction of further variability.

Summary

This chapter presented the design of Shaffler, our proposal to protect Tor against traffic confirmation attacks. We started by presenting the threat model considered. Then we presented the architecture of Shaffler, which is based on two main ideas: generating client-controlled covert traffic directed toward the entry node of the circuit and further perturbing the timings of the packets tunneled through both the target circuit and the covert OS sessions. We also presented the four custom components for modulating traffic: a *modulation instructions decoder*, a *modulation instructions encoder*, a *modulation function*, and a *cell delayer*. In addition to these components, the client will include an additional component named *cover manager* which will be responsible for setting up and maintaining covert OS sessions. In the next chapter, we present our implementation of the Shaffler system.

Chapter 4

Implementation

This chapter describes the implementation of the Shaffler system. We begin by describing the implementation of the Minimum Viable Product (MVP), which was used to test the viability of the system as well as some basic ideas. Then we describe the full implementation of the system, which includes all of the functionalities described in Chapter 3.

4.1 Exploratory Prototype

We started by developing a prototype that included all the main functionalities of the Shaffler system. This section describes the implementation details of this MVP, as well as the drawbacks that were identified, and led to the development of new solutions. To implement Shaffler we used two programming languages, C and Python. To implement traffic modulation, we modified Tor version 0.4.7.13, which is written in the C language. For the implementation of our cover traffic technique we instead used Python 3.11.2.

4.1.1 Encoding and decoding delays

An important challenge we first had to address was how and where to modulate traffic using a modulation function to delay Tor circuits' packets. Importantly, we had to devise a mechanism that, on the one hand, would support generic modulation functions, and, on the other, would not be too intrusive on Tor, requiring dramatic changes to the Tor protocols. To tackle this challenge, we began from the observation that, in the Tor network, communications are made through Tor cells. These cells contain a field reserved for commands that inform relays of the kind of operation they must perform. Some notable examples of cell commands are the CREATE command, which informs an OR of the intention of extending the circuit to include it, and the RELAY command, which informs an OR that the cell must be forwarded to the next node in the circuit, until it reaches the edge of the circuit.

A promising candidate that we identified for the encoding of delays was the relatively large difference between the maximum number of values allowed by the size of the field and the number of commands implemented. This difference is due to the capability of 1 byte to encode $2^8 = 256$ numbers, of which

only 18 were assigned to commands. This meant that it was possible to create a total of $256 - 18 = 238$ new commands. Hypothetically, this available bit space could be used for encoding delays.

However, our objective was not to create a new command as that would require larger changes to the Tor protocols, but instead to expand the existing commands into as many variants as possible, so that each variant would encode a different delay. As such, we identified two possibilities: either (i) expand all commands into $238/18 \approx 13$ variants or (ii) select the most important types of cells to delay and only expand those into variants. By using command variants instead of, for example, general delay commands, we are able to maintain all protocols as they are, simply changing the way commands are parsed to check if a command value is within a range of values, instead of checking if it matches a specific value. Besides this, only the component responsible for applying the delays is required to parse and distinguish the variants of the commands, to obtain the delay to apply.

Although we wished to be able to delay the largest variety of traffic possible, we also wanted to support a wide variety of delays. Therefore, with this trade-off in mind, we decided to expand the command that we considered to be the most important, the RELAY command, allowing us to encode the 238 different delays mentioned above. To define these command variants we first reorganized the macros defined in the `or.h` file to be contiguous. Then we defined two new macros, `CELL_RELAY_DELAY_MIN` and `CELL_RELAY_DELAY_MAX`, and increased the value assigned to the macro `CELL_COMMAND_MAX_` to our new max command value, the `CELL_RELAY_DELAY_MAX`.

4.1.2 Applying delays

Another challenging aspect of the implementation of our system is the middle node's method of applying delays. Our implementation of this method can be separated into two components: (i) the method of self-identification of a node as the middle node and (ii) the method of creating a delay between a cell's arrival and departure. We will discuss the implementation of both these components, starting with component (i).

Implementing a method for nodes to identify themselves as a middle node would be trivial if nodes of a circuit were provided with information regarding their positions. However, in Tor, only clients have a clear view of the circuits created by themselves and thus, are able to know the positions of all nodes of a circuit; the nodes themselves are, by design, not given that information. Nevertheless, there is a method that, while not perfectly reliable, can be used by ORs to identify their position. The basis of this method is the ability to distinguish addresses of ORs from "normal" addresses, through the presence of those addresses in Tor's node list. This, together with the fact that each node of a circuit knows the addresses of both the previous and next peers, allows us to describe a middle node as a node whose addresses of both directions' peers are present in the node list. Listing 4.1 shows our implementation of this method. Most of the code serves only to obtain the previous and next nodes' addresses from a `circuit_t` structure; it is only at line 25 that `nodelist` is queried for those addresses.

However, as already mentioned, this method is not perfect, since the possibility for a Tor client to simultaneously be an OR, can lead to an entry node being mistakenly identified as a middle node. In


```

1 int
2 probably_middle_node_circ(circuit_t *circ)
3 {
4     if (!circ || circ->magic == ORIGIN_CIRCUIT_MAGIC)
5         return false;
6     or_circuit_t *or_circ = TO_OR_CIRCUIT(circ);
7     if (!or_circ)
8         return false;
9     return probably_middle_node_channels(or_circ->p_chan, circ->n_chan);
10 }
11
12 int
13 probably_middle_node_channels(channel_t *p_chan, channel_t *n_chan)
14 {
15     if (!p_chan || !n_chan)
16         return false;
17     channel_tls_t *p_chan_tls = BASE_CHAN_TO_TLS(p_chan);
18     channel_tls_t *n_chan_tls = BASE_CHAN_TO_TLS(n_chan);
19     if (!p_chan_tls || !n_chan_tls)
20         return false;
21     connection_t *p_conn = &(p_chan_tls->conn->base_);
22     connection_t *n_conn = &(n_chan_tls->conn->base_);
23     tor_addr_t prev_node_addr = p_conn->addr;
24     tor_addr_t next_node_addr = n_conn->addr;
25     return nodelist_probably_contains_address(&prev_node_addr) &&
26         nodelist_probably_contains_address(&next_node_addr);
27 }

```

Listing 4.1: Code of the function that attempts to identify whether a node is the middle node of a circuit or not.

such a case, there would be two possibilities. If the entry node does not forward the delay command to the middle node, assuming that it is no longer needed, the effectiveness of the traffic modulation technique could be threatened, as discussed in Section 3.3. If the entry node instead forwards the delay command to the middle node, both nodes would end up delaying traffic, resulting in an increase in latency of communications.

After a node identifies itself as the middle node using this method, it must then proceed to apply the delay itself. To implement this, we first attempted a very simple and naive approach. This approach consisted of calling the `nanosleep()` function (defined in the `time.h` library) immediately after decoding a delay command, to put the active thread to sleep for the decoded amount of time. This approach seemed to work well when simulating a small number of clients. However, simulating a larger number of clients revealed a major oversight: Tor is a single-threaded application. This meant that delays meant to be applied to one cell were instead essentially being applied to all cells received by the OR at around the same time.

One possibility to somewhat help solve this problem would be to use a multithreaded implementation of Tor, such as the one proposed by Engler et al. [43], that distributes circuits and connections between multiple threads. However, even such an approach would not eliminate but rather simply reduce the interference between cells, since a perfect solution would require one thread to be exclusively assigned to each active connection.

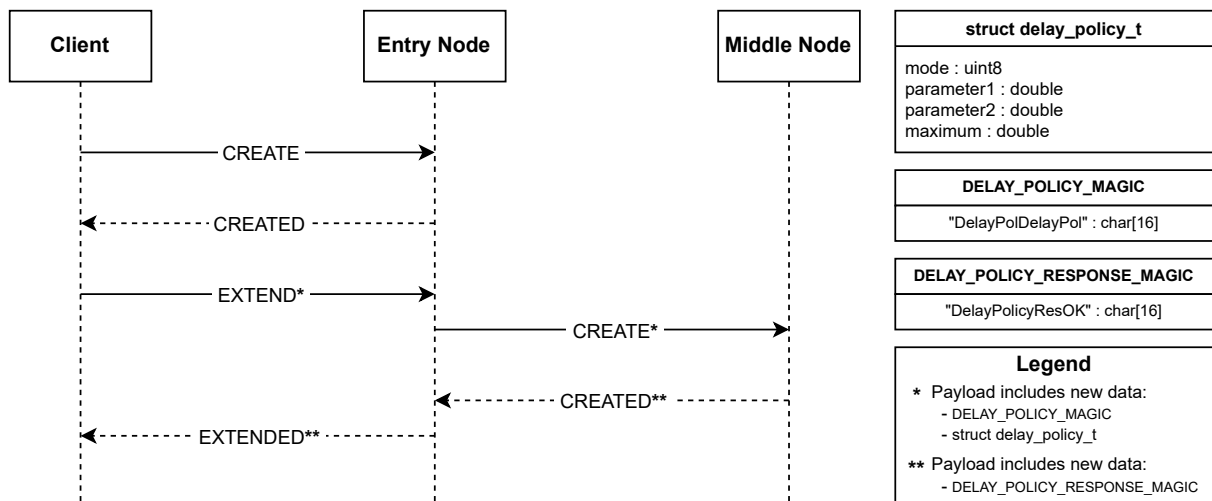


Figure 4.1: Diagram showing the additions made to the circuit creation protocol to allow the communication of delay policies.

4.2 Final Prototype

After testing and identifying issues in our MVP, we proceeded to research and implement new solutions that addressed those issues. These issues include the ones previously discussed, such as the limited amount of delays that can be encoded, the imperfect method of identifying nodes as the middle node, and the inadequacy of the `nanosleep()` function to delay cells. This section details the implementation and technical limitations of our final proposed solution for the Shaffler system.

4.2.1 Encoding and decoding delay policies

For the new version of the system, we decided to abandon the idea of having each client decide on a delay command for each cell. We opted, in turn, to allow clients to choose and customize the traffic modulation function that is used for their traffic. To make this possible, a method for the client to communicate its choices to the middle node was necessary. For this purpose, we implemented a method that piggybacks on Tor's circuit creation protocol, which can be seen in Figure 4.1, together with the added data structures.

Original protocol: The original protocol works as follows: When a client desires to create a new circuit, it sends a CREATE cell to the desired entry node and expects a CREATED cell as response, indicating that the entry node is now part of the circuit. After that, the client, to extend the circuit to a middle node, sends an EXTEND cell along the circuit, which is transformed into a CREATE cell by the entry node and sent to the desired middle node. The middle node then responds with a CREATED cell that is repackaged into an EXTENDED cell by the entry node and sent to the client, to inform him of the success in adding the middle node to the circuit.

These messages sent to add the middle node to the circuit provide an opportunity to send information towards the middle node. These four types of cell (CREATE, CREATED, EXTEND, and EXTENDED)

```

1 int
2 extend_cell_format(uint8_t *command_out, uint16_t *len_out,
3                   uint8_t *payload_out, const extend_cell_t *cell_in,
4                   delay_policy_t delay_policy)
5 {
6     (...)
7
8     /* SHAFFLER: Insert delay policy into extend cell payload */
9     if (get_options()->EnforceDelayPolicy || delay_policy.mode) {
10         if (*len_out + 16 + sizeof(delay_policy_t) > RELAY_PAYLOAD_SIZE) {
11             return -1;
12         }
13         memcpy(payload_out+*len_out, DELAY_POLICY_MAGIC, 16);
14         memcpy(payload_out+*len_out+16, &delay_policy, sizeof(delay_policy_t));
15         *len_out += 16 + sizeof(delay_policy_t);
16     }
17
18     return 0;
19 }

```

Listing 4.2: Code added to the function responsible for formatting an EXTEND cell.

do not fully occupy the 509 bytes of their payload, allowing us to use the remaining space to send a specification on how to modulate traffic to the middle node, which we call *delay policy*.

Delay policy: Regarding the *delay policy*, we define it as composed of four values: the delay mode, which specifies the modulation function to be used; the parameters to be used in the specified function; and the maximum value allowed for a delay, to force any delay higher than that value to be discarded and replaced. The delay mode is the only one of these values that is restricted, as it must correspond to: the “None” mode, which deactivates delays, the “Auto” mode, which tells the middle node to modulate traffic according to its own policy, or any of the modulation function modes.

Modified protocol: The modifications made to the protocol start when sending the EXTEND cell, whose objective is to add the middle node to the circuit. Listing 4.2 shows the necessary additions made to the `extend_cell_format()` function, which is responsible for packaging all the required information into an EXTEND cell. The client appends to the payload of the EXTEND cell a 16-byte magic number followed by the delay policy itself. Additionally, before doing that, as shown in line 10, we check if the payload truly has enough free space for all the data we want to add. If the check fails, we raise an error, which will result in the circuit creation being aborted.

Upon receiving this EXTEND cell, the entry node will check for the presence of the magic number, in the function `extend_cell_parse()`, and in case it is found, will copy the magic number and the delay policy that follows it to the payload of the CREATE cell, in a similar way to what was done previously for the EXTEND cell. Next, the entry node sends this new CREATE cell to the middle node, which will again check if the magic number is present in the payload and, if so, will proceed to extract and interpret the delay policy. Listing 4.3 shows the modifications made to the `create_cell_init()` function that is used to interpret an incoming CREATE cell.

After interpreting the contents of the cell, the middle node must process them. During this processing, the delay policy received in the CREATE cell is inserted into the corresponding `or_circuit_t` data

```

1 void
2 create_cell_init(create_cell_t *cell_out, uint8_t cell_type,
3                 uint16_t handshake_type, uint16_t handshake_len,
4                 const uint8_t *onionskin)
5 {
6     (...)
7
8     /* SHAFFLER: Parse the delay policy into the CREATE_CELL */
9     if (tor_memeq(onionskin + handshake_len, DELAY_POLICY_MAGIC, 16)) {
10         cell_out->delay_policy_is_set = 1;
11         memcpy(&cell_out->delay_policy, onionskin + handshake_len + 16, sizeof(delay_policy_t));
12     }
13     else {
14         cell_out->delay_policy_is_set = 0;
15         memset(&cell_out->delay_policy, 0, sizeof(delay_policy_t));
16     }
17 }

```

Listing 4.3: Code added to the function responsible for interpreting a CREATE cell.

```

1 static void
2 command_process_create_cell(cell_t *cell, channel_t *chan)
3 {
4     (...)
5     /* SHAFFLER: Copy delay policy from CREATE_CELL to CIRC */
6     circ->delay_policy_is_set = !get_options()->DisableDelays && create_cell->
7         delay_policy_is_set;
8     if (circ->delay_policy_is_set) {
9         log_info(LD_GENERAL, "[SHAFFLER][POLICY] Received delay policy");
10        memcpy(&circ->delay_policy, &create_cell->delay_policy, sizeof(delay_policy_t));
11    }
12    (...)
13 }

```

Listing 4.4: Code added to the function that processes a CREATE cell after it has been parsed.

structure, so that all communications from there onward may be delayed based on the specified delay policy. Listing 4.4 shows the additions we made to the `command_process_create_cell()` function to implement this processing and insert the delay policy into the circuit data structure, which is represented by the `circ` variable.

Furthermore, we implemented some `torrc` [44] options to provide some customization to the system. These options can be used in Tor’s configuration file, the `torrc` file, where a user can add and configure `torrc` options to specify many aspects of how the Tor process should behave. One of the options that we provide is called `DisableDelays`, which serves to allow ORs to fully reject delaying traffic and performing traffic modulation, which can be seen being checked in line 6. After checking the internal policy regarding the acceptance of traffic delaying and successfully obtaining the delay policy, the middle node then appends to the CREATED cell a different magic number, which will inform the entry node of the successful application of the delay policy. Similarly as before, the entry node checks the presence of the magic number and, if found, appends it again to the EXTENDED cell that is sent to the client. Finally, the client also checks the presence of the response’s magic number to confirm that the process was successful, and the delay policy was accepted.

Additionally, we provide the client with the option to enforce the usage of the delay policy, through another `torrc` option. When enabled, this option makes it so that a circuit is destroyed when the expected magic number confirming the usage of the delay policy is not recognized. Listing 4.5 shows the additions

```

1 int
2 created_cell_parse(created_cell_t *cell_out, const cell_t *cell_in)
3 {
4     (...)
5     /* SHAFFLER: Check the response magic number */
6     if (dprm_offset + 16 <= CELL_PAYLOAD_SIZE) {
7         cell_out->delay_policy_is_set = tor_memeq(cell_in->payload + dprm_offset,
8             DELAY_POLICY_RESPONSE_MAGIC, 16);
9     }
10    (...)
11 }
12
13 int
14 circuit_finish_handshake(origin_circuit_t *circ,
15     const created_cell_t *reply)
16 {
17     (...)
18     /* SHAFFLER: Check if circuit should be closed due to the delay policy failing to be set */
19     if (hop != circ->cpath && hop == circ->cpath->next &&
20         get_options()->EnforceDelayPolicy && !reply->delay_policy_is_set) {
21         log_warn(LD_CIRC, "Expected delay policy to be set by the middle node, but wasn't. Closing
22             .");
23         return -END_CIRC_REASON_TORPROTOCOL;
24     }
25     (...)
26 }

```

Listing 4.5: Code added to verify from the client side if a delay policy has been applied successfully.

made to the client to check the success of the process and to decide whether or not the circuit should be closed.

Lines 18 and 19 show the necessary checks to verify the results of the attempt to set the delay policy. First we verify that the current hop in the creation of the circuit is not the entry node, but instead the next node, which is the middle node. After these verifications, we can check if the `EnforceDelayPolicy` option is enabled, and finally check if the delay policy was applied successfully or not. If the delay policy is found to not have been successfully applied in the correct hop, when using the enforce option, then we raise an error and the circuit is closed.

Additional aspects: The reason we use magic numbers is to allow Shaffler to be compatible with the unmodified Tor, which is also the reason we piggyback on existing messages, rather than adding more types of messages to the circuit creation protocol. This way, if a middle node that is not running Shaffler receives a cell with a delay policy, it will simply ignore it and everything will proceed normally.

The delay policy and the option to enforce it are both implemented in the form of `torrc` [44] options. Additionally, each OR is also provided with options to configure its own delay policy, which will be used when a client requests the “Auto” mode. These options as well as their format and default values can be seen in Listing 4.6.

4.2.2 Applying delays

Due to the limitations discovered in our MVP’s method of applying delays, related to the inadequacy of the `nanosleep()` function for this purpose, we searched Tor’s code base for implemented utilities that could help us achieve our objective. Fortunately, we identified two promising structures already

```

1 struct or_options_t {
2     (...)
3     /* SHAFFLER: Client torrc options */
4     int EnforceDelayPolicy;
5     int DelayMode;
6     double DelayParam1;
7     double DelayParam2;
8     double DelayMax;
9     /* SHAFFLER: OR torrc options */
10    int DisableDelays;
11    int AutoDelayMode;
12    double AutoDelayParam1;
13    double AutoDelayParam2;
14    double AutoDelayMax;
15 };

```

Listing 4.6: Code used to add new torrc options to the system, allowing the configuration of a delay policy.

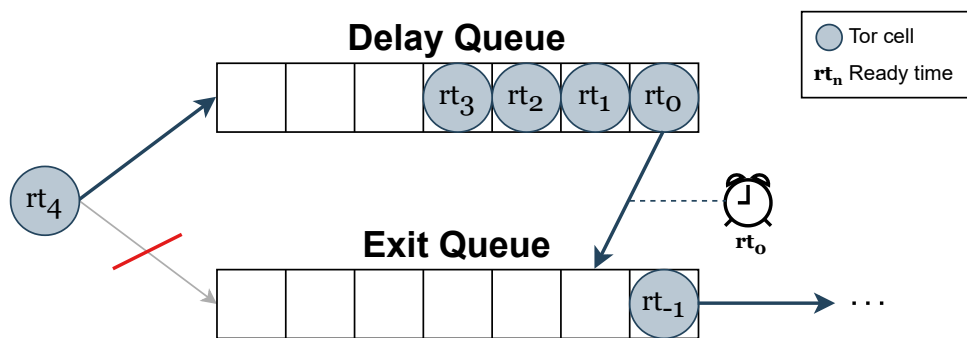


Figure 4.2: Diagram showing the process for delaying Tor cells. Includes the usage of a delay queue that stores cells ordered by ready time ($rt_n \leq rt_{n+1}$) and a single timer that waits for the first cell's ready time.

implemented in Tor that we decided to use: *queues* and *timers*. In the vanilla implementation of Tor, when a cell is prepared to be sent through a certain communication channel to a certain peer, it is placed in a queue. This queue, along with the queues of all other active channels, is then managed by a scheduler. Our implementation involves making some changes to this process, by intercepting the insertion of cells into these queues. Figure 4.2 shows a diagram representing this modified process.

We introduce an additional queue for each active channel, which we call *delay queue*. The purpose of this queue is to store all cells that must be delayed and whose delay time has not yet been completed. When a cell that must be delayed is received, it is inserted into the *delay queue* of the outgoing channel and is only moved to the same channel's cell queue upon completing their assigned delay time.

To update a cell when its assigned delay time has been completed, we use timers. These timers can be configured with callback functions that are called after a specified time has passed. This allows us to define a callback function that moves a given cell from one queue to the other, and with that, when a cell arrives, we can create a timer with the desired duration and with that function as callback.

To implement this whole behavior, several functions were developed to organize functionalities as best as possible and to keep the code readable. Listing 4.7 shows the `delay_or_append_cell()` function, a high-level function that is called where previously the `cell_queue_append()` function was called, to implement the alternative behavior of delaying cells instead of immediately appending them to the cell

```

1 void
2 delay_or_append_cell(packed_cell_t *copy, circuit_t *circ,
3                     cell_queue_t *queue, int direction)
4 {
5     if (circ->magic != OR_CIRCUIT_MAGIC) {
6         cell_queue_append(queue, copy);
7         return;
8     }
9     or_circuit_t *or_circ = TO_OR_CIRCUIT(circ);
10    if (or_circ->delay_policy_is_set && or_circ->delay_policy.mode != DELAY_MODE_NONE) {
11        cell_queue_t *delay_queue;
12        tor_timer_t *timer;
13        copy->ready_tv = get_ready_timeval(or_circ, direction);
14        if (direction == CELL_DIRECTION_OUT) {
15            delay_queue = &or_circ->n_delay_queue;
16            timer = or_circ->n_delay_timer;
17        }
18        else {
19            delay_queue = &or_circ->p_delay_queue;
20            timer = or_circ->p_delay_timer;
21        }
22        cell_queue_append(delay_queue, copy);
23        // If no timer is already set to update the delayed cells, set one up
24        if (timer == NULL) {
25            schedule_delay_timer(circ, direction);
26        }
27    }
28    else {
29        cell_queue_append(queue, copy);
30    }
31 }

```

Listing 4.7: Code of the function that intercepts arriving cells and directs them to the appropriate queue.

queue.

First, the function checks if the `circuit_t` belongs to an OR or a client. If it belongs to a client, the cell is appended to the cell queue, as before, since no cells should be delayed by clients. Otherwise, the function checks if a delay policy is set and if the delay mode is not “None”. If so, the function calculates the cell's ready time and inserts it into the delay queue. Additionally, if no timer is already set to update the delay queue, the function schedules a new timer to do so.

4.2.3 Modulating traffic

Our traffic modulation method relies on obtaining a value that we call *ready time* for each cell. When a new cell must be delayed, the first thing that is done is to generate a delay value for it, based on the delay policy. To generate this value, a function called `get_delay_timeval()` checks the delay policy and calls the corresponding function to generate the delay value. By separating the code in this way, we allow for the addition of new delay modes in the future, without requiring too many changes to the code base.

The *ready time* for cell n is then calculated by adding the delay chosen for this cell to the ready time of the last cell processed: $ready_time_n = ready_time_{n-1} + delay_n$. This ready time is stored along with each cell, so it is possible to keep track of when it is ready to be moved to the cell queue. Listing 4.8 shows the function `get_ready_timeval()` that is responsible for calculating the ready time of a cell.

This method provides an important characteristic of order preservation, which not only ensures the

```

1 struct timeval
2 get_ready_timeval(or_circuit_t *circ, int direction)
3 {
4     struct timeval previous_cell_tv, now_tv, delay_tv, ready_tv;
5     double ready; // in seconds
6
7     // Get last packet time
8     if (direction == CELL_DIRECTION_IN) previous_cell_tv = circ->p_last_ready_tv;
9     else previous_cell_tv = circ->n_last_ready_tv;
10
11     // If previous_cell_tv is too old, set it to now
12     gettimeofday(&now_tv, NULL);
13     if (previous_cell_tv.tv_sec < now_tv.tv_sec || (previous_cell_tv.tv_sec == now_tv.tv_sec &&
14         previous_cell_tv.tv_usec < now_tv.tv_usec)) {
15         previous_cell_tv = now_tv;
16     }
17
18     // Get delay
19     delay_tv = get_delay_timeval(circ, direction);
20
21     // Calculate ready time
22     timeradd(&previous_cell_tv, &delay_tv, &ready_tv);
23     ready = ready_tv.tv_sec + ready_tv.tv_usec / 1e6;
24
25     if (direction == CELL_DIRECTION_IN) circ->p_last_ready_tv = ready_tv;
26     else circ->n_last_ready_tv = ready_tv;
27
28     return ready_tv;
29 }

```

Listing 4.8: Code of the function that calculates the ready time for an incoming cell that must be delayed.

correct functioning of Tor but also allows us to check only the cell at the head of the delay queue for its ready time. It also allows us to use a single timer to transfer cells from the delay queue to the cell queue, reducing the resource usage of our solution. This timer is scheduled for the ready time of the cell found at the head of the delay queue. When the timer is triggered, all the cells in the queue that are ready are moved to the cell queue, in order, and the timer is rescheduled based on the new head of the queue. The reason we check the ready time of multiple cells, and not only the head of the queue, is due to the timer's limited resolution of one millisecond. This means that differences of less than 1 millisecond between cells' ready times might cause both to be ready when the timer is triggered. Unfortunately, this limited resolution also limits the possible delays by essentially rounding all delays to the nearest millisecond.

4.2.4 Creating cover traffic

As described in Section 3.4, all components responsible for Shaffler's cover traffic generation were conceived to run on the user's system. Since the goal is to ensure that all Tor traffic is directed towards the same guard node, all components rely on the same underlying Tor process to access the network. However, using the same Tor process in its default configuration is not sufficient to ensure that all circuits share the same guard. Therefore, in the implementation of Shaffler, the `torrc` file was edited so that the options `UseEntryGuards`, `NumEntryGuards` and `NumPrimaryGuards` were set to 1. It is relevant to note that although it is possible to achieve the same outcome by choosing a specific guard node in the `EntryNode` option, this approach could be problematic if the chosen node was unavailable.


```
1 user_pref("extensions.torlauncher.start_tor", false);
2 user_pref("extensions.torlauncher.control_port", 9051);
3 user_pref("network.proxy.socks_port", 9050);
```

Listing 4.9: Code showing the user preferences configured in Tor browser to use a custom Tor process.

The user's "regular client" can be any application just as long as it uses the aforementioned Tor process as a SOCKS proxy. In order to avoid DNS leaks, this application should be configured to use Tor as its DNS resolver as well. Shaffler's prototype used the Tor Browser v12.5.1 as the "regular client". To use the custom Tor process of the system, the default browser profile was changed to include the preferences shown in Listing 4.9.

The first preference prevents the Tor Browser from attempting to start its own Tor process (the default behaviour), while the second and third preferences specify the control and SOCKS ports used by the custom Tor process. The ports used in Shaffler's prototype matched Tor's default values of 9051 for the control port and 9050 for the SOCKS port but these could have been different as long as they matched the settings of the `torrc` configuration file. When using the Tor Browser, all DNS queries are made through Tor by default and no further configuration was needed in that regard.

Cover OS: The cover OS implementation consists of a Web Server Gateway Interface (WSGI) application written in Python 3.11.2 using the Flask v2.2.3 framework. It runs on a Gunicorn v20.1.0 WSGI server and uses Nginx v1.25.2 as a reverse proxy. In order to setup the OS, the `torrc` file was further changed to include the `HiddenServiceDir` and `HiddenServicePort` settings, making sure to redirect traffic to the Nginx port.

The Flask application can run in different modes and exposes a set of endpoints whose response is adjusted accordingly. The mode can be changed in a dedicated configuration file that allows for further customization. Four modes were created with the following behaviour:

- **Constant:** The OS responds to the `/` endpoint with a predetermined amount of randomly generated bytes. An option `Adjustable` was included to toggle the ability to adjust the size of the response. If this option is enabled, a request to the `/set/<int>` endpoint, where `int` specifies the desired response size, will result in a new default behavior.
- **Single Page:** The OS responds to requests on the `/` endpoint by serving a predetermined web page. Similarly to constant mode, the OS can be `Adjustable` or not. If this option is enabled, a request to the `/set/<string>` endpoint, where `string` is the name of the desired web page's main HTML file, will result in a new default behavior.
- **Multiple Pages:** The OS serves a web page from a selection of preloaded templates. Each request must specify which page to return. To this extent, the OS accepts requests to both the `/<int>` and `/<string>` endpoints, depending on whether one intends to identify the web page by its index in the templates directory or by the name of its main HTML file, respectively.
- **Dynamic:** The OS can respond with either randomly generated bytes or a web page. Each request

must specify the intended response. In dynamic mode the OS will respond to requests to the `/<int>` endpoint with `int` random bytes and to the `/<string>` endpoint with the respective web page.

This implementation results in an OS whose behavior can be easily customized. One can choose to run a flexible configuration, serving a large amount of different web pages, with a wide range of fingerprints, a more rigid configuration where responses consist of a fixed amount of random bytes or something in between.

Cover client: The cover client was implemented as a Python script that periodically sends requests to the OS. Its behavior is heavily dependent on the OS configuration since both the available endpoints and the responses themselves vary with it. The script relies on either the `requests` (v2.31.0) or `requests_html` (v0.10.0) python libraries, depending on whether the expected response is a collection of bytes or a web page that needs to be rendered. As is the case with every other component of the cover traffic infrastructure, the cover client accesses the Tor network via the custom Tor process mentioned above. Doing so implied configuring the cover client to use said process both as a SOCKS proxy and DNS resolver, making sure to not cache any of the OS's responses.

To orchestrate all cover traffic components, a manager was implemented using Go v1.20. The main function of the manager is to spawn and monitor all the components, ensuring that the "regular client" does not send traffic to the network unless adequate cover is being generated. To this extent, the components are spawned in a specific order. First is the Tor process, followed by the OS (both the Unicorn server and the Nginx reverse proxy), the cover client and, only if everything launched successfully, the "regular client". For each component, the manager launches a goroutine tasked with spawning the respective process. Once the process is correctly launched, the same goroutine signals `main` and proceeds to enter a monitoring mode. If, at any point, one of the processes crashes, its goroutine signals `main` to trigger a recovery sequence or a full restart of the system.

Changes required to allow testing using Shadow: The main method used to test the Shaffler system was the Shadow network simulator [45]. However, the then current implementation of Shadow lacked support for the kernel function `fork()`, which is essential for processes to be able to spawn other processes as children. This limitation made it impossible to execute the full implementation of the cover traffic technique using Shadow, since it relied not only on a manager process that would spawn all other required processes, but also on various other applications that also used the `fork()` function internally, such as Nginx and Unicorn.

For this reason, the implementation had to be adapted to simplify it as much as possible without changing its functionality. The Shadow-specific version used neither the WSGI server (Unicorn) nor the reverse proxy (Nginx), directly running a normal web server instead. Additionally, the manager, previously used to spawn both the client and the server-side processes, was abandoned. As a consequence, all instructions to launch the required processes had to be manually inserted into the simulation configuration.

```

1 - path: /usr/bin/python3
2   args: setup.py -c ../../../../conf/cover-config.json
3   start_time: 210
4   expected_final_state: {exited: 0}
5 - path: /usr/bin/python3
6   args: ./traffic_gen/os/app.py -p 8000
7   start_time: 245
8   expected_final_state: running
9 - path: /usr/bin/python3
10  args: ./traffic_gen/cover_client/client.py
11  start_time: 300
12  expected_final_state: running

```

Listing 4.10: Shadow configuration template used to add the three cover traffic processes to each client's configuration.

To insert these configurations into an already created simulation, we used a python script that copied a configuration template to each client's configuration. Listing 4.10 shows the template used, in YAML format, which includes the three processes previously mentioned.

Summary

This chapter described the implementation of the Shaffler system. We began by describing the implementation of the MVP, which was used to test the viability of the system as well as some basic ideas. Then we described the full implementation of the system, which includes all of the functionalities described in Chapter 3, such as the encoding and decoding of delay policies, the application of delays, and the creation of cover traffic. We also described the technical limitations of our solution, such as the limited resolution of the timers used to apply delays, and the modifications that we had to make to the cover traffic system to be able to simulate it on Shadow. Next, we present our evaluation of Shaffler.

Chapter 5

Evaluation

This chapter describes the evaluation of Shaffler. We begin by outlining our approach, which includes outlining our objectives, the metrics we collected and analyzed, and how we collected them. Then we present and discuss the effectiveness and performance results of multiple configurations of the two mechanisms provided by the Shaffler system, traffic modulation and cover traffic. Finally, we analyze the results, correlating and balancing the defenses provided against the performance impact.

Our evaluation will be presented using a gradual approach, studying first the effects of various configuration options of each technique in isolation, to refine our knowledge of each technique, before merging them and presenting the best results obtained for the Shaffler system, which confirm its effectiveness and viability in terms of performance.

5.1 Methodology

This section describes the methodology used to evaluate our system. We begin by presenting our evaluation's goals and procedure. Next, we list and describe the metrics used to evaluate both the defenses provided and the performance of the system. Lastly, we describe in detail our dataset collection method, which is an essential component for the ability to evaluate our system.

5.1.1 Goals and Procedure

The primary purpose of this evaluation is to assess the amount of protection that the Shaffler system can offer against traffic correlation attacks while maintaining a satisfactory level of performance. Additionally, to support this primary objective, we also aim to investigate various possible configurations of the two techniques employed by Shaffler, both independently and in conjunction, so that we can understand the effects of each customizable option, and in the end find the best configurations that the system can provide.

To measure resistance to traffic correlation attacks, we rely on DeepCoFFEA [21], the state-of-the-art traffic correlation attack. This attack requires that a machine learning model be trained with datasets of artificially correlated flows. Although it would be possible to simply use a model trained with real Tor

datasets to test our system, it would likely lead to unrealistic results of efficacy of our system, due to the substantial differences between the train and test datasets. As such, we decided to train specialized models for each configuration of our system. By combining this with the experimentation of different attack parameters, we ensure that we get closer to the maximum accuracy of the attack.

Regarding performance, we rely on the Shadow simulator [45] (v3.0.0), together with several supporting tools, such as tornettools v2.0.0 [46], OnionTrace v1.0.0 [47], and TGen v1.1.1 [48]. From all of these tools, the only one we directly use is tornettools, as it serves as a simple interface for all other tools by allowing us to generate realistic simulation configurations, to simulate networks with a fractional size of the real Tor network based on real data. Additionally, it provides tools to easily obtain various performance metrics during simulations and even export them to graphs. Furthermore, in addition to the performance results, we also use these simulations and tools to obtain the datasets mentioned above.

5.1.2 Metrics

To evaluate Shaffler, we required metrics that would allow us to measure the effectiveness of the defenses provided by the system, as well as another set of metrics that would allow us to measure the impact on the performance of the system. In this section, we describe the metrics used for both purposes.

Correlation metrics: By default, the DeepCoFFEA attack outputs, for each similarity threshold used to classify flows as correlated or uncorrelated, the following metrics:

- True Positive Rate (TPR) – the percentage of all correlated flows that are correctly identified as correlated. Calculated using Equation 5.1.

$$TPR = \frac{\#TP}{\#TP + \#FN} \quad (5.1)$$

- False Positive Rate (FPR) – the percentage of all uncorrelated flows that are wrongly identified as correlated. Calculated using Equation 5.2.

$$FPR = \frac{\#FP}{\#FP + \#TN} \quad (5.2)$$

- Bayesian Detection Rate (BDR) – the percentage of all flows identified as correlated that are truly correlated. Calculated using Equation 5.3, where $P(P)$ is the probability that a flow pair is correlated and $P(N)$ is the probability that a flow pair is not correlated.

$$BDR = \frac{TPR \times P(P)}{TPR \times P(P) + FPR \times P(N)} \quad (5.3)$$

However, these metrics are not very practical to compare the results obtained with different configurations of Shaffler, since it is hard to compare the three metrics for each configuration, while also taking into account the various similarity thresholds tested. These similarity thresholds, in our scenario, are

the values that determine how similar two flows (ingress and egress) must be for them to be considered related. This means that, for example, a higher similarity threshold will lead to a higher TPR, however, it will also come at the cost of a higher FPR and a lower BDR. As such, we require metrics that allow us to pinpoint the similarity threshold for which these trade-offs are optimized so that we can compare results more accurately and fairly. As such, we decided to calculate and use F1-score and P4. Both are compound metrics that can be calculated using the number of True and False positives (#TP and #FP) and the number of True and False negatives (#TN and #FN) of a classification problem's outcome.

$$Precision = \frac{\#TP}{\#TP + \#FP} \quad (5.4)$$

$$Recall = \frac{\#TP}{\#TP + \#FN} \quad (5.5)$$

$$F1score = \frac{2}{\frac{1}{Precision} + \frac{1}{Recall}} \quad (5.6)$$

The F1-score metric is based on both *Precision* (given by Equation 5.4) and *Recall* (given by Equation 5.5). Precision is indicative of the amount of “positive” classifications that were actually correct, while the recall, so far referred to as TPR, is indicative of the amount of positive samples actually classified as such. The F1-score is defined as the harmonic mean of precision and recall and is given by Equation 5.6. F1-score gives great importance to a classifier's behavior with respect to the “positive” samples and can, to some extent, neglect performance with respect to the “negative” ones.

$$Specificity = \frac{\#TN}{\#TN + \#FP} \quad (5.7)$$

$$NPV = \frac{\#TN}{\#TN + \#FN} \quad (5.8)$$

$$P4 = \frac{4}{\frac{1}{Precision} + \frac{1}{Recall} + \frac{1}{Specificity} + \frac{1}{NPV}} \quad (5.9)$$

Because of these characteristics of the F1-score we sought a second metric that would not base itself so heavily on the “positives”, trying instead to be as balanced as possible. This decision stems from the objective of Shaffler to not only reduce the amount of “true positives” but also reduce the amount of “true negatives”. It so happens that a new metric, P4, was recently proposed with the objective of addressing the shortcomings of the F1-score [49]. In addition to Precision and Recall, P4 is also based on *Specificity* (given by Equation 5.7) and *Negative Predictive Value (NPV)* (given by Equation 5.8). Specificity reflects the amount of “negative” samples that are correctly classified as such (the “negative” equivalent of Recall) while negative predictive value is indicative of the amount of correct predictions out of all “negative” classifications (the “negative” equivalent of precision). Similarly to the F1-score, P4 also consists of a harmonic mean (given by Equation 5.9), this time involving all four components – Precision, Recall, Specificity and Negative Predictive Value.

Performance metrics: To evaluate differences in performance, we use the metrics that are provided by `tornettools`. By default, simulations generated by `tornettools` include 100 *perfclients*, which serve

to perform several performance measurements throughout the simulation. Of all the metrics that we obtained, we found the following to be the most useful to compare different configurations of Shaffler.

- Transfer Time – the time it takes to transfer an amount of data N from a server.
($N \in \{50KiB, 1MiB, 5MiB\}$)
- Error Rate – the percentage of data streams that end in failure, for example, due to timeouts.

5.1.3 Dataset Collection

For the purpose of evaluating the effects of the Shaffler system on the DeepCoFFEA attack’s ability to correlate traffic, we required the development of a way to create compatible datasets. As previously mentioned, to generate and collect those datasets, we use the Shadow network simulator, which allows us to simulate a network running our modified version of Tor. Additionally, we use `torntools` to generate a realistic network configuration with a fractional size of the real Tor network, which we modified to as closely as possible resemble the data set collection method described by Oh et al. [21]. Due to limitations in the resources available to perform these simulations, we decided to simulate networks with 0.5% the size of the real Tor network.

The required datasets are composed of pairs of files, where one file contains the capture of a flow at the ingress of the circuit and the other contains the capture of the same flow at the egress of the circuit. More specifically, these files contain timing and size information on all packets in the relevant flow.

To make the collection of these datasets possible, it was necessary to overcome two main challenges. The first of those challenges was how to capture the timestamps and sizes of packets within the simulation. Fortunately, Shadow provides a configuration option for simulated hosts that enables the capture of all their network packets in PCAP format. All captured packets had their Transmission Control Protocol (TCP) payloads intentionally discarded, as the headers already provide us with both the timestamp and the size of the packets. Immediately discarding this unnecessary information was also possible thanks to another configuration option provided by Shadow that allowed us to limit the capture to 24 bytes per packet, which is the minimum amount necessary to capture both the Internet Protocol (IP) and TCP headers. This was essential, as storing full captures of all simulated packets would quickly become infeasible when collecting datasets of the size required by DeepCoFFEA. We activated both of these configuration options on the client hosts to capture the ingress flows and on the server hosts to capture the egress flows. This deviates slightly from the method described by Oh et al. [21] that consists of using a proxy server to capture the egress flows. This change was made possible due to the ability to set up and monitor the servers, which allows us to capture the flows without the need for a proxy server.

The second challenge was how to collect flows in an efficient manner while remaining able to keep track of the pairings of captures. Although it would be possible to simulate and capture each flow of the dataset one at a time, it would be very inefficient. However, by simulating and capturing multiple flows at the same time, it becomes harder to keep track of which flow each packet belongs to. To overcome this difficulty, we designed a method consisting of running N clients, each with a unique identifier i , and configuring all servers to listen on a range of N ports, one for each client. Clients then simply need to

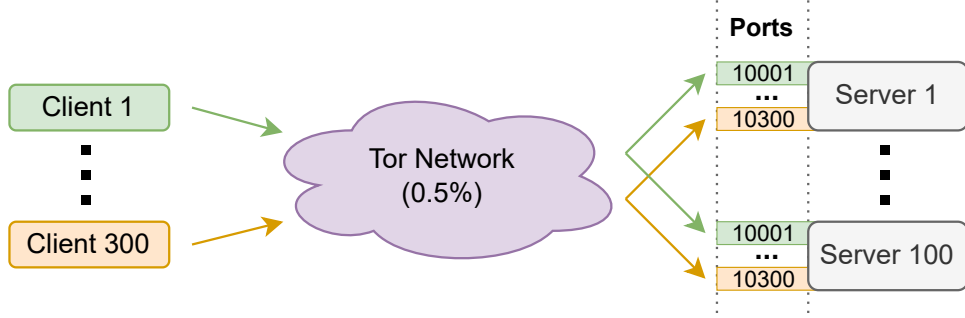


Figure 5.1: Simulation configuration used for the dataset collection.

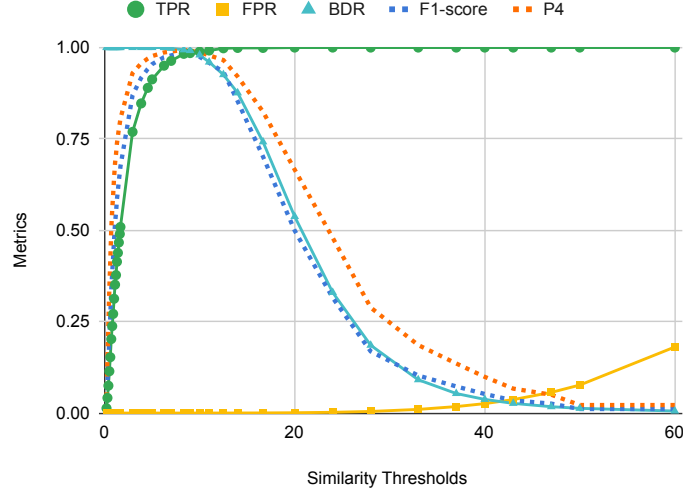
calculate the destination port for all their flows using the formula: $port_{c_i} = 10000 + i$, where $i \in [1, N]$. This setup is depicted in Figure 5.1, with $N = 300$. This number of clients results in approximately 23000 pairs of flows collected when simulated for a total time of 2 hours, where each pair spans 1 minute. Additionally, the total simulation time of 2 hours included a one-time setup period of 5 minutes, as well as 30 seconds of wait time between each flow generated by the clients.

Lastly, after running the simulation and obtaining the packet captures for all the flows, we needed to extract and parse those captures. For this purpose, we developed a Python [50] script that parses the simulation results using dpkt [51] and creates the dataset in the format used by DeepCoFFEA. The parsing script is divided into two phases. The first phase is the staging phase, where the simulation configuration file is parsed to extract and organize all the information related to each flow, such as start time, client's identifier, and corresponding destination port. This information is organized into a JSON file together with the identifiers generated for each flow and in a manner that allows the following phase to be as efficient as possible, such as sorting all flows based on start time, which prevents having to, in the worst case, traverse the whole list of flows to find the flow that a packet belongs to. The second phase is the parsing phase that is performed separately for each client and server, where all packets belonging to relevant flows are identified and their timestamps and sizes are added to the corresponding flow's output file.

The output format consists of two directories: the *inflow* directory that includes flows captured at the ingress of the circuit and the *outflow* directory that includes flows captured at the egress of the circuit. As such, a pair of flows is made up of two files with the same name, one in each of the two directories.

5.2 Traffic Modulation Results

In this section, we present and discuss the experimental results of various traffic modulation configurations of Shaffler. We start by presenting the Tor vanilla results, that is, the results of unmodified Tor, obtained through our experimental methodology. Then, we present a series of studies of various characteristics of the configuration options provided for traffic modulation, with the objective of understanding the effects of these options on the accuracy of the DeepCoFFEA attack and on the performance of the system.



$$Max(F1-score) = 98.73\%; Max(P4) = 99.36\%$$

Figure 5.2: Graph showing the results of the DeepCoFFEA attack for varying similarity thresholds, when using the “Vanilla” dataset.

Although the results of the traffic modulation technique shown in this section are less than desirable, it is important to perform this isolated evaluation to understand and compare the effects that the various configuration options may have on both the accuracy of the attack and the performance of the system. Only in Section 5.4 will we present the evaluation of traffic modulation and cover traffic generation combined, as well as the best results obtained for the Shaffler system, which confirm its effectiveness and viability in terms of performance.

5.2.1 Tor Vanilla Results

To enable the evaluation of the effects of traffic modulation on the accuracy of the DeepCoFFEA attack, we need Tor vanilla results to compare with. For this purpose, we used Tor version 0.4.7.13, instead of Shaffler, together with the collection method described above, to generate a dataset that we could run the DeepCoFFEA attack on.

Figure 5.2 shows a plot of the metrics TPR, FPR, BDR, F1-score and P4 obtained for varying similarity thresholds. The X-axis of the graph displays a selection of similarity threshold values, which, as mentioned previously, determine how similar two flows must be to be considered related. Meanwhile, the Y-axis reflects the metric values, spanning from 0 to 1. When the similarity threshold is set to 60, the BDR, F1-score and P4 values are close to 0, due to the large number of false positives that result from such a high threshold. However, we can see that a threshold of around 10 seems to be enough to achieve $TPR \approx 1.00$, making it meaningless to further increase the threshold, as it will only increase the FPR. This also explains why the BDR, F1-score and P4 metrics follow curves that achieve a maximum value for a similarity threshold close to this one, since the increase in number of false positives affects negatively all of these metrics.

By comparing these results with those obtained by Oh et al. [21] using a real traffic dataset, we see

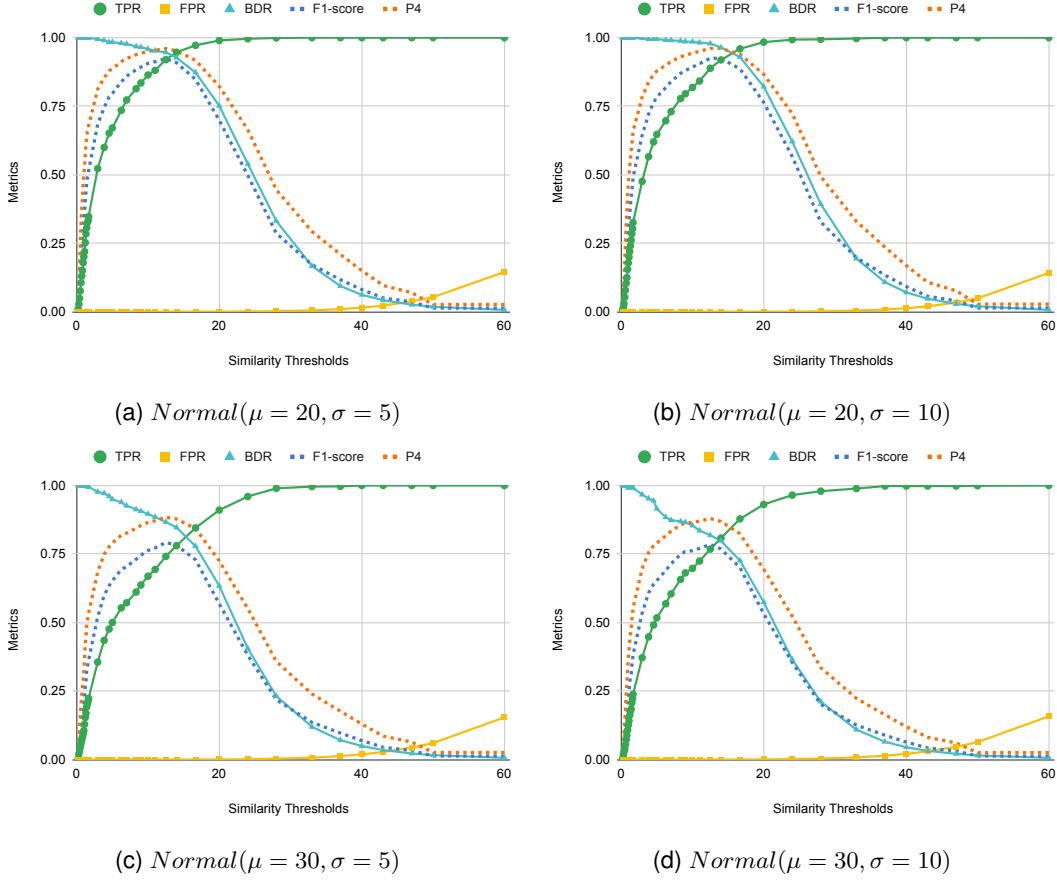


Figure 5.3: Graphs showing the results of the DeepCoFFEA attack for varying similarity thresholds, when using the normal distribution with different parameters.

that the attack was more accurate using our dataset, achieving, for a certain threshold, $TPR \approx 99\%$ and $BDR \approx 95\%$ simultaneously. This shows a difference of approximately 19% over the highest TPR obtained by Oh et al. [21] simultaneously with $BDR \geq 95\%$, which is approximately 80%. This increased accuracy is likely due to our method of collecting datasets, which uses a simulated network, instead of a real network. Although not optimal, it is acceptable, since our objective is to use these results as a baseline to compare all other results.

5.2.2 Mean and Standard Deviation of Delays

To study the effects of increasing the mean and the Standard Deviation (SD) of delay sizes, we naturally chose to use a distribution that is defined by those parameters, the normal distribution. We selected two values for each of the parameters and tested the four possible combinations of those parameters. The values we chose for the mean (μ) were 20 and 30 and the values chosen for the SD (σ) were 5 and 10.

Effectiveness Results: Figure 5.3 and Table 5.1 show the results obtained by running the DeepCoFFEA attack with its default parameters on four datasets created with each of these four variants.

As expected, the mean parameter appears to affect the efficacy of the DeepCoFFEA attack the most, with an increase of 10 milliseconds significantly decreasing the TPR and BDR curves and increasing the

Modulation Function	Max F1-score	Max P4
<i>Normal</i> ($\mu = 20, \sigma = 5$)	92.37%	96.03%
<i>Normal</i> ($\mu = 20, \sigma = 10$)	92.51%	96.11%
<i>Normal</i> ($\mu = 30, \sigma = 5$)	79.21%	88.40%
<i>Normal</i> ($\mu = 30, \sigma = 10$)	78.50%	87.95%

Table 5.1: Maximum F1-score and P4 values obtained for each of the four tested configurations of the Normal distribution modulation.

FPR curve. Additionally, a significant effect can also be observed on the F1-score and P4 values, which decrease by approximately 14% and 8%, respectively. These results suggest that the accuracy of the attack is reduced as the mean size of the delays increases.

However, it is likely to also be a result of the small window size used as the default parameter for the DeepCoFFEA attack. This parameter specifies the duration of each decision window in seconds, and it is reasonable to assume that longer delays reduce the likelihood of a flow being completely contained within a window. Therefore, further in this section, we present experiments to confirm this hypothesis and determine if a larger window size can indeed reduce the large delay’s effectiveness.

Also interesting to note is the minimal effect that the SD appears to have. In one case, a higher value of SD slightly increases the accuracy of the attack, increasing the F1-score by 0.14% and the P4 by 0.08%, and in the other case it decreases it, although by a larger amount, decreasing the F1-score by 0.71% and the P4 by 0.45%. These minimal and inconsistent changes of less than 1% do not allow us to conclude with certainty what effect SD has on the accuracy of the attack. However, they do suggest the possibility of an increase in SD resulting in a slight increase in accuracy, which is likely due to the wider range of delays from which it is possible to select.

Performance Results: To study the effects on performance of increasing the mean and SD, we also collected some performance results for the same four sets of parameters. The most notable of these results are shown in Figure 5.4.

Regarding transfer times, we can see that they increase significantly as the mean increases, as expected. For example, while the transfer time of 50KiB for both functions with mean $\mu = 20ms$ varies between 2 and 3.25 seconds, for both functions with mean $\mu = 30ms$ it instead varies between 3 and 4.25 seconds. However, SD does not appear to have such a significant effect. The results of the two sets of parameters with mean 30, suggest a negligible effect, whereas the results with mean 20 seem to suggest some minor but relevant effect. However, a detail might explain this observation, which is the fact that the normal distribution with $\mu = 20$ and $\sigma = 10$ results in a probability of 2.3% of selecting a delay lower than 0. When such a thing happens, because negative delays are not allowed, Shaffler discards them and selects a new value, resulting in a minor increase in the mean delay size. With this in mind, we expect the transfer times obtained with $\mu = 20$ and $\sigma = 10$ to be slightly higher than the times obtained with $\mu = 20$ and $\sigma = 5$, which can be observed in our results. As such, this suggests that SD has negligible or no effect on transfer times.

It is also important to note how the results of the normal distribution with $\mu = 30$ are missing from the

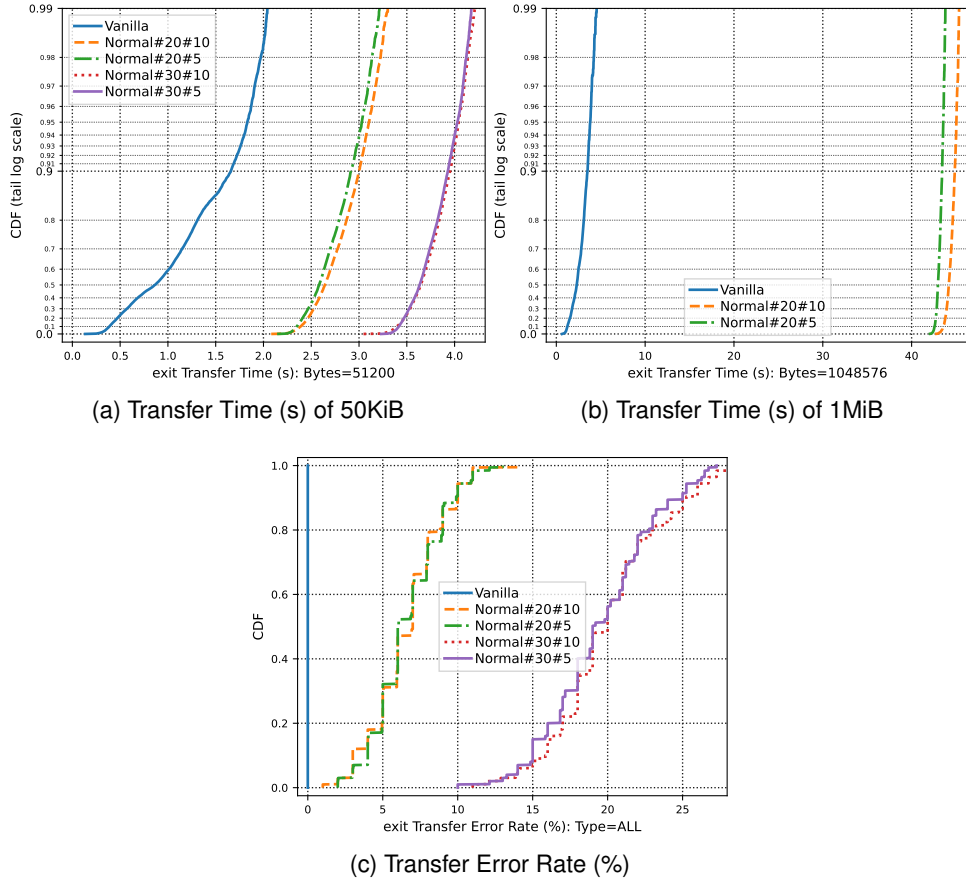


Figure 5.4: Performance results obtained by `toronettools` for all four tested configurations of the Normal distribution.

1MiB transfer times graph. This is due to the tool used, `toronettools` [46], defining the time limit for the transfer of 1MiB to be 60 seconds. This, together with the higher error rate seen in Figure 5.4 (c), allows us to conclude that when using a normal distribution with $\mu = 30$, the 1MiB transfer times are always higher than 60 seconds.

To interpret these results in a practical manner, we can relate them to the median weight of a web-page, which is approximately 2MB, according to Web Almanac 2022 [52], a report published by the HTTP Archive on the state of the web in 2022. As such, we can interpret our results as taking more than 1 minute to load half of a web page of median size, when performing traffic modulation with a mean delay size of 30 milliseconds. We can also interpret these results by comparing them with the results obtained when no traffic modulation is performed (Vanilla) to quantify the impact on performance. Based on the maximum amount of time taken to transfer 1MiB in the Vanilla simulation being 5 seconds, we can calculate the impact on the transfer time to be at least a $12\times$ increase.

Although the reduction in attack accuracy obtained may be significant, the impact on system performance shown by these results suggests that the usage of the traffic modulation technique by itself may not be a viable way to defend against traffic correlation, as it affects usability in an unacceptable way. However, the possibility remains that the traffic modulation technique may prove to be an effective and relevant addition to the system, when both techniques are combined, the results of which we will show in Section 5.4.

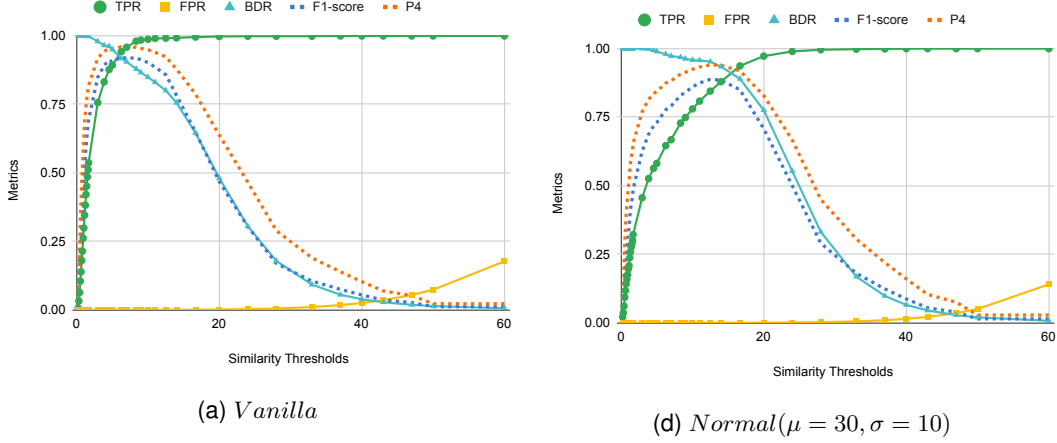


Figure 5.5: Graphs showing the effects on accuracy of increasing the window size of the DeepCoFFEA attack to 7 seconds, for the “Vanilla” dataset and the $Normal(\mu = 30, \sigma = 10)$ dataset.

5.2.3 Attack parameters

The DeepCoFFEA attack has three main parameters: the number of decision windows to use, the size in seconds of each window, and a value specifying how many seconds each window overlaps each other. All accuracy results shown previously were obtained with the default configuration: 11 windows that span 5 seconds each, with overlaps of 3 seconds. Since the usage of traffic modulation essentially expands a flow over time, increasing the distance in time between a packet’s entry and its exit from the circuit, we hypothesized that larger windows of decision would be more effective. As such, to verify our hypothesis, we trained and tested DeepCoFFEA with multiple datasets, including the same datasets used to obtain the results previously shown in Figure 5.3, but configured it to use 11 windows spanning 7 seconds each and overlapping by 4 seconds. Figure 5.5 shows the results obtained by running the attack with the modified parameters on both the “Vanilla” dataset and the $Normal(\mu = 30, \sigma = 10)$ dataset.

Comparing these results with those obtained previously, shown in Figures 5.3 and 5.2, we observed that the accuracy of the attack improved significantly for each of the datasets in which traffic modulation was performed, but reduced for the Vanilla dataset. Specifically, the Tor vanilla results revealed a decrease in F1-score of 6.42% and in P4 of 3.36%, while the $Normal(\mu = 30, \sigma = 10)$ revealed an increase in F1-score of 10.29% and in P4 of 6.11%, as shown in Table 5.2. Therefore, we can confidently conclude that our hypothesis is confirmed and that the DeepCoFFEA attack can be reconfigured to successfully reduce the protection provided by our traffic modulation technique.

However, these results are not enough to conclude that there is a linear trend of increased window sizes resulting in better accuracy. To verify this, we also tested the $Normal(\mu = 30, \sigma = 10)$ dataset with the configuration of 11 windows that span 9 seconds each, with overlaps of 5 seconds. The results of this experiment can be seen in Figure 5.6.

Comparing the F1-score and P4 values obtained for each of the tested sets of DeepCoFFEA attack parameters, summarized in Table 5.2, we can see that although increasing window sizes and overlaps appears to improve the accuracy of the attack for traffic modulation datasets, there is a limit to this improvement, as at some point the accuracy starts to decrease again, rather than continuing to increase.

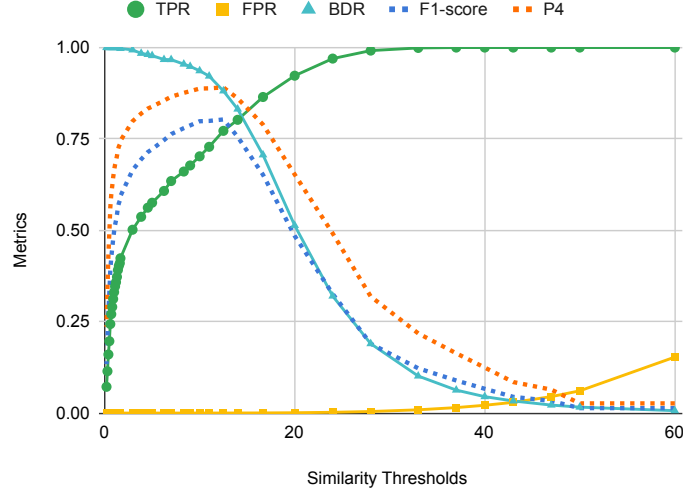


Figure 5.6: Graph showing the effects on accuracy of increasing the window size of the DeepCoFFEA attack even further to 9 seconds, for the $Normal(\mu = 30, \sigma = 10)$ dataset.

Modulation Function	Window Size	Window Overlap	Max F1-score	Max P4
<i>Vanilla</i>	5s	3s	98.73%	99.36%
<i>Vanilla</i>	7s	4s	92.31%	96.00%
$Normal(\mu = 30, \sigma = 10)$	5s	3s	78.50%	87.95%
$Normal(\mu = 30, \sigma = 10)$	7s	4s	88.79%	94.06%
$Normal(\mu = 30, \sigma = 10)$	9s	5s	80.33%	89.09%

Table 5.2: Maximum F1-score and P4 values obtained for increasing window sizes and overlaps of the DeepCoFFEA attack, when using the Normal distribution modulation.

Specifically, we see an increase of 10.29% in F1-score and of 6.11% in P4 for the first increase in window size, and a decrease of 8.46% and 4.97% in F1-score and P4, respectively, for the second increase in window size. This leads us to conclude that the parameters used with the attack must be carefully tuned to achieve its best results of accuracy. Furthermore, combined with the observation that the “Vanilla” results with increased window sizes are worse than the results with the default window size, we can conclude that the best configuration of the DeepCoFFEA attack is highly dependent on the modulation function used and its parameters.

5.2.4 Other Modulation Functions

In addition to the modulation function based on the normal distribution, which we used for simplicity of the analysis, we tested all other modulation functions implemented in Shaffler, which are listed in Section 4.2. With the objective of performing as complete a search as possible, we decided to test and evaluate a minimum of 3 sets of parameters per function. On top of that, we also experimented by varying the parameters of DeepCoFFEA, as mentioned previously, to make sure that promising looking results were not easily defeatable.

The results of all of these experiments revealed a general trend that confirmed our previous findings. That is, the effectiveness of the DeepCoFFEA attack is highly dependent on the mean size of the delays,

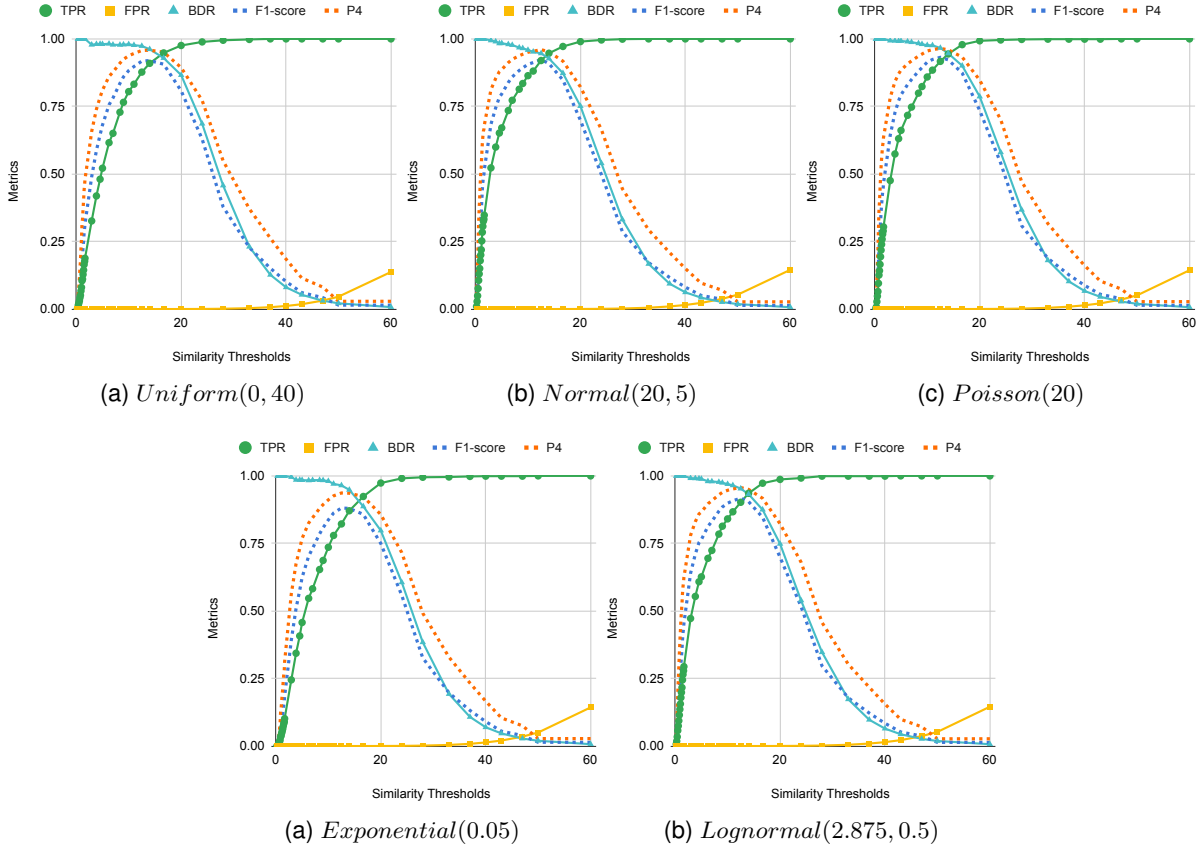


Figure 5.7: Graphs showing the results of the DeepCoFFEA attack for varying similarity thresholds, when using each modulation function with parameters configured to result in a mean of delays equal to 20 milliseconds.

with higher means resulting in lower accuracy, and modulation functions with similar means resulting in similar accuracy results. This is likely due to packets never being shuffled, instead always maintaining the same order, which allows the attack to easily identify patterns by combining volume analysis with timing analysis. Additionally, we found that the performance impact is also highly dependent on the mean size of the delays, which was highly expected, since the delays were limited with an upper bound of 100ms, which is not a large enough value to singlehandedly disrupt the performance of any communication. As such, as long as the mean size of the delays is not too high, smaller delays selected for other packets will highly likely compensate for the larger delays, resulting in a similar performance to the usage of a function that, for example, selects a constant delay of the same size as that mean size.

To further verify these findings, we performed an additional experiment, where we ran a simulation for each of the implemented modulation functions, configuring them all with the corresponding parameters that lead to a delay mean of 20 milliseconds. For some of these functions, where the mean is not obvious given the parameters, such as the Lognormal and the Exponential functions, we wrote a Python script using Scipy [53] to generate 100000 values with the corresponding distribution, and then calculate the mean of those values.

Figure 5.7 and Table 5.3 show the results obtained by running the DeepCoFFEA attack with the default parameters on all the generated datasets for this experiment.

These results confirm our previous findings, that the mean size of the delays is the most important

Modulation Function	Max F1-score	Max P4
<i>Vanilla</i>	98.73%	99.36%
<i>Uniform</i> ($\min = 0, \max = 40$)	91.98%	95.82%
<i>Normal</i> ($\mu = 20, \sigma = 5$)	92.37%	96.03%
<i>Poisson</i> ($\lambda = 20$)	93.17%	96.46%
<i>Exponential</i> ($\lambda = 0.05$)	88.11%	93.67%
<i>Lognormal</i> ($\mu = 2.875, \sigma = 0.5$)	91.73%	95.68%

Table 5.3: Maximum F1-score and P4 values obtained for modulation functions configured to result in a mean of delays equal to 20 milliseconds.

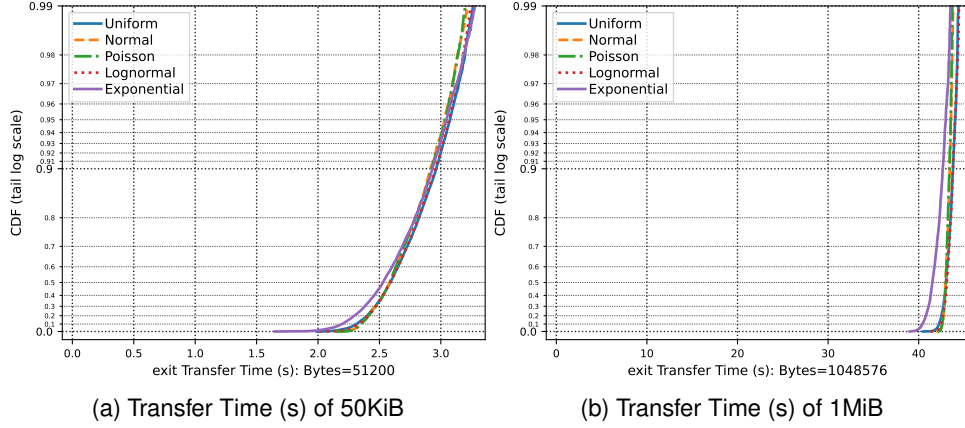


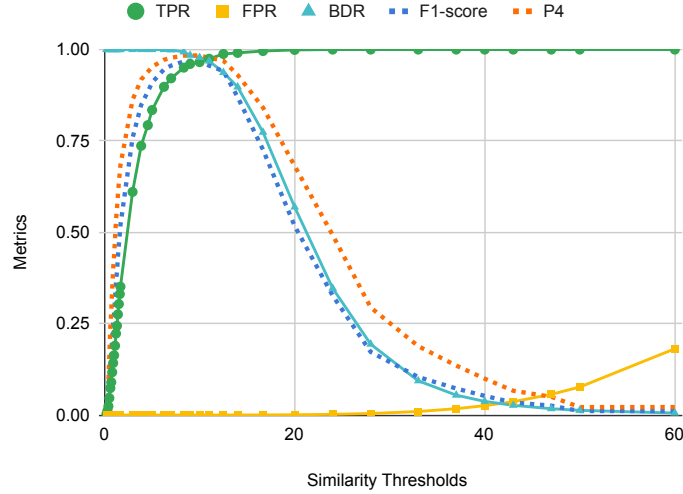
Figure 5.8: Performance results obtained by `tornettools` for each modulation function configured to have a mean of delays equal to 20 milliseconds.

aspect of a modulation function, as the accuracy of the attack is similar for all modulation functions with similar means, with the exception of the Exponential function, which is slightly less accurate. This difference is not significant enough for us to be able to conclude that the Exponential function is more effective than the other functions, however, it is interesting to note that, in general, functions with a wider range of possible delay values, such as the *Exponential*($\lambda = 0.05$) function, that can select delays as high as 100ms, appear to be more effective at reducing the accuracy of the attack than functions with a narrower range of possible delay values, such as the *Normal*($\mu = 20, \sigma = 5$), which will virtually never select a delay higher than 45ms. Specifically, we observe a difference of 4.26% and 2.36% in F1-score and P4, respectively, between the *Exponential*($\lambda = 0.05$) and the *Normal*($\mu = 20, \sigma = 5$) functions.

Furthermore, regarding the performance impact, which can be seen in Figure 5.8, we can see that the results of all modulation functions with the same mean are all extremely similar, which is expected and consistent with our previous findings.

5.3 Cover Traffic Results

In this section, we present and discuss the experimental results of various configurations of Shaffler. We start by presenting the new Tor vanilla results we had to obtain due to resource limitations. Then, we present and discuss the effects of varying the amount of cover traffic generated per request, as well as the effects of varying the amount of time between requests. Finally, we present the results of



$$\text{Max}(F1\text{-score}) = 96.95\%; \text{Max}(P4) = 98.45\%$$

Figure 5.9: Graph showing the results of the DeepCoFFEA attack for varying similarity thresholds, when using a “Vanilla” dataset generated by simulating 200 clients.

an experiment where we tested the effects of using multithreading to achieve concurrent cover traffic requests.

5.3.1 Tor Vanilla Results

Due to the additional processes required to generate cover traffic exhausting our available resources, we had to reduce the number of clients that capture traffic for the dataset creation to 200, instead of the 300 used to obtain the results shown in Section 5.2. As such, in order to make fair comparisons, we had to obtain new Tor vanilla results, which we did by running the DeepCoFFEA attack on a dataset obtained through the previously described collection method, but with only 200 clients, using Tor version 0.4.7.13.

Figure 5.9 shows a plot of the new TPR, FPR, and BDR obtained for various similarity thresholds.

Comparing these results with those shown in Subsection 5.2.1, we can see that the accuracy of the attack decreased slightly, with the maximum F1-score being lower by 1.78% and the maximum P4 being lower by 0.91%. This is expected, as the DeepCoFFEA attack benefits from having more data to train the model with. However, the difference does not seem significant enough to interfere with our analysis.

5.3.2 Amount of Traffic per Request

We started by analyzing the effects of the amount of cover traffic generated on the accuracy of the DeepCoFFEA attack. For this purpose, we selected 15 seconds as the period for the requests, i.e. the amount of time between each cover traffic request, as it seemed reasonable. We then experimented with three different amounts of cover traffic: 10KB, 100KB, and 1MB. Figure 5.10 and Table 5.4 show the results obtained by running the DeepCoFFEA attack with its default parameters on three datasets created with each of the three amounts of cover traffic tested.

As expected, the results show that the more cover traffic is generated, the less accurate the attack

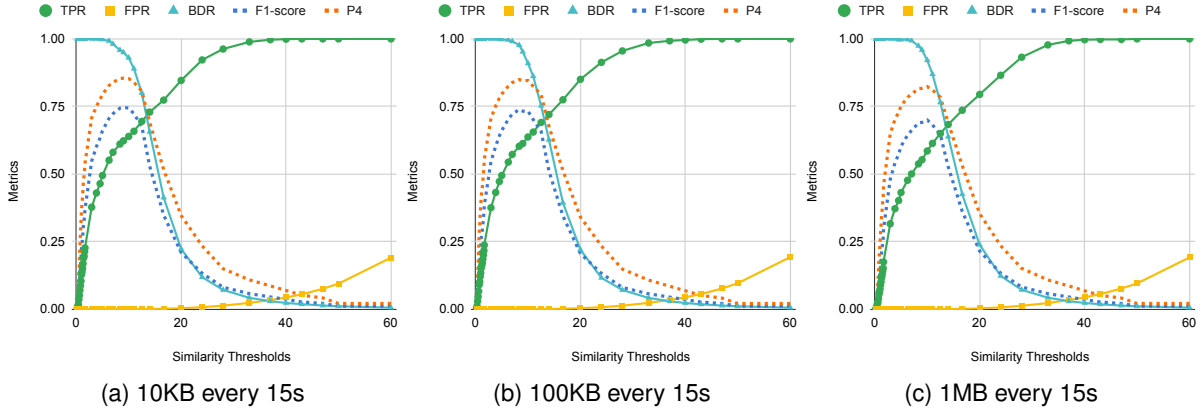


Figure 5.10: Graphs showing the results of the DeepCoFFEA attack for varying similarity thresholds, when using three different amounts of cover traffic.

Request Size	Request Period	Max F1-score	Max P4
10KB	15s	74.63%	85.46%
100KB	15s	73.83%	84.94%
1MB	15s	69.98%	82.33%

Table 5.4: Maximum F1-score and P4 values obtained for three different configurations of cover traffic with different amounts of generated data per request.

becomes. However, it is interesting to note how small of a difference each increase in the amount of cover traffic makes, especially when compared with the difference between the Tor vanilla results and the worst of these results, which reveal a massive drop of 22.32% in F1-score and of 12.99% in P4, respectively. Specifically, when increasing the amount of cover traffic by $10\times$, from 100KB to 10MB, the F1-score and P4 reduce by 3.85% and 2.61%, respectively. This suggests that, while the amount of cover traffic generated does affect the accuracy of the attack, it quickly reaches a point of diminishing returns.

Furthermore, Figure 5.11 shows the performance results obtained for three simulations, one for each previously mentioned configuration, where all clients send cover traffic following that configuration. These results suggest that these rates of generation of cover traffic have a negligible impact on performance, as the transfer times are very similar to each other and to those obtained with the Vanilla simulation, i.e., all configurations show similar Cumulative Distribution Function (CDF) curves for each transfer amount. For instance, as seen in graph (b), the time taken to transfer 1MiB of data was approximately 1 to 4 seconds for all configurations. This is likely due to the total rate of traffic generated by the clients being very low, ranging from 0.67KiB/s ($10KiB/15s$) to 66.67KiB/s ($1MiB/15s$), which is significantly lower than the rate of traffic usually generated by web browsing or video streaming, for example.

5.3.3 Period of Requests

The other basic parameter for the generation of cover traffic is the period of requests, i.e. the time between each cover traffic request made to the OS. This parameter mainly affects the frequency of occurrence of peaks in traffic sent by the OS. To evaluate its effects on the DeepCoFFEA attack's accuracy,

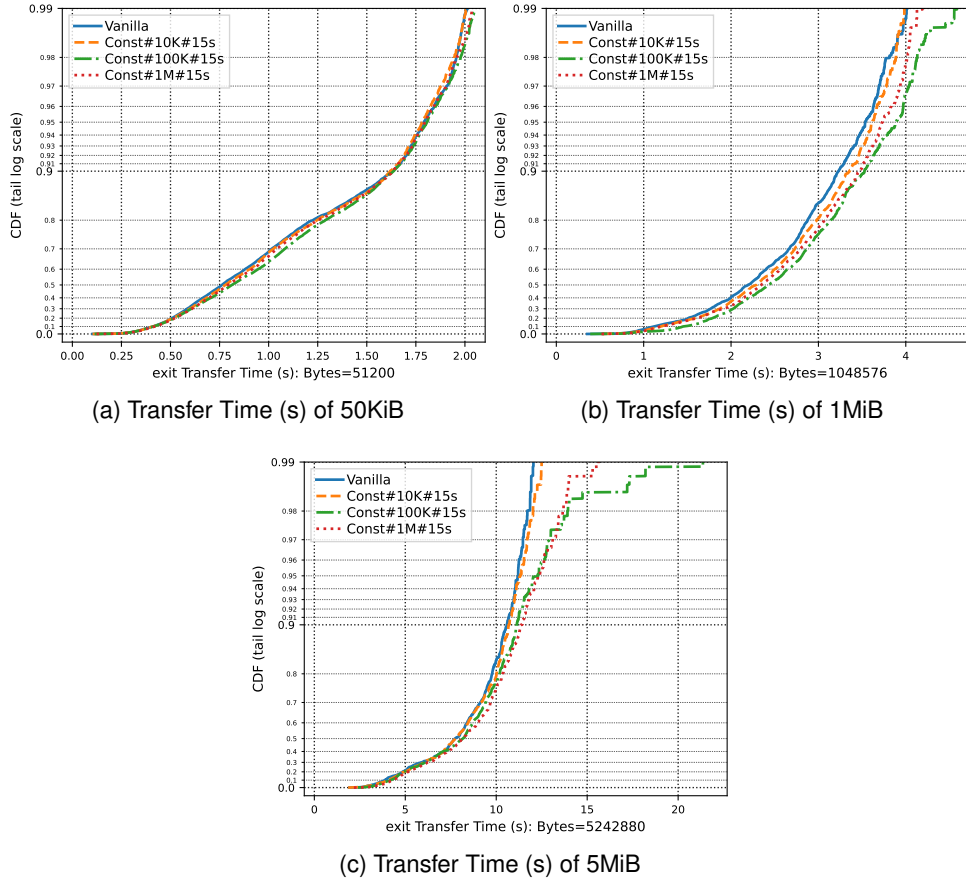


Figure 5.11: Performance results obtained by `tornettools` for three different configurations of cover traffic with different amounts of generated data per request.

Request Size	Request Period	Max F1-score	Max P4
100KB	15s	73.83%	84.94%
100KB	10s	71.58%	83.43%
100KB	5s	76.60%	86.74%

Table 5.5: Maximum F1-score and P4 values obtained for three different configurations of cover traffic with different periods of requests.

we selected two more period values in addition to the previously tested 15 seconds: 10 seconds and 5 seconds. We then tested each of these values with the three amounts of cover traffic tested previously. Figure 5.12 and Table 5.5 show only the results of the three experiments that used 100KB of cover traffic per request, as the results for the other two amounts of traffic are very similar.

The results showed that decreasing the period significantly reduced the accuracy of the attack. For example, with the reduction in period from 15s to 10s resulting in a drop in maximum F1-score and P4 of 2.25% and 1.51%, respectively. However, curiously, the 5-second period provided the worst results. After careful analysis, we concluded that this is due to 5 seconds being too little time for the generation or transfer of any significant amount of cover traffic.

Timeouts: When using a single thread or process to handle cover traffic requests, a problem arises, since a new request can only be started after the previous one has ended. Consequently, in order to

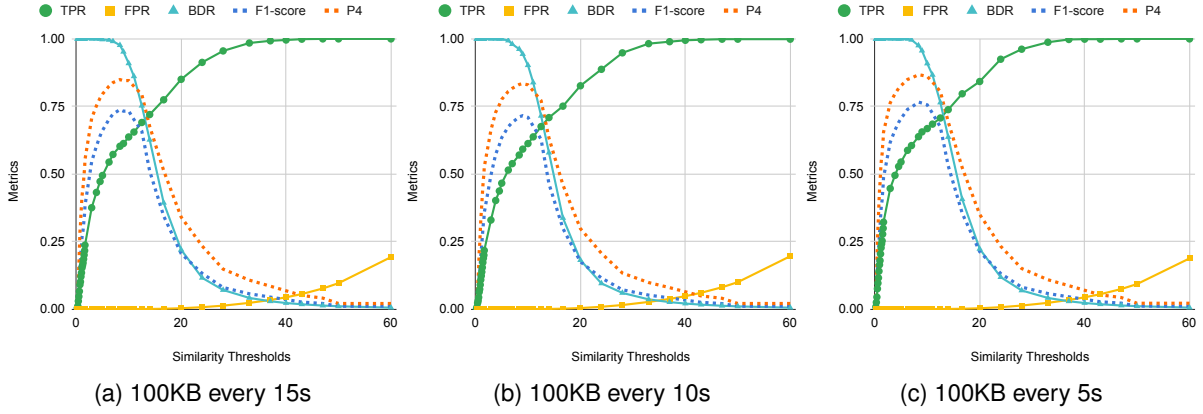


Figure 5.12: Graphs showing the results of the DeepCoFFEA attack for varying similarity thresholds, when using three different periods of requests.

maintain the specified rates, the timeouts were selected as equal to the period of requests. This resulted in experiments with low periods of requests, constantly timing out requests, limiting the amount of traffic that could be generated. This is supported by the fact that the results obtained with a period of requests of 5 seconds are exactly the same for each of the three amounts of cover traffic tested.

5.3.4 Concurrent Requests

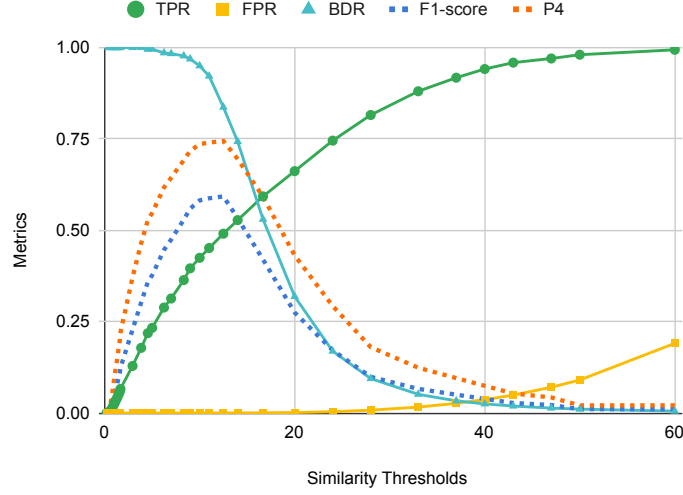
A possible solution to the limitation found when using a period of 5 seconds is to separate the timeout value of the requests from the period of the requests, allowing the timeout to be higher than the period. One way of doing this is through the usage of threads that are distributed in time, such that the requests are not made at the same time but may overlap. To evaluate the effects of this approach, we tested an adaptation of the 1MB every 15 seconds configuration to this new approach, using 2 threads, which can be summarized as 500KB every 7.5 seconds. Figure 5.13 shows the results obtained by running the DeepCoFFEA attack with its default parameters on the dataset created with this experiment.

Comparing these results with those obtained with 1MB every 15 seconds, shown in Figure 5.10 (b), we can see that the accuracy of the attack is significantly reduced. Specifically, a reduction in F1-score of 10.7% and a reduction in P4 of 7.9%. This can probably be attributed to the fact that this approach allows us to better distribute the same amount of cover traffic over time, resulting in fewer moments with no cover traffic being generated, where an attacker may be able to observe and learn real traffic patterns.

5.4 Full System Results

In this section, we present and discuss the experimental results of Shaffler with both techniques enabled: traffic modulation and generation of cover traffic.

In Section 5.2, we hypothesized that the unsatisfactory results of the traffic modulation technique were probably due to the preservation of the order of packets, which allows the DeepCoFFEA attack to easily identify patterns by combining volume analysis with timing analysis. While the obvious solution



$$\text{Max}(F1\text{-score}) = 59.28\%; \text{Max}(P4) = 74.43\%$$

Figure 5.13: Graph showing the results of the DeepCoFFEA attack for varying similarity thresholds, when using 2 threads to generate cover traffic at a rate of 500KB every 15 seconds each.

Cover Configuration	Modulation Function	Max F1-score	Max P4
None	$\text{Lognormal}(\mu = 2, \sigma = 0.5)$	96.98%	98.47%
500KB every 15s, 2 threads	None	59.28%	74.43%
500KB every 15s, 2 threads	$\text{Lognormal}(\mu = 2, \sigma = 0.5)$	20.89%	34.55%

Table 5.6: Maximum F1-score and P4 values obtained for each technique independently and when using both techniques simultaneously.

would be to shuffle the packets, our idea was to instead use cover traffic to disrupt the patterns identified, by shuffling that traffic together with the protected flow at the ingress of the circuit. Analogously, we hypothesized that the results obtained through the usage of cover traffic would also be improved by the usage of traffic modulation, as the delays added would disrupt the adversary's ability to match packets observed at the ingress and egress of the circuit, based on the expected time between observations.

To test these hypotheses, we decided to compare the results of each technique used independently with the results obtained when both techniques are used simultaneously, with the exact same parameters. For this purpose and based on our hypothesis that small delays would be enough to significantly improve traffic correlation protection, we chose a modulation function with a low mean delay size, and consequently a low impact on performance, $\text{Lognormal}(\mu = 2, \sigma = 0.5)$. Regarding the cover traffic configuration, we decided to use the same configuration that resulted in the lowest accuracy results in Section 5.3: 2 threads requesting 500KB every 15 seconds each.

Figure 5.14 and Table 5.6 show the results obtained by running the DeepCoFFEA attack with its default parameters on the three created datasets, one using only traffic modulation, another using only cover traffic, and the last one using both techniques.

By comparing the results of these three experiments, we observe that both techniques seem to complement each other, as the results obtained with both techniques are significantly better than the results in which each technique was used independently, achieving an impressive maximum F1-score of

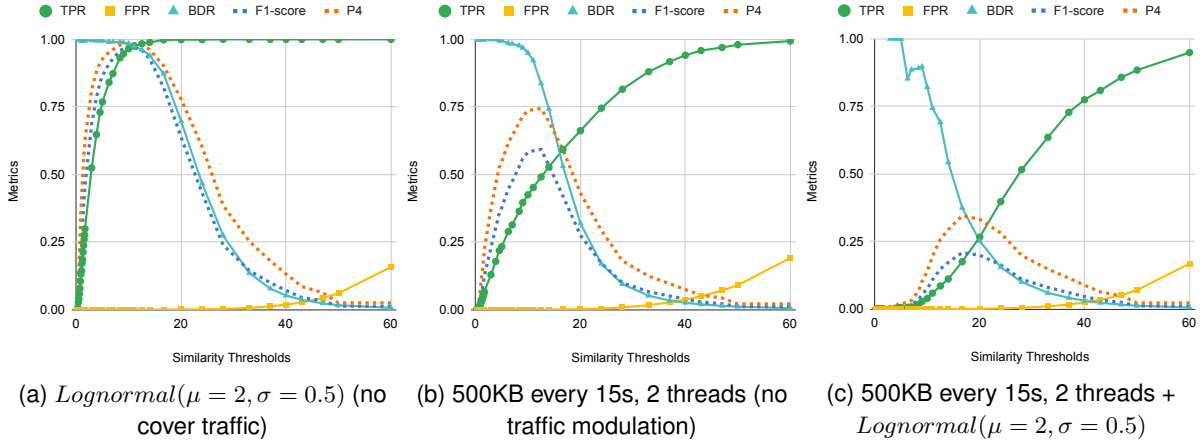


Figure 5.14: Graphs showing the results of the DeepCoFFEA attack for varying similarity thresholds, when using each technique independently and when using both techniques simultaneously.

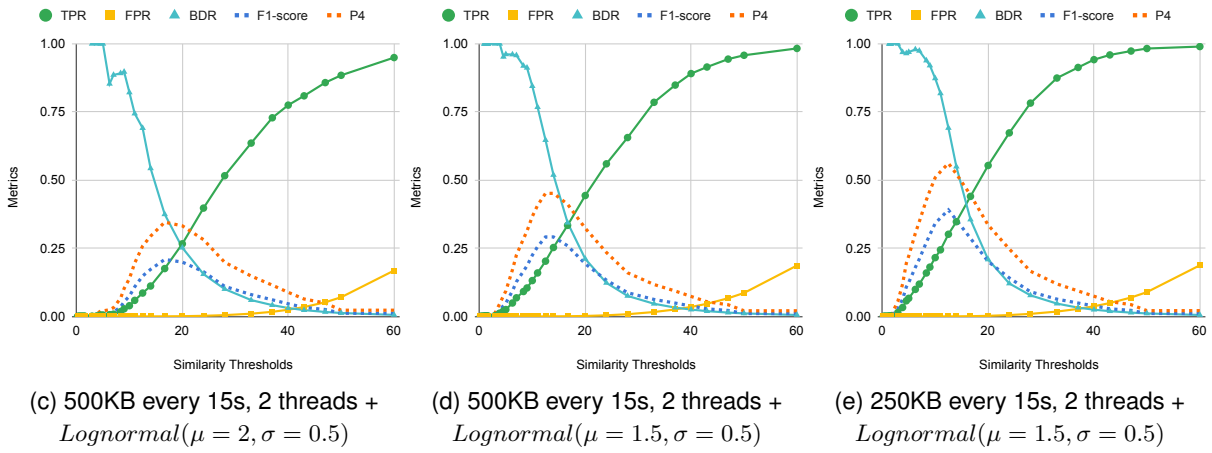


Figure 5.15: Graphs showing the results of the DeepCoFFEA attack for varying similarity thresholds, when using both techniques simultaneously, with different parameters.

20.89% and P4 of 34.55%. Furthermore, the fact that a modulation function with such a low mean delay size, which is not enough to provide any relevant protection on its own, is enough to significantly impact the accuracy of the attack when used together with cover traffic, suggests that our hypothesis might be correct and that small delays seem to be enough to disrupt timing analysis.

Although the results obtained and shown in Figure 5.14 (c) reveal an outstanding protection against traffic correlation, some might consider it more than necessary, especially considering that the impact on performance is still significant, increasing the transfer times of 1MiB by 15.5 to 17 seconds compared to Tor vanilla. As such, we decided to test altering the parameters of the *Lognormal* function to reduce the mean delay size. The parameters chosen were $\mu = 1.5$ and $\sigma = 0.5$, which result in a reduction in the mean delay size from 7 to 4.5 milliseconds, compared to the parameters previously used. After performing this test, we decided to also attempt to reduce the amount of cover traffic generated, from 500KB each 7.5 seconds to 250KB each 7.5 seconds, as we also wanted to verify the impact that reducing the amount of cover traffic generated would have when used together with traffic modulation. Figure 5.15 and Table 5.7 show the results of both experiments, as well as the results obtained with the best configuration previously shown, for comparison.

Cover Configuration	Modulation Function	Max F1-score	Max P4
500KB every 15s, 2 threads	$\text{Lognormal}(\mu = 2, \sigma = 0.5)$	20.89%	34.55%
500KB every 15s, 2 threads	$\text{Lognormal}(\mu = 1.5, \sigma = 0.5)$	29.15%	45.14%
250KB every 15s, 2 threads	$\text{Lognormal}(\mu = 1.5, \sigma = 0.5)$	39.05%	56.16%

Table 5.7: Maximum F1-score and P4 values obtained for decreasing levels of protection provided by each technique.

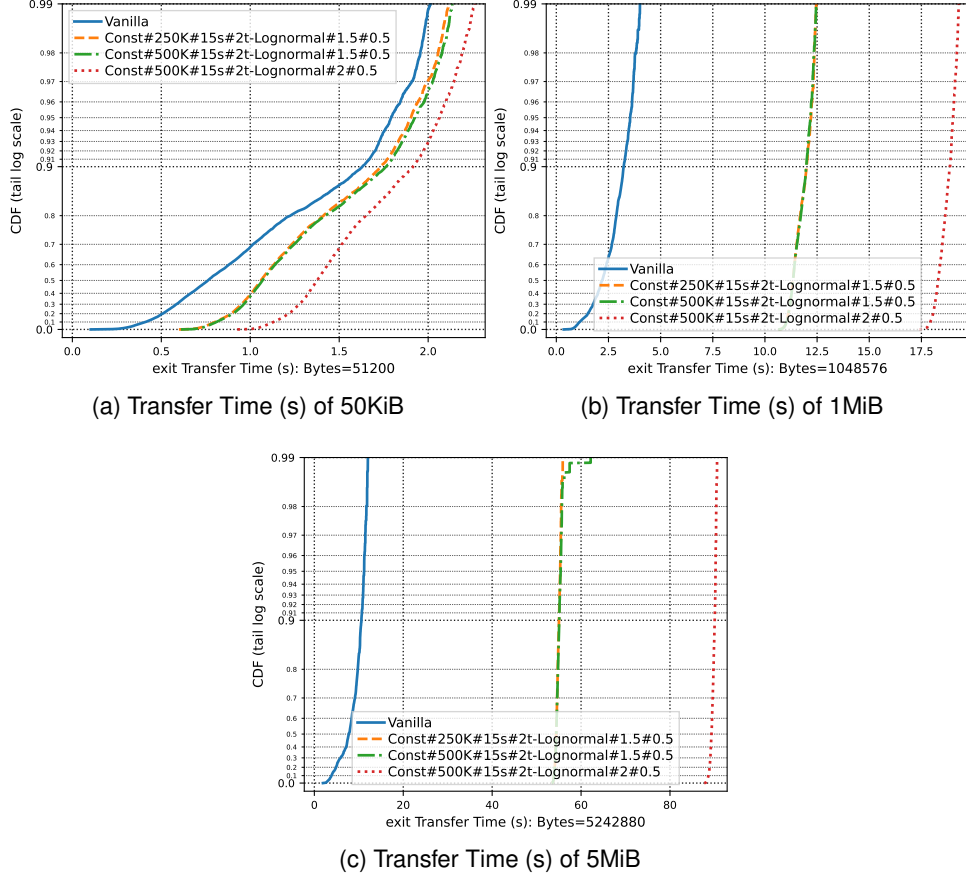


Figure 5.16: Performance results obtained by `tornettools` for three different datasets obtained by using both techniques simultaneously, with different parameters.

Furthermore, the performance results obtained for each of these experiments can be seen in Figure 5.16. These results show us that there is room in the Shaffler configuration parameters to adjust the trade-off between protection and performance. For example, the results obtained for the function $\text{Lognormal}(\mu = 1.5, \sigma = 0.5)$, which has a lower mean, show a reduction of around 7 seconds in the transfer times of 1MiB, while still being able to achieve respectable maximum F1-score and P4 values of 29.15% and 45.14%, respectively. Additionally, the similar reduction in protection observed when reducing either the mean delay size or the amount of cover traffic generated suggests that users of the system may choose which technique to reduce the strength of, based on their needs or capabilities. An example of this is a user that has a limited amount of bandwidth available, who may choose to reduce the amount of cover traffic generated, instead of reducing the mean delay size, while another user that has more available bandwidth may choose to reduce the mean delay size instead, to reduce their latency.

Summary

This chapter discusses the experimental results of running the DeepCoFFEA attack on datasets generated by Shaffler with various parameter configurations. The results highlight the attack's sensitivity to the mean delay size: higher means yield lower accuracy, and modulation functions with similar means produce similar results. Additionally, the impact on performance is also closely related to the mean delay size. The volume of traffic per cover request and the period of cover requests were also found to significantly affect the accuracy of the attack, with diminishing returns after a certain point. Finally, we found that both techniques are complementary, resulting in significantly better protection against traffic correlation when used in combination. We also found that there is room in the Shaffler configuration parameters to allow users to adjust the trade-off between protection and performance while maintaining acceptable results in both aspects. Next, we present our main conclusions of this thesis and elaborate on future directions.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

Traffic correlation attacks are a big threat to the Tor network, as they can be used to deanonymize users, defeating the purpose of the network. In this work, we have explored the problem of traffic correlation attacks on the Tor network and proposed a system to decrease their effectiveness. Our approach is based on the combination of two defensive techniques: traffic modulation and cover traffic generation.

We have designed and implemented a version of Tor that is fully compatible with the existing Tor infrastructure, and allows traffic modulation to be configured by a client and performed by the middle node of a circuit. We have also designed and implemented a method for clients to generate cover traffic to add confusion to the ingress of a circuit with minimal impact on the network, and without affecting real web services with fake requests.

Through our evaluation of this system, we discovered that the use of both techniques simultaneously results in significantly better protection against traffic correlation than the use of each technique independently. The best results have been obtained when using a moderate amount of cover traffic (2 threads generating 500KB every 15 seconds each) together with the modulation function $\text{Lognormal}(\mu = 2, \sigma = 0.5)$, which was not enough to provide any relevant protection on its own. Achieving an impressive reduction, compared to vanilla Tor, of 76.06% and 63.9% in the maximum F1-score and P4 results for the attack, respectively. Although these results come at the cost of performance, increasing the transfer times of 1MiB by 15.5 to 17 seconds, we also found that there is room in the Shaffler configuration parameters to reduce the strength of each technique. As such, users can adjust the trade-off between protection and performance while maintaining an acceptable level of protection. To exemplify this, we also presented the results of the same cover traffic configuration together with a modulation function with lower mean ($\text{Lognormal}(\mu = 1.5, \sigma = 0.5)$), which resulted in a reduction of around 7 seconds in the transfer times of 1MiB, at the cost of worsening the F1-score and P4 results for the attack by only 8.29% and 10.59%, respectively.

We have also observed that the mean delay size is a critical characteristic of all modulation functions, significantly influencing both the accuracy of traffic correlation attacks and the performance of

the system, with higher means decreasing the accuracy of the DeepCoFFEA attack and substantially increasing the time required to transfer data to a network destination. Furthermore, our findings suggest that larger ranges of possible delay values may have a negative impact on attack accuracy. However, unlike the results related to mean delay sizes, these were not as conclusive and would benefit from additional experimentation.

Regarding cover traffic generation, we discovered that, contrary to the traffic modulation technique, it can be a significantly effective defense against traffic correlation attacks by itself, without affecting the latency of real communications. Additionally, from our experiments, we also learned that higher rates and volumes of cover traffic decrease the accuracy of these attacks, as expected.

6.2 Achievements

The main achievement of the present work is the design and implementation of Shaffler, a modified version of Tor that introduces traffic modulation and cover traffic generation techniques to decrease the effectiveness of traffic correlation attacks. Additional accomplishments of this thesis include: *(i)* full compatibility of Shaffler with the current Tor infrastructure, to allow a gradual deployment; *(ii)* configurable parameters for Shaffler, to allow users to adjust the trade-off between protection and performance; *(iii)* an in-depth study of the effects of multiple configuration options of Shaffler on the effectiveness of traffic correlation attacks and on the performance of the network; *(iv)* an in-depth study of the trade-offs and viability of Shaffler's defensive techniques.

6.3 Future Work

The present work has focused on the design and implementation of Shaffler, a system that combines traffic modulation and cover traffic generation techniques to reduce the effectiveness of traffic correlation attacks. However, there are still some areas of the system that can be improved and additional research that can be done to further investigate the solutions to traffic correlation attacks proposed by our system.

To begin with, the current implementation of Shaffler uses timers to delay cells, which restrict the resolution of the delays that can be applied. Therefore, a beneficial improvement would be to implement an alternative to these timers, that is able to provide better time resolution. This would enable us to examine the effects of the increased time resolution on the effectiveness of the system.

Secondly, the current implementation of Shaffler requires users to manually configure the system parameters. Thus, another useful improvement would be to implement a mechanism to automatically adjust the parameters of the system, for example, based on a set of modes, to allow users to more easily adjust the trade-off between protection and performance.

Furthermore, the current implementation of Shaffler does not take into account the current network conditions. Consequently, another useful improvement would be to implement a mechanism to automatically adjust the parameters of the system, based on current network conditions, to avoid overloading the network. This could be done, for example, by sending identifiable cover packets through the cover

traffic loop and measuring their latency. The system could then base itself on this latency to, for instance, dynamically tweak the rate at which cover traffic is generated.

Finally, the current evaluation of Shaffler has focused on the effectiveness of the system against the DeepCoFFEA attack. Another beneficial improvement would be to conduct a study of the effectiveness of the system against other traffic correlation attacks, other than DeepCoFFEA, such as DeepCorr [20]. Additionally, all evaluation of Shaffler presented in this thesis has focused on simulation experiments. Thus, it would be useful to verify all results by performing experiments using emulation instead.

Bibliography

- [1] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The Second-Generation onion router. In *13th USENIX Security Symposium (USENIX Security 04)*, San Diego, CA, Aug. 2004. USENIX Association. URL <https://www.usenix.org/conference/13th-usenix-security-symposium/tor-second-generation-onion-router>.
- [2] R. Nithyanand, O. Starov, A. Zair, P. Gill, and M. Schapira. Measuring and mitigating as-level adversaries against tor. In *Network and Distributed System Security Symposium (NDSS)*, Feb. 2016.
- [3] A. Johnson, C. Wacek, R. Jansen, M. Sherr, and P. Syverson. Users get routed: Traffic correlation on tor by realistic adversaries. In *ACM SIGSAC Conference on Computer and Communications Security*, Nov. 2013.
- [4] S. J. Murdoch and G. Danezis. Low-cost traffic analysis of tor. In *IEEE Symposium on Security and Privacy*, May 2005.
- [5] V. Shmatikov and M.-H. Wang. Timing analysis in low-latency mix networks: Attacks and defenses. In *European Symposium on Research in Computer Security*, Jan. 2006.
- [6] M. Wright, M. Adler, B. N. Levine, and C. Shields. Defending anonymous communications against passive logging attacks. In *IEEE Symposium on Security and Privacy*, May 2003.
- [7] M. Akhoondi, C. Yu, and H. V. Madhyastha. Lastor: A low-latency as-aware tor client. In *IEEE Symposium on Security and Privacy*, May 2012.
- [8] M. Edman and P. Syverson. As-awareness in tor path selection. In *ACM SIGSAC Conference on Computer and Communications Security*, Nov. 2009.
- [9] D. Das, S. Meiser, E. Mohammadi, and A. Kate. Anonymity trilemma: Strong anonymity, low bandwidth overhead, low latency - choose two. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 108–126, 2018. doi: 10.1109/SP.2018.00011.
- [10] R. Dingledine and N. Mathewson. Anonymity loves company: Usability and the network effect. In *Workshop on the Economics of Information Security*, 2006.

- [11] S. Chakravarty, A. Stavrou, and A. D. Keromytis. Traffic analysis against low-latency anonymity networks using available bandwidth estimation. In *European Symposium on Research in Computer Security*, Sept. 2010.
- [12] R. Pries, W. Yu, X. Fu, and W. Zhao. A new replay attack against anonymous communication networks. In *IEEE International Conference on Communications*, May 2008. doi: 10.1109/ICC.2008.305.
- [13] B. Evers, J. Hols, E. Kula, J. Schouten, M. den Toom, R. van der Laan, and J. Pouwelse. Thirteen years of tor attacks. *Computer Science, Delft University of Technology*, 2016.
- [14] P. Winter, R. Ensafi, K. Loesing, and N. Feamster. Identifying and characterizing sybils in the tor network. In *USENIX Security Symposium*, Aug. 2016.
- [15] J. R. Douceur. The sybil attack. In *International workshop on peer-to-peer systems*, Mar. 2002.
- [16] K. Bauer, D. McCoy, D. C. Grunwald, and T. Kohno. Low-resource routing attacks against anonymous systems. In *ACM Workshop on Privacy in the Electronic Society*, Oct. 2007.
- [17] D. R. Figueiredo, P. Nain, and D. Towsley. On the analysis of the predecessor attack on anonymity systems. *Computer Science Technical Report*, Aug. 2004.
- [18] R. Dingledine and G. Kadianakis. One fast guard for life (or 9 months). In *Privacy Enhancing Technologies*, July 2014.
- [19] Y. Sun, A. Edmundson, L. Vanbever, O. Li, J. Rexford, M. Chiang, and P. Mittal. RAPTOR: Routing attacks on privacy in tor. In *USENIX Security Symposium*, Washington, D.C., Aug. 2015. URL <https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/sun>.
- [20] M. Nasr, A. Bahramali, and A. Houmansadr. Deepcorr: Strong flow correlation attacks on tor using deep learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS '18*, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356930. doi: 10.1145/3243734.3243824. URL <https://doi.org/10.1145/3243734.3243824>.
- [21] S. E. Oh, T. Yang, N. Mathews, J. K. Holland, M. S. Rahman, N. Hopper, and M. Wright. Deepcoffee: Improved flow correlation attacks on tor via metric learning and amplification. In *2022 IEEE Symposium on Security and Privacy (SP)*, 2022. doi: 10.1109/SP46214.2022.9833801.
- [22] Tor Project. Tor faq. <https://support.torproject.org/faq/>. Accessed: 2023-01-05.
- [23] Tor Project. Reporting bad relays. <https://gitlab.torproject.org/legacy/trac/-/wikis/doc/ReportingBadRelays>, June 2020. Accessed: 2023-01-12.
- [24] Tor Project. How to report bad relays. <https://blog.torproject.org/how-report-bad-relays>, July 2014. Accessed: 2023-01-05.

- [25] P. Winter, R. Köwer, M. Mulazzani, M. Huber, S. Schrittwieser, S. Lindskog, and E. Weippl. Spoiled onions: Exposing malicious tor exit relays. In *Privacy Enhancing Technologies*, July 2014.
- [26] S. Chakravarty, G. Portokalidis, M. Polychronakis, and A. D. Keromytis. Detecting traffic snooping in tor using decoys. In *International Workshop on Recent Advances in Intrusion Detection*, Sept. 2011.
- [27] T. Elahi, K. Bauer, M. AlSabah, R. Dingledine, and I. Goldberg. Changing of the guards: A framework for understanding and improving entry guard selection in tor. In *ACM Workshop on Privacy in the Electronic Society*, Oct. 2012.
- [28] J. Juen, A. Johnson, A. Das, N. Borisov, and M. Caesar. Defending tor from network adversaries: A case study of network path prediction. *Privacy Enhancing Technologies*, June 2015.
- [29] C. Wacek, H. Tan, K. S. Bauer, and M. Sherr. An empirical evaluation of relay selection in tor. In *Network and Distributed System Security Symposium (NDSS)*, Feb. 2013.
- [30] Z. Li, S. Herwig, and D. Levin. Detor: Provably avoiding geographic regions in tor. In *USENIX Security Symposium*, Aug. 2017.
- [31] D. Levin, Y. Lee, L. Valenta, Z. Li, V. Lai, C. Lumezanu, N. Spring, and B. Bhattacharjee. Alibi routing. In *ACM Special Interest Group on Data Communication*, Aug. 2015.
- [32] Z. Weinberg, S. Cho, N. Christin, V. Sekar, and P. Gill. How to catch when proxies lie: Verifying the physical locations of network proxies with active geolocation. In *Internet Measurement Conference*, New York, NY, Oct. 2018. Association for Computing Machinery.
- [33] K. Kohls, K. Jansen, D. Rupprecht, T. Holz, and C. Pöpper. On the challenges of geographical avoidance for tor. In *Network and Distributed System Security Symposium (NDSS)*, Feb. 2019.
- [34] J. van den Hooff, D. Lazar, M. Zaharia, and N. Zeldovich. Vuvuzela: Scalable private messaging resistant to traffic analysis. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 137–152, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450338349. doi: 10.1145/2815400.2815417. URL <https://doi.org/10.1145/2815400.2815417>.
- [35] D. Lazar, Y. Gilad, and N. Zeldovich. Karaoke: Distributed private messaging immune to passive traffic analysis. In *USENIX Symposium on Operating Systems Design and Implementation*, 2018.
- [36] A. M. Piotrowska, J. Hayes, T. Elahi, S. Meiser, and G. Danezis. The loopix anonymity system. In *26th USENIX Security Symposium (USENIX Security 17)*, Vancouver, BC, Aug. 2017. USENIX Association. ISBN 978-1-931971-40-9. URL <https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/piotrowska>.
- [37] C. Díaz, H. Halpin, and A. Kiayias. The nym network the next generation of privacy infrastructure. 2021. URL <https://api.semanticscholar.org/CorpusID:233218535>.

- [38] A. Kwon, H. Corrigan-Gibbs, S. Devadas, and B. Ford. Atom: Scalable anonymity resistant to traffic analysis. *CoRR*, abs/1612.07841, 2016. URL <http://arxiv.org/abs/1612.07841>.
- [39] C. Chen, D. E. Asoni, A. Perrig, D. Barrera, G. Danezis, and C. Troncoso. Taranet: Traffic-analysis resistant anonymity at the network layer. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2018. doi: 10.1109/EuroSP.2018.00018.
- [40] C. Chen, D. E. Asoni, D. Barrera, G. Danezis, and A. Perrig. Hornet: High-speed onion routing at the network layer. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, CCS '15*, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450338325. doi: 10.1145/2810103.2813628. URL <https://doi.org/10.1145/2810103.2813628>.
- [41] R. Meier, V. Lenders, and L. Vanbever. ditto: Wan traffic obfuscation at line rate. In *NDSS Symposium 2022*. Internet Society, 2022.
- [42] G. Danezis. The traffic analysis of continuous-time mixes. In *International Workshop on Privacy Enhancing Technologies*, pages 35–50. Springer, 2004.
- [43] S. Engler and I. Goldberg. Weaving a faster tor: A multi-threaded relay architecture for improved throughput. In *Proceedings of the 16th International Conference on Availability, Reliability and Security, ARES 21*, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450390514. doi: 10.1145/3465481.3465745. URL <https://doi.org/10.1145/3465481.3465745>.
- [44] T. Project. torrc(5). <https://manpages.debian.org/testing/tor/torrc.5.en.html>, Jan. 2023. Accessed: 2023-08-03.
- [45] R. Jansen and N. Hopper. Shadow: Running tor in a box for accurate and efficient experimentation. In *Network and Distributed System Security Symposium (NDSS)*. Internet Society, Feb. 2012.
- [46] tornettools. <https://github.com/shadow/tornettools/tree/v2.0.0>, May 2023. Accessed: 2023-08-01.
- [47] OnionTrace. <https://github.com/shadow/oniontrace/tree/v1.0.0>, Mar. 2023. Accessed: 2023-08-01.
- [48] TGen. <https://github.com/shadow/tgen/tree/v1.1.1>, July 2023. Accessed: 2023-08-01.
- [49] M. Sitarz. Extending f1 metric, probabilistic approach, Oct. 2022.
- [50] G. V. Rossum and F. L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009. ISBN 1441412697.
- [51] D. Song. dpkt. <https://dpkt.readthedocs.io/en/latest/>, Aug. 2022. Accessed: 2023-08-03.
- [52] The 2022 Web Almanac. <https://almanac.httparchive.org/en/2022/>, 2022. Accessed: 2023-09-02.

- [53] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, Í. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020. doi: 10.1038/s41592-019-0686-2.

