Ephemeral Network-Layer Fingerprinting Defenses

Tobias Pulls
Karlstad University
tobias.pulls@kau.se

Topi Korhonen
Independent
topiko1987@gmail.com

Ethan Witwer
Linköping University
ethan.witwer@liu.se

Niklas Carlsson
Linköping University
niklas.carlsson@liu.se

Abstract

Fingerprinting attacks on encrypted network traffic may reveal sensitive information about users of anonymous communication systems, such as visited websites or watched videos, linking users' activities to their identities. Defenses come at the cost of bandwidth and delay overheads, impacting the user experience and making wide-scale deployment challenging. There is a rich history of attacks and defenses, with continual improvements in deep learning as a catalyst, making deployment of defenses an ever more pressing matter. This paper introduces a new defense strategy against fingerprinting attacks—ephemeral defenses—where efficient defense search enables the generation of unique per-connection defenses. We demonstrate that ephemeral defenses are multipurpose network-layer defenses against circuit, website, and video fingerprinting attacks, achieving competitive performance compared to related work. Furthermore, we create tunable ephemeral defenses that are not overly specialized to a particular fingerprinting attack, dataset, or network conditions. Ephemeral defenses are practical, demonstrated through integration with WireGuard and deployment at Mullvad VPN for a year, serving thousands of daily users.

Keywords

anonymous communication, fingerprinting, network simulation

1 Introduction

The exact program logic of effective and efficient network traffic fingerprinting defenses is complex. This should come as no surprise, as the community has been working on defenses against fingerprinting attacks for decades [7, 11, 27, 28, 38, 40, 68, 74]. To make matters worse, advances in deep learning over the last decade have significantly enhanced fingerprinting attacks performed by relatively weak local, passive eavesdroppers, who observe patterns in encrypted traffic [6, 42, 49, 62, 66, 69, 70, 72]. The real-world threat of attacks remains debated [3, 12, 33, 35, 51, 52, 82], but better attacks have nonetheless made it more challenging to find effective defenses with minimal bandwidth and latency overheads [13, 14].

This paper builds from the insight that the probability of *random program logic* serving as a decent fingerprinting defense increases substantially if the program can be expressed within a framework dedicated to traffic analysis defenses [18, 22, 55, 59, 73, 80]. When constrained in expressivity—as opposed to general-purpose frameworks like WFDefProxy [22] (which runs arbitrary Go programs)—such a framework acts as a *domain-specific language*, defining a *search space* of possible defenses that can be expressed within it. For this work, we picked the Maybenot framework [59] with roots in the practical Tor Circuit Padding Framework [55] and traffic analysis literature [36, 71], in which defenses are expressed as probabilistic finite state machines. This structure allows us to efficiently generate and deploy randomized defenses from this search space.

Due to the use of a restricted framework, the search space is densely populated with potential defenses. However, instead of exhaustively searching this space for *the* best defense, we flip the script and search for *many* defenses that fulfill some basic overhead constraints in a simulated environment. The many defenses are then assessed on how well they defend against attacks in aggregate, with each session (or trace) randomly selecting a defense. This search process is sufficiently efficient to enable what we introduce as *ephemeral defenses*: defenses used only once per trace (e.g., a Tor circuit), similar to how ephemeral keys are used in TLS [65]. This novel defense property prevents attackers from training on the exact defenses used, enhancing resilience against adaptive attacks.

The overarching contribution of this paper is the introduction and demonstration of ***multipurpose, network-layer ephemeral defenses***—defenses that are ***not tightly tuned to any particular fingerprinting attack, dataset, or network conditions***. These defenses are effective against *circuit, website, and video fingerprinting attacks*, which fingerprint the first tens (circuit) to thousands (website) to hundreds of thousands (video) of network packets, demonstrating their multipurpose effectiveness. Further, the defenses are inherently *tunable* due to being tailored for Maybenot [59], supporting *trade-offs* between bandwidth/delay and defensive effectiveness against state-of-the-art attacks [25, 42, 70, 72]. Finally, we show that the defenses remain effective under varied network conditions by extending the Maybenot simulator with a simulated network bottleneck, and across datasets within the same and different fingerprinting domains [21, 25, 43, 76]. While our evaluations in the paper are based solely on simulation for the sake of comprehensive and fair comparisons, ephemeral defenses are practical: they have been integrated with a fork of WireGuard [17, 46] and deployed in collaboration with Mullvad VPN as an opt-in feature in their VPN client (on Android, iOS, macOS, Linux, and Windows) for a year [44], serving a growing number of thousands of daily users. This paper is the first publication describing the core technology of that opt-in feature, which Mullvad VPN refers to as "DAITA: Defense Against AI-guided Traffic Analysis" [45].

This work is licensed under the Creative Commons Attribution 4.0 International License. To view a copy of this license visit https://creativecommons.org/licenses/by/4.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

Proceedings on Privacy Enhancing Technologies YYYY(X), 1–24

© YYYY Copyright held by the owner/author(s).

https://doi.org/XXXXXXX.XXXXXXX

In support of the broader contribution of ephemeral network-layer fingerprinting defenses, the paper presents the following specific technical contributions:

- A search method for ***ephemeral fingerprinting defenses*** based on deriving random Maybenot machines that are checked against constraints in simulated environments that can be combined with polynomial growth and deployed with tunable overhead-defense trade-offs (Section 3).
- A semi-automated tuning process, showing that ephemeral defenses can provide ***tunable defense against Circuit Fingerprinting attacks in Tor*** significantly improving over existing deployed defenses (Section 4).
- A comprehensive set of Website Fingerprinting (WF) experiments in Tor that demonstrate that ephemeral defenses can be ***tuned along the Pareto front of practical padding-only and blocking WF defenses***, provide insights into how the increasingly rich feature representations and architectures of state-of-the-art WF attacks push the boundaries of closed-world evaluations, and highlight the fragility of defense and attack tuning on results and consequently real-world practicality of WF defenses (Section 5).
- Demonstrate that ephemeral blocking defenses for WF provide ***a practical trade-off between attack accuracy and overhead for Video Fingerprinting*** served over HTTPS without specific tuning of the defenses (Section 6).

Section 7 reflects on our results, including real-world deployment of ephemeral defenses together with Mullvad VPN [44, 45]. Section 8 covers key related work, and Section 9 concludes. We begin with background in Section 2.

2 Background

Traffic analysis of encrypted network tunnels can have the goal of identifying—fingerprinting—the type of circuit used in Tor (Circuit Fingerprinting, Section 2.1), the website being visited (Website Fingerprinting, Section 2.2), and the video being watched (Video Fingerprinting, Section 2.3). In those sections, we describe both state-of-the-art attacks and defenses, noting that common defenses typically fall into two categories: *padding-only* defenses, which inject dummy traffic to obscure patterns, and *blocking* defenses, which also delay traffic to control timing leakage. Finally, we introduce Maybenot, a defense framework that expresses both types of defenses as probabilistic state machines (Section 2.4).

Following standard practices of most prior fingerprinting works, throughout the paper, regardless of the fingerprinting attack considered, we assume a local, passive network adversary that can observe packet timings and sizes but cannot modify, drop, or delay traffic [38, 43, 64]. Furthermore, we focus on a *closed-world* setting, where there is a fixed number of classes (e.g., representing websites) on which attacks are trained and tested. We use average accuracy to measure the effectiveness of attacks in closed-world settings throughout, as our goal is to *compare the relative strengths and weaknesses of defenses* by controlling to the greatest extent possible all other variables (including implementations in Maybenot and fine-tuning of related work). This is in contrast to establishing the real-world efficacy of attacks (and thus need for and tuning of defenses) in the more realistic *open-world* setting [3, 12, 33, 35, 51, 52, 58, 82].

2.1 Circuit Fingerprinting

In Tor, circuits are created and used for different purposes. There are four widely used purposes, with a fifth relatively recently added: general, introductory, rendezvous, HSDir, and Conflux [78]. Conflux circuits split traffic across multiple paths in the Tor network. General and rendezvous circuits may carry significant application-layer data. In contrast, introductory and HSDir circuits typically transport less data when connecting to an onion address (a trivial distinguisher). There are more general circuits than rendezvous circuits in Tor [41], subject to some base rate.

2.1.1 Attacks. The purpose of a circuit can be reliably fingerprinted [38], despite deployed defenses [23, 37, 53, 76]. Syverson et al. [76] recently used Deep Fingerprinting [72] for this purpose.

2.1.2 Defenses. As part of the Tor Circuit Padding Framework [54, 55], two padding machine defenses are deployed in Tor: One attempts to make introductory circuits appear as HSDir circuits and the other to make rendezvous circuits look like general circuits. Both remain fingerprintable today [37, 53, 76]. Kadianakis et al. began laying the groundwork for improved defenses [37], but to our knowledge, none have yet been implemented or deployed.

2.2 Website Fingerprinting

Traffic analysis of encrypted tunnels (that hide destination IP-addresses, otherwise trivial [4]) to fingerprint visited websites is referred to as Website Fingerprinting (WF) [11, 27, 28, 40, 74].

2.2.1 Attacks. Attacks can be grouped by relying on manual feature engineering—such as k-fingerprinting [26] and CUMUL [51]—or automatic feature engineering using deep learning [1, 66, 72]. We consider three deep-learning-based attacks: Deep Fingerprinting (DF) [72], Robust Fingerprinting (RF) [70], and Laserbeak [42].

DF was the first deep-learning-based attack to significantly impact defense design, achieving high (90%+) accuracy against the WTF-PAD [36] defense, which protects against k-fingerprinting and CUMUL. DF uses only the direction of cells from a network trace as its feature representation, ignoring time.

While earlier post-DF attacks, such as Tik-Tok [62], successfully incorporated time into DF's feature representation (with directional time), the next generational leap came from RF, which improved attack robustness in the presence of defenses thanks to its Traffic Aggregation Matrix (TAM) representation. With TAM, there are two channels of features that count sent and received packets, respectively, in bins with a resolution of 10–60 ms. This two-channel robust feature representation negates many defenses [70].

Building on the success of multi-channel feature representations, Laserbeak [42] introduces six channels ("multi") with complementing feature representations, the use of attention with transformers, and various training and architectural improvements. Mathews et al. provide three versions: DF-multi, Laserbeak without attention, and Laserbeak. DF-multi is DF with enhanced feature representations and improved training. Laserbeak without attention removes the attention-based transformer blocks from the architecture, significantly improving execution time. We include all three versions.

2.2.2 Defenses. WF defenses employ a combination of padding traffic (bandwidth) and blocking (delay) to transform traffic toward some goal. Mathews et al. [43] group WF defenses into five groups: adversarial perturbation defenses, collision defenses, fixed-rate defenses, splitting defenses, and randomized defenses.

There are three groups of defenses that we do not consider in this paper but explain for the sake of completeness. Adversarial perturbation defenses aim to trick ML-based models used by adversaries. Typically, these defenses consider weak non-adaptive adversaries [43] and are limited to particular classes of attacks. Traffic splitting defenses assume that clients have one or more unobservable paths, therefore only defending against weak adversaries [5]. Collision defenses require a database of reference traces or similar to coordinate collisions between groups of websites [43].

This leaves fixed rate and randomized defenses. As the name suggests, fixed-rate defenses send traffic at some (potentially adaptive) fixed rate in both directions. Padding is sent if no real traffic is available, and real traffic is blocked until the defense dictates that traffic should be sent. Therefore, such defenses suffer from high overheads, particularly regarding delay [43]. Randomized defenses aim to obfuscate traces by randomizing traces. Such defenses—that only use padding—become less effective if the adversary can train on significant numbers of defended traces [43, 57].

2.3 Video Fingerprinting

Video Fingerprinting (VF) analyzes encrypted traffic between a client and a video server to identify the video being streamed. Modern video streaming typically uses the DASH standard [15], in which videos are encoded at multiple quality levels and split into fixed-length segments (a few seconds each), served over HTTP(S). Streaming begins with a request for a Media Presentation Description (MPD) file, which lists available qualities and URLs for each segment. The client then fetches segments individually, initially in quick succession, followed by steady-state streaming, where a segment is requested roughly once per segment duration. Finally, clients use an Adaptive Bitrate (ABR) algorithm to switch quality levels mid-stream based on network conditions.

2.3.1 Attacks. Due to the periodic nature of video traffic, it is possible to achieve remarkable success against undefended traffic by simply comparing a video's segment sizes, which are easily obtainable through the manifest, to the observed sequence of segment sizes. Thus, many attacks are based on heuristics and basic machine learning algorithms, such as Leaky Streams [64] and Walls Have Ears [24]. However, more recent attacks such as Beauty and the Burst [69] have begun to employ deep learning, with similar input formats and model structures to WF attacks. WF attacks are in fact effective out of the box [25], and the state-of-the-art VF attacks, Video-Adapted Deep Fingerprinting (vDF) and Video-Adapted Robust Fingerprinting (vRF) [10], are adaptations of DF [72] and RF [70] with time series of byte counts as input instead of cells/packets.

2.3.2 Defenses. Compared to WF, very little work has been done on defenses against VF attacks. Existing approaches include adversarial samples, differential privacy [84], and trace morphing based on GAN-generated traffic traces [79]. Unfortunately, adversarial samples are ineffective, and both differential privacy and trace morphing have limitations, including significant overhead trade-offs, unknown effects on user experience, and deployment challenges.

A recent study [25] tested adaptations of two WF defenses, FRONT [20] (randomized padding) and RegulaTor [30] (strict traffic patterns) on video traffic, finding that the former is entirely ineffective and the latter degrades user experience significantly. In response, the authors proposed Scrambler, a defense that adds random padding to segment downloads, achieving great success for all but a small subset of the tested videos. Due to Scrambler's efficacy, low impact on user experience when bandwidth is high, and implementation in Maybenot [59], we use it as a benchmark.

2.4 Maybenot

Maybenot is a framework for traffic analysis defenses implemented in Rust [59–61], based on the Tor Circuit Padding Framework [54, 55], WTF-PAD [36], and the notion of Adaptive Padding [71]. The framework is designed to be integrated with a transport protocol, such as TLS [65], Tor [16], or WireGuard [17], running on both the client-side and at a (potentially intermediate, e.g., a relay in Tor) server. It acts as a runtime for zero or more probabilistic state machines. The integrator continuously reports events that describe network activities to an instance of the framework. The framework, in turn, runs the state machines in its instance and returns actions from the machines. The possible actions are to schedule padding (a dummy packet), block outgoing packets for a duration, or update an internal timer to support more complex state machines.

An instance of Maybenot involves zero or more machines as well as *limits* imposed on all machines. Limits enable or prevent machines from scheduling actions. There are four kinds of limits, applied in the following order: (1) *Per-state limits* restrict how many actions may be scheduled on self-transitions to the same state in a machine. (2) *Per-machine absolute limits* enable a machine to create a total number of padding packets and duration of active blocking, bypassing other limits below. (3) *Per-machine fractional limits* restrict the fraction of packets that can be padding and the fraction of time that outgoing traffic can be actively blocked. (4) *Framework-wide fractional limits* are enforced across all machines, limiting global padding fraction and blocking duration.

2.4.1 Maybenot machines. A machine consists of the two above-mentioned per-machine limits and one or more *states*. A state consists of a *counter*, an *action*, and *transitions*.

The counter can be used to implement more expressive machines, as counters are incremented, decremented, set, and reach zero, triggering events that may lead to transitions. Actions are triggered (scheduled), if limits allow, upon transition to the state. Possible actions include scheduling a padding packet to be sent, scheduling the blocking of outgoing traffic for a specific duration, and updating timers. The timer, like the counter, is designed to support more expressive machines. Actions to pad or block, as well as the duration to block for, are sampled from one of 11 parameterized *distributions*, such as the uniform and Poisson distributions.

The transitions are a matrix specifying the probability of transitioning to every state in the machine for every possible *event* in the framework. There are 13 events in Maybenot, including events for packets sent/received, distinguishing between with and without padding, as well as events related to blocking, limits, and timers.

2.4.2 The Maybenot simulator. Enables rapid development of machines. Takes as input a (base) network trace, Maybenot machines run at both the client and the server, and a specified delay between the client and the server. The simulator then outputs a simulated defended network trace. We forked and enhanced the network model of the simulator to support, in addition to a delay between the client and server, a simulated network bottleneck (symmetric) in terms of packets-per-second (PPS) rate. Blocking actions—and now sent traffic exceeding the bottleneck rate—leads to *aggregate delays* of subsequent network traffic. The original Maybenot simulator and our changes conservatively overestimate the aggregate delays (not to underestimate defense overheads), similar to the simulation of Tamaraw [9] and observations in related work [32, 83]. Appendix A provides further details on our improvements.

3 Ephemeral Defenses

The intuition behind ephemeral defenses is summarized in four parts marked in Figure 1. First, we generate candidate defenses as random Maybenot machines derived from seeds (Section 3.1). Ephemeral defenses are instantiated deterministically from these seeds, enabling reproducible yet diverse defense instances across traces and simulations. Second, we repeatedly search (shaded area) for defenses that fulfill constraints in simulated environments (Section 3.2). Third, to achieve the desired number of defenses, defenses can be optionally combined with polynomial growth under constraints (Section 3.3). Finally, defenses are deployed with tunable overhead-defense trade-offs (Section 3.4). We detail each step and end with an example (Section 3.5).

3.1 Random Machines

Random machines use Maybenot machines as a domain-specific language for traffic analysis defenses. In gist, it consists of randomized machine limits (see Section 2.4), one or more random states, feature flags for turning on or off Maybenot features, and *three reference points*. The feature flags control whether blocking actions are allowed (off for padding-only defenses) and whether "expressive" features, such as counters and timers, are permitted. This paper does not explore expressive features for random machines, using only a subset of Maybenot features to limit the size of the search space. Based on feature flags, each state will perform either a block or pad action with parameters influenced by the reference points.

The count reference point is the upper bound for uniformly random sampling of allowed padding packets and per-state limits. Its default value is 100. Per-state limits have a 50% probability of being set (as a probability distribution, see below). A budget of allowed padding packets is always sampled if the feature flag is set.

The duration reference point p determines the sampled durations for when to take actions (scheduled timeouts) and the duration of blocking actions. The duration reference point is also an upper bound on all sampled durations from distributions. The default value is 100 ms. The maximum value is either p (with 50% probability) or sampled uniformly random from $[0, p]$ independently for each distribution. Note that this value provides a maximum duration for blocking outgoing traffic, with significant implications for the effectiveness and efficiency of defenses. For the randomized parameters of distributions, p influences the ranges of their values to steer the expected value of the distribution towards the reference point (for applicable distributions); see Appendix B.

The timeout reference point determines the *minimum* timeout value on all blocking and padding action timeouts – defaults to zero. If set, this restricts the ability of machines to cause large bursts by repeatedly padding without any intermediate time.

The final piece of each random state is its *transitions*. With 50% probability per relevant event (e.g., ignoring timer/counter events for non-expressive machines), add transitions to a uniformly random subset of states with random probabilities adding up to 1.0. This process is repeated until the machine's state transitions form a *strongly connected graph* of all states (using Kosaraju's algorithm) with *liveness*, i.e., form a strongly connected graph based only on transitions on events for sending and receiving normal packets with a minimum transition probability of 5%. Unlike padding or blocking, which runtime limits may prevent, these events always occur, preventing machines from getting stuck in a state.

3.2 Defense Search

By repeatedly *deriving* defenses from seeds (Section 3.2.1), we search for defenses in simulated *environments* (Section 3.2.2) that fulfill some specified load and delay *constraints* (Section 3.2.3).

3.2.1 Deriving Defenses. We define a defense as two lists of Maybenot machines: One list for the client and one for the server. Maybenot specifies a custom serialization format for machines based on serde, compression, and base64 encoding [61]. For ephemeral defenses, we adopt a more safety-focused approach, inspired by ML-KEM [48], of using seeds. Given a configuration file for defense search, a seed deterministically generates a defense. We do this by using xoshiro256 [8], a deterministic PRNG. The execution time can be bound, leading to verifiable defense derivation in the order of milliseconds. For example, using hyperfine [56] to benchmark our CLI tool to derive a defense and print its serialized machines to stdout (including all parsing of the config file, base traces, etc.) takes 43.6 ± 4.7 ms on a commodity laptop for one of our ephemeral padding-only defenses used in Section 5.

3.2.2 Environments. The environment is used to simulate a defense on network traces using the Maybenot simulator. We define an environment as a set of network traces, a finite number of simulation steps, and a network model between client and server.

The network traces should represent the type of defense being searched for, e.g., WF traces could be from the BigEnough dataset [43]. The number of traces can be relatively few compared to typical datasets; around 10–30 traces suffice.

The simulation steps are primarily bound to deal with machines that get "stuck" in action loops (e.g., repeatedly scheduling new blocking on blocking beginning). At first glance, it might be compelling to express the simulation in terms of the number of packets at the client (as captured in typical datasets), but random machines quickly find a way to mess this up. Other reasons to control simulation steps are to search for defenses that fulfill their constraints with fewer actions and to search only for defenses that target the start of traces (e.g., when defending handshakes for CF in Section 4).

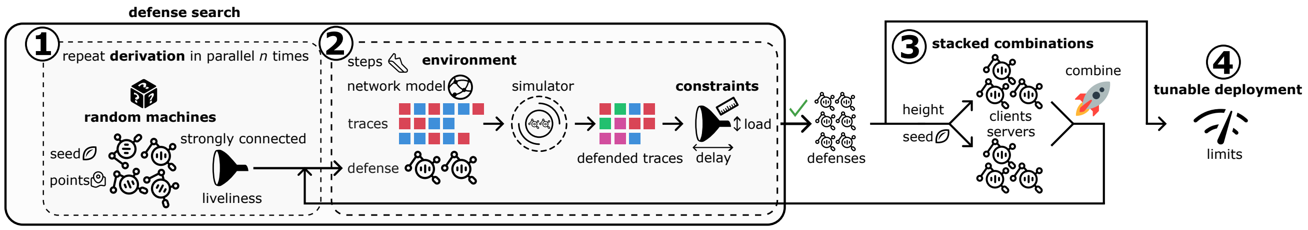


Figure 1: Overview of the defense search process. (1) Random machines with strongly connected states based on packet-level events (liveliness) are repeatedly derived (first dashed shaded area) from seeds and (2) evaluated as defenses in simulated environments (second dashed shaded area) in terms of fulfilling constraints wrt. bandwidth load and additional runtime (delay). (3) Found defenses can be used to create stacked combinations of new defenses (with polynomial growth) that are also subject to constraints. (4) Defenses can be tuned when deployed for different trade-offs in terms of overhead and protection.

The network model between client and server is part of the extended Maybenot simulator, as described in Section 2.4 and Appendix A. It consists of an RTT and a PPS rate. The RTT between the client and server affects the simulated network trace at the server and the transmission time of padding packets. The extended simulator also supports a PPS rate, specifying the maximum PPS of a simulated network bottleneck between the client and server. Packets over the PPS get additional simulated delay before being received. If normal packets get delayed, this causes aggregate delay throughout the simulated network trace.

3.2.3 Constraints. Constraints are expressed in terms of acceptable ranges of load (additional bandwidth) and delay (additional duration). Load is defined for both client and server (based on packets sent), while delay is the aggregate increase in duration, as both client and server machines contribute to the total duration. These constraints are computed as averages over all traces in the environment. For each simulated trace, there are also early constraint checks for a configurable minimum total number of normal packets and that at least a hardcoded 20% (arbitrary selected) of events from the simulator relate to packets (normal or padding). These early checks filter out many machines that prevent traffic by too aggressively blocking traffic, toggling blocking on and off, etc.

3.3 Stacked Combinations

The purpose of stacked combinations is to speed up the generation and potentially also the distribution of ephemeral defenses. With input of one or more defenses, create a list C of all client machines and a list S of all server machines in those defenses. Then, based on a provided maximum height $H \geq 1$, create new defenses by randomly selecting between $[1, H]$ random machines for the client and server, independently. No modifications to machines are necessary. Check for constraints in an environment just as in the defense search. If the machines are from defenses fulfilling identical constraints, we observe that combined defenses typically require significantly fewer attempts to fulfill constraints compared to random machines. For example, for the ephemeral padding-only defenses in Section 5, it took an average of 1.28 attempts (median 1) to create defenses fulfilling the constraints.

The total number of unique combinations is:

$$\left(\sum_{k=1}^{\min(C,H)} \binom{C}{k}\right) \times \left(\sum_{k=1}^{\min(S,H)} \binom{S}{k}\right) \tag{1}$$

Assuming a fixed H, the number of combinations grows polynomially $O(N^{2H})$ for $N = C = S$. For example, $H = 5$ with $N = 1,000$ gives 6.88×10^{25} unique defenses and $H = 6$ with $N = 10,000$ in total 1.93×10^{42} unique defenses. The polynomial growth is an advantage when deploying ephemeral defenses in settings where defenses may be distributed by a central party, saving compute.

3.4 Tunable Deployment

The Maybenot framework provides framework-wide limits, per-machine limits, and per-machine budgets for both padding packets and blocking duration (Section 2.4). These limits are also used during defense search—and we enforce that random machines have liveness (Section 3.1)—so most defenses are inherently tunable. Hitting limits only temporarily stops the defense. As soon as limits permit, actions will continue to be scheduled. As such, defenses are inherently tunable in deployment by adjusting limits, allowing for overhead-protection trade-offs.

3.5 Example: Ephemeral Padding-Only Defenses for Website Fingerprinting

Appendix C introduces the TOML configuration of our Rust CLI and provides a detailed walkthrough of the configuration for our ephemeral padding-only defenses for WF evaluated in Section 5. To summarize and highlight key insights, the configuration defines the number of defenses to search for, derivation settings for machine and environment parameters, and defense constraints. It primarily involves balancing search space exploration with computational limits. For example, environments are sampled and explored for a maximum number of attempts before being resampled, and constraints in terms of load and delay are either tightly or loosely defined in ranges of acceptable fractions. The entire search is deterministic from a seed; we use seed 0 for all ephemeral defenses as a nothing-up-my-sleeve number throughout the paper.

4 Parameter Tuning Ephemeral Defenses for Circuit Fingerprinting

To demonstrate a straightforward approach to parameter tuning ephemeral defenses, we search for CF defenses that are an essential part of protecting access to onion services in Tor.

4.1 False Positives for Onions

We sketch a defense to create false positives for a passive network adversary attempting to fingerprint onion-site traffic. The source of false positives would be general circuits misclassified as rendezvous circuits. In addition, such a defense would require that general circuits create dummy introduction and HSDir circuits. Because the base rate of general circuits is higher than onion circuits, achieving a low false positive rate in a balanced experiment (as below) would likely suffice. Therefore, the defense could be run with some probability on general circuits (always on rendezvous circuits). We emphasize that such tuning would be necessary, as the adoption of Conflux [2] circuits introduces additional complexities, and that the impact of the added latency of onion circuits (twice the circuit length) also requires careful investigation.

4.2 Parameter Tuning

Syverson et al. [76] collected a large dataset from the live Tor network to evaluate the fingerprintability of the Onion Location feature of Tor Browser [77]. From their dataset, we create a CF dataset of 10,000 samples each of general and rendezvous circuits. We truncate each sample to the first 30 cells and confirm that RF and DF still achieve perfect (99.9%) accuracy. To search for ephemeral defenses, we randomly picked 7 general and 7 rendezvous traces from the dataset for our environment.

In the context of ephemeral defenses, parameter tuning is updating the configuration file. All parameters are relevant, including those for tunable deployment, which ultimately modify the machines' budgets. The search space is practically infinite, especially when considering the environment traces. Here, we describe a semi-automated basic method for parameter tuning. Much can probably be improved in future work.

The tuning method operates in phases, with each phase executed iteratively. Each phase begins with a starting configuration and then iteratively randomly changes each parameter in the configuration with a small probability until it finds a new (previously unevaluated) one. Each new configuration searches for defenses up to a maximum number of defenses or until a maximum runtime has expired. The defenses are then simulated on a dataset and evaluated using DF, RF, and overhead calculations. This process is automated and deterministically performed from a seed. It repeats until stopped manually. Then, all results are evaluated manually, and a new starting configuration is created for the next phase.

The parameter tuning of CF began with a liberal configuration file (Appendix D), selecting 4–14 random traces from the dataset and allowing client and server loads between 0.5 and 10.0, as well as delays between 0.5 and 5.0. We searched for up to 1,000 defenses for up to 15 minutes. When simulating defenses, machine and framework budgets are scaled based on set values and tunable defense limits. In this case, we simulate scaled limits of 1.0, 0.75, and 0.5. This results in a trade-off line with three data points representing the trade-off between overhead and accuracy reduction for DF and RF. We never modify the simulation parameters as part of the tuning process to ensure ease of comparison between phases.

Parameter tuning was conducted over 11 phases, spanning six days, and evaluated a total of 356 configurations. In brief, after starting a phase, review the results in the evening before bed and initiate a new phase overnight. Then, in the morning, repeat the process. Figure 2 shows the trade-offs in accuracy and overheads[1] between the start and end configurations. When manually reviewing the results of each phase, we plotted the results as in Figure 2 and selected the configuration changes that showed the best improvement overall. This process could be automated in the future. We leave this as future work. Our research artifacts include the configuration file and evaluation results from each phase, as well as support for tuning using the `maybenot tune` CLI command (see Artifacts section at the end of the paper). Appendix D contains the complete git diff between the start and end configurations. In summary, the tuning resulted in three categories of changes:

Constraints favors server-side padding over client padding, but both with significantly increased minimums over the starting configuration and reduced required delay. Changes to computing limits only on actions between the first and last packet in the trace. For short traces such as handshakes, this makes total sense (no tail to consider).

Increased granularity by setting a duration reference point with a wide range of values and significantly increased simulation steps, allowing for defenses that take many small actions to pass through constraints.

Searched longer by greatly increasing (17×) the maximum attempts before sampling a new environment and narrowing the number of traces to 7–11. Likely, this leads to more diverse defenses.

Using the final configuration, we searched for 1,000 defenses and used those as a basis to evaluate combinations with heights 1–10. (We combine to get 20,000 defenses in total for unique defenses for the 20,000 traces in the dataset.) Height 1 showed slightly worse protection, while we found no notable differences between heights 2–10. Therefore, we picked height 2 and combined 20,000 defenses that fulfill the criteria. We also observed no gains in limits above 0.75. Finally, we selected eight limits (for spacing out trade-off lines) and evaluated them using ten-fold cross-validation, producing Figure 3.

Figure 3 shows the accuracy of DF and RF for different overhead costs. Note that this is a binary classification problem with balanced classes, so 50% accuracy serves as the baseline (i.e., guessing). In terms of overhead, it is the sum of delay and bandwidth. The delay is at most 85% increased duration. At overhead 10 and below, delay represents less than 3 seconds of blocking (average undefended duration 18.2 seconds); the rest is padding. Recall that all traces in the undefended dataset are capped at 30 cells. This means that a 10× overhead is 300 cells (150 KiB). At 90% accuracy—representing a non-negligible false positive rate—the defense increases the average handshake from 15 KiB to 80 KiB and delays it by an additional 0.9 seconds. Whether this is sufficient or even practical is arguable.

[1] We combine bandwidth and delay fractional overheads to a single value, leaving exploring inherent trade-offs [13, 14] to future work as discussed later in Section 4.2.

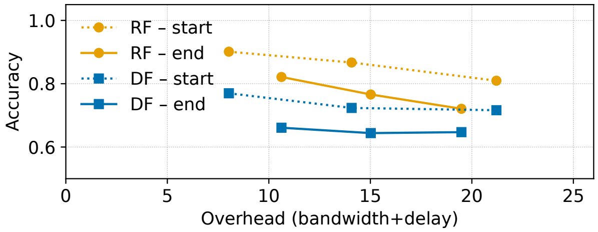


Figure 2: Trade-offs comparison between start and end configurations for the Circuit Fingerprinting tuning.

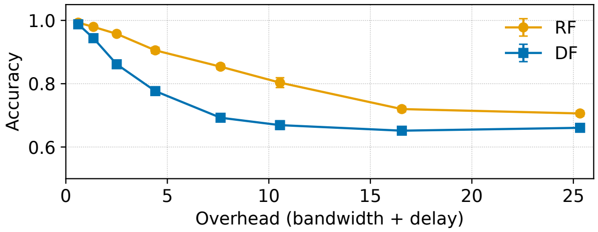


Figure 3: An ephemeral Circuit Fingerprinting defense with tunable protection-overhead (0.5 accuracy baseline).

On the one hand, circuits are usually established preemptively in Tor before being used, so the bandwidth and delay may not impact the user experience. On the other hand, bandwidth is a precious resource in the Tor network. It might be worthwhile to optimize defenses for using delay over padding. We leave this as future work.

Takeaway: Using a straightforward semi-automated tuning process, ephemeral defenses can provide tunable defense, e.g., against Circuit Fingerprinting attacks.

5 Website Fingerprinting

For WF, we implement and tune seven defenses in Maybenot for two network models (Section 5.1), evaluate the defenses using five state-of-the-art WF attacks (Section 5.2), improve the evaluation using infinite training (Section 5.3), investigate how well trained attacks generalize across defenses (Section 5.4), and how well parameter-tuned defenses (Section 5.5) generalize across datasets.

5.1 Defenses and Parameter Tuning

For the sake of accurate comparison, we implement selected related-work WF defenses in Maybenot. Some defenses are more or less suitable as state machines, limiting our selection. We stress that our implementations may have shortcomings, that related work was not necessarily designed to operate in the settings provided by our updated Maybenot simulator (e.g., where a network bottleneck may be present), and that there is ultimately no substitute for real-world deployment and experimentation.

As representative padding-only defenses we selected Break-Pad [31], FRONT [20], and Interspace [57]. FRONT, together with Interspace, were evaluated as on a Pareto front among practical padding-only defenses studied by Mathews et al. [43]. Break-Pad and Interspace target the Tor Circuit Fingerprinting framework [55] and are, therefore, directly portable to Maybenot. For FRONT, we use the implementation from Maybenot [59].

As representative fixed-rate defenses, we implemented versions of RegulaTor [30] and Tamaraw [9]. Tamaraw is implemented with a soft stop condition after a tunable window (in seconds) of no normal packets sent, as done by Gong et al. for real-world implementations of WF defenses [22]. The soft stop condition is motivated by the practical challenge of a network-layer defense implementation to determine the actual end of application-layer data. Otherwise, Tamaraw is straightforward to implement as a state machine. RegulaTor, on the other hand, is far from suitable as Maybenot state machines due to the need for per-packet timers, among other issues. We picked RegulaTor in part due to its excellent performance and in part to demonstrate the limitations of Maybenot. We spent considerable time developing and refining a practical implementation, but there is likely more that can be done.

We parameter-tuned all implementations for the BigEnough [43] standard dataset, with and without a simulated network bottleneck. Details are provided in Appendix E, including the final parameters and additional information on the implementations. Parameters were generally selected to minimize overheads (while maintaining some defense, in the case of FRONT) or to match overheads from related work. We weighed delay overhead twice as much as bandwidth overhead. For FRONT, Interspace, and ephemeral defenses, we created new instances of these defenses for each trace. The ephemeral defenses use stacked combinations of height 2. Unless otherwise stated, all results are from five-fold cross-validation, where we parameter-tuned on the first fold. Tuning and evaluation were done independently by different authors.

5.2 Evaluation and Network Bottleneck

We simulate defenses on the BigEnough (standard Tor Browser security level) dataset in a closed world setting [43] twice: Once with infinite simulated bandwidth and once with a network bottleneck based on each trace's maximum observed PPS in either direction (see Appendix A). We evaluate defenses in terms of overhead and accuracy. The overhead is in terms of additional bandwidth and delay. Accuracy comes from five attacks (see Section 2.2.1): Deep Fingerprinting (DF) [72], Robust Fingerprinting (RF) [70], Laserbeak [42], Laserbeak without attention (Laserbeak$^-$), and DF-multi. Attacks use their respective default hyperparameters (such as 30 epochs), except for their input lengths being increased to 10,000 to better capture the majority of defended traffic (as in related work [42]), unless otherwise stated. We present the results in Table 1, where ▽ indicates a simulated bottleneck. Results are split by padding-only and blocking defenses.

When it comes to the simulated network bottleneck, our results confirm that randomized padding-only defenses also add delay by causing congestion [83]. Note that we re-tuned the parameters for all defenses in their bottleneck versions. Break-Pad and FRONT, due to their bursty padding at the beginning of network traces, struggle in particular. The same is true for RegulaTor, which proved challenging to tune, as balancing the send rate caused delays due to being either too slow, thereby delaying traffic, or too high, which

Table 1: For seven implemented defenses [9, 20, 30, 31, 57] in the Maybenot framework [59], bandwidth and delay overheads, and attack accuracies from five attacks [42, 70, 72] on the BigEnough standard dataset in a closed world [43]. Results from five-fold cross-validation. Attacks use a 10,000 input length. The highest average accuracy per defense is highlighted in bold. ▽ indicates a simulated network bottleneck computed for each undefended network trace; otherwise, infinite bandwidth.

	BigEnough	Accuracy %					Overhead %	
		DF	DF-multi	Laserbeak	Laserbeak$^-$	RF	Bandwidth	Delay
	Undefended	$89.0^{\pm0.8}$	$93.5^{\pm0.4}$	$\mathbf{96.4^{\pm0.3}}$	$96.2^{\pm0.2}$	$90.1^{\pm0.8}$	$0.0^{\pm0.0}$	$0.0^{\pm0.0}$
Padding-only	Break-Pad▽	$65.6^{\pm0.8}$	$74.7^{\pm1.7}$	$88.1^{\pm0.8}$	$\mathbf{88.2^{\pm0.7}}$	$71.7^{\pm1.1}$	$75.3^{\pm0.4}$	$332.6^{\pm30.1}$
	Break-Pad	$66.6^{\pm0.8}$	$84.4^{\pm1.4}$	$89.8^{\pm0.7}$	$\mathbf{90.3^{\pm0.6}}$	$77.9^{\pm1.2}$	$75.3^{\pm0.4}$	$0.0^{\pm0.0}$
	Ephemeral Pad▽	$38.2^{\pm1.0}$	$62.0^{\pm2.3}$	$\mathbf{74.3^{\pm1.3}}$	$73.2^{\pm1.2}$	$44.1^{\pm0.9}$	$64.3^{\pm0.4}$	$43.9^{\pm7.1}$
	Ephemeral Pad	$39.7^{\pm0.7}$	$62.3^{\pm0.5}$	$\mathbf{75.2^{\pm0.9}}$	$74.2^{\pm0.9}$	$46.6^{\pm1.1}$	$58.7^{\pm0.5}$	$0.0^{\pm0.0}$
	FRONT▽	$79.7^{\pm1.3}$	$83.9^{\pm0.7}$	$\mathbf{90.8^{\pm0.8}}$	$90.7^{\pm0.5}$	$88.7^{\pm1.1}$	$18.2^{\pm0.3}$	$111.2^{\pm15.0}$
	FRONT	$59.2^{\pm0.9}$	$76.2^{\pm1.0}$	$87.7^{\pm1.1}$	$90.0^{\pm0.5}$	$\mathbf{90.8^{\pm0.9}}$	$72.7^{\pm1.3}$	$0.0^{\pm0.0}$
	Interspace▽	$55.5^{\pm2.1}$	$81.4^{\pm0.5}$	$\mathbf{87.3^{\pm1.1}}$	$87.0^{\pm1.0}$	$68.6^{\pm1.0}$	$56.3^{\pm0.6}$	$17.9^{\pm8.5}$
	Interspace	$55.8^{\pm1.2}$	$83.6^{\pm0.4}$	$\mathbf{88.3^{\pm0.7}}$	$88.2^{\pm0.5}$	$69.7^{\pm0.6}$	$56.3^{\pm0.6}$	$0.0^{\pm0.0}$
Blocking	Ephemeral Block▽	$14.0^{\pm0.6}$	$27.9^{\pm1.4}$	$33.2^{\pm1.6}$	$\mathbf{34.7^{\pm2.1}}$	$17.9^{\pm0.7}$	$78.8^{\pm0.9}$	$123.7^{\pm38.2}$
	Ephemeral Block	$10.2^{\pm0.2}$	$21.8^{\pm0.9}$	$25.2^{\pm1.9}$	$\mathbf{26.0^{\pm1.1}}$	$14.7^{\pm1.0}$	$97.5^{\pm1.1}$	$68.4^{\pm0.7}$
	RegulaTor▽	$40.5^{\pm0.8}$	$47.3^{\pm0.7}$	$51.7^{\pm0.9}$	$\mathbf{54.6^{\pm0.7}}$	$50.1^{\pm1.4}$	$38.6^{\pm0.2}$	$133.4^{\pm5.2}$
	RegulaTor	$34.3^{\pm1.1}$	$44.6^{\pm0.7}$	$47.4^{\pm1.5}$	$53.4^{\pm1.7}$	$\mathbf{66.4^{\pm0.6}}$	$89.7^{\pm0.7}$	$7.8^{\pm0.2}$
	Tamaraw▽	$29.0^{\pm0.2}$	$33.5^{\pm2.1}$	$34.9^{\pm0.9}$	$\mathbf{36.8^{\pm0.6}}$	$28.5^{\pm1.6}$	$127.9^{\pm2.8}$	$146.9^{\pm11.6}$
	Tamaraw	$25.2^{\pm0.4}$	$32.0^{\pm0.4}$	$30.7^{\pm0.7}$	$\mathbf{32.5^{\pm1.0}}$	$31.1^{\pm0.8}$	$129.3^{\pm2.8}$	$73.4^{\pm0.8}$

hit the simulated bottleneck (see Appendix E). Interspace induced the least additional delay due to the bottleneck, which is notable since it has no parameters to tune. We stress that our simulator model might be too conservative and that most defenses in the literature were not intentionally (or unintentionally) designed with a network bottleneck between client and server in mind. Real-world implementations and experiments of these defenses often take the form of Pluggable Transports (PTs) [22, 30] in Tor, where the network bottleneck in experiments is not between the PT endpoints but is highly likely to be inside the Tor network.

Takeaway: Defenses should consider network bottlenecks in their design and evaluation (simulated and real-world experiments) [3, 12, 32–34]; padding-only defenses cause delay [83].

There is a stark difference between padding-only and blocking defenses; attacks are highly effective against all padding-only defenses, with ephemeral defenses offering some protection. Laserbeak performs exceptionally well in terms of attacks, with slight differences compared to Laserbeak$^-$. This is in line with Mathews et al.'s [42] findings that the added value of attention is small at best. Blocking defenses offer significantly better protection against all attacks, albeit with higher delays. That defenses need to cause some overheads in terms of both bandwidth and delay to be effective is expected [13, 14]. We observe a high deviation in delay overhead for the ephemeral defenses in the bottleneck model, likely due to the combination of randomly selecting a defense per trace and varying actual bottlenecks for each trace. In contrast, the tuning process focuses on average overheads.

Takeaway: Ephemeral defenses are competitive padding-only and blocking defenses compared to state-of-the-art.

5.3 Data Augmentation and Infinite Training

Data augmentation and longer training times enhance attacks, especially against randomized padding-only defenses [42, 43, 57, 67]. Ephemeral defenses can create unique defenses for each simulated trace in datasets, similar to how randomized defenses like FRONT and Interspace sample parameters per instantiation. Because of this, we implement *infinite training* for all attacks. When a trace is pulled in the training loop for simulation, a new defense instance is assigned to it (or, in the case of static defenses, the same defense: This mirrors augmentation in earlier work where the same defense is simulated multiple times per trace). No epoch limit is set; the number of times traces are simulated with different (sampled) defenses is not limited. We let the training loop continue until no improvement in the validation set has been observed for 32 epochs (the termination patience). A minor modification in learning rate scheduling is necessary; we use the plateau learning rate scheduler [19] for all attacks. Specifically, when there has been no improvement in training set loss for eight epochs, the learning rate is reduced by a factor of 0.8. Otherwise, the default setting of each attack is preserved. Table 2 shows the result from infinite training for the same setting as Table 1. The minor differences in overheads between tables are due to measuring overheads in the final test set, just like attack accuracies. The training time for different attacks for Table 2 is shown in Appendix F. In total, it took 27 days of wall-clock time using a mix of 4070 Ti and L40S GPUs. Laserbeak and Laserbeak$^-$are only two-fold cross-validated for this reason.

Table 2: A repeat of Table 1, but with infinite training time for the attacks until their validation accuracy stops improving.

	BigEnough	Accuracy %					Overhead %	
		DF	DF-multi	Laserbeak	Laserbeak^{-}	RF	Bandwidth	Delay
	Undefended	$92.7^{\pm0.3}$	$95.8^{\pm0.7}$	$96.5^{\pm0.5}$	**$97.1^{\pm0.1}$**	$94.7^{\pm0.6}$	$0.0^{\pm0.0}$	$0.0^{\pm0.0}$
Padding-only	Break-Pad▽	$90.6^{\pm0.4}$	$93.7^{\pm1.9}$	$96.5^{\pm0.1}$	**$97.6^{\pm0.3}$**	$85.0^{\pm0.8}$	$75.3^{\pm0.4}$	$332.6^{\pm30.1}$
	Break-Pad	$90.6^{\pm0.3}$	$97.1^{\pm0.1}$	$97.3^{\pm0.3}$	**$97.7^{\pm0.2}$**	$86.8^{\pm0.6}$	$75.3^{\pm0.4}$	$0.0^{\pm0.0}$
	Ephemeral Pad▽	$72.3^{\pm1.3}$	$86.2^{\pm3.7}$	**$92.8^{\pm0.2}$**	$91.1^{\pm1.1}$	$66.2^{\pm0.7}$	$64.3^{\pm0.4}$	$43.9^{\pm7.1}$
	Ephemeral Pad	$71.1^{\pm0.7}$	$90.9^{\pm0.7}$	**$93.5^{\pm0.2}$**	$92.4^{\pm0.8}$	$67.0^{\pm0.7}$	$58.7^{\pm0.6}$	$0.0^{\pm0.0}$
	FRONT▽	$94.3^{\pm0.5}$	$96.2^{\pm0.4}$	**$96.9^{\pm0.6}$**	$96.2^{\pm0.3}$	$93.5^{\pm0.8}$	$18.2^{\pm0.3}$	$111.2^{\pm15.0}$
	FRONT	$91.8^{\pm0.8}$	$95.9^{\pm0.3}$	$96.2^{\pm0.4}$	**$96.3^{\pm0.4}$**	$94.9^{\pm0.9}$	$72.5^{\pm1.0}$	$0.0^{\pm0.0}$
	Interspace▽	$85.3^{\pm0.4}$	$95.1^{\pm0.4}$	$94.8^{\pm1.2}$	**$95.9^{\pm0.4}$**	$81.4^{\pm0.7}$	$56.3^{\pm0.6}$	$17.9^{\pm8.5}$
	Interspace	$85.7^{\pm0.6}$	$95.8^{\pm0.4}$	$95.9^{\pm0.4}$	**$96.4^{\pm0.4}$**	$81.7^{\pm0.9}$	$56.7^{\pm0.6}$	$0.0^{\pm0.0}$
Blocking	Ephemeral Block▽	$37.8^{\pm1.0}$	$52.1^{\pm5.7}$	**$76.1^{\pm2.0}$**	$62.1^{\pm3.3}$	$32.2^{\pm2.5}$	$78.8^{\pm0.9}$	$123.7^{\pm38.2}$
	Ephemeral Block	$29.2^{\pm1.0}$	$69.6^{\pm1.7}$	$71.8^{\pm0.8}$	**$78.3^{\pm0.2}$**	$24.3^{\pm1.5}$	$97.5^{\pm1.1}$	$68.4^{\pm0.7}$
	RegulaTor▽	$68.8^{\pm0.5}$	$76.9^{\pm0.7}$	$69.8^{\pm2.9}$	**$78.1^{\pm0.1}$**	$63.5^{\pm0.5}$	$38.6^{\pm0.2}$	$133.4^{\pm5.2}$
	RegulaTor	$68.1^{\pm0.6}$	$74.7^{\pm0.4}$	$73.5^{\pm1.1}$	$77.6^{\pm1.1}$	**$77.7^{\pm0.4}$**	$89.7^{\pm0.7}$	$7.8^{\pm0.2}$
	Tamaraw▽	$41.4^{\pm0.6}$	$45.7^{\pm0.6}$	$45.1^{\pm1.6}$	**$50.8^{\pm0.2}$**	$40.5^{\pm0.5}$	$127.9^{\pm2.8}$	$146.9^{\pm11.6}$
	Tamaraw	$35.6^{\pm0.5}$	$42.9^{\pm0.7}$	$39.0^{\pm0.5}$	**$44.8^{\pm0.2}$**	$39.5^{\pm0.9}$	$129.3^{\pm2.8}$	$73.4^{\pm0.8}$

Starting with padding-only defenses, only ephemeral defenses show a slight accuracy reduction of 3–4% compared to undefended against Laserbeak. In the same way that DF bypasses the padding-only defense WTF-PAD [36, 72], we see that Laserbeak does the same for Break-Pad, FRONT, Interspace, and ephemeral defenses (to a slightly lesser degree) given enough training time and data. Akin to the closely related Anonymity Trilemma [13, 14], defense against WF attacks seemingly requires some degree of explicit bandwidth ***and*** delay overheads. It is an open question if effective and efficient padding-only defenses are possible in the setting.

Takeaway: Given sufficient training time and data in a simulated closed-world setting, Laserbeak's effectiveness against padding-only defenses is the same as for undefended datasets.

Blocking defenses still provide protection under infinite training. Tamaraw sees the smallest increase in accuracy between the tables. As a constant-rate defense fit (parameter tuned) to the underlying dataset to minimize overheads, all information leaked is in the tail from the soft stop condition. Despite that we see 45–51% accuracy, indicating that the parameter choice for the soft stop condition is essential. Regulator and ephemeral blocking defenses are comparable in terms of accuracy, with RegulaTor having a lower overall overhead. Laserbeak, compared to other attacks, is especially effective against ephemeral blocking defenses, indicating that the Laserbeak architecture is a key strength in this regard, particularly the attention layers. We know that it is not the feature representation since DF-multi does notably worse. For RegulaTor, on the other hand, all attacks are within 15%. Figure 4 shows that ephemeral blocking defenses (no bottleneck) can be tuned, by altering deployment parameters, to trade overhead for defense against infinite trained Laserbeak^{-} (the highest accuracy attack for the defense in Table 2).

Takeaway: Blocking-based defenses remain effective, albeit to a lesser extent, in closed-world simulated evaluations with infinite training. Laserbeak's architecture is particularly effective against ephemeral blocking defenses.

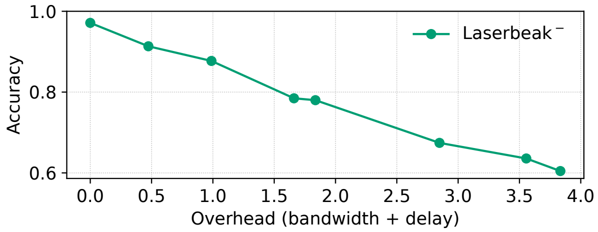


Figure 4: Ephemeral blocking (infinite network) offers tunable defense against Laserbeak^{-} with infinite training.

5.4 Generalizability of Trained Attacks

Tables 1 and 2 involved training and testing a large number of attacks. The training and testing took place on the same defenses, i.e., we trained the attack on data from the same defense as we later tested on. Here, we create attack/defense heatmaps, where we test each trained attack on every defense. We test both the standard 30 epochs trained and infinitely trained attacks.

Figure 5 shows the results for Laserbeak, with 30 epochs trained on the left and infinite training to the right. There is a stark impact on generalizability due to infinite training. For infinite training, the models trained on ephemeral defenses generalize particularly well, especially on ephemeral blocking defenses. This is probably due to the diversity of defenses observed during training. Tamaraw is the only defense where the models completely fail—likely

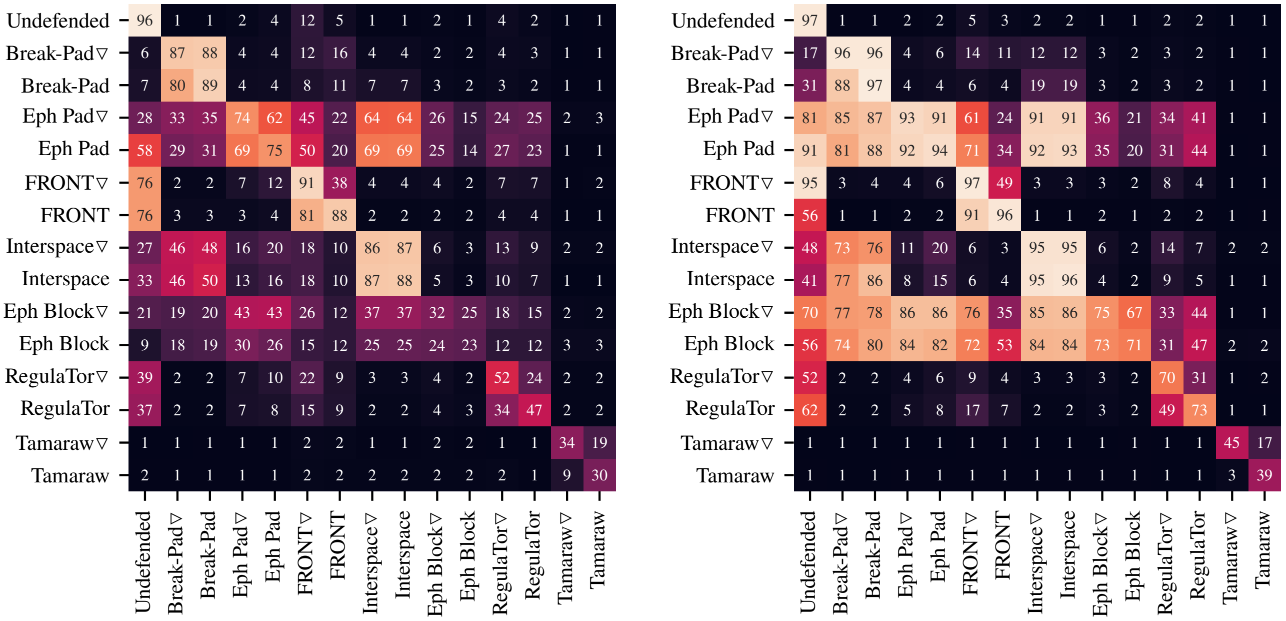


Figure 5: Accuracy on the BigEnough standard dataset [43] in a closed world for *Laserbeak* [42] trained on the defense given in row when tested on the defense given in column. Left for the 30 epochs training, right for infinite training.

because Tamaraw is the only constant-rate defense—a widely different defense strategy. The second-worst defense is RegulaTor, which uses heavy regularization (related to constant-rate defenses), further confirming the observation. The resulting heatmaps for Laserbeak⁻, DF-multi, RF, and DF are in Appendix G. They show a similar trend, albeit at lower accuracies.

Takeaway: Attacks trained infinitely on ephemeral defenses show improved generalizability for randomized defenses, likely due to the greater diversity in ephemeral defenses.

5.5 The Gong-Surakav Dataset

Parameter tuning of defenses for BigEnough was an elaborate task (Appendix E). Without re-tuning, we investigate how well the parameter tuning transfers to the closely related Gong-Surakav [21] undefended dataset. Table 3 presents the results, which are obtained by performing the same evaluation as earlier on BigEnough (Table 1) but using the Gong-Surakav dataset instead.

We start with overheads. The bottleneck network model is much less impactful across the board in terms of delay, with the most dramatic *decrease* for Break-Pad (332.6±30.1% ⟶ 18.9±10.4%), FRONT (109.7±12.8% ⟶ 3.0±4.0%) and RegulaTor (133.4±5.2% ⟶ 32.7±8.4%). For the infinite network model, Tamaraw sees a significant *increase* (73.4±0.8% ⟶ 125.8±1.3%), as do ephemeral blocking defenses (68.1±0.8% ⟶ 90.2±1.2%) to a lesser extent. For bandwidth, overhead is down for every defense, especially for padding-only defenses.

In terms of attack accuracy, results are more similar to the infinite training (Table 2) than the default attack parameters (Table 1). This can likely be explained by comparing the two dataset structures: BigEnough contains 95 classes with 20 samples each of 10 subpages of each website/class (samples of subpages split randomly), while the Gong-Surakav dataset contains 100 classes with 100 samples each of the frontpage of each website. Gong-Surakav represents a much easier classification task, both with more samples per webpage and with only 100 webpages compared to 950 webpages (over 95 classes) in BigEnough. For padding-only defenses, none of the defenses provide any significant protection, with a slight edge to padding-only ephemeral defenses.

For blocking defenses, on the one hand, ephemeral blocking and Tamaraw behave similarly to the infinite training case. RegulaTor, on the other hand, offers protection similar to padding-only defenses. The parameter tuning of RegulaTor involves fitting the distribution of one or more incoming bursts of traffic constrained by a limited budget; this is inherently dataset dependent. Tamaraw, with the soft stop condition based on a time window of no real traffic, becomes more robust as a defense. Ephemeral defenses are more general by virtue of consisting of many different defenses that fit any dataset to some extent. This is important for practical, real-world defenses, where the distribution of traffic is not well known in advance (e.g., defenses for CF deal with a more predictable distribution than WF, with VF falling somewhere in between).

Takeaway: Defenses with parameters tightly coupled to the underlying distribution of traffic may deal poorly with out-of-distribution traffic, analogous to how WF attacks may struggle in the open-world instead of a closed-world setting.

Table 3: A repeat of Table 1 with the same defense and attack parameters, but simulated using the Gong-Surakav dataset [21].

	Gong-Surakav	Accuracy %					Overhead %	
		DF	DF-multi	Laserbeak	Laserbeak^{-}	RF	Bandwidth	Delay
	Undefended	$96.1^{\pm 0.4}$	$97.5^{\pm 0.3}$	$97.6^{\pm 0.4}$	$\mathbf{97.9^{\pm 0.5}}$	$95.3^{\pm 0.7}$	$0.0^{\pm 0.0}$	$0.0^{\pm 0.0}$
Padding-only	Break-Pad▽	$83.7^{\pm 0.5}$	$92.4^{\pm 0.6}$	$94.0^{\pm 0.7}$	$\mathbf{94.4^{\pm 0.4}}$	$90.8^{\pm 0.4}$	$46.2^{\pm 0.2}$	$18.9^{\pm 10.4}$
	Break-Pad	$84.1^{\pm 0.8}$	$93.3^{\pm 0.4}$	$94.7^{\pm 0.8}$	$\mathbf{95.1^{\pm 0.4}}$	$92.0^{\pm 0.6}$	$46.2^{\pm 0.3}$	$0.0^{\pm 0.0}$
	Ephemeral Pad▽	$64.8^{\pm 1.6}$	$86.9^{\pm 1.1}$	$\mathbf{90.5^{\pm 0.5}}$	$89.9^{\pm 0.6}$	$85.0^{\pm 1.3}$	$53.7^{\pm 0.4}$	$2.1^{\pm 2.9}$
	Ephemeral Pad	$54.9^{\pm 3.2}$	$80.0^{\pm 1.1}$	$\mathbf{87.9^{\pm 1.4}}$	$86.8^{\pm 0.8}$	$79.9^{\pm 1.1}$	$52.7^{\pm 1.1}$	$0.0^{\pm 0.0}$
	FRONT▽	$92.2^{\pm 0.6}$	$95.3^{\pm 0.4}$	$95.5^{\pm 0.6}$	$\mathbf{96.2^{\pm 0.6}}$	$95.7^{\pm 0.7}$	$10.8^{\pm 0.2}$	$3.0^{\pm 4.0}$
	FRONT	$71.5^{\pm 2.7}$	$86.0^{\pm 0.7}$	$90.2^{\pm 0.9}$	$93.0^{\pm 0.9}$	$\mathbf{95.0^{\pm 0.7}}$	$43.3^{\pm 0.7}$	$0.0^{\pm 0.0}$
	Interspace▽	$68.8^{\pm 1.2}$	$91.3^{\pm 0.4}$	$\mathbf{93.7^{\pm 0.6}}$	$93.7^{\pm 0.6}$	$85.8^{\pm 0.7}$	$52.2^{\pm 0.4}$	$0.6^{\pm 0.1}$
	Interspace	$67.9^{\pm 1.7}$	$91.4^{\pm 0.7}$	$\mathbf{94.0^{\pm 0.4}}$	$93.9^{\pm 0.5}$	$86.2^{\pm 0.4}$	$52.2^{\pm 0.4}$	$0.0^{\pm 0.0}$
Blocking	Ephemeral Block▽	$23.8^{\pm 1.7}$	$56.3^{\pm 1.9}$	$\mathbf{67.8^{\pm 1.2}}$	$65.6^{\pm 1.7}$	$51.6^{\pm 1.3}$	$68.5^{\pm 1.1}$	$73.4^{\pm 5.6}$
	Ephemeral Block	$17.9^{\pm 0.4}$	$46.4^{\pm 1.5}$	$55.5^{\pm 0.6}$	$\mathbf{56.2^{\pm 0.9}}$	$44.4^{\pm 1.0}$	$81.8^{\pm 1.5}$	$90.2^{\pm 1.2}$
	RegulaTor▽	$83.7^{\pm 0.6}$	$90.6^{\pm 0.6}$	$92.0^{\pm 0.9}$	$\mathbf{92.8^{\pm 0.7}}$	$92.2^{\pm 0.8}$	$22.8^{\pm 0.2}$	$32.7^{\pm 8.4}$
	RegulaTor	$70.1^{\pm 1.8}$	$82.4^{\pm 1.5}$	$87.7^{\pm 0.4}$	$90.7^{\pm 0.7}$	$\mathbf{94.0^{\pm 1.1}}$	$54.9^{\pm 0.4}$	$9.2^{\pm 0.0}$
	Tamaraw▽	$34.3^{\pm 0.9}$	$\mathbf{42.4^{\pm 1.2}}$	$39.9^{\pm 0.9}$	$42.1^{\pm 0.5}$	$37.0^{\pm 4.2}$	$115.3^{\pm 1.4}$	$144.7^{\pm 15.9}$
	Tamaraw	$29.5^{\pm 0.8}$	$\mathbf{38.2^{\pm 0.9}}$	$32.8^{\pm 1.1}$	$34.7^{\pm 0.9}$	$32.5^{\pm 3.8}$	$121.7^{\pm 1.5}$	$125.8^{\pm 1.3}$

6 Video Fingerprinting

As a benchmark defense for VF, we use Scrambler [25], a state-of-the-art defense implemented in Maybenot designed explicitly for video traffic. Scrambler assumes reasonably high bandwidth, under which overlapping segment downloads are rare. It detects a segment download via a normal packet event and sends at a fixed rate (every δ ms) until N packets have been sent, after which padding is sent until several packets sampled uniformly from the range $[P_{min}, P_{max}]$ can be sent with no intervening data packets, as a heuristic to detect the end of a segment download.

To evaluate Scrambler and our ephemeral defenses, we utilize the LongEnough dataset [25], which was collected and used to evaluate Scrambler. LongEnough consists of 100 movies streamed with 100 Mbps constant bandwidth (default) and is also available in an extended version with four variable bandwidth scales (bw1, bw2, bw4, and bw8). In the latter case, a real-world LTE bandwidth trace is randomized, scaled by the number in the dataset name, and used to limit the bandwidth available for streaming during data collection. We use the strongest parameters reported by Hasselquist et al. after tuning Scrambler on the default LongEnough settings: $\delta = 120$, $N = 1{,}500$, $P_{min} = 400$, $P_{max} = 1{,}000$. In their evaluations, this configuration results in 284% bandwidth overhead.

We use vDF and vRF [10] for attacks with ten-fold cross-validation. For overheads, we simulate each defense ten times. The variable-bandwidth datasets of LongEnough inherently capture network fluctuations, and adding another (simulated) bottleneck may introduce confounding interactions between simulated and dataset-imposed constraints. Appendix A reports one related suspected edge case. For ephemeral defenses, we use the ephemeral blocking defenses *for WF from Section 5 without tuning for VF.*

Table 4 shows the results. We achieve nearly identical results to Carlson et al. with vDF and vRF against undefended traffic [10]. Our simulations of Scrambler align with Hasselquist et al.'s real-world deployments [25] in terms of bandwidth overhead: They report 284.0% overhead, which is 21.1% higher than our results. The nearly exponential increase in overhead we see with decreasing bandwidth is expected, as the parameters we use are not tuned for variable bandwidth, and all of Scrambler's parameters are highly sensitive to bandwidth conditions due to ABR algorithms.

Table 4: Video Fingerprinting on the LongEnough dataset (default and four with variable bandwidth conditions) [25], evaluated using vDF and vRF [10]. Scrambler is a SotA VF defense [25], ephemeral is the ephemeral blocking defense from Section 5 without any tuning.

	LongEnough	Accuracy %		Overhead %	
		vDF	vRF	Bandwidth	Delay
Undefended	Default	$99.6^{\pm 0.3}$	$100.0^{\pm 0.0}$	0.0%	0.0%
	bw8	$98.5^{\pm 0.3}$	$99.7^{\pm 0.2}$	0.0%	0.0%
	bw4	$96.8^{\pm 0.5}$	$98.0^{\pm 0.4}$	0.0%	0.0%
	bw2	$90.2^{\pm 0.6}$	$93.4^{\pm 0.7}$	0.0%	0.0%
	bw1	$80.7^{\pm 1.8}$	$87.3^{\pm 1.1}$	0.0%	0.0%
Scrambler	Default	$1.0^{\pm 0.0}$	$1.0^{\pm 0.0}$	$262.9^{\pm 0.0}$	$64.8^{\pm 0.0}$
	bw8	$1.0^{\pm 0.0}$	$1.0^{\pm 0.0}$	$459.4^{\pm 0.0}$	$57.1^{\pm 0.0}$
	bw4	$1.0^{\pm 0.0}$	$1.0^{\pm 0.0}$	$810.9^{\pm 0.1}$	$43.5^{\pm 0.0}$
	bw2	$0.7^{\pm 0.3}$	$0.9^{\pm 0.3}$	$1445.6^{\pm 0.1}$	$31.7^{\pm 0.0}$
	bw1	$0.4^{\pm 0.4}$	$0.5^{\pm 0.6}$	$2364.2^{\pm 0.2}$	$19.9^{\pm 0.0}$
Ephemeral	Default	$19.3^{\pm 1.3}$	$24.6^{\pm 1.1}$	$134.0^{\pm 0.8}$	$74.6^{\pm 0.7}$
	bw8	$10.5^{\pm 1.2}$	$23.6^{\pm 0.9}$	$154.6^{\pm 0.8}$	$75.2^{\pm 0.7}$
	bw4	$7.1^{\pm 1.0}$	$17.8^{\pm 1.6}$	$164.6^{\pm 0.8}$	$78.7^{\pm 0.8}$
	bw2	$3.1^{\pm 0.6}$	$7.8^{\pm 1.3}$	$175.3^{\pm 0.8}$	$81.8^{\pm 9.2}$
	bw1	$2.1^{\pm 0.4}$	$3.3^{\pm 0.4}$	$187.1^{\pm 0.7}$	$81.3^{\pm 11.3}$

Hasselquist et al. [25] also mention that 1.1% of playback is spent waiting for a 10-minute video on average, corresponding to roughly 1.1% delay overhead. The higher overhead we observe is likely due to aggregated delays in the simulator not fully capturing the dynamics of continuous traffic and potentially also a result of the specific traces used during defense evaluation. This suggests that ephemeral blocking defenses have lower delay overheads than we report in real-world deployments.

Regarding protection, Scrambler renders vDF and vRF completely ineffective; the highest accuracy is 1%, equivalent to random guessing. This also aligns with Hasselquist et al.'s conclusion that Scrambler provides perfect protection. The efficacy of ephemeral blocking defenses, on the other hand, depends on the bandwidth scale: vRF is the most effective attack in all cases. It achieves 25% accuracy against default LongEnough, with accuracy decreasing (as bandwidth increases) to 3% against bw1. This represents a significant weakening of the attacks at a much lower overhead than Scrambler. Bandwidth overhead increases from 134% for default to 187% for bw1, while delay overhead remains relatively stable from 75% to 82%. Thus, ephemeral blocking defenses represent a practical option for strong protection without Scrambler's bandwidth overhead or volatility in variable network conditions.

Takeaway: Ephemeral blocking defenses offer a practical trade-off between accuracy and overhead in both good and variable bandwidth conditions for VF, suggesting that they are viable in situations with long-lasting, continuous traffic.

7 Discussion

In the same way that Section 6 just demonstrated that ephemeral blocking defenses tuned for WF are viable for VF, Figure 6 compares the ephemeral blocking WF defense with the tuned ephemeral CF defense from Section 4. We see that the WF ephemeral blocking defense offers tunable protection against circuit, video, and website fingerprinting attacks. In settings where the application-layer traffic is unknown, ephemeral defenses shine. If the application-layer traffic is known, highly optimized network-layer defenses offer a better trade-off, e.g., Scrambler for VF, Tamaraw for WF, or optimizing ephemeral defenses to particular settings such as CF.

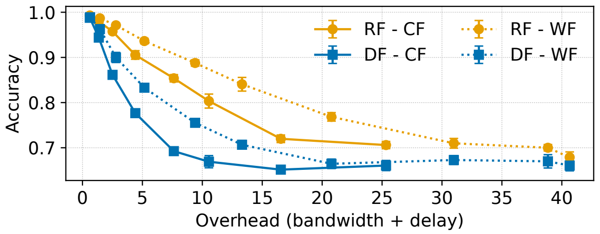


Figure 6: The ephemeral blocking WF defense from Section 5 without any tuning as a CF defense. Compared with the finally tuned ephemeral CF defense.

Takeaway: By not being tuned/overfitted to attack, dataset, or network conditions, ephemeral defenses are multipurpose network-layer defenses against fingerprinting.

One such setting where a network-layer technology carries different kinds of application-layer traffic is VPNs. In this paper, we have simulated all defenses and attacks for the sake of comparing their relative strengths and weaknesses. We can briefly report that ephemeral defenses with Maybenot have been integrated with WireGuard [17, 46] and deployed in production at Mullvad VPN for about a year [44, 45], supporting thousands (and growing) active daily connections. Note that the integration also pads all packets in the WireGuard tunnel to MTU [46]. While there is much to improve and optimize—and absolute tuning for deployment remains largely disconnected from relative tuning for closed-world comparisons [12]—we believe that there is no substitute for iteration and experience from deployment. The defense strategy of ephemeral defenses and the basic methodology of search presented in this paper are practical and effective. If anything, the user experience is acceptable (as indicated by a growing number of users), and the protection offered against fingerprinting and other forms of traffic analysis (with the notable exception of likely making censorship easier due to WireGuard modifications) is greater than that of standalone WireGuard.

With ephemeral defenses comes the question of defense distribution. In the VPN case, where there is inherent trust in the VPN servers, we built an architecture where VPN clients dynamically receive their defense and configuration (Maybenot machines and framework limits) upon establishing a connection. VPN servers, in turn, have a (large) database of ephemeral defenses that are straightforward to update and can eventually be replaced with truly unique defenses per connection as we iterate. In the case of Tor, with distributed trust in relays, the seed-based derivation of defenses (Section 3.2.1) enables verifiable defense derivation based on a mutually agreed-upon configuration, e.g., as part of the Tor consensus. The seed could incorporate the shared random value from the consensus to ensure freshness. Having both the client and relay be aware of the defenses of both parties allows them to reason about received padding from the other party, making it harder to use padding and blocking actions as a side channel [53].

8 Related Work

Interspace [57] was developed in part by evolving state machines using a genetic algorithm within the Tor Circuit Padding Framework [54, 55], which employed an associated simulator and DF in its learning loop. Our semi-automated tuning (Section 4) similarly uses a simulator, state machines, and DF. Unlike Interspace, we do not perform manual changes of machines. The manual changes introduced randomized parameters per instance of Interspace. FRONT [20] also uses randomized parameters per instance, i.e., randomized padding budget and padding window. For ephemeral defenses, every defense is potentially unique. Ephemeral defenses use random sampling, as established defense strategy [43] (Section 2.2), but take this strategy one step further with *randomly generated* random sampling defenses. Future work, building on attack generalizability (Figure 5), could strengthen the theoretical foundation for ephemeral defenses.

DeTorrent [29] by Holland et al. employs competing neural networks to generate and evaluate traffic analysis defenses, where the defense utilizes an LSTM that, at time steps based on prior traffic, determines how much padding to send. While resulting in essentially unique padding defenses, the LSTM is not implementable as state machines (as Holland et al. note). They also worked on transferability using the BigEnough dataset (but not across datasets for WF) and across traffic analysis domains. Similar to how we note that ephemeral defenses translate well across datasets and domains, they found that their WF defense was also effective at defending against Flow Correlation attacks [39, 47, 50, 63, 71, 75, 85].

Another WF defense that results in largely unique defenses is Surakav by Gong et al. [21]. Surakav uses a GAN to generate realistic traffic patterns and regulates traffic to match those patterns. Like DeTorrent, Surakav is not implementable as state machines. With its strict regularization approach, we observe that Surakav is highly application- and network-dependent.

The Laserbeak [42] closed-world evaluation was also done on BigEnough, with 10x data augmentation, and with similar attacks and defenses, albeit our defenses are all implemented in Maybenot. The results align well with our infinite training results, indicating that padding-only defenses offer little to no protection. The main difference is that our more practical implementations of RegulaTor and Tamaraw show worse protection with about 20% higher average accuracy, likely due to our implementations.

A number of works show that network congestion and (simplified) simulation can have a significant negative impact on both WF defenses and attacks in practice [3, 12, 32–34, 83]. This is closely related to the long-going discussion of the real-world practicality of WF attacks [35, 52, 82]. Our results align with these observations in several ways. For one, closed-world defense comparisons against state-of-the-art attacks, such as Laserbeak, are soon rendered pointless due to near-perfect attack accuracies, at least for padding-only defenses. Defenses need to induce both bandwidth and delay [13, 14]. We also note significant differences for defenses based on (simulated) network conditions and across datasets; sterile lab tuning of defenses generalizes poorly. On the attack side, if provided sufficient training time, we observe that attacks trained on ephemeral blocking defenses generalize better across various defenses and network conditions than if trained on other defenses.

In parallel with our work, Wails et al. published Unidentified Protocol Generation (UPGen), a protocol generator for circumventing Internet censorship [81]. UPGen can generate up to 4.2×10^{22} distinct protocols, making blocklisting infeasible for a censor. There are many parallels to ephemeral defenses. Mainly, shifting to generating defenses with fairly simple (practical) methods over careful manual design. Network adversaries, whether their goal is to fingerprint or censor network traffic, need to consider all traffic on the Internet. While each UPGen protocol or ephemeral defense on its own may offer little protection against targeted attacks, collectively, as a defense strategy, the adversary faces a daunting task due to the sheer number of possible defenses. One key difference is that UPGen protocols, by definition, define the exact protocol to use, while ephemeral defenses *shape* the protocol that Maybenot has been integrated with. Typically, such shaping does not entirely define the patterns of communication (unless in the case of a constant-rate defense), necessitating a trade-off between overheads and information leaked (anonymity provided [13, 14]). We note that such information leaks could also be used by a censor that targets not protocols but activities, such as visiting a particular website.

9 Conclusion

Ephemeral defenses are multipurpose network-layer defenses that are not tightly tuned to any particular fingerprinting attack, dataset, or network conditions. The ephemeral defense strategy introduced in this paper uses simple methods of defense search and semi-automated tuning. There is room for improvement. In general, dynamic selection, generation, or assignment (depending on the setting) of ephemeral defenses opens up for improved customization to particular network conditions—reducing the negative impact on the user experience of defenses—and may deprive attackers of the capability to train on the exact defense used by users. In a sense, this shifts Kerckhoffs's principle for fingerprinting defenses to treat defenses like cryptographic keys (with public parameters to generate, i.e., the configuration files) instead of public knowledge.

Artifacts

To promote reproducibility and further development of ephemeral network-layer defenses, we release three Rust crates available on crates.io as open-source (MIT or Apache-2.0 dual-license):

`maybenot-cli` A command-line interface for generating ephemeral defenses, as well as tweaking, simulating, and evaluating Maybenot defenses.

`maybenot-gen` A library to generate ephemeral defenses and to evaluate Maybenot defenses.

`maybenot-machines` A collection of Maybenot machines, including related work Break-Pad [31], FRONT [20], Interspace [57], RegulaTor [30], Scrambler [25], and Tamaraw [9, 22].

The Rust workspace for all Maybenot tools is at https://github.com/maybenot-io/maybenot. An archived snapshot of all necessary tools to enable more precise reproducibility of the results in this paper is available at https://github.com/maybenot-io/popets-2026.1-ephemeral-defs-paper-artifacts.

Acknowledgments

A big thank you to everyone at Mullvad VPN for your efforts on DAITA: Daniel Berntsson, Linus Färnstrand, Erik der Hagopian, Sebastian Holmin, Joakim Hulthe, Jan Jonsson, Oscar Linderholm, David Lönnhager, Odd Stranne, and Fredrik Strömberg.

This work was funded by the Knowledge Foundation of Sweden, Mullvad VPN, Karlstad Internet Privacy Lab, the Graduate School in Computer Science (CUGS) at Linköping University, and the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

The authors used the generative AI-based tools Grammarly and GitHub Copilot (no explicit prompting, continuous feedback from the LaTeX IDE) to revise the text, enhance the flow, and correct any typos, grammatical errors, and awkward phrasing. ChatGPT o3 assisted in creating our Python scripts for plotting figures.

All experiments in this paper were conducted by simulation on datasets made available by other researchers [21, 25, 43, 76]. The deployed ephemeral defenses with Mullvad VPN in DAITA are an opt-in feature. There is no additional direct harm from our research; we believe that openly advancing the state-of-the-art of practical fingerprinting defenses is in the greater interest of society.

References

[1] Kota Abe and Shigeki Goto. 2016. Fingerprinting attack on Tor anonymity using deep learning. *Proceedings of the Asia-Pacific Advanced Network*.

[2] Mashael AlSabah, Kevin Bauer, Tariq Elahi, and Ian Goldberg. 2013. The path less travelled: Overcoming Tor's bottlenecks with traffic splitting. In *Privacy Enhancing Technologies: 13th International Workshop, PETS 2006*. Springer-Verlag, LNCS 7981. https://doi.org/10.1007/978-3-642-39077-7_8

[3] Alireza Bahramali, Ardavan Bozorgi, and Amir Houmansadr. 2023. Realistic Website Fingerprinting By Augmenting Network Traces. In *CCS*.

[4] Ludovic Barman, Sandra Siby, Christopher A. Wood, Marwan Fayed, Nick Sullivan, and Carmela Troncoso. 2022. This is not the padding you are looking for! On the ineffectiveness of QUIC PADDING against website fingerprinting. *CoRR* (2022). https://doi.org/10.48550/arXiv.2203.07806

[5] Matthias Beckerle, Jonathan Magnusson, and Tobias Pulls. 2022. Splitting Hairs and Network Traces: Improved Attacks Against Traffic Splitting as a Website Fingerprinting Defense. In *WPES*.

[6] Sanjit Bhat, David Lu, Albert Kwon, and Srinivas Devadas. 2019. Var-CNN: A Data-Efficient Website Fingerprinting Attack Based on Deep Learning. *PETS*.

[7] George Dean Bissias, Marc Liberatore, David D. Jensen, and Brian Neil Levine. 2005. Privacy Vulnerabilities in Encrypted HTTP Streams. In *Privacy Enhancing Technologies, 5th International Workshop, PET 2005, Cavtat, Croatia, May 30-June 1, 2005, Revised Selected Papers*.

[8] David Blackman and Sebastiano Vigna. 2021. Scrambled linear pseudorandom number generators. *ACM Transactions on Mathematical Software (TOMS)* 47, 4 (2021), 1–32.

[9] Xiang Cai, Rishab Nithyanand, Tao Wang, Rob Johnson, and Ian Goldberg. 2014. A Systematic Approach to Developing and Evaluating Website Fingerprinting Defenses. In *CCS*.

[10] August Carlson, David Hasselquist, Ethan Witwer, Niklas Johansson, and Niklas Carlsson. 2024. Understanding and Improving Video Fingerprinting Attack Accuracy under Challenging Conditions. In *WPES*.

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

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

[13] Debajyoti Das, Sebastian Meiser, Esfandiar Mohammadi, and Aniket Kate. 2018. Anonymity Trilemma: Strong Anonymity, Low Bandwidth Overhead, Low Latency - Choose Two. In *IEEE S&P*.

[14] Debajyoti Das, Sebastian Meiser, Esfandiar Mohammadi, and Aniket Kate. 2020. Comprehensive Anonymity Trilemma: User Coordination is not enough. *PETS*.

[15] DASH-Industry-Forum. 2024. dash.js. https://dashjs.org/.

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

[17] Jason A Donenfeld. 2017. WireGuard: Next Generation Kernel Network Tunnel.. In *NDSS*.

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

[19] PyTorch Foundation. 2025. ReduceLROnPlateau — PyTorch 2.7 documentation. https://docs.pytorch.org/docs/stable/generated/torch.optim.lr_scheduler.ReduceLROnPlateau.html, accessed 2025-05-28.

[20] Jiajun Gong and Tao Wang. 2020. Zero-delay Lightweight Defenses against Website Fingerprinting. In *USENIX Security*.

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

[22] Jiajun Gong, Wuqi Zhang, Charles Zhang, and Tao Wang. 2024. WFDefProxy: Real World Implementation and Evaluation of Website Fingerprinting Defenses. *IEEE Trans. Inf. Forensics Secur.* 19.

[23] David Goulet and Mike Perry. 2020. Overcoming Tor's Bottlenecks with Traffic Splitting. https://spec.torproject.org/proposals/329-traffic-splitting.html.

[24] Jiaxi Gu, Jiliang Wang, Zhiwen Yu, and Kele Shen. 2018. Walls Have Ears: Traffic-based Side-channel Attack in Video Streaming. In *Proc. IEEE INFOCOM*.

[25] David Hasselquist, Ethan Witwer, August Carlson, Niklas Johansson, and Niklas Carlsson. 2024. Raising the Bar: Improved Fingerprinting Attacks and Defenses for Video Streaming Traffic. *PoPETs* (2024).

[26] Jamie Hayes and George Danezis. 2016. k-fingerprinting: A Robust Scalable Website Fingerprinting Technique. In *USENIX Security*.

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

[28] Andrew Hintz. 2002. Fingerprinting Websites Using Traffic Analysis. In *PETS*.

[29] James K. Holland, Jason Carpenter, Se Eun Oh, and Nicholas Hopper. 2024. DeTorrent: An Adversarial Padding-only Traffic Analysis Defense. *PETS*.

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

[31] Bin Huang and Yanhui Du. 2024. Break-Pad: effective padding machines for Tor with break burst padding. *Cybersecurity* 7, 1 (2024), 28. https://cybersecurity.springeropen.com/articles/10.1186/s42400-024-00222-y.

[32] Rob Jansen and Ryan Wails. 2023. Data-Explainable Website Fingerprinting with Network Simulation. *PETS*. See also https://explainwf-popets2023.github.io.

[33] Rob Jansen, Ryan Wails, and Aaron Johnson. 2024. A Measurement of Genuine Tor Traces for Realistic Website Fingerprinting. *CoRR* abs/2404.07892 (2024). https://doi.org/10.48550/arXiv.2404.07892

[34] Rob Jansen, Ryan Wails, and Aaron Johnson. 2024. Repositioning Real-World Website Fingerprinting on Tor. In *WPES*.

[35] Marc Juarez, Sadia Afroz, Gunes Acar, Claudia Díaz, and Rachel Greenstadt. 2014. A Critical Evaluation of Website Fingerprinting Attacks. In *CCS*.

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

[37] George Kadianakis, Theodoros Polyzos, Mike Perry, and Kostas Chatzikokolakis. 2022. Tor circuit fingerprinting defenses using adaptive padding. arXiv:2103.03831 [cs.CR]

[38] Albert Kwon, Mashael AlSabah, David Lazar, Marc Dacier, and Srinivas Devadas. 2015. Circuit Fingerprinting Attacks: Passive Deanonymization of Tor Hidden Services. In *USENIX Security*.

[39] Brian Neil Levine, Michael K. Reiter, Chenxi Wang, and Matthew K. Wright. 2004. Timing Attacks in Low-Latency Mix Systems (Extended Abstract). In *Financial Cryptography*.

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

[41] Akshaya Mani, T Wilson-Brown, Rob Jansen, Aaron Johnson, and Micah Sherr. 2018. Understanding tor usage with privacy-preserving measurement. In *IMC*.

[42] Nate Mathews, James K Holland, Nicholas Hopper, and Matthew Wright. 2024. LASERBEAK: Evolving Website Fingerprinting Attacks with Attention and Multi-Channel Feature Representation. *IEEE Transactions on Information Forensics and Security* (2024).

[43] 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.

[44] Mullvad VPN. 2024. Introducing Defense against AI-guided Traffic Analysis (DAITA), https://web.archive.org/web/20250825193659/https://mullvad.net/en/blog/introducing-defense-against-ai-guided-traffic-analysis-daita.

[45] Mullvad VPN. 2025. DAITA: Defense Against AI-guided Traffic Analysis, https://web.archive.org/web/20250504100318/https://mullvad.net/en/vpn/daita.

[46] Mullvad VPN. 2025. Go Implementation of WireGuard - Mullvad VPN fork. https://github.com/mullvad/wireguard-go/, accessed 2025-08-25.

[47] Milad Nasr, Alireza Bahramali, and Amir Houmansadr. 2018. DeepCorr: Strong Flow Correlation Attacks on Tor Using Deep Learning. In *CCS*.

[48] NIST. 2024. FIPS 203: Module-Lattice-Based Key-Encapsulation Mechanism Standard. https://csrc.nist.gov/pubs/fips/203/final [Online; accessed 7-March-2025].

[49] Se Eun Oh, Saikrishna Sunkam, and Nicholas Hopper. 2019. p1-FP: Extraction, Classification, and Prediction of Website Fingerprints with Deep Learning. *PETS*.

[50] 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 SP*.

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

[52] Mike Perry. 2013. A Critique of Website Traffic Fingerprinting Attacks, https://web.archive.org/web/20250119143043/https://blog.torproject.org/critique-website-traffic-fingerprinting-attacks/.

[53] Mike Perry. accessed 2025-04-22. Proposal 344: Information Leak Hazards for Tor Implementations. https://spec.torproject.org/proposals/344-protocol-info-leaks.html.

[54] Mike Perry and George Kadianakis. accessed 2025-03-02. Circuit Padding Developer Documentation. https://gitweb.torproject.org/tor.git/tree/doc/HACKING/CircuitPaddingDevelopment.md.

[55] Mike Perry and George Kadianakis. accessed 2025-03-02. Tor Padding Specification. https://spec.torproject.org/padding-spec/index.html.

[56] David Peter and hyperfine contributors. 2025. hyperfine, A command-line benchmarking tool. https://github.com/sharkdp/hyperfine, accessed 2025-04-09.

[57] Tobias Pulls. 2020. Towards Effective and Efficient Padding Machines for Tor. *CoRR* abs/2011.13471 (2020). https://arxiv.org/abs/2011.13471

[58] Tobias Pulls and Rasmus Dahlberg. 2020. Website Fingerprinting with Website Oracles. *PoPETs* (2020).

[59] Tobias Pulls and Ethan Witwer. 2023. Maybenot: A Framework for Traffic Analysis Defenses. In *WPES*.

[60] Tobias Pulls and Ethan Witwer. 2024. Maybenot: A Framework for Traffic Analysis Defenses. arXiv:2304.09510 [cs.CR] https://arxiv.org/abs/2304.09510

[61] Tobias Pulls and Ethan Witwer. 2025. maybenot - crates.io: Rust Package Registry. https://crates.io/crates/maybenot [Online; accessed 2-March-2025].

[62] 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. *PETS.*

[63] Jean-François Raymond. 2000. Traffic Analysis: Protocols, Attacks, Design Issues, and Open Problems. In *Designing Privacy Enhancing Technologies, International Workshop on Design Issues in Anonymity and Unobservability, Berkeley, CA, USA, July 25-26, 2000, Proceedings.*

[64] Andrew Reed and Benjamin Klimkowski. 2016. Leaky streams: Identifying variable bitrate DASH videos streamed over encrypted 802.11n connections. In *Proc. IEEE Consumer Communications & Networking Conference (CCNC).*

[65] Eric Rescorla. 2018. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446. https://doi.org/10.17487/RFC8446

[66] Vera Rimmer, Davy Preuveneers, Marc Juárez, Tom van Goethem, and Wouter Joosen. 2018. Automated Website Fingerprinting through Deep Learning. In *NDSS.*

[67] 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. *PoPETs.*

[68] T. Scott Saponas, Jonathan Lester, Carl Hartung, Sameer Agarwal, and Tadayoshi Kohno. 2007. Devices That Tell on You: Privacy Trends in Consumer Ubiquitous Computing. In *USENIX Security.*

[69] Roei Schuster, Vitaly Shmatikov, and Eran Tromer. 2017. Beauty and the Burst: Remote Identification of Encrypted Video Streams. In *USENIX Security.*

[70] Meng Shen, Kexin Ji, Zhenbo Gao, Qi Li, Liehuang Zhu, and Ke Xu. 2023. Subverting Website Fingerprinting Defenses with Robust Traffic Representation. In *USENIX Security.*

[71] Vitaly Shmatikov and Ming-Hsiu Wang. 2006. Timing Analysis in Low-Latency Mix Networks: Attacks and Defenses. In *ESORICS.*

[72] Payap Sirinam, Mohsen Imani, Marc Juarez, and Matthew Wright. 2018. Deep Fingerprinting: Undermining Website Fingerprinting Defenses with Deep Learning. In *CCS.*

[73] Jean-Pierre Smith, Luca Dolfi, Prateek Mittal, and Adrian Perrig. 2022. QCSD: A QUIC Client-Side Website-Fingerprinting Defence Framework. In *USENIX Security.*

[74] 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.*

[75] Yixin Sun, Anne Edmundson, Laurent Vanbever, Oscar Li, Jennifer Rexford, Mung Chiang, and Prateek Mittal. 2015. RAPTOR: Routing Attacks on Privacy in Tor. In *USENIX Security.*

[76] Paul Syverson, Rasmus Dahlberg, Tobias Pulls, and Rob Jansen. 2025. Onion-Location Measurements and Fingerprinting. *PoPETs.*

[77] Tor Project 2021. *Onion-Location.* https://community.torproject.org/onion-services/advanced/onion-location/ accessed 2025-05-29.

[78] Tor Project accessed 2025-05-13. *Tor specifications.* https://spec.torproject.org.

[79] Alexander Vaskevich, Thilini Dahanayaka, Guillaume Jourjon, and Suranga Seneviratne. 2021. Smaug: Streaming media augmentation using CGANs as a defence against video fingerprinting. In *Proc. IEEE NCA.*

[80] Ryan Wails, Rob Jansen, Aaron Johnson, and Micah Sherr. 2023. Proteus: Programmable Protocols for Censorship Circumvention. In *Free and Open Communications on the Internet.* https://www.petsymposium.org/foci/2023/foci-2023-0013.pdf

[81] Ryan Wails, Rob Jansen, Aaron Johnson, and Micah Sherr. 2025. Censorship Evasion with Unidentified Protocol Generation. In *USENIX Security.* https://www.usenix.org/conference/usenixsecurity25/presentation/wails

[82] Tao Wang and Ian Goldberg. 2016. On Realistically Attacking Tor with Website Fingerprinting. *PETS.*

[83] Ethan Witwer, James K. Holland, and Nicholas Hopper. 2022. Padding-only Defenses Add Delay in Tor. In *WPES.*

[84] Xiaokuan Zhang, Jihun Hamm, Michael K Reiter, and Yinqian Zhang. 2019. Statistical privacy for streaming traffic. In *Proc. NDSS.*

[85] Ye Zhu, Xinwen Fu, Bryan Graham, Riccardo Bettati, and Wei Zhao. 2004. On Flow Correlation Attacks and Countermeasures in Mix Networks. In *Privacy Enhancing Technologies, 4th International Workshop, PET 2004, Toronto, Canada, May 26-28, 2004, Revised Selected Papers.*

A Simulating and Aggregating Delays in the Maybenot Simulator

When parsing a base (undefended) trace for use in the Maybenot simulator, the simulator takes as an argument a network model. The network model consists of a round-trip time (RTT) and a packets-per-second (PPS) rate. The RTT is the simulated RTT between the client and server, while the PPS is the maximum number of packets per second that can be sent over the network.

During the parsing of a trace, the simulator creates event queues for both the client and the server. The client event queue consists of the normal *sent* events in the base trace since this is the ground truth for the simulator. The serve event queue, in turn, consists of sent events for the server to match the *recv* events at the client in the base trace. The timestamps in this queue are based on the RTT, with the server sending packets $\mathsf{RTT}/2$ before they arrive at the client, matching the base trace.

When building the event queues, we updated the simulator to also compute the maximum observed PPS in either direction of the base trace. This per-trace PPS rate is then used by the simulator in the bottleneck network model. We know, per definition, that there is some bottleneck between client and *destination* and that the packets in the base trace traversed this bottleneck. Here, we now take the worst-case scenario of modeling this bottleneck as being between the client and server. How common this scenario is, depends on where Maybenot is used. For HTTPS, the bottleneck is, by definition, somewhere between the client and the server. For VPNs, the bottleneck is likely between the client and the VPN server because VPN servers and destinations typically have excellent bandwidth. For Tor, the bottleneck is expected within the Tor network itself, and therefore, it depends on which relay is running defenses (e.g., guard, middle, or exit for general circuits).

There are two sources of *aggregating delays* during simulation in the simulator: *blocking actions* and *PPS above the bottleneck rate.* The PPS bottleneck can be disabled by setting a large PPS rate. This is what we do in the “infinite” bandwidth models. The aggregated delays will accumulate at the client and server, delaying all future traffic. A key observation for *when* aggregate delay should come into *effect* is that *packets already in flight cannot depend on packets not yet sent.* This is similarly discussed in the Tamaraw paper [9] in the context of enforcing causal ordering of packets and not violating packet dependencies for simulation. Likewise, we take a conservative approach to guarantee correctness but at the likely cost of overestimating delay overheads. For example, the result of aggregate delay due to blocking or exceeding the bottleneck PPS at the client will come into effect at the client in $2 \times \mathsf{RTT}$ and at the server in $1.5 \times \mathsf{RTT}$, when assuming the same delay between destination and server as between client and server.

When active blocking expires, the simulator considers the delta between the time of the block expiry and when the *tail* of any blocked queued *burst of packets* would have been sent without blocking. The burst window is one millisecond. We consider the tail of the burst, assuming that delaying prior packets would not allow the receiving application to respond any faster until the tail has arrived. In cases when the burst is large, the PPS bottleneck may lead to further delays.

When the PPS bottleneck is exceeded, the simulator triggers aggregate delays of $1/\mathsf{PPS}$ for the last packet sent within a one-millisecond window. On the one hand, the aggregate delay of $1/\mathsf{PPS}$ is excessive for lower PPS rates. On the other hand, by only triggering aggregate delays once per window, we underestimate the delay when a significant number of packets are sent (either due to rapid padding or long-lived blocking).

There is likely much to be done to improve further the networking side of the simulator, including resolving complex edge cases involving interactions between aggregate delays and blocking actions. For example, when running ten-fold cross-validation for the Video Fingerprinting results in Table 4, we observed that simulation seed 2 resulted in a significant spike in total average delay with ephemeral blocking defenses. To investigate, we ran simulations with seeds 0–99 (taking four days on an AMD EPYC 7713P 64-Core Processor). For 95 of the seeds, the delay was below 100%. The remaining five delays were 170%, 171%, 2855%, and 4919%, with 4919% from seed 2. Inspecting the resulting simulated traces, we found several outliers where the 10-minute video was simulated to take days to stream. We suspect this is due to a rare bug involving cascading aggregate delays. Further refinement on accurate but fast simulations is needed for ephemeral defenses.

B Random Maybenot Distributions

We distinguish between picking distributions for counts and durations. For counts, we consider the Uniform, Binomial, Geometric, Pareto, Poisson, and Weibull distributions. For timeouts, the Uniform, Normal, SkewNormal, LogNormal, Pareto, Poisson, Weibull, and Gamma distributions. Maybenot also supports the Beta distribution, but we found no clear use for it.

Distribution parameters are selected uniformly at random in relation to a reference point, specifically the count and duration reference points for their respective distributions. For reference point p where $\leftarrow$ denotes uniformly random sampling from an inclusive range, we sample floats, except for trials in the Binomial distribution. We often sample parameters starting from 0.001 instead of 0.0 to prevent extreme cases that may slow down sampling the distribution in practice (with the `rand_distr` crate[2]). While much can probably be improved here, for the ten distributions used, as a starting point, we selected to use:

Uniform (low, high) $l \leftarrow [0, p]$, $h \leftarrow [l, p]$.

Binomial (trials, probability) $t \leftarrow [10, \max(p, 11)]$, $p \leftarrow [0.001, 1.0]$. We enforce 10 trials to get spread in values.

Geometric (probability) $p \leftarrow [0.001, 1.0]$.

Gamma (scale, shape) scale $\leftarrow [0.001, p]$, shape $\leftarrow [0.001, 10.0]$. A shape of 10 already leads to a Normal-like distribution.

Pareto (scale, shape) scale $\leftarrow \max([p/100.0, p], 0.001)$, shape $\leftarrow [0.001, 10.0]$. The maximum shape 10 makes the distribution less tail-heavy.

Poisson (lambda) $l \leftarrow [0, p]$

Weibull (scale, shape) scale $\leftarrow [0.001, p]$, shape $\leftarrow [0.5, 5]$. The shape range captures a wide range of uses.

Normal (mean, stdev) $m \leftarrow [0, p]$, $s \leftarrow [0, p]$.

SkewNormal (location, scale, shape) location $\leftarrow [0.5p, 1.5p]$, scale $\leftarrow [p/100, p/10]$, shape $\leftarrow [-5, 5]$. The location is centered around p, the scale clustered around p (recall that p is often very large due to Maybenot operating in microseconds), and the shape range allows the distribution to take on asymmetric shapes.

LogNormal (mu, sigma) $mu \leftarrow [0, 20.0]$, $sigma \leftarrow [0, 1]$. The range for μ leads to values from 0 to $\approx 4.85 \times 10^8$. A small sigma leads to modest spread.

C Example: Ephemeral Padding-Only Defenses for Website Fingerprinting

To exemplify ephemeral defense search, we introduce the configuration and provide more detail by walking through the configuration for our ephemeral padding-only defenses for WF evaluated in Section 5. The defense search process from Section 3 is implemented in Rust (like Maybenot) and configured using TOML. The top table is for the search itself:

```toml
[search]
n = 1_000
seed = 0
```

The search table specifies the number of defenses to find and a seed for deterministic search. In this paper, we use seed 0 for all ephemeral defenses as a nothing-up-my-sleeve number throughout.

For each defense to derive, we specify the number of machines (at client and server) and the maximum number of attempts before sampling a new environment. The number of machines is a range, [1, 1], sampled uniformly random (inclusive). Ranges are used through the configurations. The start of the derive table:

```toml
[derive]
num_machines = [1,1]
max_attempts = 50
```

Deriving multiple machines per side is likely to result in only one sound machine. It is mainly useful in conjunction with fixed (hardcoded) machines (not shown here). The maximum attempts parameter is important. Typically, it is harder to find defenses in some areas of the search space in the configuration. On the one hand, setting a maximum number of attempts can then prevent the search from getting stuck. On the other hand, too few attempts result in defenses that only fulfill the constraints in the easier areas.

Next is the machine parameters:

```toml
[derive.machine]
num_states = [3,4]
duration_ref_point = [672_000, 950_000]
allow_blocking_client = false
allow_blocking_server = false
allow_expressive = false
allow_frac_limits = false
allow_fixed_budget = true
```

For each machine, we will sample uniformly randomly (inclusive) the number of states. We find little value in more states in machines while increasing them greatly increases the search space. One of the most important parameters is the duration reference point, `duration_ref_point`, that controls all durations in the state (Section 3.1). Here, we randomly select a point from 672 ms to 950 ms for each machine (Maybenot operates in microseconds). This creates a large difference between machines. Finally, we turn off blocking actions, expressive states, and machine (fractional) limits while allowing a fixed padding budget. Next is the environment:

```toml
[derive.env]
traces = ["DeepFingerprinting", "GongSurakav"]
num_traces = [15, 16]
rtt_in_ms = [50, 500]
packets_per_sec = [10_000, 20_000]
```

[2] https://crates.io/crates/rand_distr, accessed 2025-04-16.

```
sim_steps = [5_000, 5_000]
```

The list of string traces maps to hardcoded collections of traces (for efficiency). In this case, the traces consist of 14 traces from Gong et al.'s undefended Surakav dataset [21] and 14 traces from the DF [72] dataset. Each trace is from a distinct class and selected only on the basis of consisting of at least 1,000 cells. As the `num_traces` key suggests, we found little value in additional traces in our experimentation. As is, between 15–16 out of the 28 traces are selected at random for each environment. We also experimented with using traces from the BigEnough [43] dataset, but due to the intentionally coarse-grained timestamps (10 ms resolution, to prevent attackers from gaining an advantage), simulations resulted in defenses that simulated quite differently on datasets with accurate timestamps.

For the simulated network between client and server, we sample an RTT and PPS. The random RTT between 50–500 ms captures the variability of Tor circuits. We do not want defenses to get tailored to particular latencies. The simulated bottleneck of 10,000–20,000 packets per second is, in practice, equivalent to infinite bandwidth, as most traces are below 5,000 cells over several seconds of fetching a website over Tor. Finally, we limit the simulator to 5,000 steps. This is a conservative value in the sense that it allows most simulations to finish before the entire trace is processed (sending a normal packet consumes four steps in the simulator, with two at the sender and two at the receiver). The simulation steps greatly influence the search space; how depends on the other parameters. For example, computing constraints over a shorter number of steps constrain defenses to performing actions early. For defenses targeting handshakes, this is desirable, while it may be less so for other types of defenses where the tail of flows may be informative.

Finally, we express the constraints defenses should fulfill:

```
[derive.constraints]
client_load = [0.6, 2.6]
server_load = [0.36, 2.4]
delay = [0.0, 0.0]
client_min_normal_packets = 30
server_min_normal_packets = 100
```

We require the client load (fraction of additional bandwidth) to be between 0.6–2.6 and the server load between 0.36–2.4, turn off delay (if min and max are both 0.0), and require a minimum of 30 and 100 normal (non-padding) sent packets from the client and server, respectively. The minimum load, in particular, is important, which, in conjunction with the minimal number of normal packets (at the client, especially), filters out many machines. Note that these numbers are closely tied to the simulation steps set in the environment and the environment traces.

D Circuit Fingerprinting Tuning

Listing 1 shows the starting configuration for our CF defense search. The Git diff between starting and final configuration is in Listing 2.

Listing 1: Starting configuration.

```
[derive]
num_machines = [1,1]
max_attempts = 50

[derive.machine]
num_states = [3,4]
allow_blocking_client = true
allow_blocking_server = true
allow_fixed_budget = true
allow_expressive = false
# count_ref_point
# duration_ref_point
# min_action_timeout

[derive.env]
traces = ["TorCircuit"]
num_traces = [4, 14]
rtt_in_ms = [50, 500]
packets_per_sec = [10_000, 10_000]
sim_steps = [1_000, 1_000]

[derive.constraints]
client_load = [0.5, 10.0]
server_load = [0.5, 10.0]
delay = [0.5, 5.0]
client_min_normal_packets = 0
server_min_normal_packets = 0
include_after_last_normal = true

[search]
seed = 0
n = 1_000
max_duration_sec = 900

[sim]
base_dataset = "circuitfp-general-rend"
max_samples = 10_000
tunable_defense_limits = [1.0, 0.75, 0.5]
seed = 0

[sim.simulator.client]
padding_budget = [1000, 1000]
blocking_budget = [100_000, 200_000]
padding_frac = [0.9, 0.9]
blocking_frac = [0.9, 0.9]

[sim.simulator.server]
padding_budget = [1000, 1000]
blocking_budget = [100_000, 200_000]
padding_frac = [0.9, 0.9]
blocking_frac = [0.9, 0.9]

[sim.simulator]
rtt_in_ms = [50, 500]
packets_per_sec = [40_000, 40_000]
trace_length = 10_000
events_multiplier = 1_000
```

Listing 2: Git diff between starting and end configuration.

```
3c3
< max_attempts = 50
---
> max_attempts = 865
12c12
< # duration_ref_point
---
> duration_ref_point = [23_214, 796_961]
17c17
< num_traces = [4, 14]
---
> num_traces = [7, 11]
20c20
< sim_steps = [1_000, 1_000]
---
> sim_steps = [18_638, 47_942]
23,25c23,25
< client_load = [0.5, 10.0]
< server_load = [0.5, 10.0]
< delay = [0.5, 5.0]
---
> client_load = [2.78, 3.172]
> server_load = [7.051, 9.801]
> delay = [0.039, 1.794]
28c28
```

```
< include_after_last_normal = true
---
> include_after_last_normal = false
```

E Parameter Tuning of WF Defenses

We implemented some WF defenses in Maybenot and tuned their respective parameters to the BigEnough dataset [43]. We repeated the tuning with and without a simulated network bottleneck between the client and server. Parameters were generally selected to minimize overhead, with delay overhead being weighed twice as much as bandwidth overhead. Table 5 contains the final hyperparameters. We used the first (out of five) folds for tuning for those defenses where parameter selection has significant implications for defense efficacy. For defense efficacy, we used DF and RF with default hyperparameters and patience 10 for early stopping. Additionally, DF was modified to use up to 10,000 cells (from its original 5,000). RF parsed the first 80 s of each trace (the average undefended trace is 28.1 s in BigEnough standard).

The simulated RTT between client and server (middle relay, not to be mistaken for the destination website) were uniformly randomly selected per trace between $[50, 500]$ ms. All defenses showed similar results for a fixed RTT of 250 ms on the final parameters. The randomized defenses (Break-Pad, FRONT, and Interspace) simulated up to 40,0000 cells. In comparison, the fixed-rate defenses needed 200,000 cells to ensure that all normal (non-padding) cells from the original traces were included in the defended traces.

E.1 Break-Pad and Interspace

Both Break-Pad [31] and Interspace [57] target Tor's Circuit Padding Framework [54, 55], so implementation in Maybenot is straightforward. Break-Pad has no parameters to tune. Interspace has no parameters to tune, but several parameters are randomized on *instantiation* of the defense. We, therefore, created 100,000 instances of the defense, and for each trace in BigEnough (19,000 with no augmentation), we randomly selected an instance.

E.2 FRONT

We use the implementation of FRONT [20] from Maybenot [59]. FRONT is randomized like Interspace, so we created 100,000 instances to randomly select from per trace. For the parameters, we did a grid search of $W_{min} \in \{1, 5, 10\}$, $W_{max} \in \{12, 15, 20\}$, $N_S = N_C \in \{1500, 1700, 2000, 2500, 3000, 6000\}$, and the number of states to approximate the Rayleigh distribution $S = \{1, 10, 100\}$. For FRONT with infinite bandwidth, we found bandwidth overheads 0.18–0.75, selecting parameters in the higher range that gave the best defense against DF (no parameter selection got RF below 0.9). With a simulated bottleneck, we opted for the parameters with the lowest overhead (incidentally, the lowest delay overhead).

E.3 Tamaraw

For Tamaraw [9], we did a grid search $P_C \in [0.005, 0.03]$ in 0.005 increments, $P_S \in [0.0015, 0.0055]$ in 0.005 increments, using default $L = 100$ (clear trade-off parameter, no point in tuning), and $W \in \{2, 4, 6\}$ seconds. The stop window W is from Gong et al.'s soft stop condition for real-world implementations of WF defenses [22].

E.4 RegulaTor

We implemented a version of RegulaTor [30] in Maybenot. Exact implementation is not possible due to state machine restrictions. This has been done in related work [25, 59, 60]. On the client, ensuring that packets are never queued for more than c seconds requires us to flush all outgoing queued packets instead of just the one delayed for c seconds. For most website datasets, this difference should be negligible due to light upload traffic. On the server, the main shortcomings of our implementation are discretizing the decaying send rate into bins and the inability to send only individually queued packets once the padding budget is consumed. As for the client, we flush all queued packets according to the packet rate.

E.4.1 Infinite Bandwidth. We started hyperparameter tuning with infinite bandwidth to find parameters resulting in approximately 92% load and 7% delay—the same overhead as for the tuned RegulaTor on BigEnough in the Laserbeak paper [42]. Because of the excessive search space, we did the tuning in phases, using RegulaTor heavy parameters as a starting point.

(1) Searched the number of bins for the server-side, $b \in \{10, 50, 100, 1000\}$. Too fined-grained binning may lead to delayed surges. Anticipating challenges around keeping delay acceptable, we opted for a low bin count of $b = 10$.
(2) Grid searched $u \in \{3.53.954.0, 4.5\}$ and $c \in \{0.5, 1.0, 1.5, 1.77, 2.0\}$ for the client. We saw the lowest delay with $u = 4.5$ and $c = 1.0$.
(3) Motivated by relatively low load and high delays, we grid searched $r \in \{200, 277, 300, 400, 600, 800, 1000, 1200\}$ and $n \in \{2000, 3000, 3550, 4000, 5000, 6000\}$. For $r \leq 400$ we note high delays, while $r \geq 600$ allowed for a range of trade-offs in load and delay based on varying n. We therefore fixed $r = 600$.
(4) Varied n as in phase three, but this time also running RF and DF. We observe a clear relationship between padding budget and attacker success. We set $n = 5000$ because the load is still below our target, and we need to reduce delay.
(5) Grid searched $d \in \{0.9, 0.92, 0.94, 0.96, 0.98\}$ and $t \in \{1.0, 1.5, 2.0, 2.5, 3.0, 3.55, 4.0\}$. Delay is minimized with $d = 0.98$ and $t = 2.5$ with little difference in load.
(6) To attempt to get delay further down, grid searched $c \in \{0.5, 1.0\}$, $r \in \{600, 700\}$, and $n\ in\{5000, 6000\}$. Delay is minimized with $c = 0.5$, $r = 700$, and $n = 5000$.
(7) We search $r \in \{700, 800, 900, 1000, 1100, 1200\}$. With $r = 1200$ we hit our target 7% delay. Load is 83%.
(8) We search $n \in [5000, 6000]$ in 100 increments. With $n = 5600$ we have load 93% and delay 8%.

The final parameters for infinite RegulaTor are: $u = 4.5$, $c = 0.5$, $r = 1200$, $d = 0.98$, $t = 2.5$, $n = 5600$ with $b = 10$.

E.4.2 Network Bottleneck. We use the final parameters for infinite RegulaTor as a starting point. Recall that the difference between infinite and bottleneck is that each trace will now have a PPS bottleneck equal to the PPS needed for each trace.

(1) Baseline gives 93% load and 1938% delay. Because the PPS in the bottleneck is symmetric and sent at the client is a subset of sent from the server, we can rule out client-side parameters. The main parameter that controls the peak PPS from the server is the rate r.

Table 5: Final defense hyperparameters tuned for BigEnough with infinite and ▽bottleneck network models.

	Defense	Parameters
Padding-only	Break-Pad	—
	Break-Pad▽	—
	Ephemeral padding-only	Listing 3
	Ephemeral padding-only▽	Listing 4
	FRONT	$N_S = N_C = 6000$, $W_{min} = 5$, $W_{max} = 12$, $S = 1$
	FRONT▽	$N_S = N_C = 1500$, $W_{min} = 5$, $W_{max} = 15$, $S = 10$
	Interspace	—
	Interspace▽	—
Blocking	Ephemeral blocking	Listing 5
	Ephemeral blocking ▽	Listing 6
	RegulaTor	$U = 4.5$, $C = 0.5$, $R = 1200$, $D = 0.98$, $T = 2.5$, $N = 5600$, $B = 10$
	RegulaTor▽	$U = 4.5$, $C = 0.5$, $R = 220$, $D = 0.98$, $T = 2.5$, $N = 2000$, $B = 10$
	Tamaraw	$P_C = 0.01$, $P_S = 0.005$, $L = 100$, $W = 2$
	Tamaraw▽	$P_C = 0.01$, $P_S = 0.0055$, $L = 100$, $W = 2$

(2) We set $n = 10$ to capture the case when the rate is irrelevant (because the server sends no padding). We get 5% load and 2% delay.

(3) We search $r \in [100, 1200]$ in 100 increments. Interestingly, $r = 100$ gives 1930% delay, $r = 200$ 277% delay, and $r = 300$ 303% delay. This suggests some minima where a too-low rate leads to cascading aggregate delays and a too-high rate hits too many PPS bottlenecks.

(4) We search $r \in [100, 300]$ in 10 increments. At $r = 220$, we get 267% delay and 88% load.

(5) We search $n \in [500, 5000]$ in 500 increments and run DF and RF. At $n = 2000$, we get 138% delay, between Tamaraw and Ephemeral blocking bottleneck delays.

In gist, RegulaTor gets stuck between having a surge spike too high—hitting the PPS bottleneck—and sending traffic too slow, leading to delays due to excessive blocking. Part of this might be shortcomings in our port of RegulaTor to Maybenot. Ideally, one would want a mechanism to set the rate based on the observed bottleneck dynamically. That related work implementations did not encounter this is likely due to being implemented as Pluggable Transports (PTs) [22, 30]. The PT endpoints were run outside the Tor network with the PT server as a bridge. In this scenario, the network bottleneck between client and destination is likely somewhere within the Tor network and not between PT client and server, doubly so in experimental settings with no shared use of the PT server.

E.5 Ephemeral Defenses

The ephemeral defenses for WF were tuned in the same manner as the CF defenses in Section 4. The process was less structured as it took place in parallel with the development of the tooling for ephemeral search and tuning presented in this paper. We present the configurations for the final defenses below. They also use height 2 like the CF defenses.

Listing 3 shows the final configuration for the ephemeral padding-only defenses with the infinite network model. Listing 4 shows the diff between the infinite and bottleneck network models for padding-only defenses. Listing 5 shows the final configuration for the ephemeral blocking defenses with the infinite network model. Listing 6 shows the diff between the infinite and bottleneck network models for blocking defenses.

Listing 3: Ephemeral padding-only infinite network model.

```
[derive]
num_machines = [1,1]
max_attempts = 50

[derive.machine]
num_states = [3,4]
duration_ref_point = [672_000, 950_000]
allow_blocking_client = false
allow_blocking_server = false
allow_expressive = false
allow_frac_limits = false
allow_fixed_budget = true

[derive.env]
traces = ["DeepFingerprinting", "GongSurakav"]
num_traces = [15, 16]
rtt_in_ms = [50, 500]
packets_per_sec = [10_000, 20_000]
sim_steps = [5_000, 5_000]

[derive.constraints]
client_load = [0.6, 2.6]
server_load = [0.36, 2.4]
delay = [0.0, 0.0]
client_min_normal_packets = 30
server_min_normal_packets = 100

[search]
seed = 0
n = 1000

[combo]
n = 100_000
height = 2
max_attempts = 50

[combo.env]
traces = ["DeepFingerprinting", "GongSurakav"]
num_traces = [15, 16]
rtt_in_ms = [50, 500]
packets_per_sec = [10_000, 20_000]
sim_steps = [5_000, 5_000]

[combo.constraints]
```

```
client_load = [0.6, 2.6]
server_load = [0.36, 2.4]
delay = [0.0, 0.0]
client_min_normal_packets = 30
server_min_normal_packets = 100

[sim]
base_dataset = "bigenough-95x10x20-standard-rngsubpages/"
max_samples = 100
tunable_defense_limits = [0.75]
seed = 0

[sim.simulator]
rtt_in_ms = [50, 500]
packets_per_sec = [40_000, 40_000]
trace_length = 60_000
events_multiplier = 1_000

[sim.simulator.client]
padding_budget = [1_000, 2_000]
blocking_budget = [0, 0]
padding_frac = [0.95, 0.95]
blocking_frac = [0, 0]

[sim.simulator.server]
padding_budget = [1_000, 3_000]
blocking_budget = [0, 0]
padding_frac = [0.95, 0.95]
blocking_frac = [0, 0]
```

Listing 4: Git diff padding-only infinite and bottleneck network model.

```
3c3
< max_attempts = 50
---
> max_attempts = 769
7c7,9
< duration_ref_point = [672_000, 950_000]
---
> duration_ref_point = [247_000, 320_000]
> min_action_timeout = [196, 396]
> count_ref_point = [254, 469]
16c18
< num_traces = [15, 16]
---
> num_traces = [12, 17]
18c20
< packets_per_sec = [10_000, 20_000]
---
> #packets_per_sec = [10_000, 20_000]
22,23c24,25
< client_load = [0.6, 2.6]
< server_load = [0.36, 2.4]
---
> client_load = [2.2, 2.8]
> server_load = [1.0, 2.5]
25c27
< client_min_normal_packets = 30
---
> client_min_normal_packets = 33
35c37
< max_attempts = 50
---
> max_attempts = 769
39c41
< num_traces = [15, 16]
---
> num_traces = [12, 17]
41c43
< packets_per_sec = [10_000, 20_000]
---
> #packets_per_sec = [10_000, 20_000]
45,46c47,48
< client_load = [0.6, 2.6]
< server_load = [0.36, 2.4]
---
> client_load = [2.2, 2.8]
> server_load = [1.0, 2.5]
48c50
< client_min_normal_packets = 30
---
> client_min_normal_packets = 33
54c56
< tunable_defense_limits = [0.75]
---
> tunable_defense_limits = [0.5]
59c61
< packets_per_sec = [40_000, 40_000]
---
> #packets_per_sec = [40_000, 40_000]
73a76
>
```

Listing 5: Ephemeral blocking infinite network model.

```
[derive]
num_machines = [1,1]
max_attempts = 711

[derive.machine]
num_states = [3,4]
duration_ref_point = [290_000, 966_000]
allow_blocking_client = true
allow_blocking_server = true
allow_expressive = false
allow_frac_limits = false
allow_fixed_budget = true

[derive.env]
traces = ["DeepFingerprinting", "GongSurakav"]
num_traces = [9, 11]
rtt_in_ms = [50, 500]
packets_per_sec = [10_000, 20_000]
sim_steps = [5_000, 5_000]

[derive.constraints]
client_load = [1.5, 7.5]
server_load = [0.9, 7.4]
delay = [0.5, 5.0]
client_min_normal_packets = 30
server_min_normal_packets = 100
include_after_last_normal = true

[search]
n = 1_000
seed = 0

[combo]
n = 100_000
height = 2
max_attempts = 711

[combo.env]
traces = ["DeepFingerprinting", "GongSurakav"]
num_traces = [9, 11]
rtt_in_ms = [50, 500]
packets_per_sec = [10_000, 20_000]
sim_steps = [5_000, 5_000]

[combo.constraints]
client_load = [3.0, 4.0]
server_load = [1.0, 2.0]
delay = [0.5, 5.0]
client_min_normal_packets = 30
server_min_normal_packets = 100
include_after_last_normal = true

[sim]
base_dataset = "bigenough-95x10x20-standard-rngsubpages/"
max_samples = 100
tunable_defense_limits = [0.75]
seed = 0

```

```
[sim.simulator]
rtt_in_ms = [50, 500]
packets_per_sec = [40_000, 40_000]
trace_length = 60_000
events_multiplier = 1_000

[sim.simulator.client]
padding_budget = [1_000, 2_000]
blocking_budget = [100_000, 200_000]
padding_frac = [0.95, 0.95]
blocking_frac = [0.5, 0.5]

[sim.simulator.server]
padding_budget = [1_000, 3_000]
blocking_budget = [100_000, 200_000]
padding_frac = [0.95, 0.95]
blocking_frac = [0.5, 0.5]
```

Listing 6: Git diff blocking infinite and bottleneck network model.

```
3c3
< max_attempts = 711
---
> max_attempts = 900
6c6
< num_states = [3,4]
---
> num_states = [3,3]
18c18
< packets_per_sec = [10_000, 20_000]
---
> #packets_per_sec = [10_000, 20_000]
36c36
< max_attempts = 711
---
> max_attempts = 900
42c42
< packets_per_sec = [10_000, 20_000]
---
> #packets_per_sec = [10_000, 20_000]
46,47c46,47
< client_load = [3.0, 4.0]
< server_load = [1.0, 2.0]
---
> client_load = [1.5, 7.5]
> server_load = [0.9, 7.4]
61c61
< packets_per_sec = [40_000, 40_000]
---
> #packets_per_sec = [40_000, 40_000]
```

F Model Training Times

Table 6 shows the running time in the infinite training scenario from Section 5.3. In total 634 hours—almost a month of walltime. While examining the table, it is important to note that the defense simulation load plays a role here. That is, runtime comparison between models can only be interpreted while keeping a defense fixed, as each defense introduces a different overhead on the overall runtime due to simulation time.

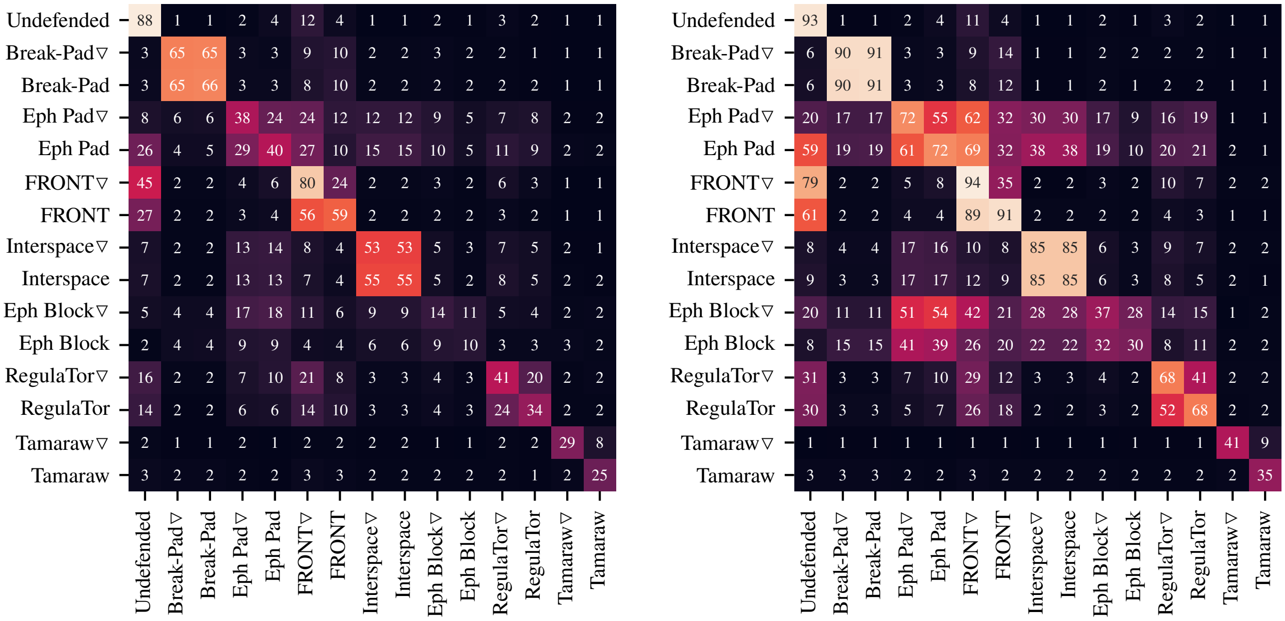
For the 30 epoch scenario, we have training times: DF 3.6 min, DF-multi 4.5 min, Laserbeak 32 min, Laserbeak⁻ 16 min, RF 3.2 min. In both cases, 30 epochs and infinite training, a fair comparison of the training times becomes hindered by Laserbeak's large vRAM usage, leading to those runs having separate hardware compared to others (DF, df-multi, Laserbeak⁻, RF). Laserbeak was evaluated using an AMD EPYC 7R13 and an NVIDIA L40S (AWS instance type g6e.4xlarge), while the others used an AMD Ryzen 9 7950X and an NVIDIA RTX 4070 Ti.

G Cross Attack/Defense Heatmaps

For the cross attack/defense heatmaps in Section 5.4, Figure 7 shows the heatmap for DF [72], Figure 8 for RF [70], Figure 9 for DF-multi [42], and finally Figure 10 for Laserbeak without attention [42].

Table 6: Running time per xv fold during infinite training.

	BigEnough	Training time (h)				
		DF	DF-multi	Laserbeak	Laserbeak$^-$	RF
	Undefended	$0.3^{\pm0.0}$	$0.8^{\pm0.2}$	$5.5^{\pm0.1}$	$2.7^{\pm0.3}$	$0.5^{\pm0.1}$
Padding-only	Break-Pad▽	$0.9^{\pm0.1}$	$0.6^{\pm0.2}$	$7.1^{\pm0.8}$	$3.9^{\pm0.3}$	$1.0^{\pm0.1}$
	Break-Pad	$0.8^{\pm0.1}$	$1.2^{\pm0.2}$	$7.7^{\pm1.7}$	$3.8^{\pm0.4}$	$0.8^{\pm0.1}$
	Ephemeral Pad▽	$1.7^{\pm0.2}$	$0.5^{\pm0.2}$	$5.6^{\pm0.4}$	$2.1^{\pm0.9}$	$1.8^{\pm0.2}$
	Ephemeral Pad	$2.3^{\pm0.3}$	$1.9^{\pm0.4}$	$6.7^{\pm0.1}$	$3.6^{\pm0.0}$	$2.3^{\pm0.2}$
	FRONT▽	$0.4^{\pm0.1}$	$0.7^{\pm0.1}$	$5.1^{\pm0.5}$	$2.3^{\pm0.6}$	$0.7^{\pm0.1}$
	FRONT	$0.5^{\pm0.1}$	$0.9^{\pm0.1}$	$5.9^{\pm0.4}$	$2.9^{\pm0.3}$	$0.6^{\pm0.1}$
	Interspace▽	$1.2^{\pm0.2}$	$0.9^{\pm0.1}$	$5.8^{\pm3.5}$	$2.3^{\pm0.1}$	$0.8^{\pm0.2}$
	Interspace	$1.2^{\pm0.1}$	$1.2^{\pm0.1}$	$8.3^{\pm0.1}$	$4.0^{\pm0.2}$	$0.8^{\pm0.1}$
Blocking	Ephemeral Block▽	$2.2^{\pm0.2}$	$0.5^{\pm0.2}$	$5.9^{\pm1.0}$	$1.1^{\pm0.0}$	$2.2^{\pm0.5}$
	Ephemeral Block	$2.8^{\pm0.4}$	$2.1^{\pm0.4}$	$8.8^{\pm1.7}$	$4.8^{\pm0.7}$	$1.9^{\pm0.3}$
	RegulaTor▽	$1.1^{\pm0.1}$	$1.8^{\pm0.2}$	$7.1^{\pm1.3}$	$2.3^{\pm0.0}$	$1.7^{\pm0.2}$
	RegulaTor	$1.8^{\pm0.2}$	$2.6^{\pm0.3}$	$10.1^{\pm3.4}$	$3.9^{\pm0.0}$	$3.2^{\pm0.6}$
	Tamaraw▽	$0.6^{\pm0.1}$	$0.7^{\pm0.1}$	$5.2^{\pm0.1}$	$1.8^{\pm0.1}$	$1.7^{\pm0.4}$
	Tamaraw	$0.5^{\pm0.0}$	$1.0^{\pm0.1}$	$4.4^{\pm0.2}$	$1.6^{\pm0.2}$	$1.8^{\pm0.3}$



	Undefended	Break-Pad▽	Break-Pad	Eph Pad▽	Eph Pad	FRONT▽	FRONT	Interspace▽	Interspace	Eph Block▽	Eph Block	RegulaTor▽	RegulaTor	Tamaraw▽	Tamaraw
Undefended	88	1	1	2	4	12	4	1	1	2	1	3	2	1	1
Break-Pad▽	3	65	65	3	3	9	10	2	2	3	2	2	1	1	1
Break-Pad	3	65	66	3	3	8	10	2	2	2	2	2	2	1	1
Eph Pad▽	8	6	6	38	24	24	12	12	12	9	5	7	8	2	2
Eph Pad	26	4	5	29	40	27	10	15	15	10	5	11	9	2	2
FRONT▽	45	2	2	4	6	80	24	2	2	3	2	6	3	1	1
FRONT	27	2	2	3	4	56	59	2	2	2	2	3	2	1	1
Interspace▽	7	2	2	13	14	8	4	53	53	5	3	7	5	2	1
Interspace	7	2	2	13	13	7	4	55	55	5	2	8	5	2	2
Eph Block▽	5	4	4	17	18	11	6	9	9	14	11	5	4	2	2
Eph Block	2	4	4	9	9	4	4	6	6	9	10	3	3	3	2
RegulaTor▽	16	2	2	7	10	21	8	3	3	4	3	41	20	2	2
RegulaTor	14	2	2	6	6	14	10	3	3	4	3	24	34	2	2
Tamaraw▽	2	1	1	2	1	2	2	2	2	1	1	2	2	29	8
Tamaraw	3	2	2	2	2	3	3	2	2	2	2	2	1	2	25

	Undefended	Break-Pad▽	Break-Pad	Eph Pad▽	Eph Pad	FRONT▽	FRONT	Interspace▽	Interspace	Eph Block▽	Eph Block	RegulaTor▽	RegulaTor	Tamaraw▽	Tamaraw
Undefended	93	1	1	2	4	11	4	1	1	2	1	3	2	1	1
Break-Pad▽	6	90	91	3	3	9	14	1	1	2	2	2	2	1	1
Break-Pad	6	90	91	3	3	8	12	1	1	2	1	2	2	1	1
Eph Pad▽	20	17	17	72	55	62	32	30	30	17	9	16	19	1	1
Eph Pad	59	19	19	61	72	69	32	38	38	19	10	20	21	2	1
FRONT▽	79	2	2	5	8	94	35	2	2	3	2	10	7	2	2
FRONT	61	2	2	4	4	89	91	2	2	2	2	4	3	1	1
Interspace▽	8	4	4	17	16	10	8	85	85	6	3	9	7	2	2
Interspace	9	3	3	17	17	12	9	85	85	6	3	8	5	2	1
Eph Block▽	20	11	11	51	54	42	21	28	28	37	28	14	15	1	2
Eph Block	8	15	15	41	39	26	20	22	22	32	30	8	11	2	2
RegulaTor▽	31	3	3	7	10	29	12	3	3	4	2	68	41	2	2
RegulaTor	30	3	3	5	7	26	18	2	2	3	2	52	68	2	2
Tamaraw▽	1	1	1	1	1	1	1	1	1	1	1	1	1	41	9
Tamaraw	3	3	3	2	2	3	2	2	2	2	2	2	2	2	35

Figure 7: Accuracy on the BigEnough standard dataset [43] in a closed world for *Deep Fingerprinting* [72] trained on the defense given in row when tested on the defense given in column. Left for the 30 epochs training, right for infinite training.

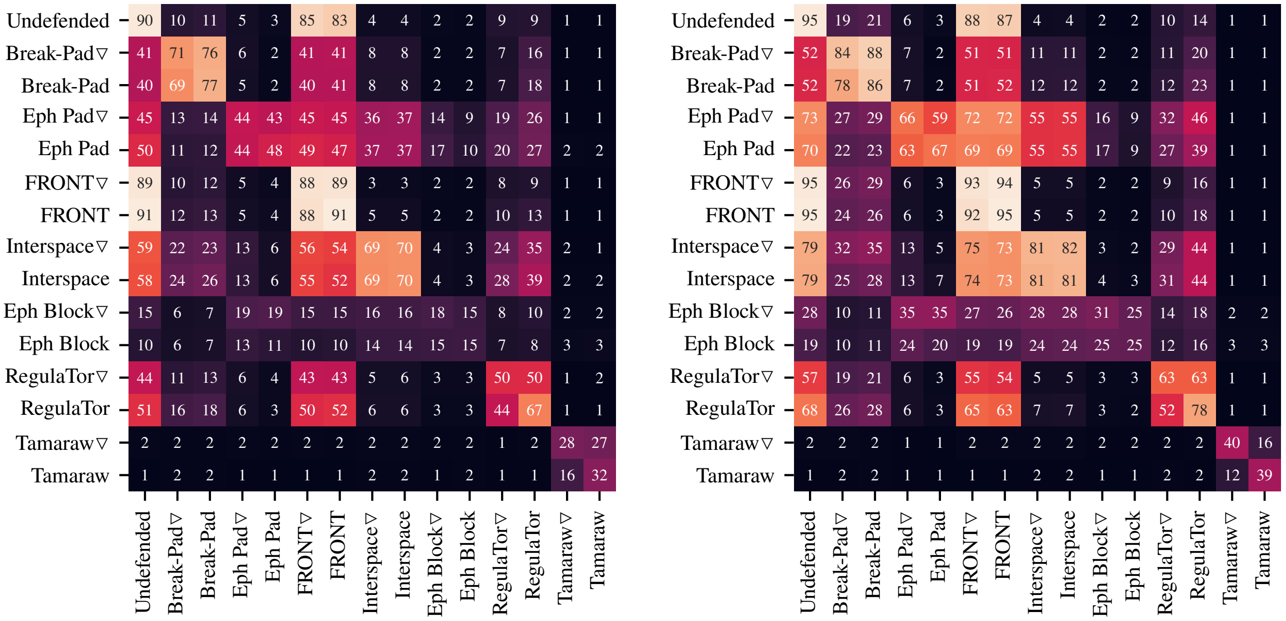


Figure 8: Accuracy on the BigEnough standard dataset [43] in a closed world for *Robust Fingerprinting* [70] trained on the defense given in row when tested on the defense given in column. Left for the 30 epochs training, right for infinite training.

	Undefended	Break-Pad▽	Break-Pad	Eph Pad▽	Eph Pad	FRONT▽	FRONT	Interspace▽	Interspace	Eph Block▽	Eph Block	RegulaTor▽	RegulaTor	Tamaraw▽	Tamaraw
Undefended	90	10	11	5	3	85	83	4	4	2	2	9	9	1	1
Break-Pad▽	41	71	76	6	2	41	41	8	8	2	2	7	16	1	1
Break-Pad	40	69	77	5	2	40	41	8	8	2	2	7	18	1	1
Eph Pad▽	45	13	14	44	43	45	45	36	37	14	9	19	26	1	1
Eph Pad	50	11	12	44	48	49	47	37	37	17	10	20	27	2	2
FRONT▽	89	10	12	5	4	88	89	3	3	2	2	8	9	1	1
FRONT	91	12	13	5	4	88	91	5	5	2	2	10	13	1	1
Interspace▽	59	22	23	13	6	56	54	69	70	4	3	24	35	2	1
Interspace	58	24	26	13	6	55	52	69	70	4	3	28	39	2	2
Eph Block▽	15	6	7	19	19	15	15	16	16	18	15	8	10	2	2
Eph Block	10	6	7	13	11	10	10	14	14	15	15	7	8	3	3
RegulaTor▽	44	11	13	6	4	43	43	5	6	3	3	50	50	1	2
RegulaTor	51	16	18	6	3	50	52	6	6	3	3	44	67	1	1
Tamaraw▽	2	2	2	2	2	2	2	2	2	2	2	1	2	28	27
Tamaraw	1	2	2	1	1	1	1	2	2	1	2	1	1	16	32

	Undefended	Break-Pad▽	Break-Pad	Eph Pad▽	Eph Pad	FRONT▽	FRONT	Interspace▽	Interspace	Eph Block▽	Eph Block	RegulaTor▽	RegulaTor	Tamaraw▽	Tamaraw
Undefended	95	19	21	6	3	88	87	4	4	2	2	10	14	1	1
Break-Pad▽	52	84	88	7	2	51	51	11	11	2	2	11	20	1	1
Break-Pad	52	78	86	7	2	51	52	12	12	2	2	12	23	1	1
Eph Pad▽	73	27	29	66	59	72	72	55	55	16	9	32	46	1	1
Eph Pad	70	22	23	63	67	69	69	55	55	17	9	27	39	1	1
FRONT▽	95	26	29	6	3	93	94	5	5	2	2	9	16	1	1
FRONT	95	24	26	6	3	92	95	5	5	2	2	10	18	1	1
Interspace▽	79	32	35	13	5	75	73	81	82	3	2	29	44	1	1
Interspace	79	25	28	13	7	74	73	81	81	4	3	31	44	1	1
Eph Block▽	28	10	11	35	35	27	26	28	28	31	25	14	18	2	2
Eph Block	19	10	11	24	20	19	19	24	24	25	25	12	16	3	3
RegulaTor▽	57	19	21	6	3	55	54	5	5	3	3	63	63	1	1
RegulaTor	68	26	28	6	3	65	63	7	7	3	2	52	78	1	1
Tamaraw▽	2	2	2	1	1	2	2	2	2	2	2	2	2	40	16
Tamaraw	1	2	2	1	1	1	1	2	2	1	1	2	2	12	39

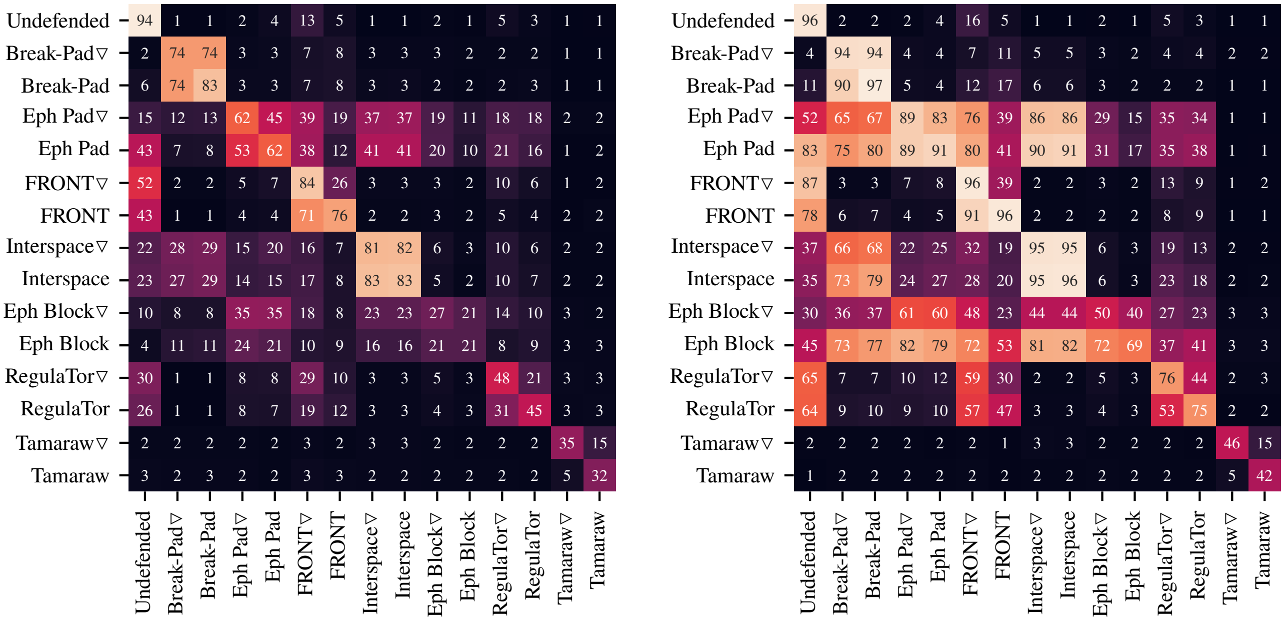
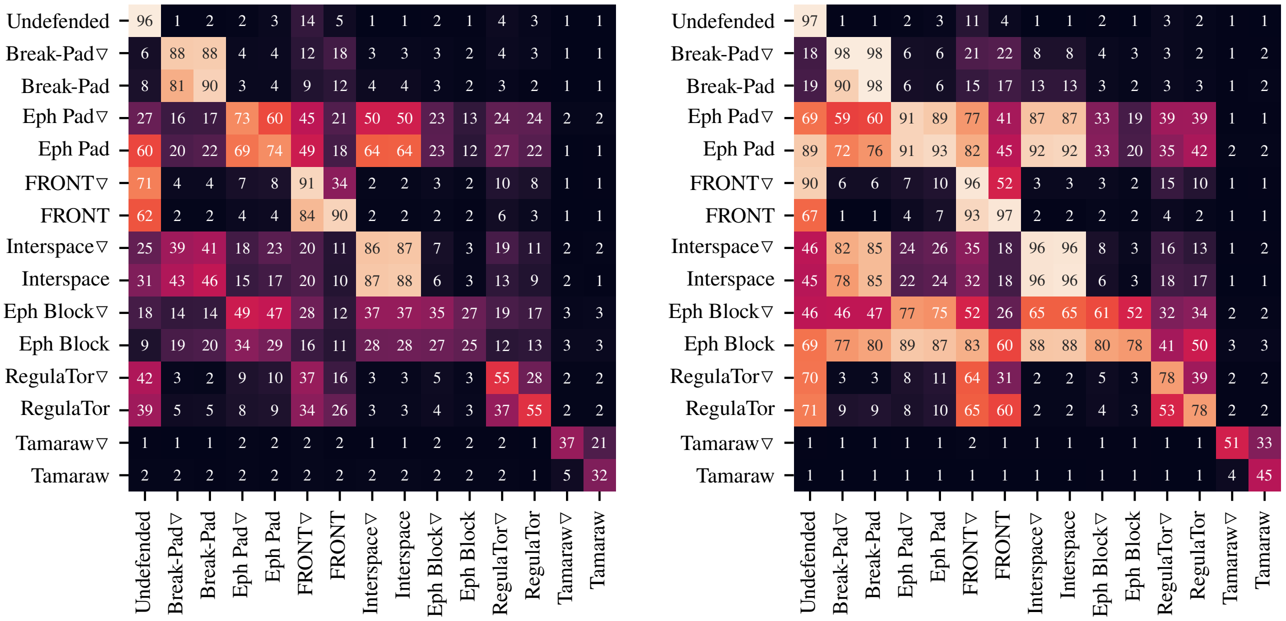


Figure 9: Accuracy on the BigEnough standard dataset [43] in a closed world for *DF-multi* [42] trained on the defense given in row when tested on the defense given in column. Left for the 30 epochs training, right for infinite training.

	Undefended	Break-Pad▽	Break-Pad	Eph Pad▽	Eph Pad	FRONT▽	FRONT	Interspace▽	Interspace	Eph Block▽	Eph Block	RegulaTor▽	RegulaTor	Tamaraw▽	Tamaraw
Undefended	94	1	1	2	4	13	5	1	1	2	1	5	3	1	1
Break-Pad▽	2	74	74	3	3	7	8	3	3	3	2	2	2	1	1
Break-Pad	6	74	83	3	3	7	8	3	3	2	2	2	3	1	1
Eph Pad▽	15	12	13	62	45	39	19	37	37	19	11	18	18	2	2
Eph Pad	43	7	8	53	62	38	12	41	41	20	10	21	16	1	2
FRONT▽	52	2	2	5	7	84	26	3	3	3	2	10	6	1	2
FRONT	43	1	1	4	4	71	76	2	2	3	2	5	4	2	2
Interspace▽	22	28	29	15	20	16	7	81	82	6	3	10	6	2	2
Interspace	23	27	29	14	15	17	8	83	83	5	2	10	7	2	2
Eph Block▽	10	8	8	35	35	18	8	23	23	27	21	14	10	3	2
Eph Block	4	11	11	24	21	10	9	16	16	21	21	8	9	3	3
RegulaTor▽	30	1	1	8	8	29	10	3	3	5	3	48	21	3	3
RegulaTor	26	1	1	8	7	19	12	3	3	4	3	31	45	3	3
Tamaraw▽	2	2	2	2	2	3	2	3	3	2	2	2	2	35	15
Tamaraw	3	2	3	2	2	3	3	2	2	2	2	2	2	5	32

	Undefended	Break-Pad▽	Break-Pad	Eph Pad▽	Eph Pad	FRONT▽	FRONT	Interspace▽	Interspace	Eph Block▽	Eph Block	RegulaTor▽	RegulaTor	Tamaraw▽	Tamaraw
Undefended	96	2	2	2	4	16	5	1	1	2	1	5	3	1	1
Break-Pad▽	4	94	94	4	4	7	11	5	5	3	2	4	4	2	2
Break-Pad	11	90	97	5	4	12	17	6	6	3	2	2	2	1	1
Eph Pad▽	52	65	67	89	83	76	39	86	86	29	15	35	34	1	1
Eph Pad	83	75	80	89	91	80	41	90	91	31	17	35	38	1	1
FRONT▽	87	3	3	7	8	96	39	2	2	3	2	13	9	1	2
FRONT	78	6	7	4	5	91	96	2	2	2	2	8	9	1	1
Interspace▽	37	66	68	22	25	32	19	95	95	6	3	19	13	2	2
Interspace	35	73	79	24	27	28	20	95	96	6	3	23	18	2	2
Eph Block▽	30	36	37	61	60	48	23	44	44	50	40	27	23	3	3
Eph Block	45	73	77	82	79	72	53	81	82	72	69	37	41	3	3
RegulaTor▽	65	7	7	10	12	59	30	2	2	5	3	76	44	2	3
RegulaTor	64	9	10	9	10	57	47	3	3	4	3	53	75	2	2
Tamaraw▽	2	2	2	2	2	2	1	3	3	2	2	2	2	46	15
Tamaraw	1	2	2	2	2	2	2	2	2	2	2	2	2	5	42



	Undefended	Break-Pad▽	Break-Pad	Eph Pad▽	Eph Pad	FRONT▽	FRONT	Interspace▽	Interspace	Eph Block▽	Eph Block	RegulaTor▽	RegulaTor	Tamaraw▽	Tamaraw
Undefended	96	1	2	2	3	14	5	1	1	2	1	4	3	1	1
Break-Pad▽	6	88	88	4	4	12	18	3	3	3	2	4	3	1	1
Break-Pad	8	81	90	3	4	9	12	4	4	3	2	3	2	1	1
Eph Pad▽	27	16	17	73	60	45	21	50	50	23	13	24	24	2	2
Eph Pad	60	20	22	69	74	49	18	64	64	23	12	27	22	1	1
FRONT▽	71	4	4	7	8	91	34	2	2	3	2	10	8	1	1
FRONT	62	2	2	4	4	84	90	2	2	2	2	6	3	1	1
Interspace▽	25	39	41	18	23	20	11	86	87	7	3	19	11	2	2
Interspace	31	43	46	15	17	20	10	87	88	6	3	13	9	2	1
Eph Block▽	18	14	14	49	47	28	12	37	37	35	27	19	17	3	3
Eph Block	9	19	20	34	29	16	11	28	28	27	25	12	13	3	3
RegulaTor▽	42	3	2	9	10	37	16	3	3	5	3	55	28	2	2
RegulaTor	39	5	5	8	9	34	26	3	3	4	3	37	55	2	2
Tamaraw▽	1	1	1	2	2	2	2	1	1	2	2	2	1	37	21
Tamaraw	2	2	2	2	2	2	2	2	2	2	2	2	1	5	32

	Undefended	Break-Pad▽	Break-Pad	Eph Pad▽	Eph Pad	FRONT▽	FRONT	Interspace▽	Interspace	Eph Block▽	Eph Block	RegulaTor▽	RegulaTor	Tamaraw▽	Tamaraw
Undefended	97	1	1	2	3	11	4	1	1	2	1	3	2	1	1
Break-Pad▽	18	98	98	6	6	21	22	8	8	4	3	3	2	1	2
Break-Pad	19	90	98	6	6	15	17	13	13	3	2	3	3	1	2
Eph Pad▽	69	59	60	91	89	77	41	87	87	33	19	39	39	1	1
Eph Pad	89	72	76	91	93	82	45	92	92	33	20	35	42	2	2
FRONT▽	90	6	6	7	10	96	52	3	3	3	2	15	10	1	1
FRONT	67	1	1	4	7	93	97	2	2	2	2	4	2	1	1
Interspace▽	46	82	85	24	26	35	18	96	96	8	3	16	13	1	2
Interspace	45	78	85	22	24	32	18	96	96	6	3	18	17	1	1
Eph Block▽	46	46	47	77	75	52	26	65	65	61	52	32	34	2	2
Eph Block	69	77	80	89	87	83	60	88	88	80	78	41	50	3	3
RegulaTor▽	70	3	3	8	11	64	31	2	2	5	3	78	39	2	2
RegulaTor	71	9	9	8	10	65	60	2	2	4	3	53	78	2	2
Tamaraw▽	1	1	1	1	1	2	1	1	1	1	1	1	1	51	33
Tamaraw	1	1	1	1	1	1	1	1	1	1	1	1	1	4	45

Figure 10: Accuracy on the BigEnough standard dataset [43] in a closed world for *Laserbeak without attention* [42] trained on the defense given in row when tested on the defense given in column. Left for the 30 epochs training, right for infinite training.