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

(extended abstract of the MSc dissertation)

Afonso Pedro Antunes da Mota Gomes
Departamento de Engenharia Informática
Instituto Superior Técnico

Advisors: Professors Nuno Santos and Kevin Gallagher

***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.**

I. Introduction

The Internet has become a fundamental part of people's lives, yet it has also presented a major issue with regards to privacy and anonymity. This is because standard protocols for Internet communication are based on strong traffic-identifying metadata, such as source and destination IP addresses. These IP addresses are necessary for communication parties to establish TCP/IP connections and answer each other's messages, making them powerful identifiers that allow network observers, such as Internet Service Providers (ISPs), to trace communications to a sender and a receiver, thus making it difficult to remain anonymous online.

Fortunately, there are systems that can offer users improved anonymity. The most well-known of these is the Tor network [1]. When using Tor, the sender's traffic is routed through a minimum of three "onion routers" before it reaches the recipient. The sender's traffic is encrypted in sequence, with all the secret keys negotiated between the sender and each node of the circuit. As traffic passes through the circuit, each node decrypts it with its key, revealing the address of the next node. This way, only the sender knows both its own and the recipient's IP addresses.

Despite the strong anonymity properties provided by the Tor onion routing protocols, there are still more complex methods to identify the parties involved in a communication. Traffic correlation attacks [1] are one of these, which involve the attempt to link patterns in traffic sent by a user to the Tor network with patterns of traffic leaving the Tor network and arriving at a remote host. Dingledine et al. [1] refer to these attacks as *end-to-end timing correlation*, when the attack is based on timing patterns, and *end-to-end volume correlation* when an attacker attempts to correlate flows through the volume of packets.

For a long period of time, it was thought difficult to launch these attacks in practice due to the geographical dispersion of Tor circuits' routes and the difficulty for a single ISP to intercept traffic at both communication endpoints of a Tor circuit. However, traffic correlation attacks are now becoming more of a realistic threat to the anonymity of Tor users, as they can be launched on a large scale by a coalition of ISPs that are similar to state-level adversaries. [2] conducted an empirical study and found that up to 40% of all Tor traffic is vulnerable to attacks by traffic correlation from network-level attackers; 42% from Autonomous Systems (ASs) that may collaborate with each other and 85% from state-level adversaries. Users in certain countries (such as 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] revealed that 80% of all users can be identified when faced with a relay-level attacker. Surprisingly, most users can be successfully identified within three months by an AS-level attacker. When ASs collaborate, they can correctly correlate users in 90 times less time. When targeting a specific web user, collaborating ASs can effectively deanonymize them in a single day.

The research community has been striving to create new defenses to protect Tor users from potential risks [2, 4–8]. Das et al. [9] coined the term *anonymity trilemma* to describe the problem of having to sacrifice at least one between strong anonymity, low latency, and low resource usage. This work investigates two defensive techniques to improve Tor's anonymity: traffic modulation, which comes at the cost of latency, and cover traffic generation, which mainly sacrifices resource usage.

	Low Latency	Resistant to TCA	Traffic Modulation	Cover Traffic	Compatible with the Tor Network
Tor [1]	✓	✗	✗	✗	✓
Vuvuzela [11]	✗	✓	✗	✓	✗
Atom [12]	✗	✓	✓	✗	✗
Loopix [13]	✓	✓	✓	✓	✗
Nym [14]	✓	✓	✓	✓	✗
Shaffler	✓	✓	✓	✓	✓

Table I
COMPARISON OF VARIOUS ANONYMITY SYSTEMS.

Our goal is to develop a system that decreases the effectiveness of traffic correlation attacks by focusing on two principles: making flows in the network indistinguishable and increasing the number of flows passing through the entry or exit nodes. Regarding the first principle, our approach is to employ traffic modulation in the middle of a circuit, making the timing patterns observed by an adversary at both ends of a circuit different. For the second principle, we explore the generation of cover traffic, where clients send cover traffic to a self-hosted Onion Service (OS), using the same guard node for all circuits, thus increasing the number of flows that pass through that entry node.

Based on these two principles, we developed a system called Shaffler, which provides a functional implementation of both techniques, traffic modulation and cover traffic, in Tor, while maintaining compatibility with the existing infrastructure. Furthermore, we present an extensive evaluation of Shaffler and its various configuration options, made possible by the use of the Shadow [10] network simulator.

II. RELATED WORK

One of the most powerful classes of attacks aimed at deanonymizing Tor circuits are **correlation attacks**. 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 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.

Another very influential work for performing flow correlation is DeepCorr [15]. It 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 that 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 [16], which is more effective than DeepCorr, while also introducing a significant speedup.

Defenses against traffic correlation: 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. [15] 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 the employing of 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.

Other anonymity systems: In addition to the defensive mechanisms studied for Tor that address the challenge of traffic correlation attacks, various alternative anonymity systems have emerged, each incorporating a wide range of techniques design to defend against traffic analysis attacks. Systems such as Loopix [13] and Nym [14] use a combination of two interesting techniques: (i) traffic modulation, which modifies the timing patterns observed in traffic, through the application of delays based on a statistical distribution; (ii) cover traffic generation, which not only modifies the volume patterns found in traffic, but also further perturbs the timing patterns. Table **??** presents a summary of some characteristics provided by Shaffler, and compares it with various other anonymity systems. Although the techniques 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.

III. 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 observe the *ingress* and *egress* flows of a circuit and may attempt to correlate them. Although this threat model does not consider global passive adversaries, 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 limit the strength of the attacker to being able to control at most two out of the three Onion Relays (ORs) used in a circuit. The most threatening of the resulting combinations are the entry and exit nodes, considering that the objective is to perform traffic correlation. 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 active attacks in addition to those considered in the original threat model. 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 based on machine learning algorithms. These adversaries could be embodied by governments, law enforcement agencies, or other organizations with the resources and capabilities to conduct mass surveillance.

IV. Design

Shaffler aims to protect Tor clients against traffic correlation 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 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, colored orange in Figure 1, at the client endpoint and at the mix nodes. 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 1, colored dark gray. Together, these components generate cover traffic and delay the transmitted data towards preventing attacks carried out by an adversary that wishes to correlate the traffic observed at both endpoints of a circuit.

A. 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 work is to provide and study modulation functions based on statistical distributions, our main focus is also to design our system in such a way that other researchers can easily implement and test their own modulation functions. Shaffler provides the following modulation functions: Uniform, Normal, Poisson, Exponential and Lognormal.

Who modulates: Considering our threat model, which states that the entry and exit nodes of a circuit may be observed by an adversary, 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: To allow clients to customize their protection, we decided on an approach in which both the client and the delaying node have a role 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

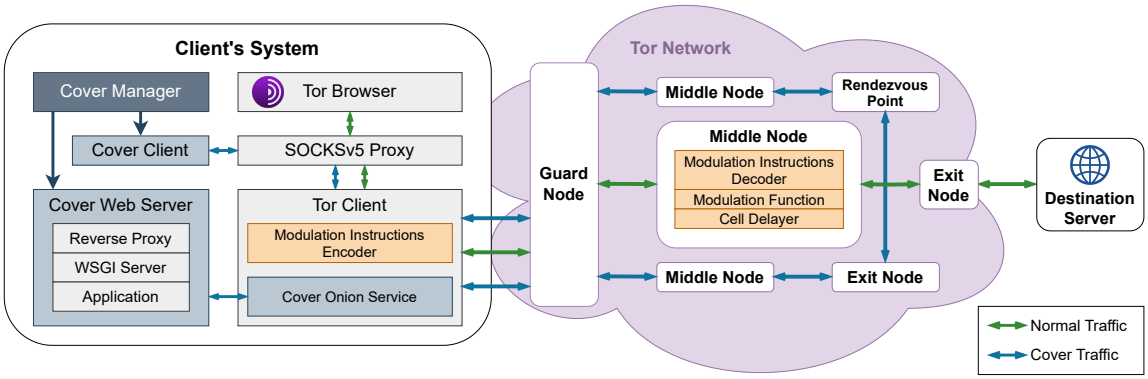


Figure 1. Shaffler system architecture.

being applied. In this case, the delay must exceptionally be applied in relation to the arrival time of the cell.

B. Generation of Cover Traffic

As mentioned previously, 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 correlation of the 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 [13]. This allows us to both increase the amount of extra traffic crossing the guard node and better control its characteristics by adjusting the server's response as needed.

A possible alternative to using an OS would be to use a simple web server, also hosted on the client's machine. However, while initially it might seem to reduce the load of the technique on the network, that is not the case. Although the number of non-target ORs that get affected by the traffic is halved, the number of times a single flow passes through the target node is also halved. Additionally, 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. Using an OS we also avoid requiring that users configure Network Address Translation (NAT) on their local network to make the web server accessible from outside the network. For these reasons, we decided to use an OS.

Besides 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 the requests and adjust the desired response from the OS. The highly customizable nature of both *cover OS* and *cover client* enables Shaffler to tailor the generation of cover traffic to the legitimate one's characteristics.

Shaffler's design requires all three 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, the packets captured near the guard now consist of a mix of web server traffic and cover traffic. Both OS requests and responses can be modulated as needed, allowing for the introduction of further variability.

V. Implementation

To implement Shaffler we used two programming languages, C and Python. To implement traffic modulation, we modified version 0.4.7.13 of Tor, which is written in the C language. For the implementation of our cover traffic technique we instead used Python 3.11.2. This section provides an overview of Shaffler's implementation details.

A. Applying delays

To implement cell delays, we identified two promising utilities already implemented in Tor: queues and timers. In the vanilla implementation of Tor, when a cell is prepared to be sent through a communication channel to a 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 2 shows a diagram representing this modified process.

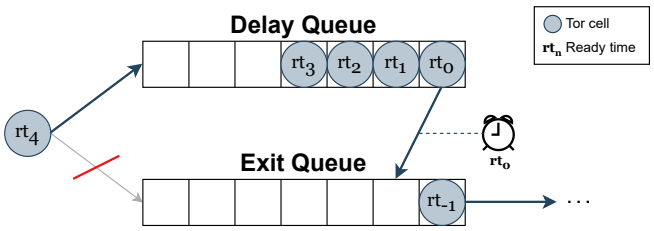


Figure 2. Diagram showing the process for delaying Tor cells. Includes the usage of a delay queue that stores cells ordered by ready time ($rt_n \leq rt_{n+1}$) and a single timer that waits for the first cell's ready time.

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.

B. 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 modulation function to generate the delay value. By separating the code in this way, we allow the addition of new delay modes in the future, without requiring too many changes in the code.

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. This method provides an important characteristic of order preservation, which not only ensures the 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 singular timer to transfer cells from the delay queue to the cell queue. This timer is scheduled for the ready time of the cell at the head of the delay queue.

When the timer fires, all the cells in the queue that are ready are moved to the cell queue, 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 because of the limited resolution of the timers, which is one millisecond. This means that delays of less than 1 millisecond between cells might cause multiple cells to be ready when the timer is triggered.

C. Encoding and decoding delay policies

To allow clients to customize the modulation performed at the middle nodes, we implemented a method that piggybacks on Tor's existing protocol for creating circuits, which is described in Figure 3. 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) 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*.

Modified protocol: The modifications start when sending the EXTEND cell, whose objective is to add the middle node to the circuit. The client appends to the payload of the EXTEND cell a 16-byte magic number followed by the delay policy itself.

Upon receiving this EXTEND cell, the entry node will check for the presence of the magic number, 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.

After interpreting the contents of the cell, the middle node inserts them into the corresponding `or_circuit_t` data structure, so that all communications from there onward may be delayed based on the specified delay policy.

Furthermore, a `torrc` [17] option is provided to ORs to allow them to fully reject delaying traffic and performing traffic modulation, called `DisableDelays`. 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, to inform the client of the successful application of the delay policy.

Additionally, we provide the client with the option to enforce the usage of the delay policy. 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.

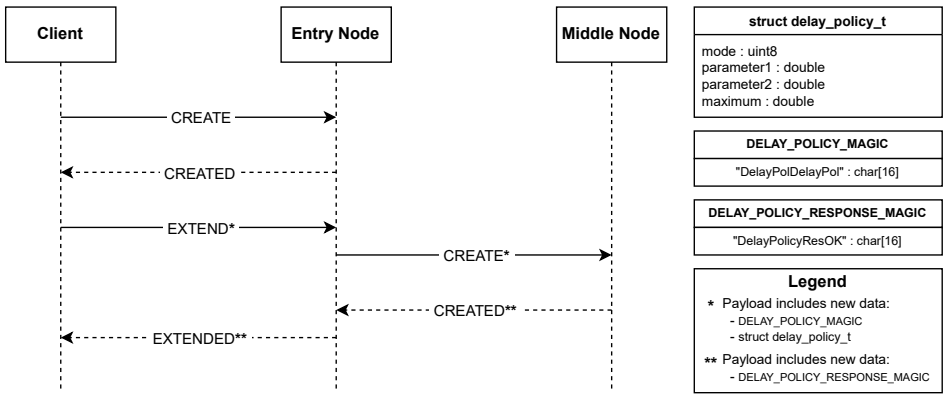


Figure 3. Diagram showing the additions made to the circuit creation protocol to allow the communication of delay policies.

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.

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.

The delay policy and the option to enforce it are both implemented in the form of torrc [17] 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.

D. Creating Cover Traffic

As described in Section IV, all components responsible for Shaffler's cover traffic generation were conceived to run on the user's system. Since the goal is to ensure 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 all circuits share the same guard. Therefore, in Shaffler's implementation, 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.

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 as the "regular client". In order to use the system's custom Tor process, the browser's default profile was configured through preferences to: (i) prevent the Tor Browser from attempting to start its own Tor process (the default behaviour), (ii) specify the control and SOCKS ports used by the custom Tor process. Additionally, 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 using the Flask framework. It runs on a Gunicorn WSGI server and uses Nginx as a reverse proxy. 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: (i) Constant, where the OS responds with a predetermined or requested amount of randomly generated bytes; (ii) Single page, where the OS responds by serving a predetermined or requested web page, which can be identified by name; (iii) Multiple pages, where the OS serves a web page from a selection of preloaded templates; (iv) Dynamic, where the OS can respond with either randomly generated bytes or a 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

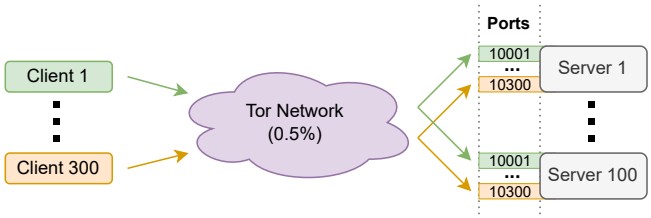


Figure 4. Simulation configuration used for the dataset collection.

vary with it. The script relies on either the `requests` or `requests_html` 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. 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 Gunicorn 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.

VI. Evaluation

This section describes our evaluation methodology and presents the results of various experiments performed to assess Shaffler's effectiveness and performance.

A. Methodology

Correlation Metrics: By default, the DeepCoFFEA attack outputs the metrics True Positive Rate (TPR), False Positive Rate (FPR) and Bayesian Detection Rate (BDR). However, these metrics are not very practical to compare results obtained with different configurations of Shaffler. The difficulty stems from the nature of the relationship between these metrics; for example, a higher TPR comes at the cost of a higher FPR and lower BDR.

Therefore, we require metrics that allow us to pinpoint the similarity threshold where these trade-offs are optimized, allowing a fair result comparison. As such, we calculated and used F1-score, which is a compound metric that can be calculated from true and false positives and negatives of a classification problem's outcome. Additionally, we also decided to use P4 [18], which was designed to address F1-score's tendency to overlook true and false negatives, providing a more balanced assessment of the effectiveness of our defenses.

Performance Metrics: By default, simulations by `tornettools` use 100 *perfclients* to perform several performance measurements. Among all metrics provided by `tornettools`, we found the key metrics for evaluating Shaffler configurations performance to be Transfer Times (for data sizes $N \in \{50KiB, 1MiB, 5MiB\}$) and Error Rate (percentage of failed data streams, for example, due to timeouts).

Dataset Collection: To evaluate the impact of the Shaffler on the DeepCoFFEA attack's traffic correlation ability, we required compatible datasets. As previously mentioned, to generate those datasets, we employed the Shadow network simulator to mimic our modified version of Tor and used `tornettools` to create a realistic network configuration, modified to resemble the dataset collection method described by Oh et al. [16]. Due to resource constraints, we simulated networks at 0.5% of the real Tor network's size.

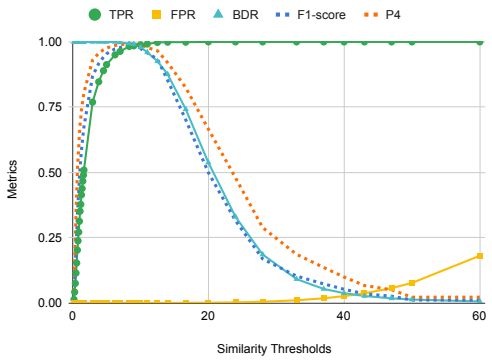
The datasets are composed of pairs of files, capturing a flow at the ingress and egress of the circuit, specifically, the timing and size details of each packet.

To collect these datasets, it was necessary to overcome two main challenges. The first of those challenges was how to capture the packet details within the simulation. We used a Shadow configuration for simulated hosts to capture packets in PCAP format, retaining essential header information while discarding unnecessary payload data. We limited captures to 24 bytes per packet using Shadow's options, crucial for manageable dataset sizes. Unlike the method described by Oh et al. [16], we directly captured egress flows on the server hosts, eliminating the need for a proxy server due to our server monitoring capabilities.

The second challenge was how to efficiently collect flows while keeping track of the pairings of captures. Simulating and capturing each flow of the dataset at a time would be inefficient, while capturing multiple flows simultaneously risks losing track of the flow pairings. To overcome this, we designed a method consisting of running N clients, each assigned a unique identifier i. Servers were configured to listen on N ports, one for each client. Clients determined the destination port for their flows using the formula: $port_{c_i} = 10000 + i$, where $i \in [1, N]$.

This setup, shown in Figure 4, with $N = 300$, collected around 23000 flow pairs for 2 hours, with each pair spanning 1 minute. The 2-hour simulation included a 5-minute one-time setup and 30-second intervals between generated client flows.

After running the simulation, we developed a Python [19] script using dpkt [20] to extract and parse the packet captures, creating the dataset in the format used by DeepCoFFEA. The script has two phases: staging, where flow details are organized into a sorted JSON file, and parsing, where timestamps and sizes are added to the corresponding flow's output file. The output format consists of two directories:



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

Figure 5. Graph showing the results of the DeepCoFFEA attack for varying similarity thresholds, when using the "Tor vanilla" dataset.

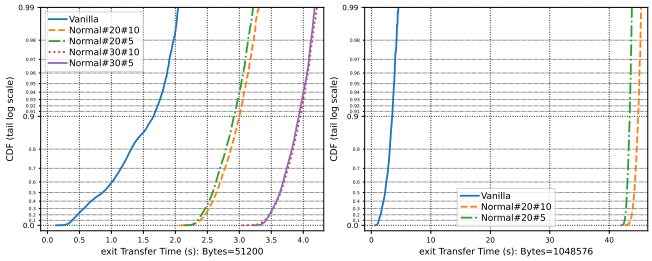


Figure 6. Transfer Times of 50KiB and 1MiB, obtained for the tested configurations of the Normal distribution.

inflow for ingress flows and *outflow* for egress flows. Each flow pair is represented by two files with matching names, one in each directory.

B. Tor Vanilla Results

To evaluate the effects of traffic modulation on the DeepCoFFEA attack accuracy, we compared results with Tor vanilla (version 0.4.7.13) using the dataset collection method described above. Figure 5 shows a plot of TPR, FPR, and BDR for different similarity thresholds.

Our dataset produced more accurate DeepCoFFEA attack results (e.g., $TPR \approx 99\%$ and $BDR \approx 95\%$ for a certain threshold) comparing to Oh et al. [16] with a real traffic dataset. This shows a difference of approximately 19% over the highest TPR obtained by Oh et al. simultaneously with $BDR \geq 95\%$, which is approximately 80%. This increased accuracy is likely due to our simulated network collection method. Although not optimal, it serves as a baseline for comparison.

C. Traffic Modulation Results

Mean and Standard Deviation of Delays: We experimented with varying mean and Standard Deviation (SD) values to assess their impact on the effectiveness of the DeepCoFFEA attack and system performance.

Larger mean delays were found to decrease the accuracy of the attack, with a mere 10 ms increase in mean delay size, from 20 ms to 30 ms, reducing the maximum F1-score by $\approx 14\%$ and P4 by $\approx 8\%$. We also found that the SD had minimal and inconsistent impact on accuracy, with results fluctuating in both directions by less than 1%. Additionally, as expected, higher mean delays led to longer transfer times, with the previously mentioned increase in mean delay size resulting in a 33% increase in the transfer time of 50KB. Additionally, as shown in Figure 6, we found that, comparing the usage of Tor vanilla with the usage of a modulation function with 20 ms of mean delay size, there is an increase of $\approx 40s$ in the transfer time of 1MiB of data. The SD, however, seemed to have negligible effects on performance. Although these performance results may seem unacceptable when considered in isolation, they must be interpreted as a trade-off for the reduction in attack accuracy achieved.

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. The 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 different parameters on multiple datasets.

We found that while, for the Tor Vanilla dataset increasing the window size and overlap parameters to 7s and 4s, respectively, reduced the accuracy of the attack slightly, for all other datasets where traffic modulation was used, the accuracy of the attack instead increased significantly. Specifically, the Tor Vanilla results revealed a decrease in maximum F1-score of 6.42% and in P4 of 3.36%, while the $Normal(\mu = 30, \sigma = 10)$ revealed an increase in maximum F1-score of 10.29% and in P4 of 6.11%. However, if the increase in these parameters is too high, the accuracy results start becoming worse again. The results we obtained suggest that the optimal parameters of the attack, when using most modulation functions, may be close to 7s and 4s, for the window size and overlap parameters, respectively.

Other Modulation Functions: We examined diverse modulation functions, including the Uniform, Normal, Poisson, Exponential, and Lognormal functions. The results of the various functions tested can be seen in Table II.

Our analyzes revealed that the mean delay size remained as the primary factor influencing attack accuracy, with modulation functions with similar means yielding similar accuracy results. Additionally, performance consistency was observed among modulation functions with the same mean delay. Interestingly, we also found a pattern where wider delay value ranges seemed to reduce the attack accuracy, with the main example of this being the Exponential function, which is able to generate delays of up to 100ms.

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

Table II
MAXIMUM F1-SCORE AND P4 VALUES OBTAINED FOR MODULATION FUNCTIONS CONFIGURED TO RESULT IN A MEAN DELAY OF $20ms$.

Cover Configuration	Modulation Function	Max F1-score	Max P4
None	$Lognormal(\mu = 2, \sigma = 0.5)$	96.98%	98.47%
500KB per 15s, 2 threads	None	59.28%	74.43%
500KB per 15s, 2 threads	$Lognormal(\mu = 2, \sigma = 0.5)$	20.89%	34.55%
500KB per 15s, 2 threads	$Lognormal(\mu = 1.5, \sigma = 0.5)$	29.15%	45.14%
250KB per 15s, 2 threads	$Lognormal(\mu = 1.5, \sigma = 0.5)$	39.05%	56.16%

Table III
MAXIMUM F1-SCORE AND P4 VALUES OBTAINED FOR EACH TECHNIQUE INDEPENDENTLY AND WHEN USING BOTH TECHNIQUES SIMULTANEOUSLY.

D. Cover Traffic Results

Amount of Traffic per Request: We tested cover traffic amounts (10KB, 100KB, and 1MB every 15 seconds) to analyze their effects on the DeepCoFFEA attack accuracy. As expected, the results show that the more cover traffic is generated, the less accurate the attack becomes. However, it is interesting to note how small of a difference each increase in the amount of cover traffic makes, when compared with the difference between the results of Tor Vanilla and the lowest amount of cover traffic, which reveal a massive drop of 22.32% in maximum 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 maximum 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.

Additionally, these amounts of cover traffic tested have a negligible impact on performance, as the transfer times are very similar to each other and to those obtained with the Vanilla simulation. 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.

Period of Requests: We also experimented with the period of requests parameter, which mainly affects the frequency of occurrence of peaks in traffic sent by the OS. We performed experiments for periods of 15, 10, and 5 seconds, all with requests of 100KB. 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. This is due to the use of a single thread to generate cover traffic requests, which results in requests constantly being timed out before being able to transfer any significant amount of cover traffic. These timeouts occur because they are configured to be equal to the period of requests, as it is not possible to send a new request without aborting the previous one.

Concurrent Requests: A possible solution for the limitation of the 5-second period 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. We tested a configuration of 2 threads generating 500KB every 15s, with a time gap of 7.5s between requests across both threads.

Comparing these results to the 1MB every 15-second single-threaded setup, the accuracy of the attack is significantly reduced. Specifically, this approach reduced the maximum F1-score by 10.7% and P4 by 7.9%. By distributing cover traffic more evenly, it creates fewer moments with no cover traffic being generated and where an attacker may be able to observe and learn real traffic patterns.

E. Full System Results

In this section, we discuss the experimental results of Shaffler, using traffic modulation and cover traffic.

In Section VI-C, we hypothesized that preserving the order of packets in traffic modulation allowed the DeepCoFFEA attack to identify patterns easily by combining volume analysis with timing analysis. While the obvious solution 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 circuit's ingress. 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 tested both techniques independently and together using the same parameters. For this purpose and based on our hypothesis that small delays would be enough to significantly improve traffic correlation protection, we selected a modulation function with low mean delay size, and consequently a low impact on performance, $Lognormal(\mu = 2, \sigma = 0.5)$. Regarding the cover traffic configuration, we used the same configuration that resulted in the lowest accuracy results in Section VI-D: 2 threads requesting 500KB every 15s each.

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

Complementary techniques: By comparing the results of the three experiments, we observed that both techniques seem to complement each other, as the results obtained with both techniques are significantly better than the results where each technique was used independently, achieving an impressive maximum F1-score of 20.89% and P4 of 34.55%. Furthermore, the effectiveness of this approach is highlighted by the impact even small delays (as low as 7 milliseconds) have on the accuracy of the attack, suggesting that our hypothesis might be correct and that small delays seem to be enough to disrupt timing analysis.

Although the results obtained reveal an outstanding protection against traffic correlation, some might consider it more than necessary, especially due to the significant impact on performance. As such, we decided to also 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. Additionally, we also attempted to reduce the amount of cover traffic generated, from 500KB each 7.5 seconds to 250KB each 7.5 seconds.

Adjustability: The results of these experiments, which can also be seen in Table III, show us that there is room in the Shaffler configuration parameters to adjust the trade-off between protection and performance. 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.

VII. Conclusions

Our study focuses on countering traffic correlation attacks in the Tor network, a threat to user anonymity. We introduce Shaffler, employing traffic modulation and cover traffic generation techniques. Implemented in a Tor version compatible with the existing infrastructure and with configurable parameters, Shaffler significantly reduces the effectiveness of the attack. However, this improvement in security comes with a performance cost. We also explored various statistical distributions for traffic modulation and cover traffic generation, revealing their complementary benefits. Shaffler allows users to balance protection and performance, offering a solution against traffic correlation attacks.

VIII. Future Work

In the present work, we developed Shaffler. Moving forward, there are areas for improvement and further research. Future work includes overcoming the limited time resolution of delays imposed by timers, providing user-friendly configuration through automated parameter adjustments based on predefined modes, adapting system parameters to real-time network conditions by monitoring the latency of loop traffic, and testing Shaffler against other traffic correlation attacks beyond DeepCoFFEA. Furthermore, while our current evaluation relied on simulations, validating our results through emulation experiments is a crucial step.

Acknowledgments

This work was supported by national funds through IAPMEI via the SmartRetail project (ref. C6632206063-00466847). Parts of this work have been performed in collaboration with other members of the Distributed Systems Group at INESC-ID, namely, Afonso Carvalho, Diogo Barradas and Kevin Gallagher.

References

[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: USENIX Association, Aug 2004. [Online]. Available: 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)*, 2018, pp. 108–126.

[10] 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.

[11] 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*, ser. SOSP '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 137–152. [Online]. Available: https://doi.org/10.1145/2815400.2815417

[12] A. Kwon, H. Corrigan-Gibbs, S. Devadas, and B. Ford, "Atom: Scalable anonymity resistant to traffic analysis," *CoRR*, vol. abs/1612.07841, 2016. [Online]. Available: http://arxiv.org/abs/1612.07841

[13] 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: USENIX Association, Aug 2017. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/piotrowska

[14] C. Diaz, H. Halpin, and A. Kiayias, "The nym network," 2021.

[15] 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*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3243734.3243824

[16] S. E. Oh, T. Yang, N. Mathews, J. K. Holland, M. S. Rahman, N. Hopper, and M. Wright, "Deepcoffea: Improved flow correlation attacks on tor via metric learning and amplification," in *2022 IEEE Symposium on Security and Privacy (SP)*, 2022.

[17] T. Project, "torrc(5)," https://manpages.debian.org/testing/tor/torrc.5.en.html, Jan 2023, accessed: 2023-08-03.

[18] M. Sitarz, "Extending f1 metric, probabilistic approach," 2022.

[19] G. Van Rossum and F. L. Drake, *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009.

[20] D. Song, "dpkt," https://dpkt.readthedocs.io/en/latest/, Aug 2022, accessed: 2023-08-03.