State Machine Frameworks for Website Fingerprinting Defenses: Maybe Not

Ethan Witwer

Submitted under the supervision of Nicholas Hopper to the University Honors Program at the University of Minnesota-Twin Cities in partial fulfillment of the requirements for the degree of Bachelor of Science, *summa cum laude* in Computer Science.

2023

Abstract

Tor is an anonymity network used by millions of people every day to evade censorship and protect their browsing activity from privacy threats such as mass surveillance. Tor conceals communication metadata, which indicates who is sending messages to whom, or which websites a user is visiting. Unfortunately, though, Tor has been shown to be vulnerable to website fingerprinting attacks, in which an adversary observes the connection between a user and the Tor network and leverages features of the encrypted traffic, such as the timing and volume of packets, to identify the websites that are being visited. This undermines the protection goals of Tor and puts its users at risk of exposure.

In response, researchers have proposed a number of defenses against website fingerprinting attacks, and a "circuit padding framework" has been added to the Tor software which supports the implementation of defenses. However, many proposed defenses cannot be implemented with this framework, because it requires defenses to be modeled as state machines which probabilistically send padding packets through a Tor circuit. Since padding packets look like normal packets to an observer but contain no useful data, they can be used to conceal traffic patterns that are taken advantage of by attacks; but a large number of defenses are deterministic, have complex behavior, or involve delaying packets, which is not supported by the circuit padding framework. Additionally, no padding-only defenses have been shown to be effective enough to merit the overhead that they incur, so none are currently implemented in Tor.

As Arti, a reimplementation of Tor in the Rust programming language, is being developed, the issue arises of whether a new state machine framework should be included or if alternative models should instead be considered for future defense implementation. We address this question by using an improved Rust-based state machine framework, Maybenot, to implement three state-of-the-art website fingerprinting defenses: FRONT, RegulaTor, and Surakav. By evaluating our implementations in terms of their similarity to the simulated versions of these defenses, overhead, and protection against attacks, we

demonstrate the potential of state machine frameworks to support effective defenses, and we highlight important features that they should contain to do so. However, our evaluation also raises uncertainty about the long-term feasibility of state machine frameworks for defense implementation. We recommend enhancements to Maybenot and substantial further evaluation, along with consideration of alternative designs, before any decision is made regarding a mechanism for implementing website fingerprinting defenses in Arti.

Summary

1 Introduction

In the face of steadily increasing mass surveillance and threats to online privacy, many people are turning to privacy-enhancing technologies such as Tor to protect their browsing activity. Tor is an anonymity network that aims to hide communication metadata, which reveals who is communicating with whom, or which websites a user is visiting [5]. Tor serves millions of users every day, including journalists, activists, whistleblowers, and ordinary people with an interest in protecting their privacy [26, 27].

Tor works by routing connections through a *circuit* consisting of three relays—guard, middle, and exit—using layered encryption so that each relay only sees the IP addresses of the previous and next hops in the circuit. The guard relay forwards traffic from the client to the middle relay; the middle relay forwards from the guard to the exit relay; and the exit relay forwards from the middle relay to the destination. A layer of encryption is stripped at each relay, revealing information about the next hop in the circuit. In this way, no single entity is aware of the identities of both the client and destination.

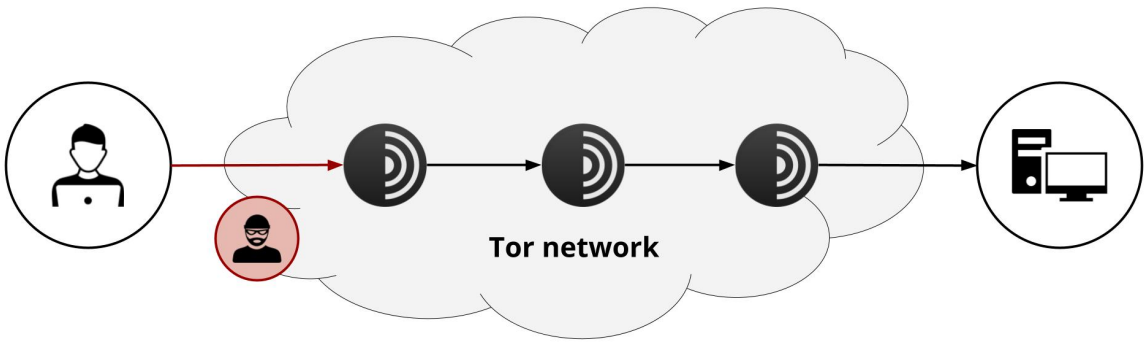


Figure 1: The website fingerprinting threat model in Tor

However, it has been demonstrated that Tor is vulnerable to a type of traffic analysis attack called *website fingerprinting*, in which a local, passive adversary identifies which websites a user is visiting by observing the traffic sent on the connection between the client and guard relay (Figure 1). In certain settings, website fingerprinting attacks have proved highly effective, identifying monitored web pages with greater than 98% accuracy [31, 35]. This defeats the goals of Tor and poses a significant threat to the privacy of its users,

making the need for effective defense mechanisms critical.

Tor's main codebase is written in C, and it contains a "circuit padding framework" that allows for the implementation of website fingerprinting defenses [25]. Defenses are modeled as state machines that control the probabilistic injection of padding packets onto the connection between a Tor client and relay, with the goal of concealing traffic patterns used in website fingerprinting attacks. However, many effective defenses that have been proposed by researchers cannot be implemented with this framework, notably those that involve delaying packets, which is not supported.

The Tor Project is currently developing Arti, a Rust implementation of Tor that will eventually completely replace the C code [19], and there are not yet any concrete plans for a website fingerprinting defense framework [21]. This raises the question of whether a state machine framework similar to the circuit padding framework should be included in Arti or if alternative designs should be explored. In this work, we address this issue by assessing the capability of state machine frameworks to support current state-of-the-art website fingerprinting defenses.

We consider a recently developed Rust-based framework for traffic analysis defenses called Maybenot [29]. Maybenot is a generalization of Tor's circuit padding framework and introduces new features including packet delays, making it more representative of the maximal capabilities of state machine frameworks. Thus, in working with Maybenot, we gather important insights into the potential of state machine frameworks in general to support proposed defenses.

We use Maybenot to implement three state-of-the-art defenses: FRONT [7], RegulaTor [13], and Surakav [8]. We compare our implementations to the simulated versions of these defenses and measure their overhead and protection against attacks. This evaluation demonstrates that state machine frameworks have the potential to support effective defenses, but it also brings up important issues and considerations related to their long-term viability for defense implementation, which are discussed in detail.

Overall, we offer the following main contributions:

- We provide two implementations of FRONT, one of RegulaTor, and a precursor to Surakav in the Rust-based state machine framework Maybenot. Their source code has been made publicly available on GitHub.
- We demonstrate that state machine frameworks can be used to implement effective website fingerprinting defenses, but we observe that certain features are needed to do so optimally and suggest improvements to the Maybenot framework.
- We discuss the implications of adopting a state machine framework for website fingerprinting defenses in Arti and recommend further evaluation and consideration of alternative designs before the decision to include one (or not) is made.

2 Background

Tor and Arti. Tor is an anonymity network that is designed to protect the metadata of its users' communications [5]. It is intended to ensure that no single entity can determine both the source and destination of any correspondence, meaning that the websites a user visits should not be discernible.

To achieve this, a client builds a *circuit* consisting of three intermediate *relays*—guard, middle, and exit—and establishes separate symmetric encryption keys with each one in a telescopic fashion. Data is sent through the circuit in 512-byte *cells* that are encrypted in layers, once with every key: each relay can only decrypt one layer, revealing the next relay to send the cell to (or the final destination, in the case of the exit relay).

The original Tor software was written in C, and it is still widely used as of 2023. However, in 2020, the Tor Project began working on Arti, a reimplementation of Tor in Rust [19]. Arti was intended to avoid common bugs in C programs and improve on the design of C Tor. It was deemed suitable for production use in 2022; though it still lacks some of the features available in C Tor, its eventual goal is to replace it entirely [20].

Website Fingerprinting. In website fingerprinting (WF) attacks, an adversary records the sequence of packets sent over the connection between a Tor client and guard relay, producing a *trace* of the traffic flow. Although the contents of packets and their ultimate destination are encrypted, the adversary can still deduce the websites a user has visited with a classifier which makes use of features such as the timing and volume of packets. These features form a unique *fingerprint* that is largely consistent for any given web page, making it possible to identify the page from a trace.

WF attacks are considered in the context of a local, passive adversary. The adversary is deemed *local* since he must only observe the connection between the client and guard relay. This can be done by a number of actors, including another user on the local network, a user's Internet service provider, or the guard relay itself. Moreover, the adversary is *passive* since he does not modify the traffic flow; he only needs the ability to observe

it. This scenario falls within Tor's threat model and represents a serious threat to the privacy of its users [5].

Evaluation of WF attacks is typically done in either the *closed-world* setting, in which an adversary has access to sample traces for all of the web pages that a user might visit; or the more realistic *open-world* setting, in which a user can additionally visit pages that are not known by the adversary. Unfortunately, it has been demonstrated that WF attacks are highly effective against Tor in both settings [1, 10, 23, 31, 35]. Other settings are even more favorable to attackers, such as the one-page setting proposed by Wang, in which an adversary attempts to detect visits to only a single web page [37].

Many effective WF attacks based on machine learning classifiers have been proposed. Panchenko et al. proposed CUMUL in 2016, which uses the "cumulative representation" of a trace along with an SVM classifier to achieve 92% accuracy in the closed-world setting [23]. In 2018, Sirinam et al. proposed Deep Fingerprinting (DF), an attack that uses deep learning to achieve 98% accuracy using only sequences of packet directions [35]. DF was extended by Rahman et al. in 2019 to include timing features, resulting in Tik-Tok [31]. Other classifiers have also obtained high accuracy against Tor [1, 10, 12].

Furthermore, although some of the assumptions made in attack evaluations have been criticized for being unrealistic [14], some work has demonstrated that certain assumptions are not required for WF attacks to be practical, such as [39]; and highly effective attacks have been carried out in the real world with a small set of monitored pages [3].

Defenses. A number of defenses against WF attacks have been proposed, most of which attempt to directly modify a trace in real time through a combination of sending padding packets and delaying packets. Some of these defenses are padding-only, meaning that they do not induce delay; they introduce extra packets into traces in ways that are intended to mask specific features used in attacks. WTF-PAD attempts to hide delays between packets that are unusually long ("statistically unlikely") [15], and another defense, FRONT, inserts padding near the beginning of a download based on the intuition

that the most useful features are present in the initial portion of download traffic [7]. Padding-only defenses are widely favored due to the assumption that they do not degrade user experience as much as those involving delay, but this is not true in practice [42].

Another class of defenses include delay as a crucial part of their design, and many of these are intended to "regularize" traffic traces, making them appear similar to all other traces or to others in an anonymity set. An extreme example of this is BuFLO, proposed in 2012, which sends traffic at a constant rate throughout an entire download, sending padding and delaying packets as necessary to do so [6]. In 2015, Wang presented an improved version of BuFLO called Tamaraw, which includes separate constant rates for download and upload traffic and pads the total length of a download up to a multiple of some chosen parameter [38]. This provides gains in overhead and protection against attacks, but BuFLO and Tamaraw are both impractical defenses due to their very high overhead. A newer regularizing defense, RegulaTor, sends traffic at an initial constant rate that decreases according to a decay function, achieving a high level of protection against attacks with more practical overhead requirements [13].

Some defenses shape traces in an attempt to make them look like other *specific* traces. Decoy, proposed in 2011, loads a second page in the background to confuse an attacker instead of directly padding or delaying packets [24]. In 2017, Wang and Goldberg described Walkie-Talkie, which makes pairs of web pages have the same trace by modifying the browser to communicate in a half-duplex mode and performing "burst molding" [40]. Larger anonymity sets can also be used: Glove, proposed by Nithyanand et al. in 2014, morphs all traces in a computed "cluster" into a single "supertrace," rendering them mutually indistinguishable [22]; a similar approach is taken by Supersequence, also presented in 2014 [41]. These defenses are limited by the fact that they require prior knowledge of web page traces, and *which* pages are grouped together in an anonymity set can have a significant impact on their efficacy [31, 37]; a recent defense, Surakav, avoids the latter issue by employing adversarial machine learning techniques to generate fake traces and shaping traffic to make it appear similar to these traces [8].

The defenses summarized so far all operate at the network layer, directly modifying traces; still more defenses work at the application layer and take advantage of the specifics of the protocol in use to defend against attacks. For instance, HTTPOS manipulates the TCP window and downloads web objects in separate chunks via the HTTP Range header and HTTP pipelining to confuse attacks [16]; however, it has been shown to be ineffective [2]. Other network-layer techniques are also possible, such as splitting traffic over multiple Tor circuits or network links, approaches which are taken by TrafficSliver [4] and HyWF [11], respectively. In this work, we only consider standard network-layer defenses, as they provide broader protection and are suitable for direct implementation in Tor. We implement FRONT, RegulaTor, and Surakav as exemplars of the three categories of network-layer defenses that have been discussed.

Defense Frameworks. Shmatikov and Wang proposed *adaptive padding* in 2006 to defend against timing analysis attacks in mix networks [34]. This technique involves maintaining a likely distribution of inter-packet time intervals. When a packet is received by a mix, an interval is sampled: if another packet arrives before the interval expires, it is forwarded and a new interval is sampled; otherwise, a padding packet is sent before sampling another interval. This has the effect of making inter-packet delays roughly follow an expected distribution. Adaptive padding can also operate in a dual mode, which minimizes padding sent within bursts of traffic and focuses on protecting gaps between bursts. In this mode, when a packet is received by a mix, a larger interval is sampled; if an interval expires and padding is sent, a shorter one is sampled next.

The dual mode of adaptive padding (and the basic algorithm) is implemented using histograms: when the first packet of a connection is received by a mix, a histogram is constructed with bins representing ranges of inter-packet delays. Each bin is filled with *tokens*: a configurable number of values are sampled from the inter-packet interval distribution, and if a value falls within a bin's range, a token is added to that bin. Then, when a packet is received by the mix, a random token is selected and removed from its

bin, and an inter-packet interval is sampled from the bin's range. If the interval expires without any traffic being received, a padding packet is sent; otherwise, the received packet is forwarded, the removed token is returned to its bin, and a token is removed from the bin corresponding to the observed inter-packet delay. In the former case, the next interval is sampled from the "low-bins set;" otherwise, one is sampled from the "high-bins set."

In 2016, Juarez et al. extended the dual mode of adaptive padding to develop the WF defense WTF-PAD [15]. This included (1) adding control messages so that the client is in charge of padding decisions and (2) allowing padding to be sent in response to packets received *and* sent, since Tor clients do not forward traffic as do mixes; separate distributions are maintained for this purpose. A circuit padding framework based on WTF-PAD was developed for C Tor in 2019 [25]. Clients can negotiate the use of padding state machines with any relay in a circuit, and padding will be sent probabilistically according to the state machine in use; histograms can be used along with parameterized probability distributions, as was done by Pulls in APE, a defense similar to WTF-PAD [28]. However, packet delays are not supported nor are particularly complex behaviors.

Besides APE and WTF-PAD, which has been defeated [35], a few other WF defenses have been developed for the circuit padding framework. In 2018, Mathews et al. presented the Random Extend Bursts defense, which adds padding to *bursts* of cells (uninterrupted sequences of cells sent in a single direction on a connection), but it is not effective against DF [17]. Pulls used genetic algorithms with the circuit padding framework in 2020 and manually modified the best evolved state machine to develop the defenses Spring and Interspace, which are effective against DF but have prohibitive bandwidth overhead [30]. No WF defenses are currently deployed in Tor.

Maybenot. In this work, we use Maybenot [29], a more recent framework written in Rust. Maybenot is a generalization of Tor's circuit padding framework; it allows for more sophisticated defenses, eliminates histograms in favor of parameterized probability distributions, and includes support for packet delays (temporarily blocking traffic from

being sent). An application using Maybenot provides as input *events* related to an encrypted communication channel, such as receiving or sending a packet, and is presented with *actions* that should be taken to defend the channel, i.e., sending padding or blocking outgoing traffic. *Machines* define which action to take when a given event occurs. Several machines may operate in parallel to build more complex defenses.

Each machine consists of a number of *states*, which are characterized by an action, three distributions, and vectors of probabilities for state transitions. The action is either to send padding or block outgoing traffic. An action distribution specifies either the amount of padding or the duration of blocking; a timeout distribution is used to sample the time that should pass before the action is applied; and a limit distribution indicates how many self-transitions can occur before a *LimitReached* event is triggered. A map is maintained between each event and a corresponding state transition vector, allowing different events to have their own probabilities of transitioning to any given state. More specific features of Maybenot will be discussed as relevant throughout the paper.

3 Implementing FRONT

3.1 Description

The FRONT defense is based on two primary observations: (1) the beginning of a trace contains most of the features used in WF attacks, and (2) trace-to-trace randomness can be employed to create strong defenses, as it reduces an attacker's ability to effectively train a classifier on defended traces [7]. Thus, a high volume of padding cells are sent at the beginning of each trace, and the number and timing of these cells differ among traces.

This is accomplished by using a Rayleigh distribution to schedule padding cells before a download begins. A number of time values are sampled, and a padding cell is then sent at each of these times relative to download start. Trace-to-trace randomness is achieved by varying the distribution's scale parameter and the number of values sampled among downloads. The entire sequence of steps involved in FRONT is as follows:

1. A padding count is sampled from a discrete uniform distribution: n_c is sampled from the range $[1, N_c]$ by the client, and n_s is sampled from $[1, N_s]$ by the relay. The parameters N_c and N_s are the client and relay padding budgets, respectively.

2. A padding window, which is the scale parameter σ of the Rayleigh distribution, is sampled from a continuous uniform distribution: w_c is sampled from $[W_{min}, W_{max}]$ by the client, and w_s is sampled from the same range by the relay.

3. Padding cells are scheduled: n_c values are sampled from the Rayleigh distribution by the client, and n_s values are sampled by the relay.

4. During the download, a padding cell will be sent at each sampled time. No further padding is sent after the download completes.

3.2 First Machine: Maybenot FRONT

We sought to implement FRONT in Maybenot using a single machine design, since both client and relay perform the same straightforward sequence of steps. Because padding cells are scheduled *before* a download starts in FRONT, this could not be done directly; instead, we aimed to approximate the sending rate of padding cells that would result from sampling time values from a Rayleigh distribution.

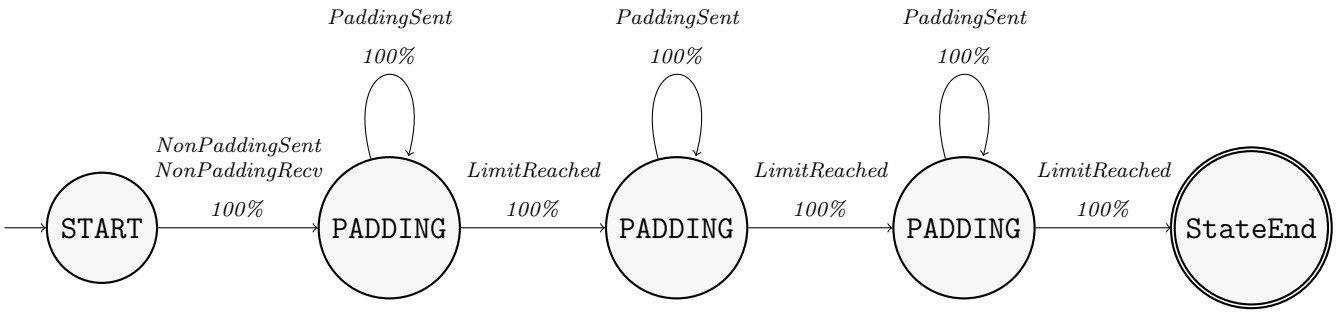


Figure 2: Maybenot FRONT machine with three `PADDING` states

Our first FRONT machine design consists of a `START` state, a number of `PADDING` states, and the pseudo-state `StateEnd` provided by the Maybenot framework, as shown in Figure 2. Each machine is characterized by its padding budget N, maximum padding window W_{max}, and number of `PADDING` states ψ. We call this design Maybenot FRONT.

The Maybenot FRONT machine transitions from `START` to the first `PADDING` state when a download begins, which is considered to occur when the first non-padding cell is sent or received (when a *NonPaddingSent* or *NonPaddingRecv* event is triggered). It then proceeds sequentially through the remaining `PADDING` states until it reaches `StateEnd`.

As their name suggests, `PADDING` states generate *Padding* actions; a uniform action distribution with parameters $a = 512$ and $b = 512$ is used so that a single padding cell is sent per action. The delay before a cell should be sent is sampled from a normal timeout distribution, and trace-to-trace randomness is achieved with a uniform limit distribution.

When a `PADDING` state is transitioned to, it generates a *Padding* action with a timeout value sampled from its timeout distribution. The application using Maybenot (i.e., Arti) will send a padding cell after this timeout expires and trigger a *PaddingSent* event,

causing a self-transition. This will occur repeatedly until the state's limit is reached, at which time a *LimitReached* event will be triggered, causing a transition to the next state.

Each `PADDING` state is modeled as corresponding to a fixed time slice of a download; its timeout distribution parameters are selected to approximate the distribution of inter-packet delays that would result during that interval if time values were sampled from a Rayleigh distribution with $\sigma = W_{max}$. In a machine with ψ `PADDING` states, the timeout distribution parameters of a `PADDING` state that spans the interval from t_1 to t_2 are:

$$\mu = \frac{\psi}{N} \cdot (t_2 - t_1) \tag{1}$$

$$\sigma = \frac{W_{max}^2}{\sqrt{\pi}} \cdot (\frac{N}{\psi} \cdot \frac{t_1 + t_2}{2})^{-1} \tag{2}$$

μ is selected to be the inter-packet delay that would result in exactly N/ψ cells being sent during the interval from t_1 to t_2. The equation for σ is partially derived from the results of preliminary simulations and trace comparisons; it allows for greater variation of inter-packet delays near the beginning of a download, and variation is increased for larger values of W_{max}. To prevent excessive variation, timeout values are bounded to be in the range $[0, 2 \cdot \mu]$ by specifying a `max` parameter for the timeout distribution.

If each state had a constant limit corresponding to the number of cells that would most likely be sent during its interval, this would allow for a precise approximation of the sending rate of padding cells that would result from a Rayleigh distribution. However, such a design would not account for the trace-to-trace randomness FRONT is intended to achieve: a machine's padding count would be constant, and variation of the padding window would be small and only due to differences in sampled timeout values.

To mimic the sampling performed by FRONT, we instead use a uniform distribution with range $[1, N/\psi]$ for each state's limit. Thus, the padding count for a download is

effectively sampled from a uniform *sum* distribution with range $[\psi, N]$. Note that this change allows for variation in the padding count as well as the padding window, as the times at which state transitions occur become more variable.

3.3 Second Machine: Pipelined FRONT

A limitation of Maybenot FRONT is that the timeout distribution parameters of `PADDING` states are calculated using values that are fixed for each machine. Although the padding count and window do vary among downloads, inter-packet timing is less variable, which reduces the efficacy of the defense. To remedy this, we introduce Pipelined FRONT, a machine based on the same principles as Maybenot FRONT but with multiple *pipelines* that have different padding budgets, as illustrated in Figure 3.

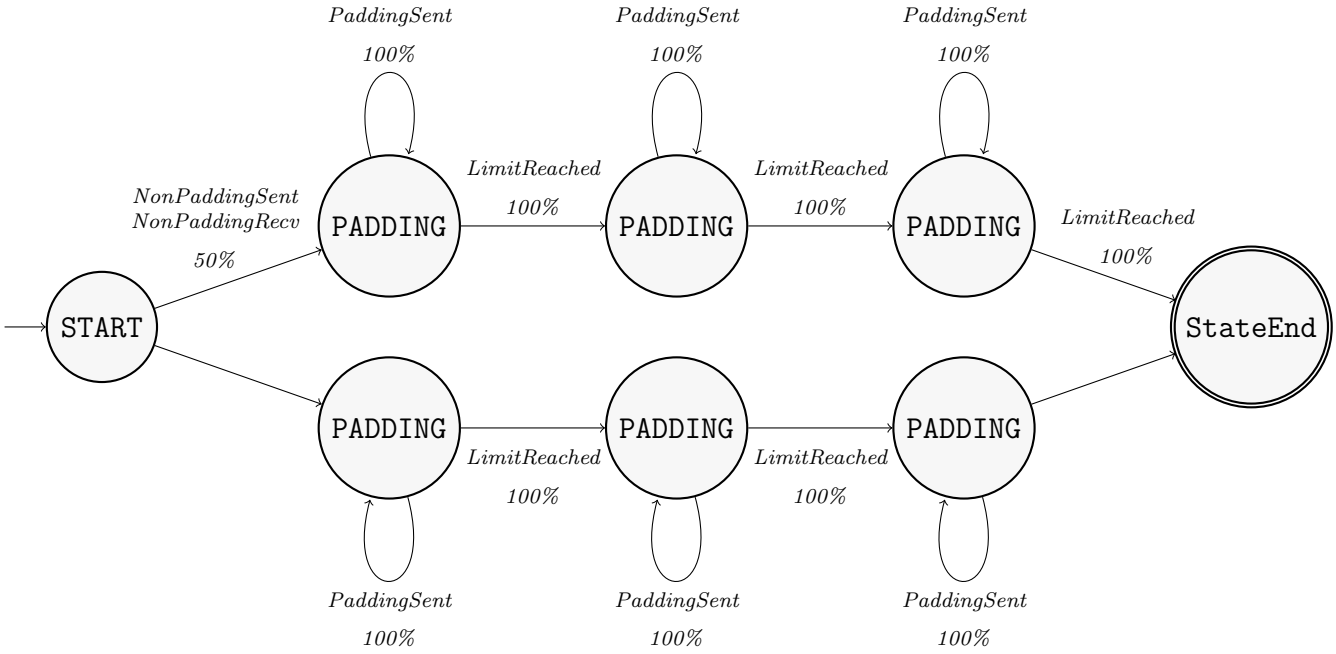


Figure 3: Pipelined FRONT machine with two pipelines, three `PADDING` states each

In this machine, the first state to transition to is chosen from a set of `PADDING` states which all have equal probability, and each one leads to a different pipeline. This allows for variation of the padding count and window, as with Maybenot FRONT, as well as inter-packet timing, which greatly improves trace-to-trace randomness.

3.4 Evaluation

3.4.1 Experimental Setup

In our evaluations, we used the BigEnough dataset collected by Mathews et al. between November 2021 and January 2022 [18]. Specifically, we used the monitored set, which consists of 19,000 traces collected from 95 websites. To build the monitored set, 10 subpages from every website were visited 20 times each, for a total of 200 traces per site. This was done using the "Safest" configuration in the Tor Browser.

We used the simulation scripts provided by Gong et al. [7] to produce FRONT-defended traces and the Maybenot simulator [29] to defend traces with Maybenot FRONT and Pipelined FRONT. A delay of 10 ms was simulated between the client and server by the Maybenot simulator; the simulator uses delays to set up event queues, and the value selected had a marginal impact on the resultant defended traces.

Using the FRONT simulation scripts, we generated two defended datasets, one for each of the two configurations of FRONT presented by Gong et al.: `FT-1`, a lightweight configuration with $N_s = N_c = 1700$, $W_{min} = 1s$, and $W_{max} = 14s$; and `FT-2`, with $N_s = N_c = 2500$ and the same window parameters as `FT-1` [7].

We produced four defended datasets with the Maybenot simulator: Maybenot `FT-1`, Maybenot `FT-2`, Pipelined `FT-1`, and Pipelined `FT-2`. The Maybenot simulator accepts machines in their serialized form (character strings) and defends input traces with them; to obtain Maybenot FRONT and Pipelined FRONT machines, we developed two simple

Defense	Parameters			
	N	W_{min}	W_{max}	ψ
Maybenot FT-1	1500	1 s	14 s	30
Pipelined FT-1	3000	1 s	14 s	30×30
Simulated FT-1	1700	1 s	14 s	—
Maybenot FT-2	2500	1 s	14 s	50
Pipelined FT-2	4500	1 s	14 s	45×45
Simulated FT-2	2500	1 s	14 s	—

Table 1: FRONT parameters

Rust programs that generate them based on supplied parameters.

We chose parameters for our implementations to match the bandwidth overhead incurred by simulated FRONT; the final parameters selected for each implementation are summarized in Table 1. Note that, for Pipelined FRONT, ψ represents number of pipelines followed by the number of `PADDING` states per pipeline.

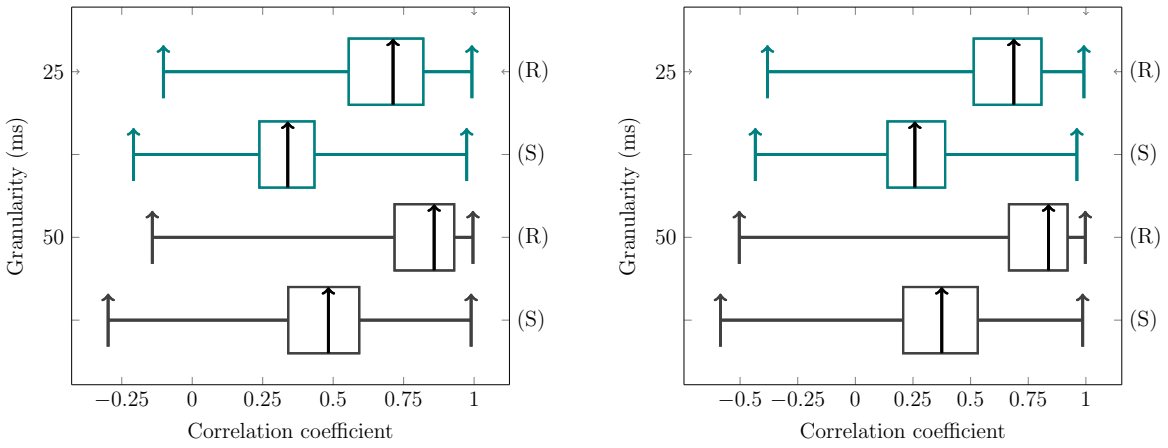
After running the Maybenot simulator, we removed trailing padding cells from each trace in the four defended datasets to better compare with simulated FRONT. Our implementations do not include any mechanism to detect the end of a download, and Maybenot provides no such event; this issue is discussed in Section 6.2.

3.4.2 Trace Comparison

We determined the similarity of corresponding traces defended with simulated FRONT and our machines, adopting the methodology of Smith et al. [36] We represented each trace with two *aggregated time series*, one for upload traffic and another for download traffic. We computed these by partitioning total download time into fixed-length windows of I ms and creating sequences consisting of the total number of bytes sent and received by the client during each window. We set $I = \{25, 50\}$ for our evaluations.

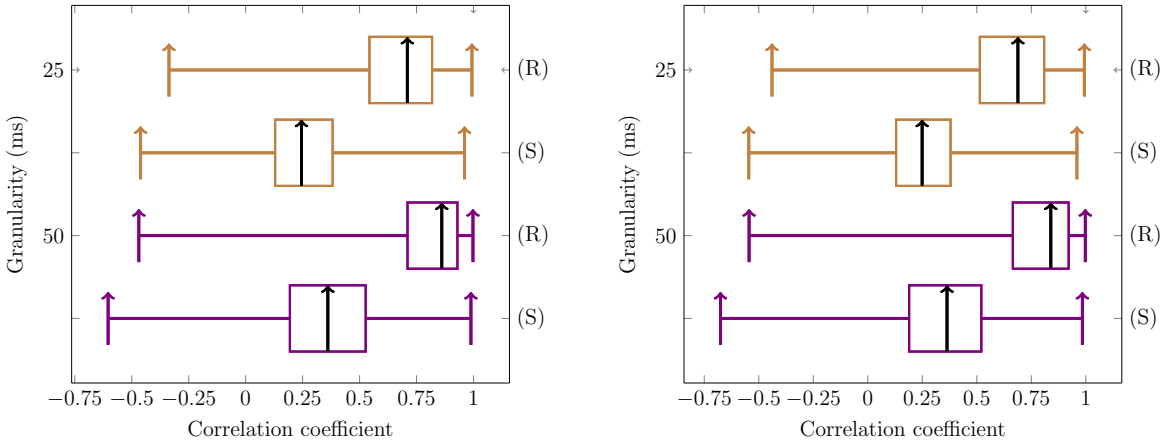
We compared corresponding traces by calculating the Pearson correlation coefficient and a longest common subsequence (LCSS) measure on their matching aggregated time series–once for upload traffic and again for download traffic. We calculated the LCSS measure by dividing the length of the longest common subsequence by the shorter of the lengths of the two aggregated time series being compared. Correlation coefficient results are shown in Figure 4 for Maybenot FRONT and Figure 5 for Pipelined FRONT. LCSS results are displayed in Figures 6 and 7.

The correlation coefficient data for both `FT-1` and `FT-2` indicates that Maybenot FRONT and Pipelined FRONT padded download traffic similarly to simulated FRONT in most cases. With the `FT-1` configuration, both defenses have a median correlation of approximately 0.71 at 25 ms granularity and 0.86 at 50 ms granularity. Interquartile



(a) `FT-1` configuration (b) `FT-2` configuration

Figure 4: Correlation coefficient, simulated FRONT and Maybenot FRONT



(a) `FT-1` configuration (b) `FT-2` configuration

Figure 5: Correlation coefficient, simulated FRONT and Pipelined FRONT

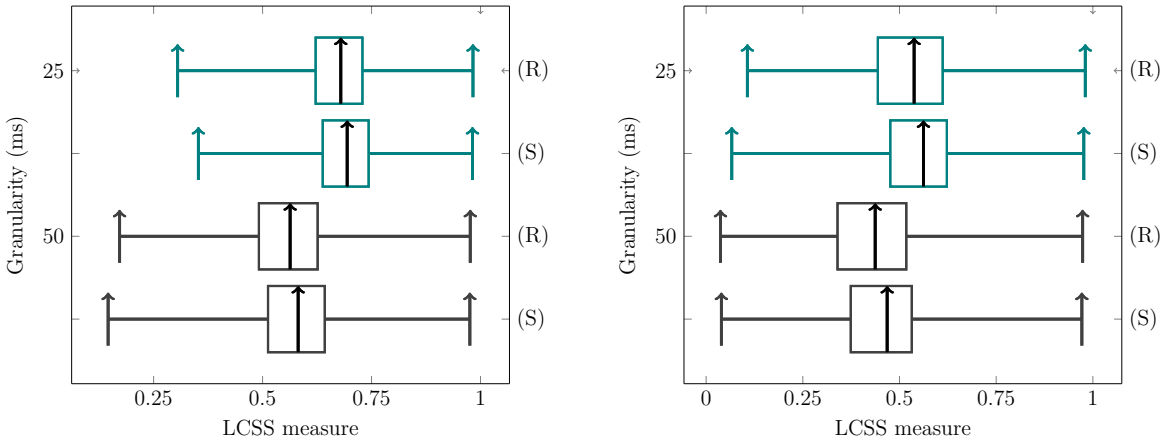
range is about [0.56, 0.82] with $I = 25$ and [0.72, 0.93] with $I = 50$ for Maybenot FRONT; it is nearly identical for Pipelined FRONT. Similar results are observed with `FT-2`.

This indicates a strong correspondence between our implementations and simulated FRONT for at least 75% of traces. However, a negative correlation is observed for some traces; this is likely due in part to the use of individual states' limit distributions to induce variation of the padding count and window. It is possible for the limit values selected for adjacent `PADDING` states to differ, which reduces correspondence to the target

Rayleigh distribution shape. We also note that each implementation may have selected different values for the padding count and window when defending the same trace, since simulations were run independently; the Appendix provides further discussion.

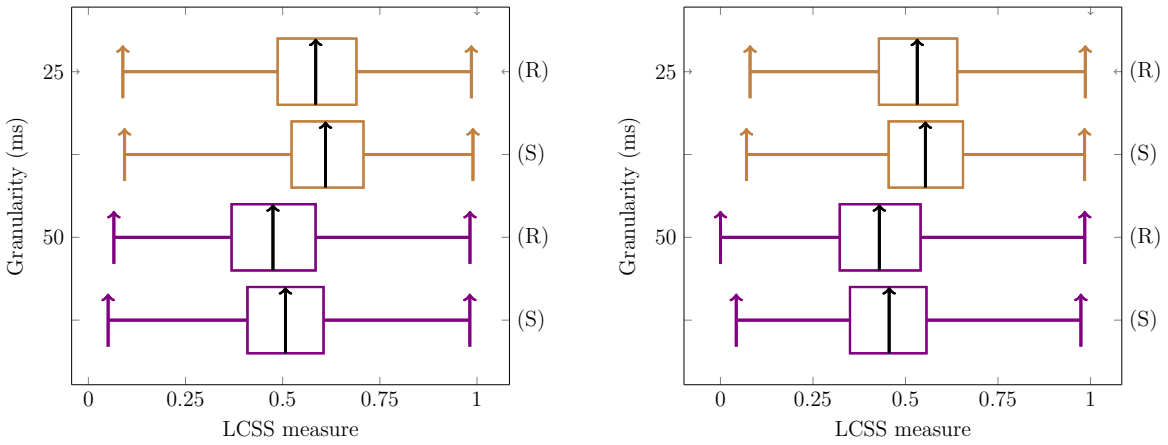
Upload traffic was not approximated as well as download traffic: with Maybenot `FT-1` and `FT-2`, there is a median correlation of about 0.34 when $I = 25$ and 0.48 when $I = 50$; these values drop to 0.25 when $I = 25$ and 0.35 when $I = 50$ with Pipelined FRONT. We attribute this to the higher ratio of padding cells to non-padding cells in upload traffic: in the case of download traffic, there is a higher density of non-padding cells, so any "misplaced" padding cells would have a smaller effect on the correlation coefficient.

The minimum median LCSS observed for download traffic is 0.53 when $I = 25$ and 0.43 when $I = 50$ with a narrow interquartile range. Upload traffic has a higher median LCSS in all cases despite lower correlation, which is likely due in part to small quantities of upload traffic resulting in aggregated time series with many low-valued windows. This would occur more often near the end of a trace when less traffic (both padding and non-padding) is sent, suggesting that most of the variation observed in the correlation results is due to differences in padding between implementations for the reasons described above.



(a) `FT-1` configuration (b) `FT-2` configuration

Figure 6: LCSS measure, simulated FRONT and Maybenot FRONT



(a) `FT-1` configuration

(b) `FT-2` configuration

Figure 7: LCSS measure, simulated FRONT and Pipelined FRONT

3.4.3 Overhead Measurement

We continue our evaluation with a comparison of the overhead of simulated FRONT, Maybenot FRONT, and Pipelined FRONT. We consider *bandwidth overhead*, which refers to the total number of padding bytes in a defended trace divided by the total number of non-padding bytes. We further distinguish between *receive* and *send* bandwidth overhead (from the client's perspective), counting cells in only one direction.

Latency overhead is another standard metric used in defense evaluations, but FRONT does not delay cells, so no latency overhead is observed in simulated traces. However, we note that "zero-delay" defenses *can* cause increased latency in live network deployment scenarios, as described in [42]. We leave this more robust evaluation to future work and focus here on comparing our implementations to FRONT's expected behavior and providing a preliminary idea of their cost.

Mean bandwidth overhead results are presented in Table 2. About 80% bandwidth overhead was incurred by `FT-1` and 125% by `FT-2`; there was little variation among implementations since the parameters of Maybenot FRONT and Pipelined FRONT were selected to match their bandwidth overhead to that of simulated FRONT.

To accomplish this for Maybenot `FT-1`, N was decreased from 1700 to 1500. This

Defense	Bandwidth overhead (%)		
	Send	**Receive**	**Overall**
Maybenot FT-1	597.51	41.84	78.24
Pipelined FT-1	613.45	43.13	80.49
Simulated FT-1	642.97	44.80	83.98
Maybenot FT-2	998.77	70.05	130.89
Pipelined FT-2	922.24	64.60	120.79
Simulated FT-2	952.91	66.24	124.32

Table 2: FRONT average bandwidth overhead

was necessary because the use of a separate uniform distribution for each `PADDING` state's limit effectively resulted in a uniform *sum* distribution for padding count, which has a higher expected value. This is also apparent with `FT-2`, since N was maintained at 2500, and this resulted in 6.57% greater bandwidth overhead than with simulated FRONT.

However, for Pipelined FRONT, N had to be increased from 1700 to 3000 for `FT-1` and from 2500 to 4500 for `FT-2`. We attribute this to the use of pipelines that are based on different padding budgets: there is only a $1/\psi$ probability of choosing a pipeline that can send N cells, and further reduction of padding count occurs within pipelines.

3.4.4 Attack Performance

We evaluated CUMUL [23], DF [35], and Tik-Tok [31] in the closed-world setting against undefended traffic, simulated FRONT, Maybenot FRONT, and Pipelined FRONT. We did this using the scripts provided by Gong et al. for CUMUL [7] and those provided by Rahman et al. for DF and Tik-Tok [31]. We performed 10-fold cross-validation for all attacks, and we used the model parameters suggested by the attacks' authors, with one exception: the input size of DF and Tik-Tok was changed from 5,000 to 10,000 cells to account for padding. The results are in Table 3.

All attacks achieved at least 94% accuracy on the undefended dataset; these results are similar to those reported by the attacks' authors, but slightly lower values are observed since each class in the BigEnough dataset consists of multiple web pages.

Simulated FRONT decreased CUMUL's accuracy to 12.06% with the `FT-1` configu-

ration and 9.12% with `FT-2`. It reduced the accuracy of DF and Tik-Tok to 48.32% and 49.47%, respectively, with `FT-1`; and it reduced their accuracy to 40.95% and 45.79% with `FT-2`. These values are used as a benchmark to evaluate our implementations.

Maybenot FRONT was much less effective than simulated FRONT: it reduced Tik-Tok's accuracy to 50.84% with the `FT-2` configuration, but it only decreased accuracy to a minimum of 64% in all other cases. Since there is a high correlation between simulated FRONT and Maybenot FRONT for download traffic but not for upload traffic, this seems to suggest that Maybenot FRONT did not conceal useful features of upload traffic.

However, Pipelined FRONT has a similarly low correlation with simulated FRONT for upload traffic, but it was much more effective than Maybenot FRONT against all attacks: DF was the best attack against it, attaining 58% accuracy with `FT-1` and 49.37% accuracy with `FT-2`. Thus, we caution that similarity metrics should not be used to conjecture about a defense implementation's protection against attacks; nevertheless, lower values of I might reveal more significant differences between implementations.

We attribute Pipelined FRONT's success to high variation in inter-packet timing, padding count, and padding window, confirming that trace-to-trace randomness can be leveraged to create an effective WF defense. However, simulated FRONT still provided the best protection, highlighting the limits of our approximation approach: FRONT can be implemented effectively with Maybenot, but not precisely.

Defense	Accuracy (%)		
	CUMUL	**DF**	**Tik-Tok**
Undefended	94.66	95.89	94.00
Maybenot FT-1	27.68	72.11	64.00
Pipelined FT-1	15.72	58.00	55.89
Simulated FT-1	12.06	48.32	49.47
Maybenot FT-2	23.41	68.11	50.84
Pipelined FT-2	13.45	49.37	48.32
Simulated FT-2	9.12	40.95	45.79

Table 3: FRONT performance in closed-world setting, BigEnough dataset

4 Implementing RegulaTor

4.1 Description

RegulaTor is based on the observation that Tor traffic consists of occasional "surges" of cells sent within a short period of time along with intervening periods of lower cell volume [13]. Surges are typically present at the beginning of a download, and the amount of traffic decreases exponentially as time elapses. Since most cells are sent in surges, these contain important features that can be leveraged by WF attacks.

In essence, RegulaTor is intended to regularize surges, thereby reducing their uniqueness and, consequently, usefulness as WF attack features. This is accomplished by sending download traffic at a set initial rate which decreases according to a decay function; if the number of queued cells exceeds a threshold, a new surge begins, and traffic is once again sent at the initial rate. Upload traffic is sent at a fraction of the rate of download traffic.

Provisions are also included to reduce overhead and ensure that progress is made. RegulaTor samples a padding count for each download, and it will stop sending padding cells to achieve a constant sending rate after this count has been exceeded, instead delaying non-padding cells to *cap* the sending rate. It will also send any queued upload cells immediately after they have been waiting for a configurable amount of time.

The entire sequence of steps involved in RegulaTor for the relay is:

1. Sample a padding count from the discrete uniform distribution $[0, N]$, where N is a parameter specifying the padding budget.

2. Wait until 10 cells are queued, then set the surge start time to the current time.

3. Set the target rate according to the decay function RD^t, where R is the initial rate, D is the decay parameter, and t is time elapsed since the surge start time (seconds).

4. If the number of queued cells is greater than a threshold parameter, T, multiplied by the target rate, reset the surge start time to the current time.

5. Send a cell if one is queued; otherwise, send a padding cell if the padding count has not yet been exceeded.

6. Repeat steps 3-5 every time a cell should be sent (determined by the target rate: every $rate^{-1}$ seconds) until the download is finished.

The client simply sends cells at a constant fraction of the rate of download traffic: one cell is sent for every U cells received. However, if any cells have been queued for more than C seconds, they will be sent at once so that downloads continue to make progress.

4.2 Maybenot RegulaTor

Two machines were created to approximate RegulaTor, one for clients and one for relays. We refer to these machines collectively as Maybenot RegulaTor.

These machines are based on the *bypass* and *replace* flags of states in Maybenot [29]. If a state with the *bypass* flag set enables blocking, then setting this flag in a padding state allows it to circumvent the blocking and send padding anyway. When *Padding* actions are generated by padding states with the *replace* flag set, the application is allowed to send a queued non-padding cell instead of generating a padding cell.

This combination allows for constant-rate traffic: blocking can be enabled with both flags set, and only padding generated by states with the *bypass* flag set will be allowed through. Such padding can be sent at a constant rate, and by additionally setting the *replace* flag, the application can send non-padding cells instead of padding cells whenever any are queued. When a padding cell is replaced, a *PaddingSent* event is generated as well as a *NonPaddingSent* event. We use this paradigm extensively in Maybenot RegulaTor.

4.2.1 Relay Machine

The relay machine can be seen as proceeding through three distinct stages: (1) infinite blocking is enabled with the *bypass* and *replace* flags set; (2) until 10 cells have been sent, a constant traffic rate of 10 cells/second is maintained; and (3) constant-rate `SEND` states

are used to approximate the sending rate imposed by RegulaTor's decay function. The design of this machine is depicted in Figure 8.

When the first *NonPaddingSent* event is triggered, the machine transitions to the `BLOCK` state, which enables infinite blocking with the *bypass* and *replace* flags set; this allows for constant traffic rates to be set later, as described previously. Once the application has carried out the blocking action, a *BlockingBegin* event will be triggered, causing the machine to transition to the `BOOT_0` state.

Each `BOOT` state generates a *Padding* action with the *bypass* and *replace* flags set and a 100 ms timeout. When the corresponding *PaddingSent* event is triggered, a self-transition occurs: this results in a constant traffic rate of 10 cells/second. When a *NonPaddingSent* event is triggered, a transition is made to the next `BOOT` state or, in the case of `BOOT_8`, the `SEND_0` state. Including the *NonPaddingSent* event that causes a transition to the `BLOCK` state, then, exactly 10 non-padding cells are sent before the `SEND_0` state is reached.

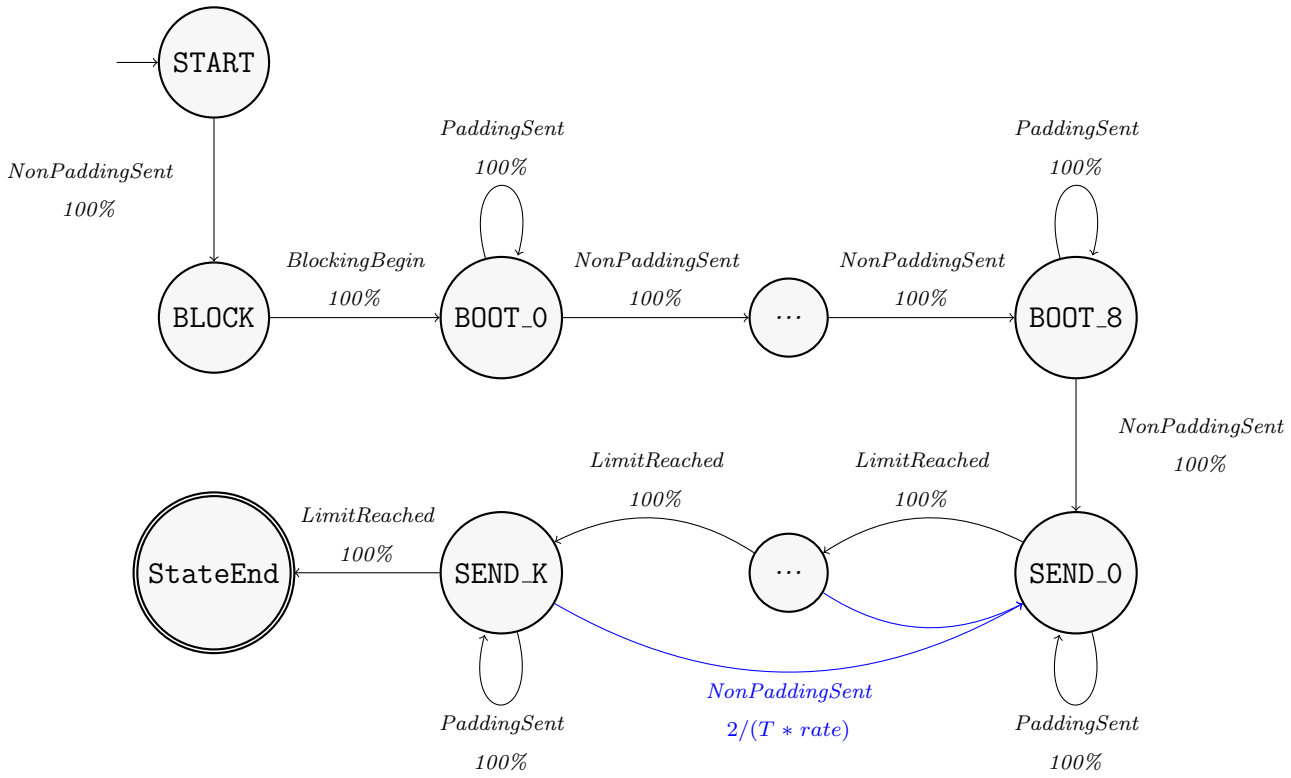


Figure 8: Maybenot RegulaTor relay machine with K `SEND` states

The SEND states each have the same limit and set a constant traffic rate (timeout) to approximate RegulaTor's decay function. RegulaTor also specifies that if a certain threshold of queued cells is exceeded, a new surge is said to have started and the rate should be increased back to its initial value. Since no queue-related events are present in Maybenot, we implement this behavior probabilistically with a small chance of transitioning back to SEND_0 when a *NonPaddingSent* event is triggered.

Our implementation also excludes the N parameter of RegulaTor, allowing machines to send an unlimited amount of padding during a download. Although Maybenot has features to set padding limits, these will prevent a machine from generating any actions [29]; there is no way to specify that the sending rate should be capped.

4.2.2 Client Machine

The client machine sends one cell for every U cells received. It consists of a configurable number of COUNT states arranged in sequence, which enable infinite blocking with the *bypass* and *replace* flags set, transitioning to the next state when a *PaddingRecv* or *NonPaddingRecv* event is triggered; and a single SEND state, which generates a *Padding* action with no timeout and the *bypass* and *replace* flags set, transitioning to the first COUNT state when a *PaddingSent* event is triggered.

If U is a whole number, this machine consists of U COUNT states that each have a 100% probability of transitioning to either the next COUNT state or, in the case of the last COUNT state, the SEND state when a *PaddingRecv* or *NonPaddingRecv* event is triggered.

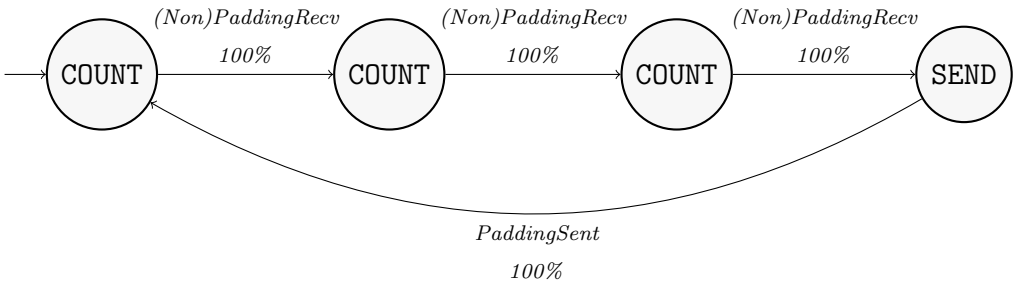


Figure 9: Maybenot RegulaTor client machine with $U = 3$

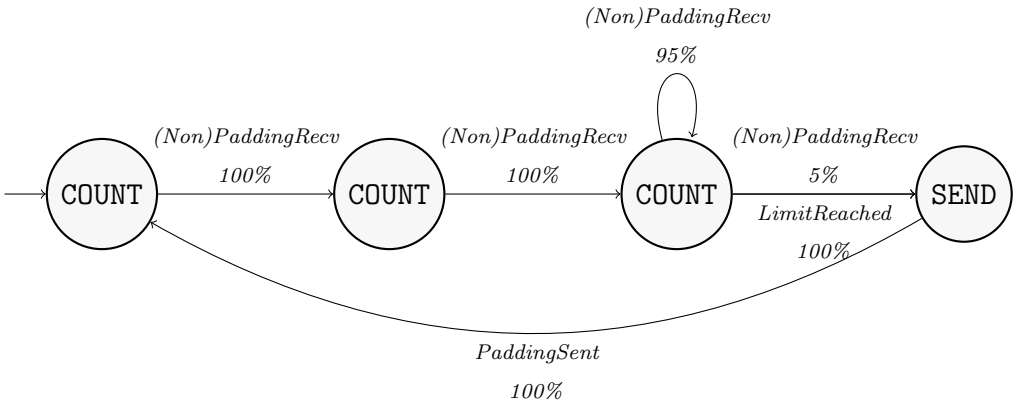


Figure 10: Maybenot RegulaTor client machine with $U = 3.95$

Thus, exactly one cell is sent for every U cells received; this is the case in Figure 9.

If U is *not* a whole number, which is acceptable in RegulaTor, there are $\lfloor U \rfloor$ `COUNT` states, and the probability of transition from the last `COUNT` state to the `SEND` state is set to $1 - (U - \lfloor U \rfloor)$; if this does not occur, a self-transition does. The next cell received will cause an immediate transition to `SEND`: the limit for the last `COUNT` state is fixed at 2, and the *LimitReached* event causes a transition to `SEND` with 100% probability.

This design is intended to probabilistically approximate the expected behavior of non-integral values of U; that is, it should still be the case that one cell is sent for every U cells received on average. Thus, if the fractional part of U is 0.95, there will be a 5% chance of reaching `SEND` after $\lfloor U \rfloor$ cells are received and a 95% chance of reaching `SEND` after $\lceil U \rceil$ cells are received. This situation is depicted in Figure 10.

While both of these machines effectively mimic the RegulaTor client's behavior, they do not include the C parameter, which determines the maximum amount of time that a cell can be queued for before being sent immediately. Thus, a cell might be queued indefinitely, which could result in download progress being slower than with an exact implementation of RegulaTor.

Defense	Parameters						
	R	D	T	N	U	C	ω
Maybenot RT-Light	324	0.86	3.75	—	4.02	—	20
Simulated RT-Light	206	0.86	3.75	1650	4.02	2.08	—
Maybenot RT-Heavy	238	0.94	3.55	—	3.95	—	20
Simulated RT-Heavy	220	0.94	3.55	2815	3.95	1.77	—

Table 4: RegulaTor parameters

4.3 Evaluation

4.3.1 Experimental Setup

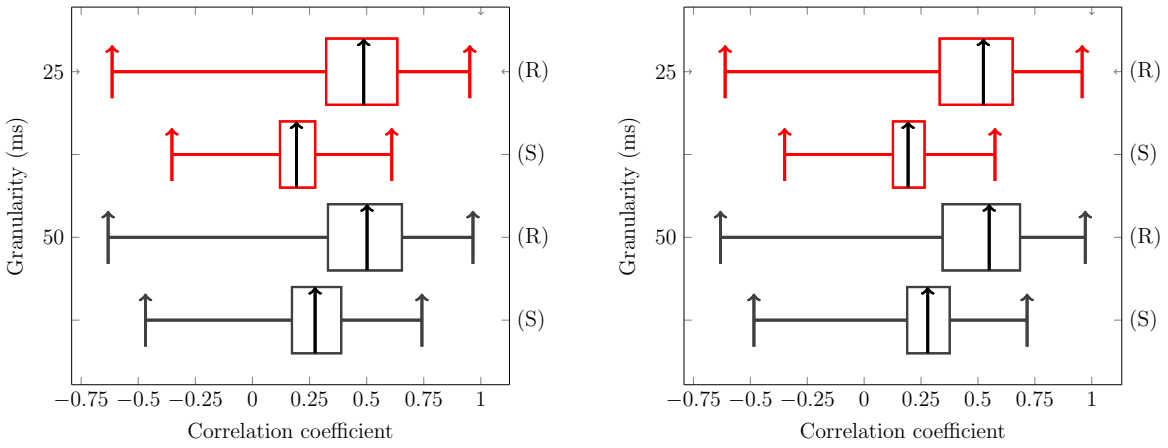
We used the BigEnough dataset [18] to evaluate RegulaTor, as detailed for FRONT in Section 3.4. We defended traces with the RegulaTor simulation scripts provided by Holland and Hopper [13] as well as the Maybenot simulator [29] with the machines described above. Trailing padding cells were removed from the Maybenot-defended datasets before evaluation, as was done for FRONT.

We considered the two configurations of RegulaTor presented by Holland and Hopper: RegulaTor-Light (`RT-Light`) and RegulaTor-Heavy (`RT-Heavy`). We derived parameters from the ones used in their paper based on the tuning process they described [13]. This consisted of multiplying R and N by a ratio between the average number of cells per trace in the BigEnough dataset and the average cells per trace in their dataset. We additionally modified R for Maybenot RegulaTor to match the *latency* overhead of simulated RegulaTor, leaving other parameters unchanged. ω represents the number of cells per state in Maybenot RegulaTor. All parameters are summarized in Table 4.

4.3.2 Trace Comparison

We generated two aggregated time series for each trace from simulated RegulaTor and Maybenot RegulaTor, using the process described in Section 3.4, with $I = \{25, 50\}$. The correlation results are shown in Figure 11, and LCSS results are in Figure 12.

Very similar correlation coefficients are observed with both configurations. There is a moderate correlation for download traffic: with `RT-Light`, the median is approximately



(a) `RT-Light` configuration

(b) `RT-Heavy` configuration

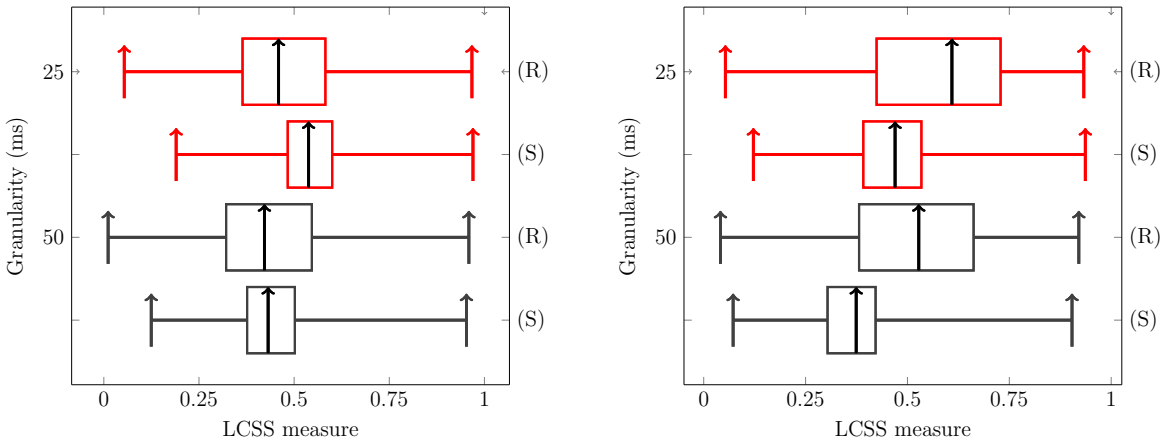
Figure 11: Correlation coefficient, simulated RegulaTor and Maybenot RegulaTor

0.49 with $I = 25$ and 0.50 with $I = 50$. `RT-Heavy` has a slightly higher median correlation in both cases: it is 0.52 with $I = 25$ and 0.55 with $I = 50$. Interquartile range is $[0.32, 0.63]$ for `RT-Light` with $I = 25$; similar results are seen for $I = 50$ and `RT-Heavy`.

Maybenot RegulaTor sent 20 cells per state, allowing for the rate prescribed by the decay function to be matched closely. The moderate correlation observed is likely due to two more significant factors. Maybenot RegulaTor uses a small probability of transitioning to `SEND_0` on a *NonPaddingSent* event as a heuristic to mimic RegulaTor's surge restarting behavior, which could result in surges being restarted at different times, greatly decreasing correlation. Also, Maybenot RegulaTor sets a constant rate throughout a download, whereas simulated RegulaTor caps the sending rate after a padding count has been exceeded; this could lead to more divergence towards the end of a trace.

Since upload traffic is simply sent at a constant fraction of the rate of download traffic, the low correlation observed for it is likely due to the same factors and the omission of the C parameter in Maybenot RegulaTor, causing some cells to be sent later.

The median LCSS of `RT-Light` is about 0.46 with $I = 25$ and 0.42 with $I = 50$; although `RT-Heavy` has a higher median LCSS in both cases (0.61 with $I = 25$ and 0.53 with $I = 50$), its interquartile range is much greater. This indicates that traces were more



(a) `RT-Light` configuration

(b) `RT-Heavy` configuration

Figure 12: LCSS measure, simulated RegulaTor and Maybenot RegulaTor

similar near the beginning and that much of the observed variation is due to different surge restart times: the probability of restarting a surge in Maybenot RegulaTor decreases as sending rate increases, and sending rate was initially higher with `RT-Heavy`, resulting in more surge restarts later in a download. Lower LCSS for upload traffic with `RT-Heavy` also suggests that the C parameter is important, since there was more upload traffic with this configuration and many cells were likely sent later with Maybenot RegulaTor.

4.3.3 Overhead Measurement

We measured both the bandwidth and latency overhead–total time to the last non-padding cell of a trace after being defended compared to original download time–of simulated RegulaTor and Maybenot RegulaTor; mean results are in Table 5.

Defense	Bandwidth overhead (%)			Latency overhead (%)
	Send	**Receive**	**Overall**	
Maybenot RT-Light	747.98	138.23	178.18	21.11
Simulated RT-Light	424.62	44.93	69.80	22.01
Maybenot RT-Heavy	1091.88	151.35	212.96	15.31
Simulated RT-Heavy	537.66	73.86	104.24	17.52

Table 5: RegulaTor average bandwidth and latency overhead

Simulated `RT-Light` incurred 69.80% bandwidth overhead and 22.01% latency overhead; `RT-Heavy` resulted in a higher 104.24% bandwidth overhead and slightly lower latency overhead (17.52%), a consequence of its faster sending rate.

With both configurations, Maybenot RegulaTor had comparable latency overhead to simulated RegulaTor, but its bandwidth overhead was much greater: Maybenot `RT-Light` incurred 178.18% overhead, a 108.38% increase over simulated `RT-Light`; and Maybenot `RT-Heavy`'s overhead was 212.96%, which is 108.72% higher than simulated `RT-Heavy`.

This is due to the lack of the N parameter in Maybenot RegulaTor: there is no mechanism to limit padding in the relay machine, so a constant traffic rate is set throughout a download. Since surges are restarted probabilistically, it is also likely that this happened more often than necessary. An approximate 108% increase in bandwidth overhead makes Maybenot RegulaTor too costly for implementation in Tor.

4.3.4 Attack Performance

We evaluated CUMUL [23], DF [35], and Tik-Tok [31] in the closed-world setting against simulated RegulaTor and Maybenot RegulaTor using scripts provided by Gong et al. for CUMUL [7] and Rahman et al. for DF and Tik-Tok [31]. We performed 10-fold cross-validation and changed the input size of DF and Tik-Tok from 5,000 to 10,000 cells to account for padding. The results are in Table 6.

Simulated `RT-Light` was effective, reducing the accuracy of CUMUL to 5.65% and DF to 6.42%, but Tik-Tok attained 22% accuracy against it. Similarly, simulated `RT-Heavy` lowered the accuracy of CUMUL to 4.53% and DF to 5.79%, and Tik-Tok was the best attack, achieving 15.16% accuracy.

Maybenot RegulaTor provided better overall protection than simulated RegulaTor. Although CUMUL and DF achieved slightly higher accuracy against it with the `RT-Light` configuration (6.38% and 6.63%, respectively), Tik-Tok's accuracy was only 9.89%. Similarly, CUMUL and DF were slightly more effective against Maybenot `RT-Heavy`, but Tik-Tok had lower accuracy than it did against simulated `RT-Heavy`.

Defense	Accuracy (%)		
	CUMUL	**DF**	**Tik-Tok**
Undefended	94.66	95.89	94.00
Maybenot RT-Light	6.38	6.63	9.89
Simulated RT-Light	5.65	6.42	22.00
Maybenot RT-Heavy	6.88	8.11	10.00
Simulated RT-Heavy	4.53	5.79	15.16

Table 6: RegulaTor performance in closed-world setting, BigEnough dataset

This is likely due to the same factors that increased Maybenot RegulaTor's bandwidth overhead: there was no padding limit, so traffic was sent at a constant rate throughout each download; and surges were restarted probabilistically rather than deterministically, so precise information about the number of queued packets was not leaked. Nevertheless, Maybenot RegulaTor's bandwidth overhead would need to be decreased for it to be a viable candidate for implementation in Tor.

5 Implementing Surakav

5.1 Description

Surakav is intended to achieve the benefits of regularization defenses while decreasing the overhead they often incur [8]. It works by generating *reference traces*, which appear similar to real cell traces, and using them to shape the traffic pattern of each download. It consists of two components: a *generator*, which uses a GAN to generate reference traces; and a *regulator*, which uses these reference traces to shape traffic.

Both real and reference traces are viewed as *burst sequences*, in which a single burst is an uninterrupted sequence of cells sent in the same direction. An entire burst sequence, then, consists of alternating outgoing/incoming bursts (from the client's perspective), which are characterized by their size, or number of cells.

The defense operates in rounds: during each round, two bursts (outgoing and incoming) are taken from a reference trace. After waiting for a sampled delay, the client sends a burst of real traffic based on the size of the outgoing reference burst. This is followed by a message to the relay to inform it of the size of the incoming reference burst, which it uses to determine the size of its response burst.

The client and relay determine the size of real bursts based on the number of queued cells, the size of the reference burst, and a tolerance parameter δ. A lower and upper threshold are computed as follows, where b is the size of the reference burst:

$$\perp = \lfloor (1 - \delta) \cdot b \rfloor \tag{3}$$

$$\top = \lfloor (1 + \delta) \cdot b \rfloor \tag{4}$$

If the number of queued cells is less than $\perp$, the real burst size is $\perp$; similarly, if it is greater than $\top$, real burst size is $\top$. Otherwise, real burst size is equal to the number of

queued cells. Thus, Surakav essentially replays reference traces, modifying burst sizes to reduce overhead; values of δ closer to zero provide better protection at greater cost.

A mechanism called "random response" is also included to decrease overhead: if there are no queued cells at the relay, it can skip sending a response burst with probability q, which is sampled from the range $(0, 1)$ for each download. An additional parameter, ρ, defines the maximum time gap between outgoing bursts; it is fixed at 100 ms.

5.2 Maybenot Surakav

We found that Maybenot is unable to support an accurate implementation of Surakav, principally due to the coordination required between client and relay. We first present two machines that precisely mimic a given reference trace, which we refer to collectively as Maybenot Surakav. We then explore why this design cannot be extended to include burst adjustment or random response.

The Maybenot Surakav relay machine is shown in Figure 13; its corresponding client machine is depicted in Figure 14. These machines consist of a `START` state; a `BLOCK` state, which enables infinite blocking with the *bypass* and *replace* flags set; alternating `SEND` and `RECV` states; and the pseudo-state `StateEnd`.

When the first *NonPaddingSent* or *NonPaddingRecv* event is triggered, both machines transition to the `BLOCK` state, which enables infinite blocking; the *bypass* and *replace* flags are used to allow precise control over the number of outgoing cells. Once blocking is in effect, a *BlockingBegin* event will be triggered, causing the client machine to transition to the `SEND` state and the relay machine to transition to the `RECV` state.

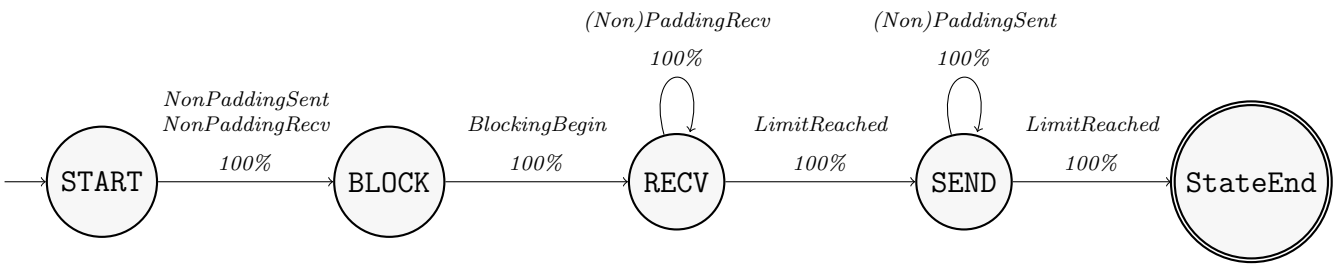


Figure 13: Maybenot Surakav relay machine with two bursts

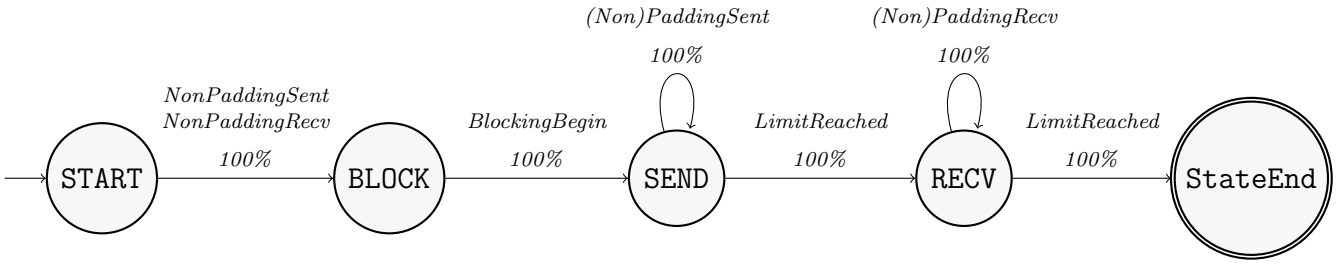


Figure 14: Maybenot Surakav client machine with two bursts

In both machines, the `SEND` state's action is to pad with the *bypass* and *replace* flags. It has a uniform action distribution with parameters $a = 512$ and $b = 512$ to send one cell for each action; a uniform timeout distribution with $a = 5$ and $b = 5$ to delay sending a cell for 5 µs, allowing for a sending rate of 200 cells per second; and a uniform limit distribution which specifies the number of cells to send, based on the size of the corresponding burst from the reference trace.

Each `RECV` state complements a `SEND` state; it is configured such that state transitions are synchronized between client and relay machines. The `RECV` states are set to enable infinite blocking with no timeout and the *bypass* and *replace* flags, which has no effect because this is done immediately by the `BLOCK` state at the beginning of a download. They have the same limits as their corresponding `SEND` states.

Though these machines allow for precise replication of a reference trace, they cannot be extended to support burst adjustment due to Maybenot's lack of queue-related events. A further complication is that, even with events for queued cells, queue size information would not be shared by the client and relay: some mechanism would be needed to communicate the size of a burst. Similarly, random response is primarily infeasible because of missing queue information, but it may also require coordination facilities.

We considered the possibility of a heuristic to approximate burst adjustment, as we did with RegulaTor's surge restarting. We created an experimental design that sent up to $\top$ cells per burst and used the number of padding cells sent during a burst as an early stop condition, but the program that generated these machines consistently triggered the out-of-memory killer on our computer with 256 GB of RAM. This is because multiple

states were needed per burst, and a single reference trace can contain thousands of these.

Machines must also be sufficiently small in order to be serialized and stored as character strings. Of the 38,000 Maybenot Surakav machine pairs we generated, the majority of serialized machines were about 8.4 MB in size. A machine with more states would require even more storage to be represented, and several machines would need to be stored at a time for future downloads, reducing practicality for ordinary users. This suggests the need for mechanisms to create heuristics that do not require many states.

5.3 Evaluation

5.3.1 Experimental Setup

For the sake of completeness, we compare simulated Surakav and Maybenot Surakav, despite major differences between implementations. We used the BigEnough dataset [18] to evaluate Surakav, as described for FRONT in Section 3.4. We defended traces with the Surakav simulation scripts provided by Gong et al. [8] and the Maybenot simulator [29]. Trailing padding cells were removed from the Maybenot-defended datasets before evaluation, as was done for FRONT and RegulaTor.

We simulated the two configurations of Surakav used by Gong et al.: Surakav-Light ($\delta = 0.6$) and Surakav-Heavy ($\delta = 0.4$). To train the GAN, we used the CW_{100} dataset collected by Rimmer et al. starting in January 2017 [32]. This dataset consists of 2,500 traces from the homepage of each of the Alexa top 100 websites; we used 1,000 traces from each website. This is similar to the approach taken by Gong et al., who used the CW_{900} dataset, choosing 100 websites at random and using 1,000 traces from each [8].

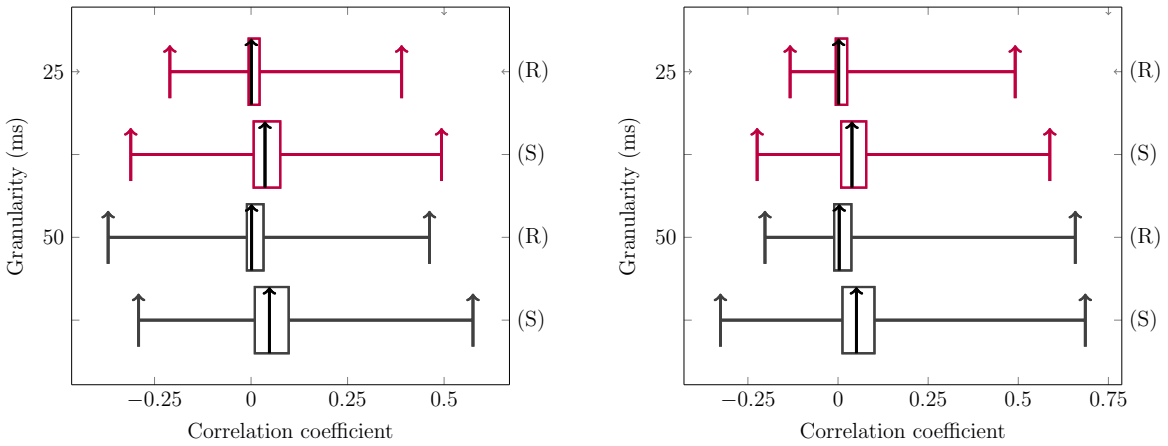
We note that most of the reference traces generated after training with this dataset contained a lot of small bursts. To account for this, we increased the size of the reference trace for each download from 10 to 80 times the size of the undefended trace; however, we set a factor of 480 to defend four particularly unusual traces in the BigEnough dataset. We also modified the Surakav simulation scripts to save the reference traces they generated.

We produced a total of 38,000 Maybenot Surakav machine pairs, one for each configuration and trace in the BigEnough dataset, with the saved reference traces from simulated Surakav. All machines were limited to 8,000 bursts, resulting in many reference traces being truncated, and they did not require any parameters. They were used to create two defended datasets, one with reference traces corresponding to Surakav-Light and another with reference traces from Surakav-Heavy.

5.3.2 Trace Comparison

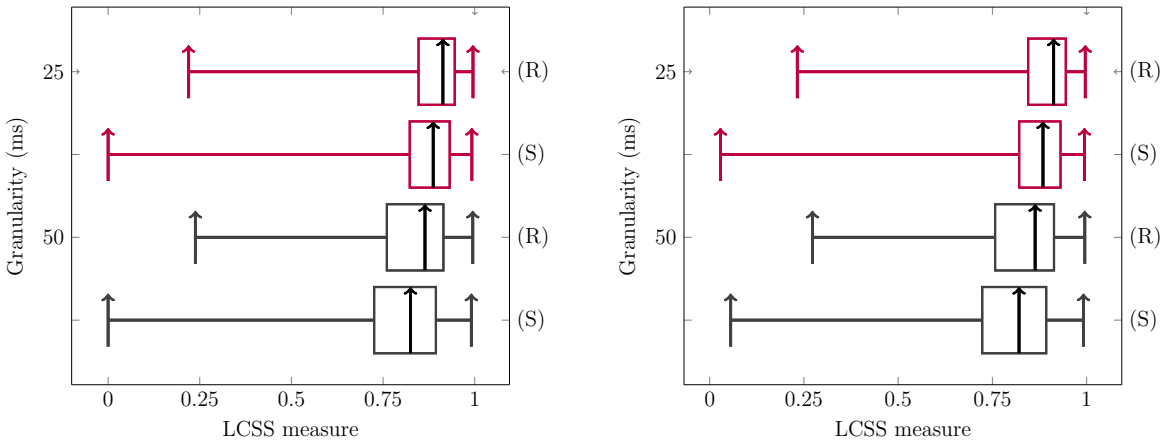
We generated two aggregated time series for each trace from simulated Surakav and Maybenot Surakav, using the process described in Section 3.4, with $I = \{25, 50\}$. The correlation results are shown in Figure 15, and LCSS results are in Figure 16.

As expected, there is a low correlation between simulated Surakav and Maybenot Surakav: the median, 25th percentile, and 75th percentile are nearly zero with both configurations for upload and download traffic. This can be attributed to the lack of burst adjustment and random response in Maybenot Surakav. Upload traffic likely has slightly higher correlation due to its low volume, resulting in less pronounced differences than download traffic.



(a) Surakav-Light configuration (b) Surakav-Heavy configuration

Figure 15: Correlation coefficient, simulated Surakav and Maybenot Surakav



(a) Surakav-Light configuration

(b) Surakav-Heavy configuration

Figure 16: LCSS measure, simulated Surakav and Maybenot Surakav

However, some traces had relatively high correlation: with Surakav-Light, maximum correlation is 0.46 for download traffic and 0.57 for upload traffic when $I = 50$; similar results are observed when $I = 25$. This is likely due to some traces in the BigEnough dataset more closely matching their reference traces. In such cases, simulated Surakav's burst adjustment and random response would not be as pronounced, causing defended traces to be more similar to Maybenot Surakav's. For the same reason, Surakav-Heavy (with $\delta = 0.4$) has a higher maximum correlation: when $I = 50$, it is 0.66 for download traffic and 0.69 for upload traffic.

Interestingly, LCSS is very high for both configurations and values of I: with Surakav-Light, median LCSS is 0.89 for upload traffic and 0.91 for download traffic when $I = 25$; it is 0.82 for upload traffic and 0.86 for download traffic when $I = 50$. High median LCSS is also observed with Surakav-Heavy. Maybenot Surakav traces are quite long since more bursts must be sent in total without burst adjustment or random response; thus, we suspect that large portions of simulated Surakav traces were found to be subsequences of their corresponding Maybenot Surakav traces, resulting in a very high LCSS measure.

Defense	Bandwidth overhead (%)			Latency overhead (%)
	Send	**Receive**	**Overall**	
Maybenot Surakav-Light	15601.91	226.57	1233.81	257.66
Simulated Surakav-Light	531.53	65.51	96.04	31.36
Maybenot Surakav-Heavy	15514.69	225.42	1227.02	257.14
Simulated Surakav-Heavy	608.66	92.47	126.29	33.05

Table 7: Surakav average bandwidth and latency overhead

5.3.3 Overhead Measurement

We measured both the bandwidth and latency overhead of simulated Surakav and Maybenot Surakav; mean results are in Table 7.

Simulated Surakav-Light incurred 96.04% bandwidth overhead and 31.35% latency overhead. With Surakav-Heavy, bandwidth overhead was 125.30%, and latency overhead was 33.05%; both of these increases are due to lower tolerance for burst adjustment.

Maybenot Surakav's bandwidth and latency overhead were significantly higher than those of simulated Surakav. With the Surakav-Light configuration, bandwidth overhead was 1233.81%, and latency overhead was 257.66%. Similarly, with Surakav-Heavy, bandwidth overhead was 1227.02%, and latency overhead was 257.14%.

These results confirm the need for burst adjustment and random response in Surakav: requiring that reference traces be replayed exactly imposes too strict of a traffic pattern, resulting in very high overheads, which make Maybenot Surakav impractical for implementation in Tor in its current state.

5.3.4 Attack Performance

We evaluated CUMUL [23], DF [35], and Tik-Tok [31] in the closed-world setting against simulated Surakav and Maybenot Surakav using scripts provided by Gong et al. for CUMUL [7] and Rahman et al. for DF and Tik-Tok [31]. We performed 10-fold cross-validation and changed the input size of DF and Tik-Tok from 5,000 to 10,000 cells to account for padding. The results are in Table 8.

Defense	Accuracy (%)		
	CUMUL	**DF**	**Tik-Tok**
Undefended	94.66	95.89	94.00
Maybenot Surakav-Light	4.08	1.05	1.58
Simulated Surakav-Light	15.86	20.32	23.37
Maybenot Surakav-Heavy	4.52	1.05	2.21
Simulated Surakav-Heavy	15.41	15.47	15.47

Table 8: Surakav performance in closed-world setting, BigEnough dataset

The best attack against simulated Surakav-Light was Tik-Tok, which attained 23.37% accuracy; CUMUL and DF achieved 15.86% and 20.35% accuracy, respectively. Simulated Surakav-Heavy was slightly more effective and provided more consistent protection across all attacks: CUMUL reached 15.41% accuracy against it, while DF and Tik-Tok both had 15.47% accuracy.

Maybenot Surakav provided a significant level of protection against all attacks due to its exact mimicking of reference traces. With Maybenot Surakav-Light, CUMUL's accuracy was 4.08%, and the accuracy of both DF and Tik-Tok was less than 2%. Similar results are seen with Maybenot Surakav-Heavy, though Tik-Tok's accuracy was slightly higher, at 2.21%. Regardless, Maybenot Surakav would need to have much lower overhead to be feasible for inclusion in Tor.

6 Discussion

It is worth considering *why* a framework is better than direct implementation of defenses. The primary appeal is increased flexibility: as observed by Pulls, a framework such as Maybenot would allow for evolving defenses, coordination of multiple defenses, and the use of tailored defenses distributed to clients in real time [29].

All of these benefits relate to the fact that machines are serialized and represented as character strings, so they can be loaded and used in a plug-and-play fashion. Meanwhile, a direct implementation would require recompilation of part or all of an application's codebase; this is even the case with Tor's existing circuit padding framework [25]. Ideally, the inclusion of a framework would give way to an increased emphasis on defense *design*, and the *integration* of defenses would be simple.

However, providing a framework as the sole or primary mechanism for defense implementation may bind authors of defenses to a specific model, which must be sufficiently expressive and capable of supporting effective defense designs. Even if a framework is flexible in terms of the way defenses are integrated, it will not provide much benefit to users of privacy-enhancing technologies if it cannot be used to implement *effective* defenses, and it may end up in disuse, as has occurred with Tor's circuit padding framework.

6.1 Suggested Improvements to Maybenot

Our evaluation has shown that Maybenot *can* be used to approximately implement proposed website fingerprinting defenses. However, it lacks a few features that would significantly improve these implementations.

Based on our experience with RegulaTor, we believe that the ability to monitor queues is critical. If Maybenot included an event for cells being queued, a machine could be created to track the *number* of cells queued, but further support would be needed to track the *time* that cells have been in the queue for. One possibility for this is a timer, which would also be useful for other purposes.

Thus, we suggest the inclusion of two new mechanisms: a *PacketQueued* event, which would allow for the implementation of length-based queue thresholds; and a timer, consisting of an action to start it as well as *TimerStart* and *TimerEnd* events, similar to blocking. This may allow for time-based queue thresholds and could also improve the implementations of a variety of defenses, including Surakav.

However, we consider that a counter would be most beneficial in solving the issues encountered with Surakav, and it could complement the *PacketQueued* event, avoiding the necessity of many states or a separate machine to keep track of queue size. One way of implementing this would be with *CounterIncrement* and *CounterDecrement* actions along with an event triggered when the counter's value is zero after being decremented.

With all of our proposed mechanisms, FRONT could be reimplemented to increment a counter for padding count by a sampled value, avoiding the necessity of multiple pipelines and larger, more complex machines. RegulaTor could also be implemented with the N parameter due to the ability to monitor queues based on length, and the C parameter may be possible to approximate with a timer. Although Surakav probably could not be implemented precisely due to the coordination it requires, the addition of queue capabilities, a timer, and a counter may allow for the development of effective heuristics to approximate burst adjustment and random response.

Another consideration relates to the use of multiple machines in synthesis to enact more sophisticated defenses: we considered such designs for the defenses implemented in this work, but Maybenot has no mechanism for deliberate signaling between machines. The only option that seemed viable was to re-enable blocking that was already in effect, which was also done for "no-op" states in the Maybenot RegulaTor client machine and Maybenot Surakav machines. The *BlockingBegin* event could then cause transitions in other machines. However, this would be rather cumbersome and would not work in all situations; internal events and actions for signaling could allow machines to more easily cause state transitions in others upon certain conditions being met.

6.2 Download Distinction and Deployment Context

Though the features described above would significantly improve Maybenot's support for proposed defenses, they do not directly address the issue of detecting the end of a download. Maybenot has no events related to download completion; this limitation was circumvented in this work by removing trailing padding cells from defended datasets.

It is important to note that these considerations arise because the defenses we have considered were evaluated on individual traces. Deploying Maybenot at the level of Tor circuits may change their behavior in unexpected ways, since multiple TCP connections can be multiplexed over a single circuit, and users can perform concurrent downloads. Even if Arti could run a separate instance of the framework for each connection, this would not fully solve the problem: traces in evaluation datasets are made up of multiple web requests to retrieve content embedded in fetched HTML pages, and determination of which connections should be grouped together would not be a trivial task.

Since guard and middle relays are not aware of individual TCP connections [5], an event for download completion could only be triggered on the client side, and, in light of the issues described above, this would need to happen on a per-connection basis. Other framework designs do not inherently solve this problem, and further modifications to Tor and any applications that use it would likely be required to do so, which is undesirable. We thus advocate for defenses that are designed to work well at the circuit level, and we believe that all of the defenses we have evaluated could do so with a soft stop condition, though their overhead and protection against attacks may be affected.

The timer we propose could be used to implement a soft stop condition by running a dedicated machine that started a timer when a non-padding cell was sent and in some way signaled other machines to transition upon its expiration. We note that if such a condition were based on cells *received*, complications could arise in situations of congestion or other network-related issues, since a download may not have actually finished. We suggest carefully designed soft stop conditions, which may be defense-specific. The necessity of soft stop conditions further justifies the new features we propose for Maybenot.

6.3 Alternatives to State Machine Frameworks

Adding the features we propose would have the effect of generalizing the capabilities of Maybenot. It may be envisioned that features could be continually added to support more defenses, but this would add significant complexity to the framework. Moreover, even if this approach were taken, some defenses could still only be approximated in Maybenot due to the nature of state machine frameworks.

Ultimately, adding additional features to Maybenot would have the effect of assimilating its behavior to that of a high-level extension language. This may in fact be a better design for defense implementation: it could allow for a significant level of flexibility and would not necessarily limit defense designers to any particular model. One example of this approach is WFDefProxy, a framework which allows for use of the Go programming language to implement website fingerprinting defenses on Tor bridges [9]. The main drawback of WFDefProxy is that deploying defenses on bridges does not account for the entry point to the Tor network (i.e., the bridge itself) being a potential WF attacker.

WF defenses could also be deployed using Flexible Anonymous Networks (FAN) [33]: with this scheme, Arti would be modified to contain *hooks* in certain code locations, and *plugins* could be deployed to interact with desired hooks. This could be done on a per-connection basis, allowing defenses to be encoded as FAN plugins which Tor users could then instruct relays to use. Such a design would also allow defenses to operate at the middle relay, accounting for the threat of a malicious guard; but WFDefProxy and FAN plugins represent only two possible alternatives to a state machine framework.

It is also worth considering that support for some types of defenses or certain features may not be necessary; *effective* defenses are the ultimate goal. Unfortunately, it cannot be said with any certainty *which* features are in fact needed and which are not, because attacks are continually improving, and defenses that were previously thought to be highly effective are no longer useful. Given this, it would be wise to choose an implementation strategy for website fingerprinting defenses that allows for a wide range of capabilities, even if these might not be necessary for effective defenses: it is easier not to use certain

features than to later implement ones which turn out to be required.

Thus, we do not recommend Maybenot for Arti in its current state, and we draw no particular conclusions about whether a state machine framework is a good way forward, except that there are some defenses that they will only ever be able to approximate. Adding the features we suggest would enhance Maybenot's capabilities significantly, and further evaluation could provide more insights into whether it is a viable candidate for inclusion in Arti. We also believe that exploring the possibility of an alternative model could be a fruitful direction for future work.

7 Conclusions and Future Work

We presented approximate implementations of FRONT, RegulaTor, and Surakav in the state machine framework Maybenot. We evaluated these implementations in terms of similarity to the simulated versions of the defenses, overhead, and protection against attacks. This evaluation demonstrates that Maybenot has the potential to support effective defenses, but the addition of certain features would greatly enhance its ability to do so.

We recommend improvements to Maybenot and extensive further evaluation before its inclusion in Arti is considered, but we believe that it could be a good framework for defense implementation if made more expressive. We also recommend consideration of alternative models for defense implementation, since they could support more complex defenses which may be needed as attacks continue to improve.

An immediate avenue for future work would be to attempt to implement more defenses in Maybenot. The defense implementations we presented could also be evaluated with different combinations of parameters, which may provide improvements in terms of overhead and protection against attacks. Evaluation could be carried out with additional datasets and attacks, including attacks performed in the one-page setting to better account for the maximal capabilities of an attacker.

Adding the features we propose to Maybenot and evaluating them with new defense implementations represent an abundance of possibilities for future work. Alternative models for website fingerprinting defenses could also be explored, such as Flexible Anonymous Networks. Ideally, such a model would allow for negotiation of defenses with the middle relay in a Tor circuit due to the possibility of the guard relay being a WF attacker, and it should be capable of supporting proposed defenses.

Availability

The code for all of the described defense implementations is available on GitHub at `https://github.com/ewitwer/maybenot-defenses`.

Acknowledgements

I would like to thank my advisor, Nick Hopper, for his assistance and helpful feedback over the course of this project as well as his guidance throughout my undergraduate career. He has shaped my experience as a researcher, and I truly appreciate his patience and support for my work.

I would also like to thank James Holland for the useful discussions we have had and reviewing this paper. I've learned a good deal about website fingerprinting from my work with him, and his feedback has been very helpful in my research.

I also extend my gratitude to Kangjie Liu for his review and his helpful feedback and suggestions. I appreciate his insightful perspectives on security that I learned about in his Introduction to Computer Security class.

Special thanks to Tobias Pulls for sharing the code and specifications for the Maybenot framework, providing important insights and feedback throughout this project, and reviewing the paper. Without his help, this project would not have been possible.

Additional thanks to Namitha Binu, Jack Milless, Shana Watters, and Alice Zhang for reviewing a draft of this paper and their helpful suggestions.

References

[1] Sanjit Bhat et al. "Var-CNN: A Data-Efficient Website Fingerprinting Attack Based on Deep Learning". In: *Proceedings on Privacy Enhancing Technologies* 2019.4 (July 2019), pp. 292–310.

[2] Xiang Cai et al. "Touching from a Distance: Website Fingerprinting Attacks and Defenses". In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. CCS '12. New York, NY, USA: Association for Computing Machinery, 2012, pp. 605–616.

[3] Giovanni Cherubin, Rob Jansen, and Carmela Troncoso. "Online Website Fingerprinting: Evaluating Website Fingerprinting Attacks on Tor in the Real World". In: *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 753–770.

[4] Wladimir De la Cadena et al. "TrafficSliver: Fighting Website Fingerprinting Attacks with Traffic Splitting". In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. CCS '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 1971–1985.

[5] Roger Dingledine, Nick Mathewson, and Paul Syverson. "Tor: The Second-Generation Onion Router". In: *13th USENIX Security Symposium (USENIX Security 04)*. USENIX Association, Aug. 2004, pp. 303–320.

[6] Kevin P. Dyer et al. "Peek-a-Boo, I Still See You: Why Efficient Traffic Analysis Countermeasures Fail". In: *2012 IEEE Symposium on Security and Privacy*. 2012, pp. 332–346.

[7] Jiajun Gong and Tao Wang. "Zero-delay Lightweight Defenses against Website Fingerprinting". In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 717–734.

[8] Jiajun Gong et al. “Surakav: Generating Realistic Traces for a Strong Website Fingerprinting Defense”. In: *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society. 2022, pp. 1525–1525.

[9] Jiajun Gong et al. *WFDefProxy: Modularly Implementing and Empirically Evaluating Website Fingerprinting Defenses*. 2021. arXiv: 2111.12629 [cs.CR].

[10] Jamie Hayes and George Danezis. “k-fingerprinting: A Robust Scalable Website Fingerprinting Technique”. In: *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Aug. 2016, pp. 1187–1203.

[11] Sébastien Henri et al. “Protecting against Website Fingerprinting with Multihoming”. In: *Proceedings on Privacy Enhancing Technologies* 2020.2 (2020), pp. 89–110.

[12] Dominik Herrmann, Rolf Wendolsky, and Hannes Federrath. “Website Fingerprinting: Attacking Popular Privacy Enhancing Technologies with the Multinomial Naive-Bayes Classifier”. In: *Proceedings of the 2009 ACM Workshop on Cloud Computing Security*. CCSW ’09. New York, NY, USA: Association for Computing Machinery, 2009, pp. 31–42.

[13] James K Holland and Nicholas Hopper. “RegulaTor: A Straightforward Website Fingerprinting Defense”. In: *Proceedings on Privacy Enhancing Technologies* 2022.2 (2022), pp. 344–362.

[14] Marc Juarez et al. “A Critical Evaluation of Website Fingerprinting Attacks”. In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’14. New York, NY, USA: Association for Computing Machinery, 2014, pp. 263–274.

[15] Marc Juarez et al. “Toward an Efficient Website Fingerprinting Defense”. In: *European Symposium on Research in Computer Security*. Springer. 2016, pp. 27–46.

[16] Xiapu Luo et al. “HTTPOS: Sealing Information Leaks with Browser-Side Obfuscation of Encrypted Flows”. In: *NDSS*. Vol. 11. 2011.

[17] Nate Mathews, Payap Sirinam, and Matthew Wright. “Understanding Feature Discovery in Website Fingerprinting Attacks”. In: *2018 IEEE Western New York Image and Signal Processing Workshop (WNYISPW)*. 2018, pp. 1–5.

[18] Nate Mathews et al. “SoK: A Critical Evaluation of Efficient Website Fingerprinting Defenses”. In: *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society. 2022, pp. 344–361.

[19] Nick Mathewson. *Announcing Arti, a pure-Rust Tor implementation*. `https://blog.torproject.org/announcing-arti/`. Accessed on June 1, 2023. 2021.

[20] Nick Mathewson. *Arti: reimplementing Tor in Rust*. `https://gitlab.torproject.org/tpo/core/arti/-/blob/main/README.md`. Accessed on June 1, 2023. 2023.

[21] Nick Mathewson. *Implement circuit padding machines (#63)*. `https://gitlab.torproject.org/tpo/core/arti/-/issues/63`. Accessed on June 1, 2023. 2021.

[22] Rishab Nithyanand, Xiang Cai, and Rob Johnson. “Glove: A Bespoke Website Fingerprinting Defense”. In: *Proceedings of the 13th Workshop on Privacy in the Electronic Society*. WPES ’14. New York, NY, USA: Association for Computing Machinery, 2014, pp. 131–134.

[23] Andriy Panchenko et al. “Website Fingerprinting at Internet Scale”. In: *NDSS*. 2016.

[24] Andriy Panchenko et al. “Website Fingerprinting in Onion Routing Based Anonymization Networks”. In: *Proceedings of the 10th Annual ACM Workshop on Privacy in the Electronic Society*. WPES ’11. New York, NY, USA: Association for Computing Machinery, 2011, pp. 103–114.

[25] Mike Perry and George Kadianakis. *Circuit Padding Developer Documentation*. `https://github.com/torproject/tor/blob/main/doc/HACKING/CircuitPaddingDevelopment.md`. Accessed on June 1, 2023. 2020.

[26] Tor Project. *Tor Metrics*. https://metrics.torproject.org/. Accessed on June 1, 2023. 2023.

[27] Tor Project. *Who uses Tor?* https://2019.www.torproject.org/about/torusers.html.en. Accessed on June 1, 2023. 2020.

[28] Tobias Pulls. *Adaptive Padding Early (APE)*. https://www.cs.kau.se/pulls/hot/thebasketcase-ape/. Accessed on June 1, 2023. 2016.

[29] Tobias Pulls. *Maybenot: A Framework for Traffic Analysis Defenses*. 2023. arXiv: 2304.09510 [cs.CR].

[30] Tobias Pulls. *Towards Effective and Efficient Padding Machines for Tor*. 2020. arXiv: 2011.13471 [cs.CR].

[31] Mohammad Saidur Rahman et al. "Tik-Tok: The Utility of Packet Timing in Website Fingerprinting Attacks". In: *Proceedings on Privacy Enhancing Technologies* 2020.3 (July 2020), pp. 5–24.

[32] Vera Rimmer et al. "Automated Website Fingerprinting through Deep Learning". In: *Proceedings of the 25th Network and Distributed System Security Symposium (NDSS 2018)*. Internet Society. 2018.

[33] Florentin Rochet and Tariq Elahi. *Towards Flexible Anonymous Networks*. 2023. arXiv: 2203.03764 [cs.CR].

[34] Vitaly Shmatikov and Ming-Hsiu Wang. "Timing Analysis in Low-Latency Mix Networks: Attacks and Defenses". In: *European Symposium on Research in Computer Security*. Springer. 2006, pp. 18–33.

[35] Payap Sirinam et al. "Deep Fingerprinting: Undermining Website Fingerprinting Defenses with Deep Learning". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS '18. Association for Computing Machinery, 2018, pp. 1928–1943.

[36] Jean-Pierre Smith et al. “QCSD: A QUIC Client-Side Website-Fingerprinting Defence Framework”. In: *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Aug. 2022, pp. 771–789.

[37] Tao Wang. “The One-Page Setting: A Higher Standard for Evaluating Website Fingerprinting Defenses”. In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’21. New York, NY, USA: Association for Computing Machinery, 2021, pp. 2794–2806.

[38] Tao Wang. “Website Fingerprinting: Attacks and Defenses”. PhD thesis. 2016.

[39] Tao Wang and Ian Goldberg. “On Realistically Attacking Tor with Website Fingerprinting.” In: *Proceedings on Privacy Enhancing Technologies* 2016.4 (2016), pp. 21–36.

[40] Tao Wang and Ian Goldberg. “Walkie-Talkie: An Efficient Defense Against Passive Website Fingerprinting Attacks”. In: *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Aug. 2017, pp. 1375–1390.

[41] Tao Wang et al. “Effective Attacks and Provable Defenses for Website Fingerprinting”. In: *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 143–157.

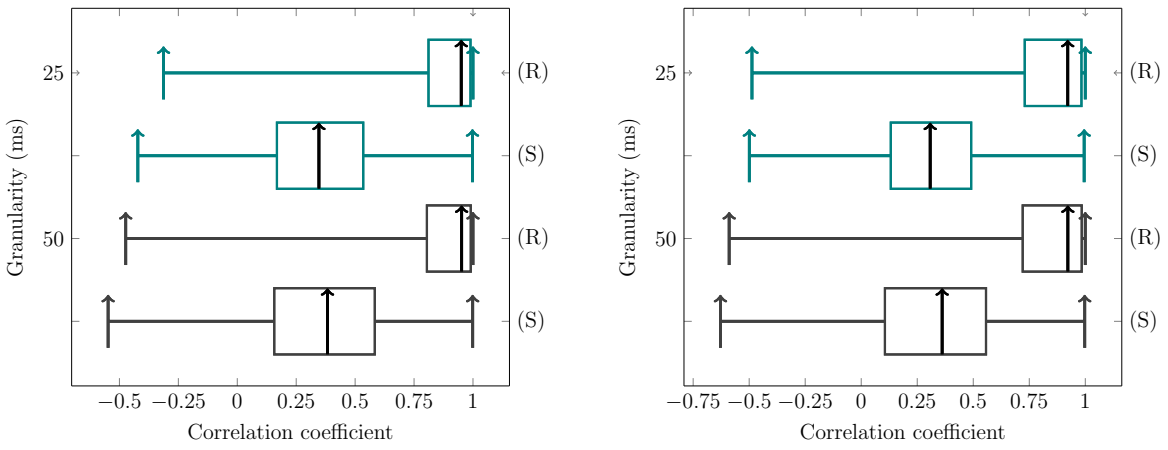
[42] Ethan Witwer, James K. Holland, and Nicholas Hopper. “Padding-Only Defenses Add Delay in Tor”. In: *Proceedings of the 21st Workshop on Privacy in the Electronic Society*. WPES’22. Association for Computing Machinery, 2022, pp. 29–33.

Appendix

To provide further context for the trace comparisons performed for FRONT, RegulaTor, and Surakav, we also compared datasets defended with the simulated version of each defense on separate occasions. This allowed us to quantify how much of the observed differences was due to selection of different parameters for each trace.

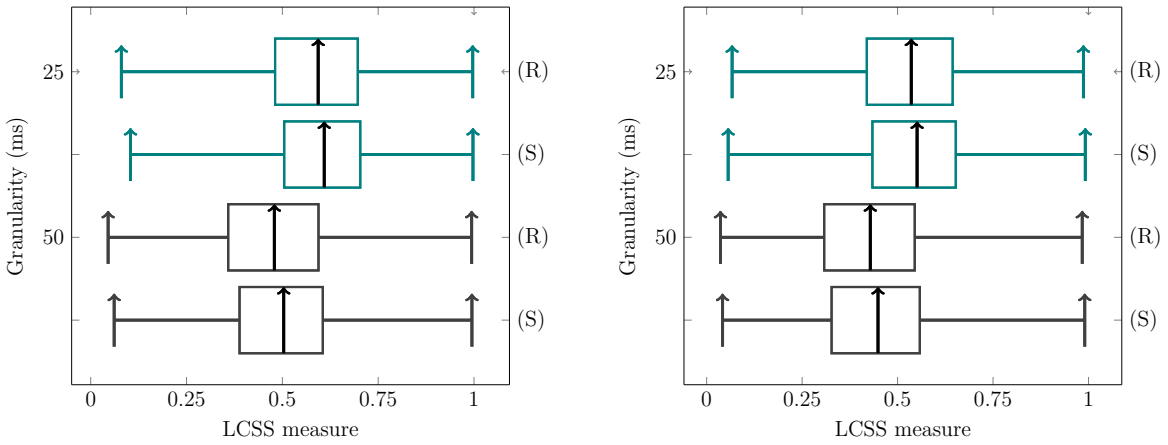
FRONT. Correlation results for FRONT are shown in Figure 17, and LCSS results are in Figure 18. Median correlation for download traffic is very high: the median is 0.95 with `FT-1`, and interquartile range is [0.81, 0.99] when $I = 25$ and [0.80, 0.99] when $I = 50$. Similar results are seen with `FT-2`. This suggests that the slightly lower correlation with Maybenot FRONT and Pipelined FRONT is due to implementation differences.

There is also a low correlation for upload traffic in all cases: median correlation with `FT-1` is 0.35 when $I = 25$ and 0.38 when $I = 50$; it is even lower (0.31 when $I = 25$ and 0.36 when $I = 50$) with `FT-2`. Since low correlation arises from differences only in padding count and window, this supports our conclusion that the low correlation for upload traffic with Maybenot FRONT and Pipelined FRONT can be attributed primarily to a higher ratio of padding to non-padding cells and partly to our implementations.



(a) `FT-1` configuration (b) `FT-2` configuration

Figure 17: Correlation coefficient, two runs of simulated FRONT



(a) `FT-1` configuration (b) `FT-2` configuration

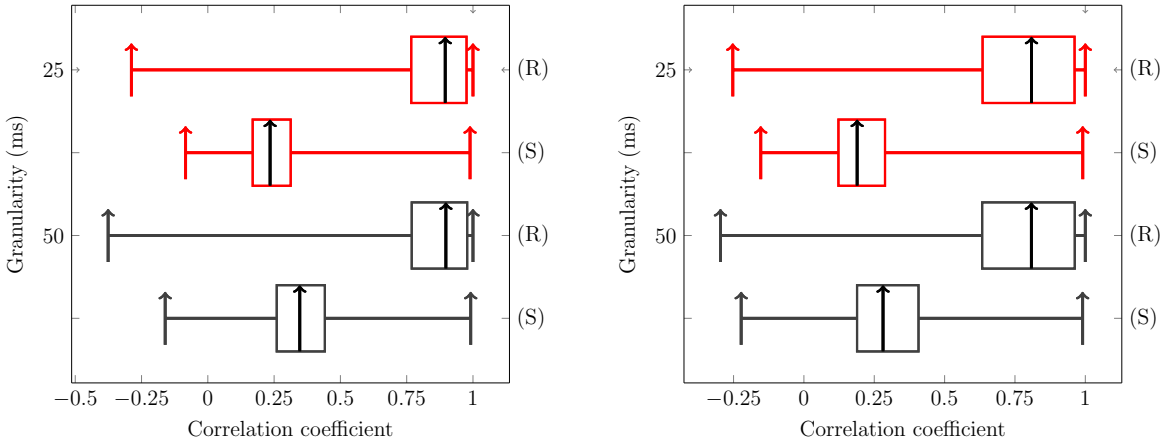
Figure 18: LCSS measure, two runs of simulated FRONT

The minimum median LCSS observed for download traffic is 0.54 when $I = 25$ and 0.43 when $I = 50$. In fact, the results are nearly identical to those of Maybenot FRONT and Pipelined FRONT comparisons; this is likely because padding differences between traces are always present near the beginning of a download, with little variation in the latter portion as few padding cells are sent.

RegulaTor. Correlation results for RegulaTor are shown in Figure 19, and LCSS results are in Figure 20. As with FRONT, there is a high median correlation for download traffic (0.90 with `RT-Light` and 0.81 with `RT-Heavy`, both values of I) and narrow interquartile range, but median correlation for upload traffic is low. This is because the only variation between traces defended with simulated RegulaTor arises from the selection of different padding counts, which has a stronger effect on upload traffic due to its lower volume.

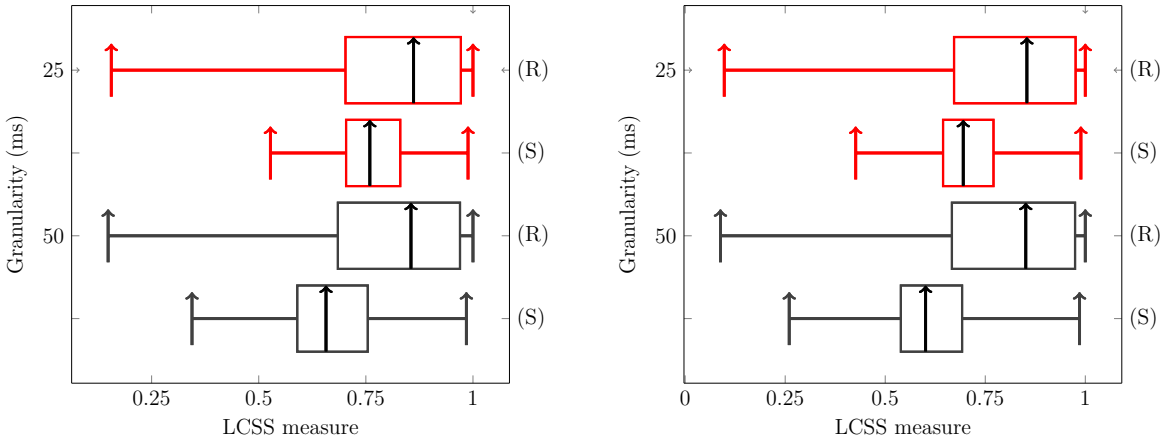
This also accounts for the strong median LCSS for both download and upload traffic. However, though upload traffic is typically sent at a constant fraction of the rate of download traffic, its median LCSS is lower: the median LCSS for download traffic is 0.86 with `RT-Light`, but it is only 0.76 when $I = 25$ and 0.66 when $I = 50$ for upload traffic. This may be due to lower padding counts causing queue sizes to increase at the client. Decreasing the padding count lowers the download traffic rate, which in turn lowers

the upload traffic rate. This could result in less queued upload cells being sent near the beginning of a download, and more cells will be queued for longer than C seconds, causing them to be sent immediately. This behavior can change the overall traffic pattern.



(a) `RT-Light` configuration (b) `RT-Heavy` configuration

Figure 19: Correlation coefficient, two runs of simulated RegulaTor



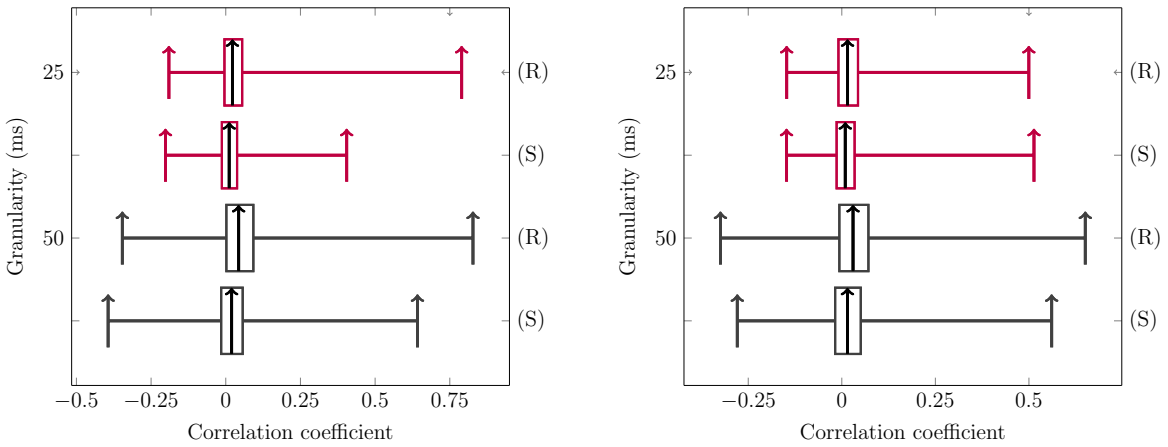
(a) `RT-Light` configuration (b) `RT-Heavy` configuration

Figure 20: LCSS measure, two runs of simulated RegulaTor

Surakav. We reused the same reference traces to compare different runs of simulated Surakav, so any variation should be due only to different random response probability between compared traces. Correlation results for Surakav are displayed in Figure 21, and LCSS results are in Figure 22.

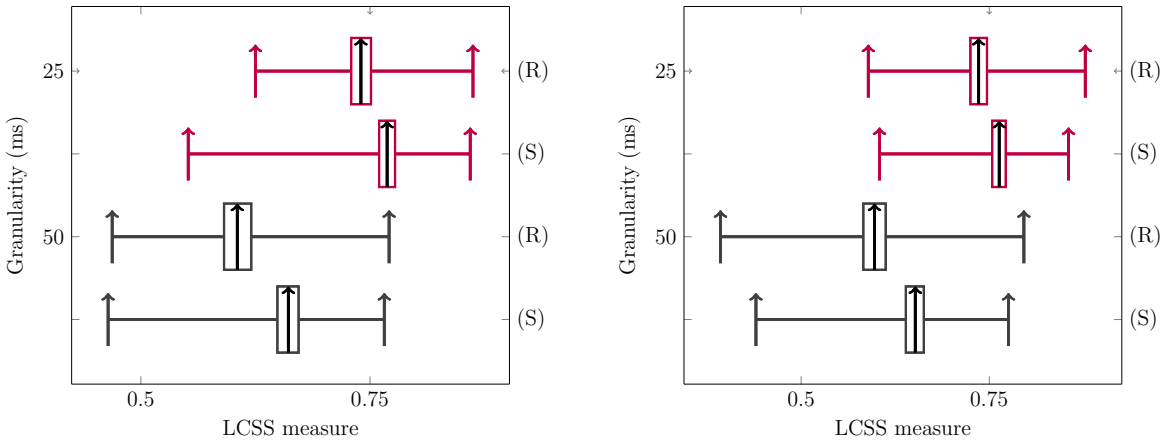
Surprisingly, median, 25th percentile, and 75th percentile correlation is nearly zero in all cases. This suggests that the random response probability selected for each download can have a significant impact on the resultant defended trace. As a consequence, if Maybenot Surakav were updated to include burst adjustment and random response, it may be expected that the correlation results we report would not change much.

However, LCSS varies markedly from that of simulated Surakav and Maybenot Surakav. With Surakav-Light, the median LCSS for download traffic is 0.74 when $I = 25$ and 0.61 when $I = 50$; median LCSS is 0.77 when $I = 25$ and 0.66 when $I = 50$ for upload traffic. Similar results are seen with Surakav-Heavy. This is likely because corresponding defended traces are of comparable length (as opposed to Maybenot Surakav defended traces), and the decision to skip a burst at the relay can have cascading effects on the remainder of a trace which are more apparent with greater values of I.



(a) Surakav-Light configuration (b) Surakav-Heavy configuration

Figure 21: Correlation coefficient, two runs of simulated Surakav



(a) Surakav-Light configuration

(b) Surakav-Heavy configuration

Figure 22: LCSS measure, two runs of simulated Surakav