Maybenot: A Framework for Traffic Analysis Defenses

Tobias Pulls
Karlstad University
Sweden
tobias.pulls@kau.se

Ethan Witwer
University of Minnesota
USA
witwe004@umn.edu

ABSTRACT

In light of the increasing ubiquity of end-to-end encryption and the use of technologies such as Tor and VPNs, analyzing communications metadata—traffic analysis—is a last resort for network adversaries. Traffic analysis attacks are more effective thanks to improvements in deep learning, raising the importance of deploying defenses. This paper introduces Maybenot, a framework for traffic analysis defenses. Maybenot is an evolution and generalization of the Tor Circuit Padding Framework by Perry and Kadianakis, designed to support a wide range of protocols and use cases. Defenses are probabilistic state machines that trigger padding and blocking actions based on events. A lightweight simulator enables rapid development and testing of defenses. In addition to describing the Maybenot framework, machines, and simulator, we implement and thoroughly evaluate the state-of-the-art website fingerprinting defenses FRONT and RegulaTor as Maybenot machines. Our evaluation identifies challenges associated with state machine-based frameworks as well as possible enhancements that will further improve Maybenot's support for effective defenses moving forward.

CCS CONCEPTS

• **Security and privacy** → **Privacy-preserving protocols**; • **Networks** → **Network privacy and anonymity**.

KEYWORDS

website fingerprinting defenses, traffic analysis, framework

ACM Reference Format:
Tobias Pulls and Ethan Witwer. 2023. Maybenot: A Framework for Traffic Analysis Defenses. In *Proceedings of the 22nd Workshop on Privacy in the Electronic Society (WPES '23), November 26, 2023, Copenhagen, Denmark.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3603216.3624953

1 INTRODUCTION

End-to-end encryption is now prevalent after decades of effort, with protocols such as QUIC [35], HTTP/3 [8], TLS 1.3 [63], DoH [31], DoQ [34], MLS [6], and others defaulting to encryption. Concurrently, there is growing acceptance of technologies that make end-users and their IP addresses unlinkable under different threat models, such as Tor [18], VPNs [3, 19, 62], and Apple's iCloud Private Relay [5, 53]. These trends make it more difficult for network operators to detect and block harmful traffic and for attackers to identify and target individual users. The final frontier is traffic analysis: inferences based on metadata of encrypted traffic.

Although there has been significant research on traffic analysis attacks [15, 41, 46, 67, 77], few defenses are deployed on the Internet today. Most existing defenses have been designed for technologies striving for user and IP address unlinkability, like Tor. However, they are modest regarding the overhead they generate, limiting their effectiveness against many attacks [56]. Current protocol standards do incorporate essential building blocks and acknowledge their potential use in defending against traffic analysis, such as the support for the PADDING frame in QUIC [35, §21.14] and the contemplation of traffic analysis in TLS 1.3 [63, §3]. However, they do not mandate any specific defenses or use.

Several factors contribute to the scarcity of deployed traffic analysis defenses. Firstly, the substantial negative performance impact of strong defenses due to padding and increased latency raises significant usability concerns [18, 56]. Given the steep costs, the advantages must be evident: unfortunately, the community is far from in agreement [14, 39, 52, 54, 75]. The landscape of traffic analysis attacks and defenses, driven by advancements in deep learning and artificial intelligence, is changing rapidly, exacerbating this issue [45]. Finally, it is critical to understand that achieving broad adoption of end-to-end encryption took decades. Data or payload leakage often seems more threatening than metadata, especially when robust encryption is missing. However, thanks to the increasing deployment of encryption, the public discourse is changing [1].

This paper presents Maybenot, a work in progress on a *framework* for traffic analysis defenses. Maybenot gets its name and logo from its purpose: to shed doubt on an attacker's ability to draw conclusions from traffic analysis of a protocol's encrypted network traffic. The logo of Maybenot is the thinking face emoji 🤔 (Unicode U+1F914). Maybenot is, by design, easy to integrate into existing protocols with the goal of being simple yet expressive enough to realize a wide range of traffic analysis defenses.

In the same vein as related work [23, 55, 56, 70], we think integrating a framework rather than a specific defense might be motivated short-term given the rapid developments around traffic analysis attacks and defenses. Figure 1 shows example protocols where Maybenot integration could protect against traffic analysis attackers in different threat models. For example, Maybenot could be integrated with TLS as part of HTTPS to protect against web*page* fingerprinting (Figure 1a), as opposed to web*site* fingerprinting as in VPNs (Figure 1b) or Tor (Figure 1c).

We make the following contributions:

- A detailed description of the Maybenot traffic analysis defense framework. Maybenot is an evolution of the Tor Circuit Padding Framework [55, 56], generalizing and improving its design, as well providing a clean abstraction as a library (Rust crate) enabling ease of integration beyond Tor (Section 3).

WPES '23, November 26, 2023, Copenhagen, Denmark

ACM ISBN 979-8-4007-0235-8/23/11.
https://doi.org/10.1145/3603216.3624953

- A framework with support for Maybenot machines, supporting probabilistic transitions, blocking actions, and the ability to construct constant-rate defenses (Section 4).
- The simple bare-bones Maybenot simulator for rapid development and testing of Maybenot machines by simulating their impact on undefended network traces (Section 5).
- Implementation and evaluation of the state-of-the-art website fingerprinting defenses FRONT [22] and RegulaTor [33] as Maybenot machines (Section 6).
- Discussion of key challenges, trade-offs, and opportunities around state-based defenses based on lessons learned from our evaluation (Section 7).

Section 2 briefly presents background, Section 8 related work on traffic analysis defense frameworks, and Section 9 conclusions.

2 BACKGROUND

Maybenot primarily originates from Website Fingerprinting (WF) defense work, closely linked to end-to-end flow correlation/confirmation. We will focus on WF in Section 2.1 and the broader influence of traffic analysis defenses in Section 2.2.

2.1 Website Fingerprinting

In the WF setting, a local passive attacker monitors encrypted client traffic to a website via a proxy or relay, typically Tor [18], but also possibly a VPN [11, 12, 29, 30, 44, 71] or a new standard like MASQUE related to iCloud Private Relay [5, 53]. The attacker's goal is to deduce the visited website or its subpage. We overview WF attacks in Section 2.1.1 and defenses in Section 2.1.2. We survey existing frameworks for defense implementation in Section 2.1.3.

2.1.1 Attacks. WF attacks use manual or automatic feature engineering based on packet sequences, directions, timestamps, and sizes. Important manual feature-engineered attacks include CUMUL [52] and k-fingerprinting [27]. However, post-2016 advances in deep learning facilitated superior automatic feature engineering [2, 7, 61, 64, 69]. Deep Fingerprinting (DF) [69] and Tik-Tok [61] are noteworthy, showing significant improvements over manual feature engineering.

2.1.2 Defenses. The defense community has been working on various techniques, including imitation, regulation, alteration, traffic splitting, and adversarial techniques [45]. Imitation defenses make traffic from one website resemble that from another [50, 74, 76]. Regulation defenses, like Tamaraw [10] and RegulaTor [33], regulate traffic to match a target trace [24, 40]. Alteration defenses, like FRONT [22] and DeTorrent [32], unpredictably modify the traffic to make traces harder to classify. Traffic splitting defenses send traffic unpredictably over multiple paths [28, 43, 73], and adversarial techniques aim to thwart deep learning-based attacks [48, 60].

2.1.3 Frameworks. Several frameworks have been developed to provide a platform for the implementation of defenses. The Tor Circuit Padding Framework [55, 56] allows for padding-based defenses to be modeled using state machines and negotiated between clients and relays. QCSD [70] shapes QUIC [35] traffic at the client side without requiring server support, which boosts the potential for adoption of defenses. WFDefProxy [23] makes use of the obfs4 [4] Pluggable Transport [57] to support defenses between Tor clients and bridges and enable real-world evaluation of these defenses. We review these three frameworks in detail in Section 8.

2.2 Real-World Impact of Traffic Analysis

The real-world impact of traffic analysis attacks is widely discussed in both the WF [14, 39, 52, 54, 75] and wider traffic analysis communities [15–18, 41, 46, 47, 51, 65]. When evaluating defenses, we consider empowered adversaries who may have unrealistic capabilities or simplified assumptions. However, these might reflect something other than the actual traffic analysis threat. A crucial conclusion is the inherent trade-off between defense effectiveness and efficiency. Any defense framework's ability to adjust defenses to fit different use cases is a primary feature. Hence the name Maybenot: the goal is to enable doubt about an attacker's conclusions from traffic analysis.

3 MAYBENOT FRAMEWORK

Figure 2 shows an overview of the Maybenot framework and its integration with an encrypted communication protocol. Once integrated, Maybenot *machines* will trigger *actions* to take based on *events* describing the communication. Machines are probabilistic finite state machines, described in detail later in Section 4. The framework is simply a means of running zero or more machines. We implemented Maybenot in Rust as a library (crate). Basic type definitions are used throughout this and the following section to explain Maybenot and its integration, taking a top-down approach. Further implementation details are available in the complementary pre-print version of this paper [59].

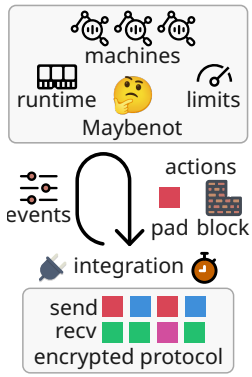


Figure 2: Overview of the Maybenot framework.

3.1 Instantiation

The first step in integrating the framework is to create an instance, e.g., as part of circuit creation in Tor or when establishing a peer connection in WireGuard. Creating an instance requires zero or more machines, blocking and padding limits, the current time, and the MTU of the connection.

Machines are provided by reference and read-only, enabling multiple instances of the framework to safely and efficiently share machines. Instead, each instance maintains a minimal runtime state per machine (64 bytes, regardless of machine size). For the sake of simplicity, the framework does not support dynamically adding or

(a) TLS client and server. (b) VPN client and relay. (c) Tor client and middle relay.

Figure 1: Example protocols and settings where Maybenot can be integrated to protect against traffic analysis by attackers.

removing machines. Instead, creating an instance of the framework is a lightweight operation.

Limits are fractions of the total duration spent blocking and the total bytes of padding sent. The framework keeps track of and enforces these limits for all running machines and respective individual limits set per machine (see Section 4). Limitations are fundamental to traffic analysis defenses, typically offering efficiency and effectiveness trade-offs.

Passing the current time to the framework—instead of having the framework keep track on its own—results in a more straightforward framework that is easier to test and use in simulation. It also makes constructing defenses that operate in steps easier by using time as a counter. Finally, the MTU of the communication channel is needed to restrict the size of padding packets in padding actions.

Once instantiated, the integration between the protocol and framework entails triggering events (Section 3.2) and processing the resulting actions (Section 3.3).

3.2 Triggering Events

Events describe the communication channel. The integrator periodically triggers one or more events in the framework together with the current time. The events are shown in Figure 3.

```
pub enum TriggerEvent {
    NonPaddingRecv { bytes_recv: u16 },
    PaddingRecv { bytes_recv: u16 },
    NonPaddingSent { bytes_sent: u16 },
    PaddingSent { bytes_sent: u16, machine: MachineId },
    BlockingBegin { machine: MachineId },
    BlockingEnd,
    LimitReached { machine: MachineId },
    UpdateMTU { new_mtu: u16 },
}
```

Figure 3: Events to trigger in the Maybenot framework.

Normal traffic is referred to as non-padding. Events that describe padding and non-padding all require the size of the data in bytes. In addition, when padding is sent, the machine that generated the padding is identified for sake of machine-specific padding limits and scoping events in the framework to the relevant machine.

Blocking actions (see Section 3.3) either begin or end, and when they begin, the responsible machine is identified in the corresponding event to enable the tracking of machine-specific limits. The `LimitReached` event is an internal event triggered by the framework when a machine reaches a state limit (explained in Section 4). Finally, `UpdateMTU` provides a way to update the MTU without recreating the framework.

3.3 Scheduling and Performing Actions

Triggering one or more events in the framework returns zero or more actions that should be *scheduled* by the integrator. Each machine running in the framework has at most one scheduled action at any point in time. Figure 4 shows the possible actions. The `Cancel` action simply cancels any scheduled action for a machine.

```
pub enum Action {
    Cancel { machine: MachineId },
    InjectPadding {
        timeout: Duration,
        size: u16,
        bypass: bool,
        replace: bool,
        machine: MachineId,
    },
    BlockOutgoing {
        timeout: Duration,
        duration: Duration,
        bypass: bool,
        replace: bool,
        machine: MachineId,
    },
}
```

Figure 4: Actions in the Maybenot framework.

The `InjectPadding` action specifies that a padding packet of a particular size in bytes should be sent after a specific `timeout`. Similarly, the `BlockOutgoing` action specifies that all outgoing traffic should be blocked for a particular *duration* of time after a timeout. Both actions specify the identifier of the machine, to be used when triggering the corresponding event after timeout.

The `bypass` and `replace` flags, and their interactions, get a little complicated. When blocking begins, the `bypass` flag determines if the blocking can be bypassed by padding with the `bypass` flag set. This enables *bypassable* blocking as well as the construction of defenses that fail closed (blocking without the flag set). The `replace`, for blocking, determines if the blocking should replace any existing ongoing blocking, or if the longest remaining duration of blocking should be set. For padding, the `replace` flag allows the padding to be injected to be replaced by any normal (non-padding) data queued or recently sent (within use case-specific limits as determined by the integrator). Together, the `bypass` and `replace` flags enable *constant-rate defenses* by first setting bypassable blocking followed by constant-rate padding with both flags set, resulting in either padding or normal traffic being sent at a fixed rate.

4 MAYBENOT MACHINES

A machine encodes the logic of when to take what action based on events. Machines are probabilistic finite state machines, building on "padding machines" (nondeterministic finite state machines) in the Tor Circuit Padding Framework [55, 56], further compared in

Section 8. Figure 5 shows the definition of a Maybenot machine. A machine consists of four fields concerning limits, a vector of one or more states, and a flag `include_small_packets` indicating if the framework should consider packets of small sizes (helpful in ignoring ACKs in, e.g., WireGuard).

```
pub struct Machine {
    pub allowed_padding_bytes: u64,
    pub max_padding_frac: f64,
    pub allowed_blocked_microsec: u64,
    pub max_blocking_frac: f64,
    pub states: Vec<State>,
    pub include_small_packets: bool,
}
```

Figure 5: A Maybenot machine.

The fraction limits behave like the framework-wide limits but apply only to the machine in question. In addition, it is possible to provide padding and blocking budgets (in absolute terms) that circumvent all limits. Such budgets are particularly useful at the start of connections, where fraction-based limits are impractical.

4.1 State

A machine must have at least one state. Figure 6 shows the definition of a state. For now, know that `Dist` represents a probability distribution that can be sampled; more details are provided in Section 4.2. Upon transitioning to a state (from another state or itself), the framework samples a `timeout` for when a particular action should be taken. We look closer at transitions, actions, and limits.

```
pub struct State {
    pub timeout: Dist,
    pub action: Dist,
    pub action_is_block: bool,
    pub bypass: bool,
    pub replace: bool,
    pub limit: Dist,
    pub limit_includes_nonpadding: bool,
    pub next_state: HashMap<Event, Vec<f64>>,
}
```

Figure 6: The state of a machine.

4.1.1 Transition. A machine has a current state, tracked by the framework as part of its runtime. The first state is state 0 in a machine's `states` vector. The `next_state` map of a state maps events to probability vectors. For each event (see Figure 3), there is a vector of probabilities to transition to each state in the machine as well as two meta events: `StateCancel` and `StateEnd`. The probabilities p sum to at most one; $0 \leq \sum_i^{n+2} p_i \leq 1$, for a machine with n states. The meta event `StateCancel` cancels any scheduled action without transitioning to a new state. The meta event `StateEnd` does not cancel any scheduled action but permanently ends the machine, preventing future transitions. Note that the probabilities above do not need to sum to one to support machines that only transition to any state with a small probability.

4.1.2 Action. There are two possible actions (in addition to the `StateCancel` action triggered by a state transition): blocking and padding, determined by the `action_is_block` flag. The `action` distribution either specifies the duration to block for or the size of the padding. The action is subject to the `bypass` and `replace` flags, as described in Section 3.3.

4.1.3 Limit. The per-state limit is distinct from the per-machine or per-framework limits on padding and blocking. The per-state limit gets sampled upon transitioning to the state from *another* state. The limit gets decremented on each transition to itself, i.e., the same state. If the limit reaches 0, it prevents the scheduling of future actions. The non-padding sent event also decreases the limit with the `limit_includes_nonpadding` flag set. This behavior can be helpful in conjunction with padding actions to send an exact number of packets, regardless of whether they are padding or non-padding packets.

4.2 Distribution

Maybenot supports a number of distributions, listed in Figure 7, provided by the rand_dist[1] crate. Together with picking the distribution and its relevant parameters, each sample is rounded to a positive value, potentially to a discrete value (bytes for padding), and optionally clamped to a min and max value. The unit of time in the framework is *microseconds*.

```
pub enum DistType {
    None,
    Uniform,
    Normal,
    LogNormal,
    Binomial,
    Pareto,
    Poisson,
    Weibull,
    Gamma,
    Beta,
}
```

Figure 7: Supported distributions.

4.3 Serialization

Machines can be serialized to and from hex-encoded strings using either Maybenot's space inefficient but simple and safe format, or just with Serde[2] (a commonly used framework for serialization and deserialization in Rust). This enables machines to be dynamically shared and changed at runtime with minimal overhead.

5 MAYBENOT SIMULATOR

The goal of the Maybenot simulator is to support the rapid development and testing of Maybenot machines. Instead of collecting new datasets of defended network traces, developers can simulate how existing network traces would change with one or more machines running at the client and server. Using the simulator entails (i) parsing a base (existing) network trace and (ii) running the simulation.

The goal of the simulator is not to be a perfect simulator—whatever that now would entail given that the framework is designed to be integrated into a wide range of protocols—but to be a helpful simulator. Hopefully, most development work can be done with the simulator, and only fine-tuning is needed for later integration. For example, in the case of Tor, early development can happen

[1] https://crates.io/crates/rand_distr
[2] https://serde.rs/

in the Maybenot simulator, with later large-scale experiments using Shadow [36], followed by real-world deployment.

Figure 8 shows the average simulation time for the Maybenot simulator for nine different websites. Each simulation simulated up to 10,000 events with heavy Maybenot machines on both client and server generating blocking and padding actions, similar to RegulaTor [33]. Generated using Criterion.rs, a statistics-driven benchmarking library for Rust, with 100 samples per website each over 5s runtime on a 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz laptop. Simulation is single-threaded and uses on the order of 1MiB of memory. Much can probably be optimized.

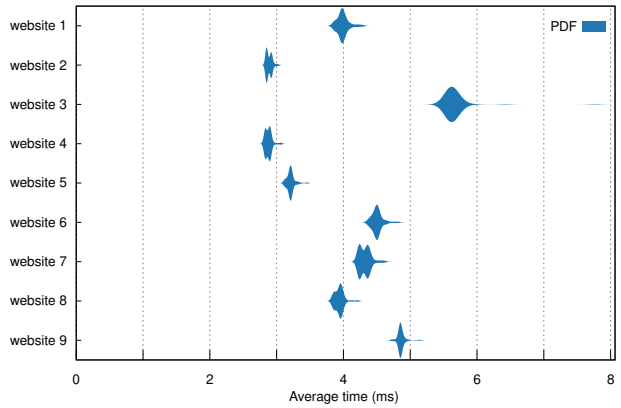


Figure 8: Violin plot of simulation time for nine websites.

5.1 Base Network Traces

Figure 9 shows an example network trace of ten first packets from WireGuard when visiting google.com and how to parse it. The trace is a string of lines, where each line is a packet with the format "timestamp,direction,size\n". The timestamp is the number of nanoseconds since the start of the trace, the direction is either "s" for sent or "r" for received (from the perspective of the client), and the size is in bytes.

To parse the trace, the simulator also takes a delay, which is the latency between the client and server. The delay is used to simulate *event queues* for the client and server, such that the packets are sent and arrive at the client exactly as in the provided trace. This is a crude approximation of the network between the client and server and should probably be improved to make the simulator more useful in the long term [37].

```
let raw_trace = "0,s,52
19714282,r,52
183976147,s,52
243699564,r,52
1696037773,s,40
2047985926,s,52
2055955094,r,52
9401039609,s,73
9401094589,s,73
9420892765,r,191";

let delay = Duration::from_millis(10);
let mut input_trace = parse_trace(raw_trace, delay);
```

Figure 9: Parsing an example trace.

5.2 Simulating Machines

Figure 10 shows an example of simulating a machine on the trace from Figure 9. The simulator supports zero or more machines running at the client and server. Because machines may run forever (e.g., sending more padding on padding being sent), it is possible to set the maximum number of events (client and server) to simulate and a flag to filter out events only related to network packets. The output of the simulator is a vector of events (see Figure 3) describing the simulated trace, each annotated with a timestamp and flag (for client or server), which is straightforward to parse.

```
use maybenot_simulator::sim;
use maybenot::machine::Machine;
use std::{str::FromStr, time::Duration};

let s = "789cedca2101...";
let m = vec![Machine::from_str(s).unwrap()];

let trace = sim(
    vec![m], // client machines
    vec![], // server machines
    &mut input_trace,
    delay,
    100, // max events
    true, // only return packet events?
);
```

Figure 10: Simulating a machine on a trace.

6 IMPLEMENTING DEFENSES

To evaluate Maybenot's support for proposed website fingerprinting defenses, we created machines that implement two state-of-the-art defenses: FRONT [22] and RegulaTor [33].

6.1 FRONT

FRONT [22] is a padding-only defense intended to conceal useful features present at the beginning of a trace. It samples time values from a Rayleigh distribution before every download, and a padding packet is sent at each of these times relative to download start.

To initialize the defense, the client samples a *padding count* n_c from a discrete uniform distribution with range $[1, N_c]$, and the server samples n_s from the range $[1, N_s]$; parameters N_c and N_s are the client and server's padding budgets, respectively. The client and server also sample a *padding window* (w_c and w_s) from a continuous uniform distribution with range $[W_{min}, W_{max}]$.

After selecting parameters, the client samples n_c time values (in seconds) from a Rayleigh distribution with $\sigma = w_c$, and the server samples n_s values from a Rayleigh distribution with $\sigma = w_s$. A padding packet is then sent at each of these times relative to download start, and no further padding is sent once a download completes. The use of varying padding counts and windows allows for trace-to-trace randomness, which reduces the ability of a classifier to train effectively on defended traces.

Maybenot FRONT. Since the client and server follow the same steps to enact the defense, we implemented FRONT in Maybenot using a single machine design, which we refer to as Maybenot FRONT. It consists of a `START` state and a number of `PADDING` states arranged in sequence, as shown in Figure 11. Each machine is

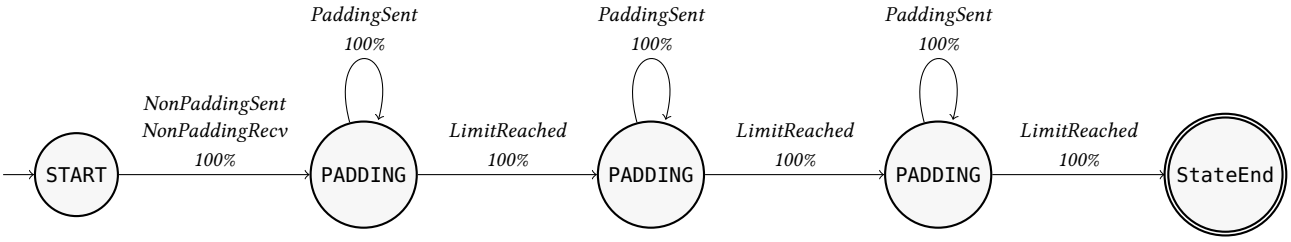


Figure 11: Maybenot FRONT machine with three `PADDING` states.

characterized by its padding budget N, maximum padding window W_{max}, and number of `PADDING` states ψ.

A transition occurs from `START` to the first `PADDING` state when the first packet is sent or received on a connection, as determined by the *NonPaddingSent* and *NonPaddingRecv* events. Maybenot FRONT then proceeds sequentially through the remaining `PADDING` states until it reaches `StateEnd`, stopping the defense.

When a `PADDING` state is transitioned to, it generates a padding action with a sampled timeout value; size is sampled from a uniform action distribution with $a = b$, so the integrator will send a single packet after the timeout expires. As a result, a *PaddingSent* event will be triggered, 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; a normal timeout distribution is used, and parameters are selected to approximate the distribution of inter-packet delays that would result during the 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:

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

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

μ is selected to be the inter-packet delay that would result in exactly N/ψ packets 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 packets that would most likely be sent during its interval, this would allow for a precise approximation of the sending rate of padding packets that would result from a Rayleigh distribution. However, such a design would exclude 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, padding count 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.

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 design based on the same principles as Maybenot FRONT but with multiple *pipelines* that have different padding budgets.

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. See Appendix A for further details.

6.2 RegulaTor

RegulaTor [33] is a regularization defense for Tor [18] based on the observation that Tor traffic consists of "surges" of cells sent within a short period of time along with intervening periods of lower cell volume. Surges are typically present at the beginning of a download, and traffic decreases exponentially as time elapses. RegulaTor leverages this fact to reduce the uniqueness of surges and, consequently, their usefulness as WF attack features.

This is accomplished by sending download traffic at a set initial rate R (s^{-1}) which decreases according to a decay function: at any given time, the current rate is calculated by RD^t, where D is a decay parameter and t is the time elapsed since surge start. The surge start time is initialized to download start time, but if the number of queued cells exceeds a threshold ($T\cdot$ rate), a new surge begins, and traffic is once again sent at the initial rate.

However, to reduce overhead, the relay samples a padding count for each download from a discrete uniform distribution with range $[0, N]$, where N is a parameter specifying the padding budget. The relay 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.

Upload traffic is sent at a constant fraction of the rate of download traffic: the client sends one cell for every U cells received. There is one exception to this: to ensure download progress, any queued cells will be sent immediately after they have been waiting for a configurable amount of time (C seconds).

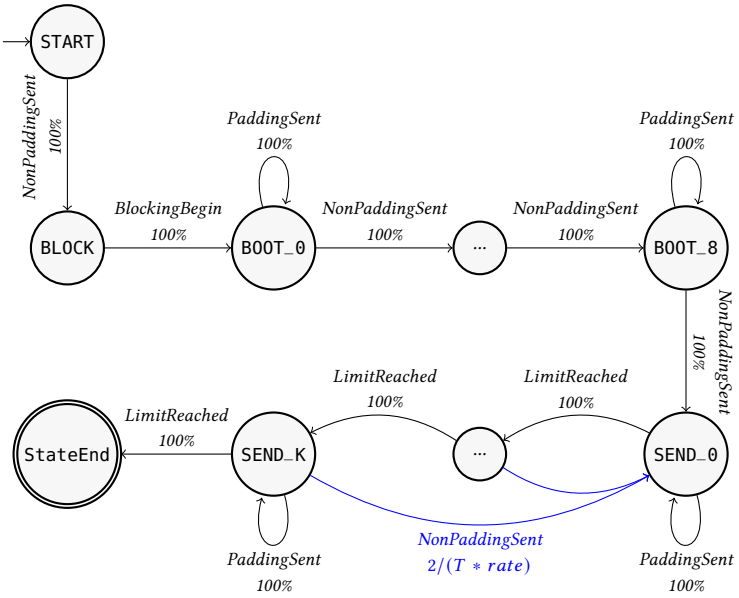


Figure 12: Maybenot RegulaTor relay machine with K `SEND` states.

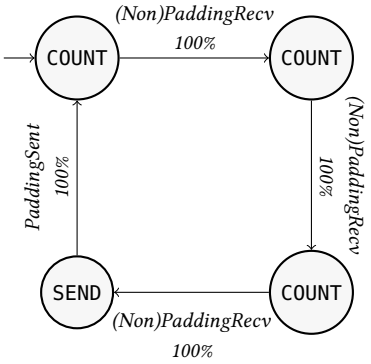


Figure 13: Maybenot RegulaTor client machine with $U = 3$.

Maybenot RegulaTor. We created two machine designs to approximate RegulaTor, one for clients and one for relays. We refer to these machines collectively as Maybenot RegulaTor.

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. This machine is depicted in Figure 12.

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. Once the integrator 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.

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. We implement this behavior probabilistically with a small chance of transitioning back to `SEND_0` when a *NonPaddingSent* event is triggered.

Our implementation 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; there is no way to specify that the sending rate should be capped.

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. Thus, one cell is sent for every U received, as in Figure 13.

If U is *not* a whole number, 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.

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.

6.3 Evaluation

We evaluated our implementations of FRONT and RegulaTor using the BigEnough dataset [45] collected by Mathews et al. Specifically,

Table 1: Parameters selected for each implementation of FRONT and RegulaTor.

(a) FRONT parameters.

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	—

(b) RegulaTor parameters.

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	—

we used the monitored set, which consists of 19,000 traces of web page visits over Tor. 95 websites were chosen based on popularity metrics from the Open PageRank Initiative, and 10 subpages from each site were visited 20 times, for a total of 200 traces per site.

We created defended datasets with Gong et al.'s FRONT simulator [22] and Holland and Hopper's RegulaTor simulation scripts [33]. We also developed three Rust programs [78] to generate machines for Maybenot FRONT, Pipelined FRONT, and Maybenot RegulaTor based on provided parameters, and we supplied them to the Maybenot simulator to defend the BigEnough dataset. To aid in comparison, we removed trailing padding packets from all traces in the Maybenot-defended datasets.

We considered the two configurations of each defense discussed by their authors: `FT-1` and `FT-2` for FRONT, and `RT-Light` and `RT-Heavy` for RegulaTor. We selected parameters for Maybenot FRONT and Pipelined FRONT to match the bandwidth overhead of simulated FRONT, and we matched the *latency* overhead of Maybenot RegulaTor to that of simulated RegulaTor. Parameters are summarized in Table 1. Note that ψ is number of pipelines followed by number of `PADDING` states per pipeline for Pipelined FRONT, and ω is cells per state in Maybenot RegulaTor.

Using the defended datasets, we calculated the similarity between traces defended by simulation and those defended with Maybenot FRONT, Pipelined FRONT, and Maybenot RegulaTor; the bandwidth and latency overhead of each implementation; and performance against the CUMUL [52], DF [69], and Tik-Tok [61] attacks.

6.3.1 Trace Comparison. Following the methodology of Smith et al. [70], we represented each trace in our defended datasets with two *aggregated time series*, one for download traffic and one for upload traffic. Each aggregated time series was computed by partitioning total download time into fixed-length windows of I ms and creating a sequence of the number of packets received or sent by the client during each window. For our evaluations, we set $I = \{25, 50\}$.

We measured trace similarity by computing the Pearson correlation coefficient and a longest common subsequence (LCSS) measure on the aggregated time series of corresponding traces. LCSS was calculated by dividing the length of the longest common subsequence by the shorter of the lengths of the two aggregated time series being compared. The results for FRONT are shown in Figures 14 and 15, and the results for RegulaTor are in Figure 16.

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. Similar results are observed with `FT-2`. Correlation for upload traffic is lower, which is likely due to a higher ratio of padding to non-padding traffic. The minimum median LCSS among all cases is 0.43, which reflects the fact that padding differences are most apparent near the beginning of a trace.

We attribute low correlation for some traces to the use of individual states' limit distributions to induce variation of the padding count and window. It is possible for limits sampled for adjacent `PADDING` states to differ, reducing correspondence to the Rayleigh distribution shape. We also note that each implementation may have selected a different padding count and window when defending the same trace, since simulations were run independently.

A moderate correlation for download traffic is observed with Maybenot RegulaTor. This is due primarily to two 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; and it sets a constant rate throughout a download, whereas simulated RegulaTor caps the sending rate after a padding count has been exceeded. 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.

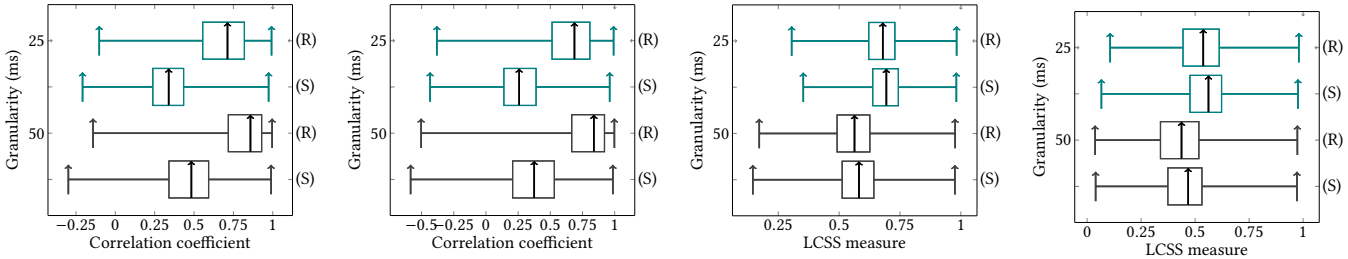
The median LCSS of `RT-Light` is about 0.46 with $I = 25$ and 0.42 with $I = 50$; although `RT-Heavy` has a higher median LCSS in both cases, its interquartile range is much greater. This indicates that traces were more similar near the beginning and that much of the observed variation is due to different surge restart times: the probability of restarting a surge in Maybenot RegulaTor decreases as sending rate increases, and it was 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.

6.3.2 Overhead. We calculated bandwidth overhead by dividing the number of padding bytes in each defended trace by the number of non-padding bytes, and latency overhead refers to the time to the last non-padding packet in a defended trace compared to original download time. Mean results are in Table 2.

About 80% bandwidth overhead was incurred by `FT-1` and 125% by `FT-2`; there was little variation among implementations since

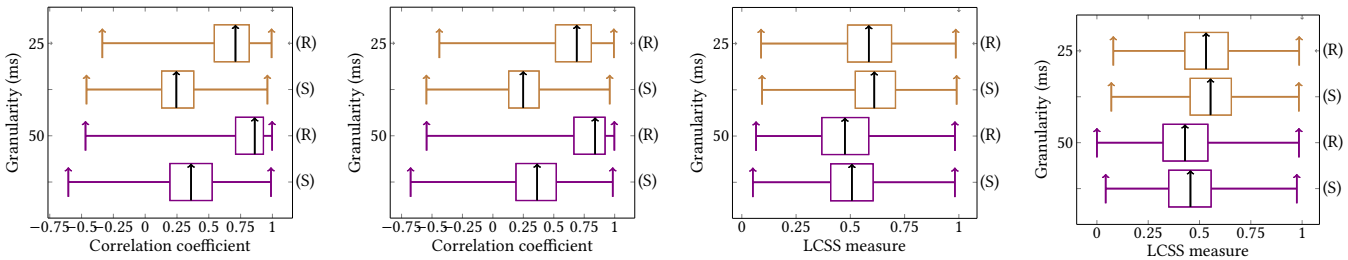
Table 2: Closed-world attack performance, average bandwidth and latency overhead.

Defense	Attack accuracy (%)			Bandwidth overhead (%)			Latency overhead (%)
	CUMUL	**DF**	**Tik-Tok**	**Send**	**Receive**	**Overall**	
Undefended	94.66	95.89	94.00	–	–	–	–
Maybenot FT-1	27.68	72.11	64.00	597.51	41.84	78.24	–
Pipelined FT-1	15.72	58.00	55.89	613.45	43.13	80.49	–
Simulated FT-1	12.06	48.32	49.47	642.97	44.80	83.98	–
Maybenot FT-2	23.41	68.11	50.84	998.77	70.05	130.89	–
Pipelined FT-2	13.45	49.37	48.32	922.24	64.60	120.79	–
Simulated FT-2	9.12	40.95	45.79	952.91	66.24	124.32	–
Maybenot RT-Light	6.38	6.63	9.89	747.98	138.23	178.18	21.11
Simulated RT-Light	5.65	6.42	22.00	424.62	44.93	69.80	22.01
Maybenot RT-Heavy	6.88	8.11	10.00	1091.88	151.35	212.96	15.31
Simulated RT-Heavy	4.53	5.79	15.16	537.66	73.86	104.24	17.52



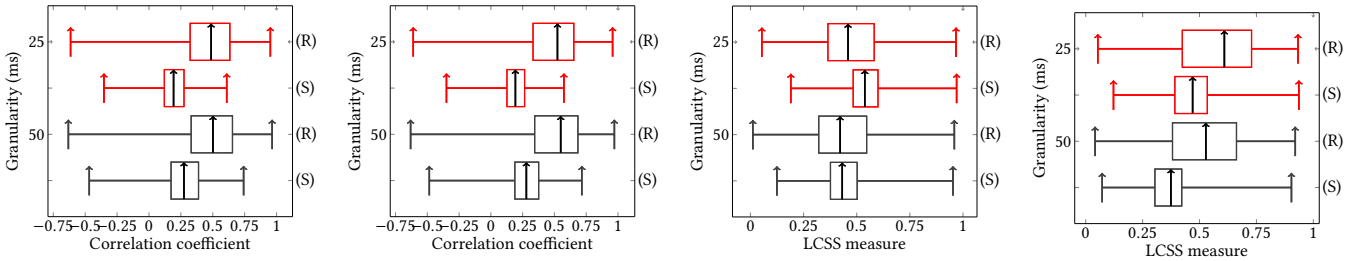
(a) Correlation, `FT-1`. (b) Correlation, `FT-2`. (c) LCSS, `FT-1`. (d) LCSS, `FT-2`.

Figure 14: Trace comparison results, simulated FRONT and Maybenot FRONT.



(a) Correlation, `FT-1`. (b) Correlation, `FT-2`. (c) LCSS, `FT-1`. (d) LCSS, `FT-2`.

Figure 15: Trace comparison results, simulated FRONT and Pipelined FRONT.



(a) Correlation, `RT-Light`. (b) Correlation, `RT-Heavy`. (c) LCSS, `RT-Light`. (d) LCSS, `RT-Heavy`.

Figure 16: Trace comparison results, simulated RegulaTor and Maybenot RegulaTor.

Maybenot FRONT and Pipelined FRONT's parameters 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 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 with 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.

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. A 108% increase in bandwidth overhead makes Maybenot RegulaTor impractical in its current state.

6.3.3 Attacks. We evaluated CUMUL [52], DF [69], and Tik-Tok [61] in the closed-world setting against undefended traffic, simulated FRONT and RegulaTor, and our implementations. We did this using the scripts provided by Gong et al. for CUMUL [22] and those provided by Rahman et al. for DF and Tik-Tok [61]. 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 2.

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` configuration 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`.

Maybenot FRONT was much less effective than simulated FRONT: it reduced Tik-Tok's accuracy to 50.84% with `FT-2`, but it only decreased accuracy to a minimum of 64% in all other cases. Pipelined FRONT was more effective: DF was the best attack against it, attaining 58% accuracy with `FT-1` and 49.37% accuracy with `FT-2`. We attribute Pipelined FRONT's success to high variation in inter-packet timing, padding count, and padding window.

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`.

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 feasible for use in Tor.

7 DISCUSSION

7.1 Challenges Expressing Defenses

Our evaluation demonstrates Maybenot's potential to support proposed website fingerprinting defenses, but it also exposes some inherent limitations of the framework along with areas of improvement, which will allow for more expressive and concise defense implementations in future versions.

We found that it is possible to effectively approximate FRONT, though fairly complex machines are required to incorporate sufficient trace-to-trace randomness. Even so, Pipelined FRONT provides slightly less protection against attacks than simulated FRONT, highlighting the challenges associated with implementing handcrafted defenses, many of which are not designed specifically to be implemented with a state machine model.

Though we were able to closely match the fundamental aspects of RegulaTor's behavior, several challenges lay in features that required knowledge of queued cells and counting. We found that it was not possible to directly implement padding limits, surge restarting, or the C parameter in the client machine. With the addition of a few new events and actions, Maybenot could support a more practical implementation of RegulaTor.

However, certain defenses do not lend themselves to implementation in the framework. We attempted to implement Surakav [24], which aims to make cell sequences match a *reference trace*, with certain adjustments to decrease overhead. We successfully created a machine to replay reference traces [78], but we could not include any of the remaining features of Surakav due to the coordination required between client and server.

Fortunately, despite Maybenot's inherent limitations, the same features that would improve our implementations of FRONT and RegulaTor may allow for heuristics that approximate more complicated defenses such as Surakav. Based on our experience with FRONT and RegulaTor, we believe that new events for queue monitoring, and allowing machines to create their own timers as well as counters, would serve both of these purposes.

With these improvements, 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 using a timer.

7.2 Why a Framework

What is the merit of developing a defense framework instead of directly deploying defenses? For one, the last decade saw significant advancements in traffic analysis attacks, notably website fingerprinting. In parallel, as attacks have improved, so have defenses. A prime example here is from the Tor Project. They started with the goal of implementing the WTF-PAD defense by Juárez et al. [40], but the Deep Fingerprinting attack by Sirinam et al. [69] significantly reduced its effectiveness compared to earlier evaluations against (among others) the k-fingerprinting attack by Hayes and Danezis [27]. So instead of implementing WTF-PAD, the Tor Circuit Padding Framework was born [55, 56].

A framework also allows for effortlessly combining multiple defenses. Combining defenses is effective [24, 28, 73]. The selection of combined defenses could also be dynamic and adaptive, e.g., based on the current non-padding traffic to hide moments of inactivity or disabled for bulk downloads. A framework is part of *orchestrating* traffic analysis defenses.

Another aspect is moving from web*site* fingerprinting to web*page* fingerprinting defenses. While most web traffic is encrypted, it goes directly between the client and involved servers. Therefore, any network attacker can perform website fingerprinting in most cases by simply observing all relevant IP addresses [68]. In a webpage fingerprinting setting, optimal defenses would be per website and optimized for the pages distributed by that website (and all involved web servers hosting third-party content [68]). Application-layer knowledge can help create more effective and efficient defenses [13]. A framework for defenses integrated into, e.g., QUIC or HTTP/3, would allow for tailored per-site defenses. Websites could distribute serialized defenses to clients upon connection establishment, or the server could implement the client side of the defense partly in the application layer (the inverse of QCSD [70]).

7.3 On Expressiveness

Related work on defense frameworks (further described in Section 8) provides varying levels of defense expressiveness, ranging from supporting the Go programming language in WFDefProxy [23] to a fixed trace regularization algorithm in QCSD [70] and state machines in the Tor Circuit Padding Framework [55, 56]. In a similar vein, in the censorship circumvention space, Proteus [72] and Marionette [20] take different approaches. Marionette models define probabilistic state machines where actions are blocking or non-blocking arbitrary Python functions that trigger transitions to other states. Proteus, in turn, specifies a domain-specific language for circumvention protocols executed within a locked-down runtime, where both the expressiveness of the language and its runtime are restricted (e.g., no dynamic memory, concurrency, or floating point arithmetic, while the standard system library/API is an allowlist to allow, e.g., cryptographic operations and running timers). Finally, yet another example is Flexible Anonymous Networks (FAN) [66] that makes anonymous networks programmable by using eBPF [21] running sandboxed in userspace [49] (eBPF is typically used within the Linux kernel) as plugins hooked within the same process as the anonymity network implementation.

A key consideration for the expressiveness of a framework comes down to the value of safety and security of protocol updates, as well as the value of updates in and of themselves (as discussed in Section 7.2). For example, in Tor the hourly consensus could contain updates from a trusted source. Despite the inherent (necessary) trust in the consensus, such potentially frequent updates could benefit from security and safety guarantees by design in the protocol update mechanism/framework. Full expressiveness in, e.g., Go or Python is obviously potentially unsafe; even limited approaches for allowing programmability based on more restricted languages like eBPF or Webassembly with limited allow-listed APIs are fraught with challenges necessitating runtime overheads and careful considerations [9, 25, 26, 38, 42, 72].

For Maybenot, we opt for slowly expanding the capabilities of Maybenot state machines. While we already had modest success in porting FRONT and RegulaTor as-is, we know from the evaluation of Mathews et al. [45] that defenses tailored to state machines are competitive (see Interspace [45, 58]). As we expand Maybenot machines to support richer expressiveness of defenses, we aim to keep machine definitions safe and secure to support dynamic and adaptive use cases. The goal is to be able to run effective and efficient defenses within Maybenot, not necessarily to support every possible type of defense. Regardless, as an API for integrating a runtime for traffic analysis defenses, the interface of Maybenot described in Section 3 is fundamentally solid: any defense will have to hook into a protocol to collect events and use a combination of bandwidth- and latency-inducing actions [16, 17, 79]. If future defense frameworks opt for different runtimes, e.g., based on a unified runtime with censorship circumvention protocols like Proteus, any existing integration efforts from Maybenot will be well spent.

8 RELATED WORK

Maybenot stems from the Tor Circuit Padding Framework, developed by Perry and Kadianakis [55, 56]. In turn, the Tor Circuit Padding Framework evolved from WTF-PAD [40], a website fingerprinting defense by Juárez et al. based on Adaptive Padding, a concept introduced by Shmatikov and Wang [67].

8.1 Tor Circuit Padding Framework

As its name implies, the Tor Circuit Padding Framework is a framework for generating padding within Tor circuits. It is part of Tor's C implementation. The framework facilitates clients' negotiations with network relays to enable hardcoded "padding machines". At the time of writing, only a pair of simple padding machines are live on the Tor-network [56]. These machines aim to conceal the configuration of client-initiated onion service circuits by ensuring that regular circuits may also generate an identical cell sequence.

Maybenot stands out from the Tor Circuit Padding Framework because its design enables incorporation into various protocols thanks to being a standalone Rust library. In contrast, the Tor Circuit Padding Framework is tightly integrated into Tor and supports negotiating padding machines—a feature that Maybenot lacks. Maybenot machines support probabilistic state transitions and blocking

actions, control associated bypass and replacement flags, and handle padding of varying lengths. Additionally, Maybenot supports a broader range of distributions but removes histogram support. Maybenot also omits support for RTT-based estimates as timer offsets, mainly due to the need for clearly defined use cases, making Maybenot more streamlined.

8.2 QCSD

Smith et al. developed QCSD [70], a framework dedicated to traffic analysis defenses optimized for QUIC [35]. QCSD is a client-side framework that generates padding and induces traffic delays at the server using features of QUIC and HTTP/3 [8]. This focus on the client side is a boon for traffic analysis defense adoption as it eliminates the need for the server or any intermediate entity to support the framework. However, this strength is also its main weakness. The reliance of QCSD on QUIC and HTTP/3 at endpoints prevents it from only defending traffic between the client and intermediate relays/proxies. Websites that typically include resources from multiple domains/endpoints are another complication.

The QCSD counterpart to Maybenot machines is a client-side regularization algorithm. This algorithm shapes the connection to conform to a target trace (static or dynamically generated). This regularization algorithm could be split and implemented as Maybenot machines following the target trace. Shaping server-to-client traffic involves control messages, adding overhead, and the challenging task of precise server traffic shaping. Smith et al. propose that QUIC extensions could enable clients to shape server traffic more accurately.

8.3 WFDefProxy

Gong et al. present WFDefProxy [23], a framework for implementing website fingerprinting defenses and empirically assessing the defenses within real-world networks. WFDefProxy builds upon obfs4 [4], a Pluggable Transport (PT) [57] used by Tor, and is implemented as a bridge. Clients connect directly to a bridge before relaying traffic typically into the Tor network. This setup confines WFDefProxy defenses to protection against network adversaries between the client and bridge, excluding those in the Tor network or controlling the bridge (i.e., the bridge is a trusted entity). The development of defenses within WFDefProxy utilizes the Go programming language, offering a richer language environment than Maybenot machines, padding machines in the Tor Circuit Padding Framework, and QCSD's regularization algorithm.

Gong et al. provide an implementation of FRONT in WFDefProxy that is structured as a finite state machine; however, they describe it as operating on "trace-level" events [23] rather than the packet-level events which are building blocks of both Maybenot and the Tor Circuit Padding Framework. This implementation consists of three states–Ready, Start, and Stop. State transitions occur when packets are received, indicating the start of a download; and when the traffic rate falls below a threshold, which is assumed to indicate download completion. However, the actions taken upon each transition are rather complex, such as scheduling padding and coordinating explicitly with the relay.

Maybenot FRONT detects the beginning of a download similarly with the *NonPaddingRecv* and *NonPaddingSent* events, and a soft stop condition could be included with the addition of a timer to Maybenot, as described in Section 7. However, Maybenot does not support the complex actions possible with WFDefProxy: Maybenot FRONT simply sends a packet upon transition to a `PADDING` state, and events result directly from the actions of the machine. This highlights the primary functional difference between Maybenot and WFDefProxy: while Maybenot provides a set of simple actions, WFDefProxy allows actions to be programmed specifically for a defense, improving flexibility in implementation.

9 CONCLUSION

We presented Maybenot, a work-in-progress framework for traffic analysis defenses heavily inspired by the Tor Circuit Padding Framework [55, 56]. Defenses are implemented as probabilistic state machines, and the framework provides a standard interface for integrating them into protocols such as Tor [18], Wireguard [19], and QUIC [35]. To assist in developing defenses, we provide a simulator to simulate how provided network traces could change if provided machines were running at the client and server. By implementing and evaluating FRONT [22] and RegulaTor [33] we identified key challenges related to the limited expressiveness of state machines, paving the way for further improvements to Maybenot while being conservative for the sake of safety and security.

Our goal with Maybenot is to contribute towards widespread real-world use of traffic analysis defenses. We hope that Maybenot will be helpful for researchers, protocol implementers, and defenders. With the monumental progress in AI and machine learning, traffic analysis defenses will become increasingly important. Because we are in the middle of this AI revolution, a framework is likely worthwhile in the short to medium term until the dust settles. It took us decades to get to where we are today, making encrypted end-to-end communication the norm. We will need similar time to get to where we want to be with traffic analysis defenses.

ARTIFACT AVAILABILITY

The Maybenot Framework and Simulator are available at https://crates.io/crates/maybenot and https://crates.io/crates/maybenot-simulator. They are both dual-licensed under either the MIT or Apache 2.0 licenses.

Our implementations of FRONT and RegulaTor are available on GitHub under the BSD-3-Clause license at https://github.com/ewitwer/maybenot-defenses.

ETHICAL CONSIDERATIONS

We see no ethical concerns with our work. We used existing datasets of network traces to simulate the defenses.

ACKNOWLEDGMENTS

For their valuable feedback, we would like to thank Matthias Beckerle, Namitha Binu, Rasmus Dahlberg, Grégoire Detrez, Linus Färnstrand, David Hasselquist, Nick Hopper, Jan Jonsson, Jack Milless, Mike Perry, Florentin Rochet, Odd Stranne, Fredrik Strömberg, Björn Töpel, Shana Watters, Alice Zhang. This work was partially funded by Mullvad VPN, the Swedish Internet Foundation, and the Knowledge Foundation of Sweden. The paper was written responsibly using ChatGPT, Github Copilot, and Grammarly.

REFERENCES

[1] Paul Mozur Aaron Krolik and Adam Satariano. accessed 2023-07-03. Russia Seeds New Surveillance Tech to Squash Ukraine War Dissent - The New York Times. https://www.nytimes.com/2023/07/03/technology/russia-ukraine-surveillance-tech.html.

[2] Kota Abe and Shigeki Goto. 2016. Fingerprinting attack on Tor anonymity using deep learning. *Proceedings of the Asia-Pacific Advanced Network* 42 (2016), 15–20.

[3] Daniel Agnew. 2020. Google Trends Reveals Surge in Demand for VPN. https://www.namecheap.com/blog/vpn-surge-in-demand/.

[4] Yawning Angel. accessed 2023-02-18. obfs4. https://github.com/Yawning/obfs4/blob/master/doc/obfs4-spec.txt.

[5] Apple. 2021. iCloud Private Relay Overview. https://www.apple.com/privacy/docs/iCloud_Private_Relay_Overview_Dec2021.PDF.

[6] Richard Barnes, Benjamin Beurdouche, Raphael Robert, Jon Millican, Emad Omara, and Katriel Cohn-Gordon. 2023. *The Messaging Layer Security (MLS) Protocol.* Internet-Draft draft-ietf-mls-protocol-20. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-ietf-mls-protocol/20/ Work in Progress.

[7] Sanjit Bhat, David Lu, Albert Kwon, and Srinivas Devadas. 2019. Var-CNN: A Data-Efficient Website Fingerprinting Attack Based on Deep Learning. *Proc. Priv. Enhancing Technol.* 2019, 4 (2019), 292–310. https://doi.org/10.2478/popets-2019-0070

[8] Mike Bishop. 2022. HTTP/3. RFC 9114. https://doi.org/10.17487/RFC9114

[9] Daniel Borkmann. 2023. BPF and Spectre: Mitigating transient execution attacks – Daniel Borkmann, Isovalent. https://www.youtube.com/watch?v=6N30Yp5f9c4 accessed 2023-06-09.

[10] Xiang Cai, Rishab Nithyanand, Tao Wang, Rob Johnson, and Ian Goldberg. 2014. A Systematic Approach to Developing and Evaluating Website Fingerprinting Defenses. In *ACM SIGSAC*. 227–238. https://doi.org/10.1145/2660267.2660362

[11] Xiang Cai, Xin Cheng Zhang, Brijesh Joshi, and Rob Johnson. 2012. Touching from a distance: website fingerprinting attacks and defenses. In *CCS*.

[12] Heyning Cheng and Ron Avnur. 1998. Traffic analysis of SSL encrypted web browsing. *Project paper, University of Berkeley* (1998).

[13] Giovanni Cherubin, Jamie Hayes, and Marc Juárez. 2017. Website Fingerprinting Defenses at the Application Layer. *PETS* (2017).

[14] Giovanni Cherubin, Rob Jansen, and Carmela Troncoso. 2022. Online Website Fingerprinting: Evaluating Website Fingerprinting Attacks on Tor in the Real World. In *USENIX Security*.

[15] George Danezis. 2004. The Traffic Analysis of Continuous-Time Mixes. *PETS* (2004).

[16] Debajyoti Das, Sebastian Meiser, Esfandiar Mohammadi, and Aniket Kate. 2018. Anonymity Trilemma: Strong Anonymity, Low Bandwidth Overhead, Low Latency - Choose Two. In *IEEE SP*. 108–126. https://doi.org/10.1109/SP.2018.00011

[17] Debajyoti Das, Sebastian Meiser, Esfandiar Mohammadi, and Aniket Kate. 2020. Comprehensive Anonymity Trilemma: User Coordination is not enough. *Proc. Priv. Enhancing Technol.* 2020, 3 (2020), 356–383. https://doi.org/10.2478/popets-2020-0056

[18] Roger Dingledine, Nick Mathewson, and Paul F. Syverson. 2004. Tor: The Second-Generation Onion Router. In *USENIX Security*.

[19] Jason A. Donenfeld. 2017. WireGuard: Next Generation Kernel Network Tunnel. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society.

[20] Kevin P. Dyer, Scott E. Coull, and Thomas Shrimpton. 2015. Marionette: A Programmable Network Traffic Obfuscation System. In *USENIX Security*.

[21] eBPF Community. accessed 2023-07-16. eBPF: Dynamically program the kernel for efficient networking, observability, tracing, and security. https://ebpf.io/.

[22] Jiajun Gong and Tao Wang. 2020. Zero-delay Lightweight Defenses against Website Fingerprinting. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 717–734. https://www.usenix.org/conference/usenixsecurity20/presentation/gong

[23] Jiajun Gong, Wuqi Zhang, Charles Zhang, and Tao Wang. 2021. WFDefProxy: Modularly Implementing and Empirically Evaluating Website Fingerprinting Defenses. *CoRR arXiv* (2021). https://arxiv.org/abs/2111.12629.

[24] Jiajun Gong, Wuqi Zhang, Charles Zhang, and Tao Wang. 2022. Surakav: Generating Realistic Traces for a Strong Website Fingerprinting Defense. In *IEEE S&P*.

[25] WebAssembly Community Group. 2023. WebAssembly System Interface. https://github.com/WebAssembly/WASI accessed 2023-05-08.

[26] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and J. F. Bastien. 2017. Bringing the web up to speed with WebAssembly. In *PLDI*.

[27] Jamie Hayes and George Danezis. 2016. k-fingerprinting: A Robust Scalable Website Fingerprinting Technique. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, Thorsten Holz and Stefan Savage (Eds.). USENIX Association, 1187–1203. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/hayes

[28] Sébastien Henri, Gines Garcia-Aviles, Pablo Serrano, Albert Banchs, and Patrick Thiran. 2020. Protecting against Website Fingerprinting with Multihoming. *PETS* (2020). https://doi.org/10.2478/popets-2020-0019

[29] Dominik Herrmann, Rolf Wendolsky, and Hannes Federrath. 2009. Website fingerprinting: attacking popular privacy enhancing technologies with the multinomial naïve-bayes classifier. In *CCSW*.

[30] Andrew Hintz. 2002. Fingerprinting Websites Using Traffic Analysis. In *PET*.

[31] Paul E. Hoffman and Patrick McManus. 2018. DNS Queries over HTTPS (DoH). RFC 8484. https://doi.org/10.17487/RFC8484

[32] James K. Holland, Jason Carpenter, Se Eun Oh, and Nicholas Hopper. 2023. DeTorrent: An Adversarial Padding-only Traffic Analysis Defense. *CoRR arXiv* (2023). https://arxiv.org/abs/2012.06609.

[33] James K Holland and Nicholas Hopper. 2022. RegulaTor: A Straightforward Website Fingerprinting Defense. *PETS* (2022).

[34] Christian Huitema, Sara Dickinson, and Allison Mankin. 2022. DNS over Dedicated QUIC Connections. RFC 9250. https://doi.org/10.17487/RFC9250

[35] Jana Iyengar and Martin Thomson. 2021. QUIC: A UDP-Based Multiplexed and Secure Transport. RFC 9000. https://doi.org/10.17487/RFC9000

[36] Rob Jansen and Nicholas Hopper. 2012. Shadow: Running Tor in a Box for Accurate and Efficient Experimentation. In *NDSS*.

[37] Rob Jansen and Ryan Wails. 2023. Data-Explainable Website Fingerprinting with Network Simulation. In *PETS*.

[38] Jinghao Jia, Raj Sahu, Adam Oswald, Dan Williams, Michael V. Le, and Tianyin Xu. 2023. Kernel extension verification is untenable. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems, HOTOS 2023, Providence, RI, USA, June 22-24, 2023*.

[39] Marc Juárez, Sadia Afroz, Gunes Acar, Claudia Díaz, and Rachel Greenstadt. 2014. A Critical Evaluation of Website Fingerprinting Attacks. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, Gail-Joon Ahn, Moti Yung, and Ninghui Li (Eds.). ACM, 263–274. https://doi.org/10.1145/2660267.2660368

[40] Marc Juárez, Mohsen Imani, Mike Perry, Claudia Díaz, and Matthew Wright. 2016. Toward an Efficient Website Fingerprinting Defense. In *ESORICS*.

[41] Dogan Kesdogan, Dakshi Agrawal, and Stefan Penz. 2002. Limits of Anonymity in Open Environments. In *Information Hiding*.

[42] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. 2020. Spectre attacks: Exploiting speculative execution. *Commun. ACM* 63, 7 (2020), 93–101.

[43] Wladimir De la Cadena, Asya Mitseva, Jens Hiller, Jan Pennekamp, Sebastian Reuter, Julian Filter, Thomas Engel, Klaus Wehrle, and Andriy Panchenko. 2020. TrafficSliver: Fighting Website Fingerprinting Attacks with Traffic Splitting. In *CCS*.

[44] Marc Liberatore and Brian Neil Levine. 2006. Inferring the source of encrypted HTTP connections. In *CCS*.

[45] Nate Mathews, James K Holland, Se Eun Oh, Mohammad Saidur Rahman, Nicholas Hopper, and Matthew Wright. 2022. SoK: A Critical Evaluation of Efficient Website Fingerprinting Defenses. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 344–361.

[46] Steven J. Murdoch and George Danezis. 2005. Low-Cost Traffic Analysis of Tor. In *IEEE S&P*.

[47] Milad Nasr, Alireza Bahramali, and Amir Houmansadr. 2018. DeepCorr: Strong Flow Correlation Attacks on Tor Using Deep Learning. In *CCS*. 1962–1976. https://doi.org/10.1145/3243734.3243824

[48] Milad Nasr, Alireza Bahramali, and Amir Houmansadr. 2021. Defeating DNN-Based Traffic Analysis Systems in Real-Time With Blind Adversarial Perturbations. In *USENIX Security*.

[49] Big Switch Networks. 2023. uBPF. https://github.com/iovisor/ubpf accessed 2023-07-16.

[50] Rishab Nithyanand, Xiang Cai, and Rob Johnson. 2014. Glove: A Bespoke Website Fingerprinting Defense. In *Proceedings of the 13th Workshop on Privacy in the Electronic Society, WPES 2014, Scottsdale, AZ, USA, November 3, 2014*, Gail-Joon Ahn and Anupam Datta (Eds.). ACM, 131–134. https://doi.org/10.1145/2665943.2665950

[51] Se Eun Oh, Taiji Yang, Nate Mathews, James K. Holland, Mohammad Saidur Rahman, Nicholas Hopper, and Matthew Wright. 2022. DeepCoFFEA: Improved Flow Correlation Attacks on Tor via Metric Learning and Amplification. In *IEEE S&P*.

[52] Andriy Panchenko, Fabian Lanze, Jan Pennekamp, Thomas Engel, Andreas Zinnen, Martin Henze, and Klaus Wehrle. 2016. Website Fingerprinting at Internet Scale. In *NDSS*.

[53] Tommy Pauly, David Schinazi, Alex Chernyakhovsky, Mirja Kühlewind, and Magnus Westerlund. 2023. *Proxying IP in HTTP.* Internet-Draft draft-ietf-masque-connect-ip-13. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-ietf-masque-connect-ip/13/ Work in Progress.

[54] Mike Perry. 2013. A Critique of Website Traffic Fingerprinting Attacks. https://blog.torproject.org/critique-website-traffic-fingerprinting-attacks.

[55] Mike Perry and George Kadianakis. accessed 2023-02-07. Circuit Padding Developer Documentation. https://gitweb.torproject.org/tor.git/tree/doc/HACKING/

CircuitPaddingDevelopment.md.
[56] Mike Perry and George Kadianakis. accessed 2023-02-07. Tor Padding Specification. https://gitweb.torproject.org/torspec.git/tree/padding-spec.txt.
[57] Tor Project. accessed 2023-02-18. Pluggable Transport Specification (Version 1). https://gitweb.torproject.org/torspec.git/tree/pt-spec.txt.
[58] Tobias Pulls. 2020. Towards Effective and Efficient Padding Machines for Tor. *CoRR arXiv* (2020). https://arxiv.org/abs/2011.13471.
[59] Tobias Pulls and Ethan Witwer. 2023. Maybenot: A Framework for Traffic Analysis Defenses. arXiv:2304.09510 [cs.CR]
[60] Mohammad Saidur Rahman, Mohsen Imani, Nate Mathews, and Matthew Wright. 2021. Mockingbird: Defending Against Deep-Learning-Based Website Fingerprinting Attacks With Adversarial Traces. *IEEE Transactions on Information Forensics and Security* (2021).
[61] Mohammad Saidur Rahman, Payap Sirinam, Nate Mathews, Kantha Girish Gangadhara, and Matthew Wright. 2020. Tik-Tok: The Utility of Packet Timing in Website Fingerprinting Attacks. *Proc. Priv. Enhancing Technol.* 2020, 3 (2020), 5–24. https://doi.org/10.2478/popets-2020-0043
[62] Reethika Ramesh, Leonid Evdokimov, Diwen Xue, and Roya Ensafi. 2022. VPNInspector: Systematic Investigation of the VPN Ecosystem. In *NDSS*.
[63] Eric Rescorla. 2018. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446. https://doi.org/10.17487/RFC8446
[64] Vera Rimmer, Davy Preuveneers, Marc Juárez, Tom van Goethem, and Wouter Joosen. 2018. Automated Website Fingerprinting through Deep Learning. In *NDSS*. http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_03A-1_Rimmer_paper.pdf
[65] Vera Rimmer, Theodor Schnitzler, Tom van Goethem, Abel Rodríguez Romero, Wouter Joosen, and Katharina Kohls. 2022. Trace Oddity: Methodologies for Data-Driven Traffic Analysis on Tor. *PETS* (2022).
[66] Florentin Rochet and Tariq Elahi. 2022. Towards Flexible Anonymous Networks. https://arxiv.org/abs/2203.03764.
[67] Vitaly Shmatikov and Ming-Hsiu Wang. 2006. Timing Analysis in Low-Latency Mix Networks: Attacks and Defenses. In *ESORICS*.
[68] Sandra Siby, Ludovic Barman, Christopher Wood, Marwan Fayed, Nick Sullivan, and Carmela Troncoso. 2023. Evaluating practical QUIC website fingerprinting defenses for the masses. *PETS* (2023).
[69] Payap Sirinam, Mohsen Imani, Marc Juárez, and Matthew Wright. 2018. Deep Fingerprinting: Undermining Website Fingerprinting Defenses with Deep Learning. In *CCS*.
[70] Jean-Pierre Smith, Luca Dolfi, Prateek Mittal, and Adrian Perrig. 2022. QCSD: A QUIC Client-Side Website-Fingerprinting Defence Framework. In *USENIX Security*.
[71] Qixiang Sun, Daniel R. Simon, Yi-Min Wang, Wilf Russell, Venkata N. Padmanabhan, and Lili Qiu. 2002. Statistical Identification of Encrypted Web Browsing Traffic. In *IEEE S&P*.
[72] Ryan Wails, Rob Jansen, Aaron Johnson, and Micah Sherr. 2023. Proteus: Programmable Protocols for Censorship Circumvention. (2023).
[73] Mona Wang, Anunay Kulshrestha, Liang Wang, and Prateek Mittal. 2022. Leveraging strategic connection migration-powered traffic splitting for privacy. *PETS* (2022).
[74] Tao Wang and Ian Goldberg. 2013. Improved website fingerprinting on Tor. In *WPES*.
[75] Tao Wang and Ian Goldberg. 2016. On Realistically Attacking Tor with Website Fingerprinting. *PETS* (2016).
[76] Tao Wang and Ian Goldberg. 2017. Walkie-Talkie: An Efficient Defense Against Passive Website Fingerprinting Attacks. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, Engin Kirda and Thomas Ristenpart (Eds.). USENIX Association, 1375–1390. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/wang-tao
[77] Xinyuan Wang, Douglas S. Reeves, and Shyhtsun Felix Wu. 2002. Inter-Packet Delay Based Correlation for Tracing Encrypted Connections through Stepping Stones. In *ESORICS*.
[78] Ethan Witwer. 2023. State Machine Frameworks for Website Fingerprinting Defenses: Maybe Not. arXiv:2310.10789 [cs.CR]
[79] Ethan Witwer, James K. Holland, and Nicholas Hopper. 2022. Padding-only Defenses Add Delay in Tor. In *WPES*.

A PIPELINED FRONT

Pipelining is a technique to unify multiple machines: rather than probabilistically selecting from a pool of machines upon connection establishment, a single machine incorporates multiple *pipelines*. From the initial state, any action has a determined probability of causing a transition to each of the configured pipelines, which operate independently. Because the framework distinguishes between machine definition and runtime, large machines can efficiently be shared between instances of the framework. One downside with pipelining is that different machine-specific padding and blocking limits of unified machines cannot be expressed.

Figure 17 depicts a Pipelined FRONT machine, whose design is described in Section 6. The *NonPaddingSent* and *NonPaddingRecv* events both have an equal probability of resulting in transition to a number of `PADDING` states, each of which leads to a different sequence of states that will be followed for the remainder of the download. This allows for parameter variation (in this case, of the padding budget) which cannot be encoded within individual states to still be represented with one machine.

We emphasize that pipelining is not an inherent feature of the framework; it is merely a possible use case that illustrates some of Maybenot's capabilities. For certain defenses, other representations to account for dynamic parameter selection may be more efficient. Specifically, it may be useful to have convergence/divergence behavior with overlapping states between pipelines, among other possible behaviors. We encourage defenders to consider all of Maybenot's features when designing hand-crafted machines.

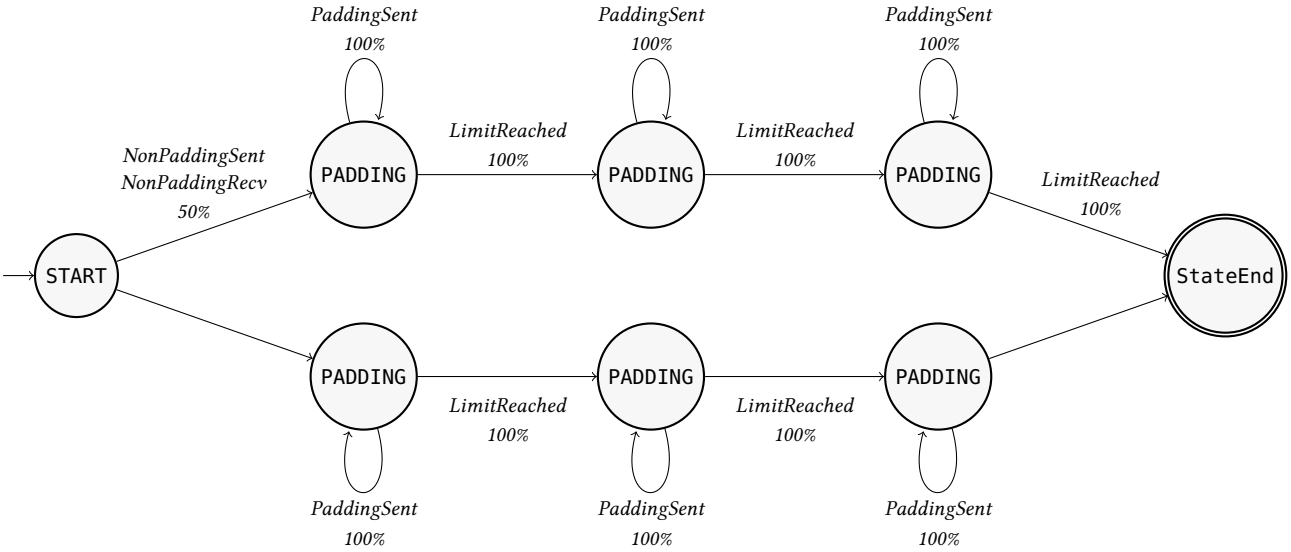


Figure 17: Pipelined FRONT machine with two pipelines, three `PADDING` states each.