Aggregate-Based Congestion Control for Pulse-Wave DDoS Defense

Paper #615; 12 pages (without references and appendices)

ABSTRACT

Pulse-wave DDoS attacks are a new type of volumetric attack consisting in series of short, high-rate traffic pulses. Such attacks target the Achilles' heel of even state-of-the-art DDoS defenses today: their reaction time. Indeed, by continuously adapting their attack vectors, pulse-wave attacks prevent defense systems to mitigate them effectively.

In this paper, we leverage programmable switches to build an in-network DDoS defense system effective against pulse-wave attacks. To do so, we revisit Aggregate-based Congestion Control (ACC), a mechanism proposed two decades ago to manage congestion events caused by high-bandwidth traffic aggregates. While ACC proved efficient in inferring and controlling DoS attacks, its design unfortunately cannot keep up with the speed requirements of pulse-wave attacks.

We propose *ACC-Turbo*, a renewed version of ACC, which: (i) infers attack patterns by applying online-clustering techniques directly in the network; and (ii) mitigates them by using programmable packet scheduling. Doing so, *ACC-Turbo* is able to infer attacks at line-rate and in real-time; and to rate-limit attack traffic on a per-packet level.

We fully implement *ACC-Turbo* in P4 and evaluate it over a wide range of attack scenarios. Our evaluation shows that *ACC-Turbo*: (i) autonomously identifies DDoS attack vectors in an unsupervised manner; and (ii) rapidly mitigates pulse-wave DDoS attacks. We also show that *ACC-Turbo* runs on existing hardware (Intel Tofino).

1 INTRODUCTION

Pulse-wave DDoS attacks have recently managed to take down critical network infrastructure while causing enormous financial and reputational damages [3, 40, 46, 54]. In contrast to conventional DDoS attacks, which grow steadily and persist longer in time, pulse-wave DDoS attacks consist of high-rate short-lived bursts. Each burst typically leverages a different attack vector (e.g., NTP, DNS, Memcached), and can reach hundreds of Gbps [34, 47, 49].

The threat in pulse-wave attacks resides in that they target the Achilles' heel of existing DDoS mitigation systems: their reaction time. Most in-network DDoS defenses today (both in research and production) rely on some sort of offline facility that can either directly scrub traffic [6, 17, 20, 30, 38, 55], orchestrate a routing-based defense [43, 48], or deploy an in-network pre-configured mitigation [31, 51]. The time required for an in-network defense to reach this external facility, and deploy the corresponding defense, can be in the order of seconds to minutes [49]. Pulse-wave attacks exploit this vulnerability by sending traffic pulses that force the DDoS defense to repeat this control loop over and over.

Designing a pulse-wave DDoS defense is an intricate task. Same as conventional-DDoS defenses, an ideal pulse-wave defense needs to be, first, *generic*, to identify a broad variety of attack vectors at different granularity [4, 15]. Generic techniques usually require unsupervision, and incur the risk of misclassifying traffic. Thus, an ideal defense also needs to be *measured* in responding to attacks [44]. Most DDoS defenses already fail at the first condition. For example, signature-based defenses [31, 51, 55] are not generic. They only cover a small set of attack vectors, and can not keep up with the constantly-growing list of new attacks [4, 15]. Similarly, most congestion-management tools (e.g., heavy-hitter detectors or active queue management) lack granularity: they only work at e.g., the per-flow level, or the whole-traffic level. Other DDoS defenses fail at satisfying the second condition. For example, drop- or routing-based defenses [18, 35] strongly degrade performance in case of misclassification.

Aggregate-based Congestion Control (ACC) was proposed twenty years ago, *satisfying the two* design conditions [32]. ACC is a canonical mechanism to reduce the impact of congestion caused by *generic* traffic aggregates, with a *measured* bandwidth control. At a high-level, ACC is a feedback loop which iteratively: *(i)* infers the aggregates causing the congestion; before *(ii)* reducing their throughput to a reasonable level. To infer the aggregates, ACC clusters the headers of packets dropped by a Random Early Detection (RED) queuing discipline. ACC then rate-limits the inferred aggregates to keep the total traffic throughput below the link capacity.

While ACC is effective at inferring and controlling conventional DDoS attacks, it fails at mitigating pulse-wave attacks, as it cannot keep up with the fast reaction times required to mitigate them. The reason is two-fold: (i) ACC relies on *offline* inference and control mechanisms, which run on either a separate server or a control plane, and (ii) ACC leverages a threshold-based defense activation. By running *offline*, ACC suffers from slow reaction time, which opens the door to pulse-wave DDoS attacks. By relying on threshold-based activation, ACC either further slows down reaction time, or increases the probability of false positives, which negatively impacts its performance. In our experiments, even with the best configuration, ACC drops $\approx 35\%$ of benign traffic in the event of a pulse-wave attack (§2.2).

	Activation	Inference	Control
ACC [32]	Threshold-based	Offline clustering on RED drops	Uniform rate limiting using estimated rates
ACC-Turbo	Always-on	Online clustering on all traffic	Programmable scheduling using exact rates

Table 1: *ACC-Turbo* techniques vs. ACC.

Our work. In this paper, we redesign ACC for pulse-wave DDoS defense. We propose *ACC-Turbo*: the *first* sub-second-reaction-time aggregate-based congestion control mechanism that mitigates pulse-wave DDoS attacks by running at line-rate on commodity hardware. *ACC-Turbo* leverages *online clustering* directly in the data plane to infer attacks, and *programmable scheduling* to mitigate them (cf. Table 1).

First, *ACC-Turbo* offloads the clustering process to the data plane, and analyzes *all* the traffic at line rate (instead of just a sample). This allows *ACC-Turbo* to substantially speed up reaction time. Further, since data-plane processing does not impact traffic latency, *ACC-Turbo* runs the clustering algorithm *continuously*. This eliminates the need for a threshold-based activation, which can be vulnerable to pulse-wave attacks and opens the door to potential false positives. The *always-on* design enables *ACC-Turbo* to anticipate the congestion events, achieving yet-faster reaction time.

Second, *ACC-Turbo* leverages programmable scheduling to deprioritize malicious traffic instead of dropping or rate-limiting it. Doing so has three advantages: (i) it accommodates fine-grained assessments which can adjust to the requirements of individual aggregates, increasing fairness; (ii) it adapts at the per-packet level, being able rapidly react to traffic variations, achieving faster and more-accurate bandwidth allocations; and (iii) it only leads to hard drops under congestion, being transparent (and therefore safe) otherwise.

Evaluation. We implement *ACC-Turbo* in P4 [9] and run it on programmable switches (Intel Tofino). We show that *ACC-Turbo* effectively mitigates pulse-wave DDoS attacks, in real-time (i.e., less than 1*s* reaction time), rapidly adapting to attack variations. We also compare *ACC-Turbo* with Jaqen [31], a state-of-the-art DDoS defense. Our results show that *ACC-Turbo* is *at least* 10× faster than Jaqen in mitigating a much broader range of attacks, while also being safer.

Contributions. We make the following contributions:

- An online-clustering approach to infer pulse-wave DDoS attacks at scale directly from the network (§4).
- A scheduling algorithm to mitigate pulse-wave attacks and minimize their impact on background traffic (§5).
- An implementation[1] of *ACC-Turbo* in Python/P4 (§6).
- A comprehensive evaluation showing *ACC-Turbo*'s practicality and its ability to run on hardware (§7, §8).

[1]We will make all our code publicly available.

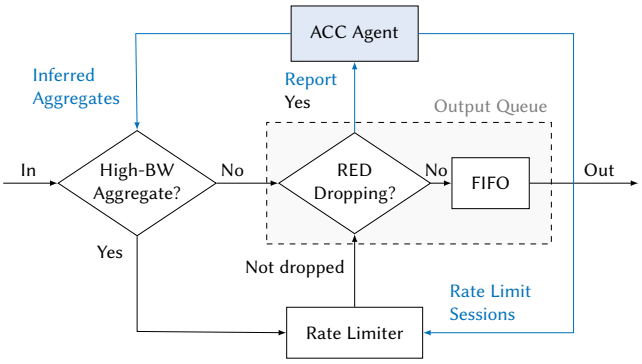


Figure 1: ACC architecture [32].

2 BACKGROUND

We now review ACC's design, as well as its limitations in mitigating pulse-wave DDoS attacks.

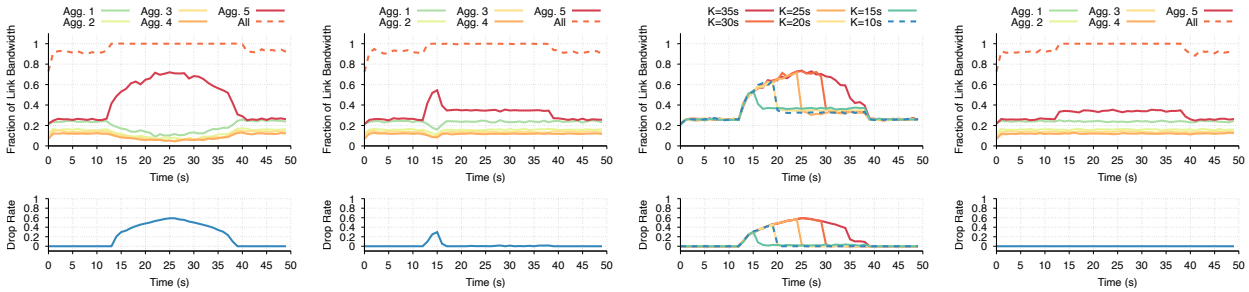
2.1 Aggregate-based Congestion Control

ACC is a mechanism for detecting and controlling high-bandwidth aggregates that persistently overload a link in the network. In this paper, we focus on its local version, which runs on the switch that gives access to the congested link.[2]

Fig. 1 shows the architecture of an ACC-enabled switch. The core of ACC is composed of a RED module, implemented on top of a first-in first-out (FIFO) queue. RED monitors the average queue size of the FIFO queue and drops packets probabilistically depending on its size. If the FIFO queue is almost empty, then all incoming packets are accepted. As the queue grows, the probability of dropping an incoming packet increases. When the queue is full, the probability is at its maximum and all the incoming packets are dropped.

Whenever the RED module decides to drop a packet, it reports the dropped-packet's header to an ACC Agent. The ACC Agent periodically analyzes the packet headers of all dropped packets and tries to infer the traffic aggregates responsible for the congestion. When the ACC Agent identifies an aggregate, it creates a rate-limit session to police the aggregates' throughput. From that point onwards, all upcoming packets belonging to the aggregate will be classified and rate limited before being processed by the RED module.

[2]The original work also includes a "pushback" mechanism to extend the rate-limiting policies to upstream switches. This part is out of our scope.



(a) No ACC (b) ACC (c) ACC: Impact of K (d) *ACC-Turbo*

Figure 2: Comparison between ACC and *ACC-Turbo* with the original papers' experiment [32].

Identifying congestion. The ACC Agent is activated when the output queue experiences sustained high congestion. This occurs when the drop rate in the output queue exceeds a pre-defined value, p_{high}, during a pre-defined period, K.

Inferring aggregates. The ACC Agent infers traffic aggregates based *solely* on IP prefixes. At the high level, it extracts a list of either source or destination IP addresses that account for more-than-twice the mean number of drops, and clusters them into 24-bit prefixes. To minimize collateral damage, it then walks down the prefix subtree trying to identify longer prefixes that still contain most of the drops.

Rate-limiting aggregates. For each aggregate inferred, the ACC Agent then computes the bandwidth to which it should be rate-limited. This limit is computed such that the drop rate at the output queue, gets below a pre-defined value, p_{target}.

To that end, the ACC Agent, first, sorts the list of inferred aggregates by their number of drops (highest, first). Then, it computes the excess arrival rate at the output queue, R_{excess}, defined as the amount of traffic that should be dropped in order for the drop rate to get below p_{target}. Finally, the ACC Agent determines the minimum number of aggregates that should be rate-limited, $|\mathcal{A}|$, and the rate to which they should be limited, L, such that the total rate is reduced by R_{excess}:

$$\sum_{i=1}^{|\mathcal{A}|}(Aggregate[i].rate - L) = R_{excess}$$

With this design, ACC manages to be generic (inferring attacks agnostically to their characteristics), and measured (rate-limiting inferred attacks instead of just dropping them), being a great defense against conventional DDoS attacks.

Experiment. We illustrate ACC's performance by using packet-level simulations (cf., §8 for details). We reproduce ACC's original experiment [32], which consists of scheduling five aggregates over a bottleneck link using FIFO, and ACC. Aggregates 1-4 are constant-bit-rate flows. Aggregate 5 is a variable-rate flow which represents an attack, and starts increasing (resp. decreasing) its rate at $t = 13s$ (resp. $t = 25s$).

Fig. 2a and Fig. 2b show the bandwidth share across the five aggregates (top), and the drop rate at the output queue (bottom), when no protection (i.e., FIFO) and ACC are used, respectively. ACC is configured with $p_{high} = 0.1$, $K = 2s$, and $p_{target} = 0.05$. Appendix B details the rest of parameters. Without ACC, we see how the attack traffic captures most of the link bandwidth, degrading the performance of the other aggregates. With ACC, the attack is efficiently mitigated. Indeed, when the drop rate at the output queue exceeds p_{high}, the ACC Agent infers the attack, and rate-limits it sufficiently to reduce its impact on background traffic.

Finally, we also see how ACC's reaction time is $\approx 4s$. This is the time since the first attack packets arrive (at $t = 13s$), until the defense is deployed (at $t = 17s$). This time is mostly driven by the monitoring-window size, K. For smaller K values, ACC checks more often whether p_{high} is exceeded, being able to detect faster when it happens. A lower K, however, does not always imply a faster reaction time (cf. Fig. 2c). For example, $K = 10s$ achieves a slower reaction time than $K = 15s$. Even though the threshold is first checked at $t = 10s$, it is not triggered until $t = 20s$, when p_{high} is reached.

2.2 Limitations of ACC

While ACC successfully defends against conventional DDoS attacks, it is vulnerable against pulse-wave DDoS attacks. Indeed, ACC uses *offline* inference and control mechanisms, which run on either a separate server or a control plane [32]; and it relies on a threshold-based defense activation.

Pulse-wave DDoS attacks are especially crafted to target the time required by such control planes to compute and deploy the DDoS defenses. They do so, by sending short high-rate traffic pulses which morph over time. These pulses congest the link resources while the inference and control processes are running. By the time they manage to converge, and the attack is mitigated, another pulse comes in, forcing the control loop to start the mitigation process all over again.

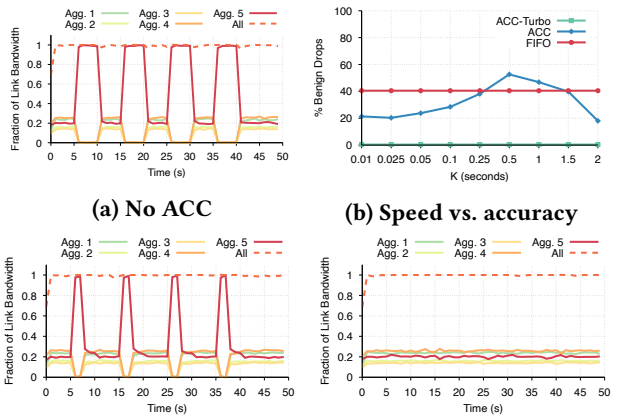


Figure 3: Performance under morphing attack.

Threshold-based defense activation introduces a second vulnerability to pulse-wave attacks. Indeed, if the threshold is too small (to speed up reaction time), the probability of false positives increases (i.e., benign traffic-bursts identified as attacks), as we prove in §8. In contrast, if the threshold is too large (to improve accuracy) reaction time gets slower, opening the door to pulse-wave attacks. In ACC, false positives are especially concerning under attack, given that (i) ACC rate-limits all aggregates to the same amount, and (ii) its rate-limiting policies have long-lasting effects (cf. §2.1).

Example. We evaluate ACC's mitigation efficiency in the case of a pulse-wave attack composed of four vectors (starting at 5*s*, 15*s*, 25*s*, and 35*s*). For simplicity, we represent the four pulses as a single "attack" aggregate (i.e., aggregate 5). We leverage four constant-bit-rate flows as background traffic (i.e., aggregates 1-4), which transmit at ≈ the link capacity.

Fig. 3a and Fig. 3c illustrate the bandwidth share at the output link when FIFO and ACC (configured as in §2.1) are used. We see how ACC fails at mitigating the attack, only managing to save ≈ 50% of benign traffic. Due to its threshold-based design, when p_{high} is reached, ACC infers the five aggregates, rate-limiting *both*, the attack *and* the benign traffic.

Fig. 3b shows how reducing the size of the monitoring window, K, does not help. In fact, due to false positives, the performance even gets worse for the smallest K values, resulting in up to ≈ 56% of benign-traffic being dropped.

Given ACC's inability to mitigate pulse-wave attacks, we propose *ACC-Turbo*. With an under-second reaction time, and disposing of the threshold-based activation, *ACC-Turbo* manages to successfully mitigate *both* conventional *and* pulse-wave DDoS attacks (cf., Fig. 2d and Fig. 3d, respectively).

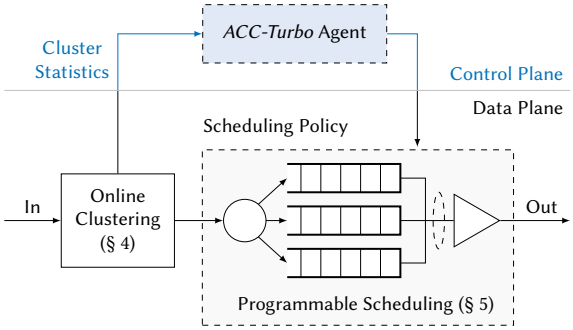


Figure 4: *ACC-Turbo* architecture.

3 OVERVIEW OF *ACC-TURBO*

ACC-Turbo is a switch-native aggregate-based congestion control mechanism that leverages programmable switches to mitigate pulse-wave DDoS attacks. It consists of two parts: an *online-clustering* module, that runs entirely in the data plane, and a *programmable-scheduling* module, that runs in both, the control plane and the data plane (see Fig. 4).

With this hybrid design, *ACC-Turbo* balances the need for fast reaction time (by running the inference process in the data plane), and accuracy, within the limited resources of the switch (by dedicating most of the data-plane resources to the inference process, while offloading the remaining less-time-sensitive components to the control plane).

The online-clustering module is in charge of extracting a set of features from the arriving-packets' headers and uses them to cluster similar packets together. It runs entirely in the data plane, processing *all* packets at line rate. As such, it can run *continuously*, regardless on whether there is congestion or not, eliminating the need for a threshold-based activation. Further, by running continuously, it anticipates inference decisions to the actual attack, speeding up reaction time.

The programmable-scheduling module involves both, the control and the data plane. The control plane periodically polls information about the extracted clusters, including statistics about the clusters and their exact arrival rates. Then, it assesses the probability that each cluster contains attack traffic. It maps this probability into a scheduling policy, which deprioritizes clusters with higher probability of conforming an attack. This policy is finally deployed into the data plane, and executed for all the packets belonging to the cluster.

By using programmable scheduling, *ACC-Turbo* can adapt to traffic variations at a per-packet granularity, being able to rapidly react to attack changes. Further, programmable scheduling enables *ACC-Turbo* to run continuously, even in the case of no attack. This is because programmable scheduling does not hurt traffic: it only drops packets in case of severe congestion (being transparent otherwise), and always starts by those with higher chances of conforming an attack.

4 TRAFFIC-AGGREGATE INFERENCE

We now describe the theoretical basis behind *ACC-Turbo*'s inference component. First, we phrase the problem formally and propose a practical solution based on online clustering (§4.1). Second, we introduce the design decisions that make *ACC-Turbo* implementable in existing programmable switches (§4.2). The resulting algorithm is in Appendix C.

4.1 Problem definition

Let us define a packet p as a set of features $\mathcal{F}$, where each feature corresponds to a field from the packet header (e.g., *sport, dport, ip.ttl, ip.proto*). For each feature $f \in \mathcal{F}$, packet p has a specific value associated: p_f. We distinguish two types of features: *ordinal* features, for which closer proximity between their values implies stronger similarity (e.g., *saddr, daddr, ip.len, ip.ttl*), and *nominal* features, for which closer values do not necessarily imply similarity (e.g., *sport, dport*).

Let us define an aggregate a as the same set of features $\mathcal{F}$. For each *ordinal* feature f, the aggregate a has a *range* of values associated: $f(a) = [min_f(a), max_f(a)]$. Analogously, for each *nominal* feature f, aggregate a has a *set* of discrete values associated $f(a) = \{min_f(a), ..., max_f(a)\}$. With this definition, the aggregate represents *all* the packets with feature values included in its ranges and sets.

Objective. At the high-level, given a set of incoming packets, our goal is to infer a set of aggregates that represent *all* the observed packets as *precisely* as possible. We need to limit the number of aggregates to infer by a parameter, $|\mathcal{A}|$. Otherwise, one could simply list each observed packet as a separate aggregate and obtain perfect precision.

Definition 4.1 (Aggregate-inference problem). Let $\delta_f(a)$ be the cost of an aggregate a, for feature f, which measures the number of values that it represents. For ordinal features, $\delta_f(a) = max_f(a) - min_f(a)$. For nominal features, $\delta_f(a) = |f(a)|$ (i.e., the number of values in the set). Let $\delta(a) = \prod_{f \in \mathcal{F}} \delta_f(a)$ be the cost of the entire aggregate, as the product of each individual feature cost. For a given feature f, a set of packets $\mathcal{P}$, and a limit on the number of inferred aggregates ($|\mathcal{A}|$), find the set of aggregates $\mathcal{A}^* = a_1, ...a_{k'} (k' \leq |\mathcal{A}|)$ that represent $\mathcal{P}$ and minimize $\delta_f(\mathcal{A}^*) = \sum_{a \in \mathcal{A}^*} \delta_f(a)$.

The presented problem is equivalent to the intent-inference problem in [27], which is NP-hard. As such, we propose a heuristic solution based on online-clustering. Our solution aims at approximating the optimal result, while enabling per-packet processing. The cost function in Def. 4.1 estimates the number of different packets that a represents, but it is also a measure of similarity of the packets in a. Indeed, packets that can be represented in a narrow aggregate have higher similarity than those which require a broader aggregate. With this intuition, we build an online-clustering algorithm that groups similar packets by minimizing the cost in Def. 4.1.

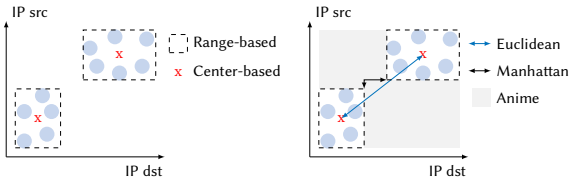


Figure 5: Cluster representations and distances.

Definition 4.2 (Online-clustering framework [11]). For a sequence of points p in $\mathcal{P}$, maintain a collection of $|C|$ clusters such that, when each input point p is presented, either it is assigned to one of the current clusters or it starts off a new cluster while two existing clusters are merged into one.

The online-clustering framework is characterized by an endless stream of data, where each data point (i..e, packet) is seen only once: it comes in, is processed, and then it goes away never to return. As such, the algorithm is required to take an irrevocable action after the arrival of each point [22].

4.2 Design decisions

The proposed framework allows three design decisions: (i) the number of clustering possibilities to compare at each iteration, (ii) the type of information to store about each cluster, and (iii) the distance to use to assess the clustering decisions. We make these decisions with the goal of maximizing performance, while staying within the resource constraints of existing data planes (to achieve an in-network design).

4.2.1 *Clustering search (fast vs. exhaustive).*

When a new packet arrives, the online-clustering algorithm can either: (i) merge it to the closest cluster, or (ii) merge two existing clusters and create a new cluster for the new packet.

[✓] Fast search. If the clustering algorithm only supports step (i), we call it *fast*. This approach follows a linear search and requires only $|C|$ distance computations: one for each existing cluster. *ACC-Turbo* relies on this type of search, as it can be implemented on programmable data planes.

[X] Exhaustive search. If the clustering algorithm supports steps (i) and (ii), we call it *exhaustive*. This approach follows a quadratic search, and requires $\binom{|C|}{2}$ additional distance computations. For a given clustering decision, the exhaustive approach outperforms, since its search space includes (but is not limited to) the one of the fast approach. However, it is not implementable *at line-rate* in existing programmable data planes. Indeed, it requires accessing multiple times each cluster's information, while registers in today's pipelines can only be accessed once per packet.

4.2.2 ***Cluster representation (ranges vs. center).*** A naive way to represent a cluster is as a mere collection of packets. This is, keeping track of *all* the packets and their respective feature values. Such representation is not practical as it does not scale. Therefore, we study two alternatives.

[X] Center-based representation. We can represent clusters by just a single point (e.g., the center of the cluster). The advantage of this representation is that the distance computation is simple, and centers can be easily updated following some pre-defined learning rate (e.g., online k-means [52]). However, we lose a lot of information, such as how big the cluster is, or which packets does it contain, which is useful for cluster assessment and traceability, respectively (cf. §9).

[✓] Range-based representation. Following the problem formulation in §4.1, *ACC-Turbo* represents each cluster c with a range of values for each ordinal feature $[min_f(c), max_f(c)]$, and a set of discrete unique values for each nominal feature $\{min_f(c), ..., max_f(c)\}$. Ranges (resp. sets) represent the feature values of packets in a cluster (Fig. 5). This representation preserves information about the cluster sizes, and simplifies interpretability by providing the exact mapping of packets to clusters. Further, ranges and sets are easy to compute and update, which facilitates its implementation on programmable data planes. For instance, the ranges of a new cluster that merges two existing clusters c_i and c_j for feature f are: $[\min(min_f(c_i), min_f(c_j)), \max(max_f(c_i), max_f(c_j))]$. Sets can be implemented as admission lists, using bloom-filters (cf. §6).

4.2.3 ***The distance function.*** We first introduce two distance functions for reference, and then derive the distance function that we use in *ACC-Turbo*. For simplicity, we illustrate the case with only ordinal features.

[X] Anime distance [27]. The cost function in Def.4.1 can be translated to the online-clustering framework as:

$$\delta_{Anime}(C) = \sum_{c_i \in C} \delta(c_i) = \sum_{c_i \in C} (\prod_{f \in \mathcal{F}} max_f(c_i) - min_f(c_i)) \quad (1)$$

This cost function sums the cost of each individual cluster, which accounts for the number of packets that the cluster represents, and estimates the similarity across packets in the cluster. We now derive a distance function that can be used to assess clustering decisions while trying to minimize this cost. Assuming a range-based cluster representation, we define the distance between clusters c_i and c_j, $\delta(c_i, c_j)$, as the amount of increase in cost produced by merging the two clusters, compared to the total cost of the two clusters if they are not merged: $\delta(c_i, c_j) = \delta(c_i \cup c_j) - (\delta(c_i) + \delta(c_j))$.
The biggest drawback of the Anime distance is the size of the output space. It can be measured as the product of the maximum ranges that each feature can allocate. For instance, with the following features and sizes {*len (16b), id (16b), f_offset (13b), ttl (8b), proto (8b), saddr (32b), daddr (32b), sport (16b), dport (16b)*}, the maximum cost is 2^{157}. Data structures in existing data planes can store blocks of maximum 64 bits, which is not enough to represent such cost.

[X] Euclidean distance. Alternatively, we could derive a cost function from the broadly-used Euclidean distance [52]:

$$\delta_{Euclid.} = \sum_{c_i \in C} \delta'(c_i) = \sum_{c_i \in C} \left(\sum_{f \in \mathcal{F}} \sum_{p \in c_i} \|p_f - r_f(c_i)\|^2 \right) \quad (2)$$

In that case, the output-space size is much smaller. For the same example, the maximum cost could be represented with less than 20 bits. However, computing the Euclidean distance involves squares and root operations which, by default, are not straightforward to implement on existing data planes.

[✓] Manhattan distance. In *ACC-Turbo* we use an alternative distance metric that gathers the benefits of both. Starting from the Anime distance and trying to reduce the size of its output space, we substitute the product of all the individual feature distances by a summation. With this modification, the overall cost function to be minimized can be written as:

$$\delta_{Manh.} = \sum_{c_i \in C} \delta''(c_i) = \sum_{c_i \in C} (\sum_{f \in \mathcal{F}} max_f(c_i) - min_f(c_i)), \quad (3)$$

and the resulting distance function to compare a new packet p with one of the clusters c_i can be written as: $\delta(p, c_i) = \delta(p \cup c_i) - (\delta(p) + \delta(c_i))$. We know that $\delta(p) = 1$ will have a fixed value, so it will not impact the comparison. First, $\delta(p \cup c_i) = \sum_{f \in \mathcal{F}} \delta_f(p \cup c_i)$, where:

$$\delta_f(p \cup c_i) = \begin{cases} max_f(c_i) - p_f, & \text{if } p_f < min_f(c_i), \\ p_f - min_f(c_i), & \text{if } p_f > max_f(c_i), \\ 0, & \text{otherwise.} \end{cases} \quad (4)$$

Second, we can write $\delta(c_i) = \sum_{f \in \mathcal{F}} \delta_f(c_i)$, where $\delta_f(c_i) = [max_f(c_i) - min_f(c_i)]$. Finally, $\delta(p, c_i) \approx \delta(p \cup c_i) - \delta(c_i) = \sum_{f \in \mathcal{F}} [\delta_f(p \cup c_i) - \delta_f(c_i)] = \sum_{f \in \mathcal{F}} \delta_f(p, c_i)$, where:

$$\delta_f(p, c_i) = \begin{cases} min_f(c_i) - p_f, & \text{if } p_f < min_f(c_i), \\ p_f - max_f(c_i), & \text{if } p_f > max_f(c_i), \\ 0, & \text{otherwise.} \end{cases} \quad (5)$$

What results from the simplification of the Anime distance into a one-dimensional space, is the Manhattan distance from the packet to its closest point of the cluster. For a single dimension (i.e., feature), the Manhattan distance and the Anime distance are equivalent. For higher dimensions, the Manhattan distance compresses the output space, losing information with respect to the original Anime distance. However, it is easier to compute, and it generates outputs in a linear space (as we have sums instead of products). This makes it implementable in existing data planes.

5 CONTROLLING AGGREGATES

We now describe how *ACC-Turbo* uses programmable scheduling to mitigate attacks. First, we introduce the scheduling design space (§5.1), and then, *ACC-Turbo*'s scheduler (§5.2).

5.1 Design space

Programming a scheduler consists in defining the order in which it should drain packets from a given buffer. This is done by tagging each packet with a *rank* that indicates the priority with which it should be drained.[3] These ranks are then processed by a (programmable) scheduler that dequeues the packets trying to follow the order specified [21, 42, 53].

Ranking algorithms for pulse-wave attacks. From our definition in §4, a ranking algorithm to mitigate pulse-wave attacks should deprioritize aggregates of high bandwidth and high packet similarity. A number of ranking algorithms can be proposed with this criteria. For example, $rank(p) = throughput(c_i)$, $rank(p) = num.packets(c_i)$, and $rank(p) = throughput(c_i)/size(c_i)$ deprioritize packets by throughput, packet rate, and a combination between throughput and cluster size, respectively. These algorithms can be easily computed with the available data-plane resources. Indeed, aggregate rates can be obtained from packet counters, and packet similarity can be extracted from cluster sizes (cf. §4).

5.2 Scheduling algorithm

To use the proposed ranking algorithms for pulse-wave DDoS defense *today*, we need to make them fit into the limited resources of existing data planes. This is hard for three reasons. First, these resources need to be shared with *ACC-Turbo*'s online-clustering module, which is resource exhaustive (e.g., 12 stages for 4 clusters and 4 features (§6)). Second, existing programmable switches do not support schedulers able to process ranks, forcing us to "build" our own. While it is possible to approximate this scheduling logic using priority queues [21, 53], doing so requires additional resources (e.g., one stage per priority queue [21]). Third, the clustering and scheduling modules need to operate sequentially, since the ranking algorithm requires the clustering results.

Design. We build a programmable scheduler on top of priority queues, and offload the rank computation and the queue mapping to the control plane. Specifically, the control plane periodically (i) polls information about the extracted clusters from the data plane, (ii) assesses clusters' maliciousness and maps them to a priority queue, and (iii) deploys this mapping into the data plane such that *future packets* of each cluster can be scheduled accordingly. This design dedicates all the data-plane resources to the inference process, maximizing its accuracy, and preserving line-rate processing.

[3] Generally, a lower rank indicates a higher priority.

6 IMPLEMENTATION

We implement *ACC-Turbo* in $P4_{16}$ [9] on Intel Tofino Wedge 100BF-32X [1], with 2360 lines of code. Our prototype uses 12 stages and supports 4 features and 4 clusters. For each incoming packet, *ACC-Turbo* (i) computes its distance to each cluster, (ii) selects (and updates) the closest cluster, and (iii) enqueues the packet with the selected-cluster's priority.

Cluster selection. For each cluster c, we store the minimum and maximum values of its ordinal-feature's ranges $[min_f(c), max_f(c)]$ using registers. We store its nominal-feature's sets using an admission list, implemented as a bloom filter (cf. §4).

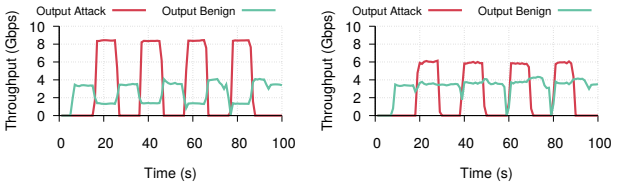
For each arriving packet, we compute its distance to each cluster by computing all the per-cluster per-feature distances, and aggregating them per cluster (cf. Appendix C). For *ordinal* features, we compute the per-cluster per-feature distances by, first, accessing the register containing $min_f(c)$, and checking if the packet feature, p_f, is below the stored value. If that is the case, we set the distance to $d_f(p, c) = min_f(c) - p_f$ within the register's ALU. If not, we access the register containing $max_f(c)$ and set the distance to $d_f(p, c) = p_f - max_f(c)$ if $p_f > max_f(c)$. Since the two registers are accessed sequentially, this takes two stages. However, computation for different cluster-feature pairs can be parallelized. For *nominal* features, we set the distance to 1 if the bloom-filter entry matched by p_f is empty. This takes one stage.

We aggregate the per-cluster per-feature distances into per-cluster distances, by progressively summing two distances at each stage using non-stateful ALUs. This requires $log_2|\mathcal{F}|$ stages, being $|\mathcal{F}|$ the number of features. Finally, we find the minimum distance by progressively comparing two distances at each stage also using non-stateful ALUs. This requires $log_2|C|$ stages, being $|C|$ the number of clusters. The distance-computation and distance-aggregation operations can also be parallelized to reduce the number of stages.

Cluster update. When the minimum distance is not zero, we know that the packet is outside the selected-cluster's coverage. In that case, we update the cluster's ranges and sets to accommodate the new packet, by using resubmission.

Queue selection. We enqueue the packet to the priority queue assigned to its cluster, which is given by a match-action table populated by the controller. The controller computes the clusters' priorities following the policy in §5.

Resource requirements. Our implementation (in Tofino 1 [1]) is restricted by the number of stages required, which limits the number of supported clusters and features. Newer programmable switches (e.g., Tofino 2 [36] and Tofino 3 [24]), support higher number of stages, allowing more-performant implementations with more clusters and features.



(a) FIFO **(b) *ACC-Turbo***

Figure 6: Mitigation of a pulse-wave DDoS attack.

7 HARDWARE-BASED EVALUATION

We evaluate our hardware implementation of *ACC-Turbo* on the Intel Tofino Wedge 100BF-32X [1]. First, we evaluate *ACC-Turbo*'s performance under a pulse-wave DDoS attack (§7.1). Second, we compare *ACC-Turbo*'s performance to the one of Jaqen [31], the state-of-the-art DDoS defense (§7.2).

7.1 *ACC-Turbo*'s performance

We generate traffic between two servers, connected by a Tofino switch via 100 Gbps (sender→Tofino) and 10 Gbps (Tofino→receiver) interfaces. As in previous work [31, 55], we replay CAIDA traces as background traffic [10] and add attack traffic on top using MoonGen [19]. The attack is composed of four UDP-flood pulses which have a duration of 10 seconds and are followed by a 10-second interleave. Each pulse targets a different IP address within a common subnet, and a different port. At its peak, the traffic reaches 40.789 Gbps. We configure *ACC-Turbo* to use 4 clusters, and to use the last two bytes of the IP destination address, the source port, and the destination port, as clustering features. We use a throughput-based ranking algorithm and update the cluster priorities at the controller's maximum speed.

Recovery rate of background traffic. Fig. 6a shows the traffic-throughput evolution under no protection. The background traffic is severely impacted by the attack, with a throughput reduction of ≈ 61%. Note that we are replaying traffic traces, and we do not see the impact of end-host congestion-control. Such effect would worsen performance even further. When *ACC-Turbo* is used instead, as soon as the attack is inferred and its traffic deprioritized, background traffic fully recovers its original throughput (Fig. 6b).

Reaction time. Fig. 6b illustrates the range of possible reaction times in *ACC-Turbo*. While *ACC-Turbo* reacts to the first pulse almost immediately, it takes up to ≈ 1s, to react to the last two pulses. This reaction time is the time it takes for the control plane to (i) poll the throughput of each cluster, (ii) update their priorities, and (iii) deploy them to the data plane. In our testbed, this is done via a non-optimized Python controller which takes several milliseconds.

	Detection	Reaction	Mitigation
ACC-Turbo	Clustering	Always-on	Programmable scheduling
Jaqen [31]	Signature (sketches)	Threshold-based	Rate limit / Drop

Table 2: *ACC-Turbo*'s techniques vs. Jaqen's.

Benign packet drops (%)	FIFO	Jaqen†	Jaqen‡	*ACC-Turbo*
No Attack	0.00	2.50	3.68	0.00
Single Flow	89.85	2.67	3.95	14.79
Carpet Bombing	89.88	73.19	3.95	19.98
Source Spoofing	89.87	88.16	88.51	14.80

Table 3: Mitigation efficiency under attack variations.

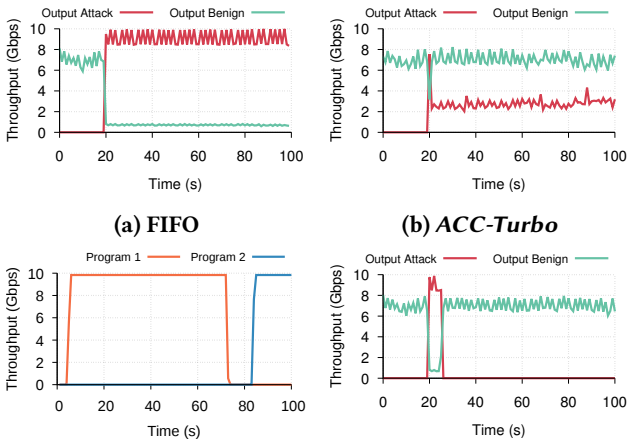
7.2 Comparison to the state-of-the-art DDoS-mitigation technique

We compare *ACC-Turbo* to Jaqen [31], the state-of-the-art DDoS defense. Jaqen uses sketch-based signatures to detect attacks and rate-limiting/dropping to mitigate them (cf. Table 2). With respect to Jaqen, we show that *ACC-Turbo* ...

(1) ...is more generic (§7.2.1), since it infers attacks agnostically, and not relying on pre-configured signatures.
(2) ...achieves faster reaction time (§7.2.2). This is because *ACC-Turbo* can mitigate attacks without reprogramming the switch (which is slow, as we show).
(3) ...avoids the "threshold-based activation" vulnerability (§7.2.3), by running continuously on all traffic.

We leverage the same setup as in the previous section, but using the four bytes of the destination IP address as features. We replay CAIDA traces as background traffic at twice their speed (reaching ≈ 7 *Gbps*), during 100 seconds, and run crafted attacks on top using MoonGen [19]. We generate attacks at maximum capacity, reaching up to ≈ 99 *Gbps*.

7.2.1 ***Genericity***. We evaluate *ACC-Turbo*'s and Jaqen's genericity by analyzing how robust their mitigation is to variations in attack traffic. Specifically, we consider a UDP-flood attack which initially consists of a single UDP flow (all the packets share the 5 tuple). We then modify the attack traffic by using: (i) UDP carpet bombing [8, 14, 23] (i.e., the attack targets a /24 destination prefix instead of a single IP); and (ii) UDP source spoofing. We configure Jaqen with a sketch that detects heavy hitters either by monitoring the 5-tuple (Jaqen†) or the source IP (Jaqen‡). We tune each sketch for optimum performance. We measure the percentage of benign packets dropped and show the results in Table 3.



(a) FIFO (b) *ACC-Turbo* (c) Reprogramming time (d) Jaqen

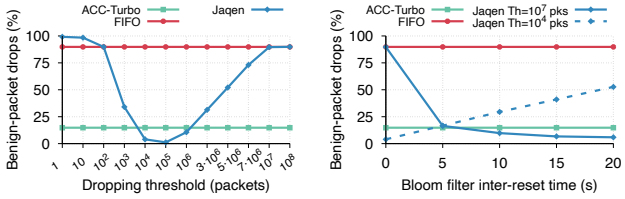
Figure 7: Reaction-time evaluation.

We can see that Jaqen is only effective when the right defense is deployed, but its performance drastically decreases as soon as the attack shape varies. Instead, *ACC-Turbo* performs well generally, being more robust to attack variations. This is expected: while *ACC-Turbo* infers attacks agnostically, without any initial assumption of the attack characteristics, Jaqen is signature-based, and relies on pre-configured defenses for a fixed subset of attacks.

7.2.2 ***Reaction time.*** We analyze the *reaction time* of *ACC-Turbo* and Jaqen, defined as the time since they see the first attack packet, until they start mitigating the attack. For *ACC-Turbo*, the reaction time is the time it takes for the control plane to poll the cluster statistics, update the cluster priorities, and deploy them to the data plane. For Jaqen, it is the time it needs to: detect the attack, compute the right mitigation, orchestrate the network to reroute legitimate traffic, replicate the switch state to the controller, and reprogram the switch with the right mitigation and the replicated state.

We measure *ACC-Turbo*'s and Jaqen's reaction times empirically. We consider two cases for Jaqen: (i) when it needs to install a new mitigation module (worst case); and (ii) when the right mitigation module already runs in the switch (best case). Our results show that *ACC-Turbo* reacts at least 11× (resp. 10×) faster than Jaqen in the first (resp. second) case.

***ACC-Turbo*'s reaction time**. We generate a simple UDP-flood attack (all packets sharing the 5-tuple) on top of the CAIDA trace (Fig. 7a), and measure the time it takes for *ACC-Turbo* to react. As depicted in Fig. 7b, *ACC-Turbo* takes ≈ 1 second to react to the attack. This is the time required by the (here, unoptimized) control plane to poll the cluster statistics, and deploy the updated priorities to the data plane.



(a) Threshold setting (b) Speed

Figure 8: Threshold-configuration sensitivity.

Jaqen's reaction time (defense not deployed). We measure how fast can Jaqen deploy a mitigation module which is not already running in a switch. To do so, we measure how long it takes for a hardware switch to swap between two (trivial) programs which simply rewrite the source IP of all packets. We execute the first program for one minute before instructing the switch to swap to the second program which has been pre-compiled and cached in advance. We send traffic during the experiment and measure the corresponding downtime. We repeat the experiment 10 times.

On average, it takes 11.5 seconds for Jaqen to load a new mitigation module. This is 11× slower than *ACC-Turbo*'s reaction time. Fig. 7c shows the result for one of the iterations.

Jaqen's reaction time (defense already deployed). If the mitigation module is already loaded, Jaqen's reaction time is the time it takes to detect the attack from the control plane and to deploy a dropping rule to the data plane. We measure this reaction time when Jaqen is configured with the simplest defense (to ensure it is as fast as possible): a sketch that monitors the number of attack packets, and a controller that reads it periodically, activating a dropping action when an attack is detected. We read the sketch entries at maximum speed and optimize the threshold value for performance.

As shown in Fig. 7d, Jaqen's reaction time is ≈ 10 seconds. This is *still* 10× slower than *ACC-Turbo*. This the time required by the controller to read the sketch and deploy the drop action, and the time it takes for the threshold to be reached not once, but twice, since Jaqen's only considers attacks when detected in two consecutive time windows.

Jaqen's slow reaction time, of ≈ 10 seconds, even in the ideal case, makes it vulnerable to pulse-wave DDoS attacks.

7.2.3 ***Threshold-configuration sensitivity.*** We measure how sensitive Jaqen is to the threshold-based defense activation vulnerability introduced in §2.2. To that end, we take Jaqen's simplest possible defense: the 5-tuple heavy hitter in §7.2.1 (Jaqen†). Such defense relies on two parameters: the *threshold* over which traffic is considered to be an attack, and the *periodicity* at which this threshold is checked. We analyze the mitigation efficiency when the two parameters are modified, in the case of a simple UDP-flood attack on

top of a CAIDA trace. We measure the percentage of benign-traffic dropped, and compare the results to the case in which no-defense (i.e., FIFO), and *ACC-Turbo*, are used.

Fig. 8a illustrates Jaqen's high sensitivity to threshold configuration. Slight variations in the threshold value (e.g., from $1M$ packets to $7M$ packets) can result in a drastic increase of benign-packet drops (from $\approx$ 10% to $\approx$ 75%). This is expected, and aligned with our arguments in §2.2. Indeed, the best threshold value depends on the dynamics of both attack and benign traffic, which change over time. As also expected, too-low thresholds can result even worse than not having any defense, due to benign-traffic drops under no congestion.

Fig. 8b shows how the periodicity at which the threshold is checked can further impact a certain threshold's performance. For example, a threshold of $10^4 packets$, which outperforms at the controller's maximum periodicity, performs very poorly if the periodicity decreases. At the same time, a bad-performing threshold at maximum periodicity (e.g., $10^7 packets$), can perform good at lower periodicity.

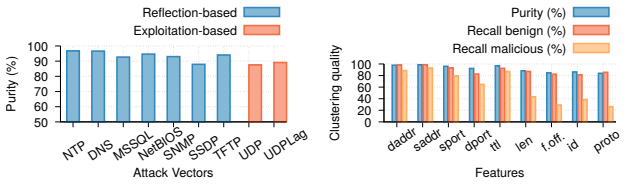
ACC-Turbo avoids this threshold-based vulnerability by running continuously on all traffic. Further, *ACC-Turbo* does not perform *binary* assessments on the traffic maliciousness based on its absolute traffic rate. Instead, it performs fine-grained decisions that prioritize traffic based on their relative rates. As a result, even though *ACC-Turbo*'s performance is not as good as Jaqen's when Jaqen is configured optimally, it outperforms when Jaqen is not perfectly tuned.

8 SIMULATION-BASED EVALUATION

Given the limitations of our Tofino prototype (§6), we extend *ACC-Turbo*'s evaluation by using packet-level simulations in Netbench [2, 26]. We evaluate *ACC-Turbo*'s performance when it schedules a mix of benign traffic and DDoS attacks. We study the impact of its different design decisions (§4), and analyze the performance of more-complete implementations of *ACC-Turbo* (e.g., by using Tofino2 or Tofino3 [24, 36]).

Methodology. We use the CICDDoS-2019 trace [41], which consists of a diverse set of attacks executed on top of benign traffic. We feed the trace into a (simulated) switch running *ACC-Turbo* and adjust the capacity of the output link to obtain the desired congestion levels. By default, we configure *ACC-Turbo* to support 10 clusters, and to use each byte of the *ip.src* and *ip.dst*, *sport*, *dport*, *ip.ttl*, and *ip.len* as features.

Overall performance. While *ACC-Turbo*'s performance depends on the characteristics of benign and attack traffic, we illustrate its practicality by evaluating it over a realistic dataset covering a wide range of attack vectors. For all attacks, *ACC-Turbo*'s online-clustering algorithm manages to distinguish packets coming from attack and benign distributions (cluster's purity of $\approx$ 90%), with as little as 10 clusters.



(a) Attack split **(b) Feature split**

Figure 9: Performance by attack type and features.

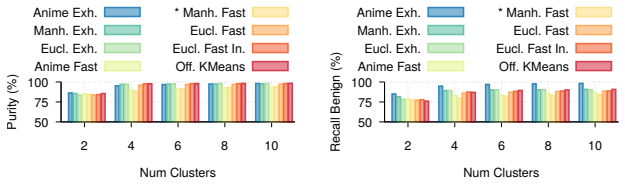
ACC-Turbo's performance is better for well-defined traffic aggregates. For reflection attacks with high packet similarity, *ACC-Turbo* manages to save up to 29% more of benign traffic than traditional FIFO queues, just 5.13% away from an ideal scheduler with full attack knowledge (cf. Fig. 11b, at 50Mbps).

8.1 Characterizing the clustering strategy

We evaluate *ACC-Turbo*'s inference process by measuring the *purity* and the *recall* of the extracted clusters. These metrics measure the accuracy of the clustering algorithm in mapping packets from different distributions into distinct clusters. We compute purity by (i) labeling each cluster as either majority-benign or majority-malicious, based on the amount of packets clustered of each type, (ii) counting the number of packets that match the cluster's label, and (iii) dividing it by the total number of packets [33]. We compute *recall* of benign (resp. malicious) packets, as the percentage of benign (resp. malicious) packets that are mapped into majority-benign (resp. -malicious) clusters. We monitor the metrics in one-minute time windows and average the result. We only analyze periods with both attack and benign traffic.

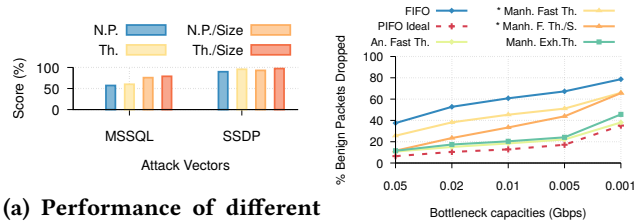
Feature selection and attack vectors. Fig. 9a shows the clustering performance across different attack vectors. In all cases the achieved purity is above 87%. Clustering performance is strongly related to attacks' feature variance. Reflection-based attacks achieve, on average, 5.4% better purity than exploitation-based attacks. Within reflection-based attacks, MSSQL and SSDP, which have higher feature-value variance (e.g., use mulitple *sport* values), perform worst

Fig. 9b shows the performance of clustering on individual features. For this particular dataset, IP addresses and source port are good identifiers of malicious traffic. In contrast, fields like IP protocol are less useful as attacks use both UDP and TCP. While the absolute values are tied to the trace characteristics, the split illustrates the different performance levels that *ACC-Turbo* can achieve. While "narrow" attacks achieve good performance if we look at the right features, their performance drastically drops as soon as attacks become diverse or features do not provide a clear signature.



(a) Purity (%) **(b) Recall benign (%)**

Figure 10: Performance of clustering strategies.



(a) Performance of different ranking algorithms

(b) Overall performance

Figure 11: Impact of scheduling for mitigation.

Number of clusters. Fig. 10 shows the performance of different clustering strategies when the number of clusters varies from 2 to 10. First, as expected (cf. §4), a higher number of clusters provides better purity and recall.[4] While selecting the optimal number of clusters is typically a challenge, in *ACC-Turbo* it is imposed by the hardware constraints (§6). Second, increasing the number of clusters is more beneficial for fewer clusters (e.g., the purity in *ACC-Turbo* improves by 4% when we move from 2 clusters to 4, while it only improves 1% when we move from 8 to 10). Since *ACC-Turbo* is designed to run in an environment where the number of clusters is limited, it builds upon this insight and dedicates all data plane resources to maximize the number of clusters (running the non-inference operations in the control plane).

Clustering search: *fast* vs. *exhaustive*. As also expected (cf. §4.2.2), we observe that *exhaustive* approaches generally outperform their respective *fast* versions, even though the difference gets smaller as the number of clusters increases. This is especially clear in the Anime and Manhattan approaches, which use range-based representations and have more information to assess clustering decisions (e.g., 93.24% to 98.09% purity increase for Anime with 10 clusters).

Cluster representations and distances. Overall, center-based distances "suffer" less when downgraded from exhaustive to fast (e.g., just 0.79% purity decrease with 10 clusters). There are two reasons for that. First, since they carry little information about clusters, the potential improvement of checking more combinations is limited. Second, range-based approaches are more sensitive to updates, since they directly include the new points as they are analyzed. In center-based approaches, we just move the center in the direction of the new data point using some learning rate (i.e., 0.3 in our case).

Fast vs. offline and baselines. We compare *ACC-Turbo*'s performance to an *offline* k-means approach with unlimited resources. In all cases, *ACC-Turbo* is very close to the *offline* case (e.g., 4.19% of difference in purity for 10 clusters, cf. Fig. 10). We also study the case of a hybrid approach which computes the cluster centers periodically offline, and updates them online with the newly arriving packets. While the hybrid approach performs better, we believe it is not significant enough to justify the increase in complexity that it requires.

[4] In fact, a naive way to achieve perfect purity is to have as many clusters as packets, mapping each packet to its own cluster. From a scheduling perspective, however, such approach would have no value, since clusters would not give any insight about the maliciousness of the packets contained.

8.2 Characterizing the scheduling scheme

Finally, we study *ACC-Turbo*'s scheduling performance. First, we evaluate the schedulers in §5, analyzing how often they prioritize benign traffic over malicious traffic. We measure a *score*, defined as the percentage of one-second intervals in the simulation where the average priority given to benign traffic is higher than the one given to malicious traffic. Second, we measure the amount of benign-packets dropped when the trace is scheduled by a FIFO queue, *ACC-Turbo*, and an ideal scheduler which prioritizes benign traffic. We use 10 clusters and the 10 most representative features for the trace (cf. §8.1).

Ranking algorithms. Fig. 11a shows the performance of the scheduling algorithms under the two most complex reflection attacks (cf. Fig.9a). While the absolute values are specific to the dataset, we can see how adding the similarity factor to the rank definition (i.e., the cluster sizes) improves performance. This result strongly supports *ACC-Turbo*'s design decision of using a range-based representation scheme.

Bottleneck. Fig. 11b analyzes the number of dropped packets for various bottleneck capacities. First, we see that the *ACC-Turbo* version that is implementable today (i.e., Manhattan distance, fast approach), performs almost on-par with the ideal case for smaller bottlenecks. Indeed, at 50 Mbps it saves 29% more of benign traffic than FIFO queues, being just 5.13% worse than an ideal PIFO with the ground truth. Second, by comparing its performance to the one of the Manhattan exhaustive, and Anime fast approaches, we see how the loss in performance of today's *ACC-Turbo* comes from the two design decisions required to make it fit in existing data planes: using a fast approach, and reducing the size of the distance space (§4). With newer programmable switches, some of which are already available today [24, 36], more-complete versions of *ACC-Turbo* become implementable.

9 DISCUSSION

Are today's DDoS still "aggregates"? Yes. Even though DDoS attacks can *theoretically* be arbitrarily complex, most DDoS attacks today are still formed by well-defined traffic subsets [23, 28], thus being characterizable as "aggregates" [32]. We study some examples in Appendix A.

Is it possible to evade *ACC-Turbo*? We identify three techniques by which attackers could try to "outplay" *ACC-Turbo*. First, attackers may generate diverse packets, making it impossible for *ACC-Turbo* to identify any relation. Second, attackers could generate traffic to resemble production traffic, with the goal of triggering *ACC-Turbo* to deprioritize both the victim traffic and the attack. Finally, attackers could use $|C|$ spread-out attack vectors, so that each is mapped into a different cluster. While possible in theory, all approaches are difficult to execute in practice. Indeed, simply generating diverse traffic is not enough, since it has to still reach the target link. While specific scenarios may exist, execution requires a higher level of complexity than current DDoS attacks.

What is the impact of leaving *ACC-Turbo* always-on? Under sporadic congestion, the possible packet reorderings produced by *ACC-Turbo* could be happening without our solution, e.g. if the deprioritized packets suffered from longer delays on the network. In case of sustained congestion (not an attack), *ACC-Turbo* will deprioritize groups of packets with higher rates (heavy hitters) and give more priority to less aggressive groups of packets. The impact could be similar to a fair-queuing scheme, with the difference that the definition of a flow is inferred dynamically.

What about reordering? Assuming that the features used are common across all packets of a given flow, all these packets will be mapped into the same cluster. As such, reordering can only happen when the priority given to the flow's cluster increases over time while there are still packets of the same flow in the queue with the old (lower) priority. Considering that we update priorities in time periods of the order of milliseconds to seconds, potential reordering would only impact large flows, which anyway have high flow completion times.

What about interpretability? *ACC-Turbo*'s range-based clustering allows operators to know which packets are being mapped into each cluster, as well as the exact mapping of these clusters to the priority queues. Contrary to a black-box approach, an operator has access to the complete information of every action performed in real time. An operator could further modify the table entries in *ACC-Turbo* to reduce the number of priority queues to be used, to drop specific parts of traffic, or to treat some known-benign traffic preferentially.

Ethical issues. This work does not raise any ethical issues.

10 RELATED WORK

Given the novelty of pulse-wave DDoS attacks, just a few works have studied their mitigation [13]. Thus, in this section, we cover in-network defenses for conventional DDoS attacks, which are the closest related to our work. Another branch of attacks, called shrew or pulsing DDoS attacks, also leverage traffic pulses to disrupt the victim's connectivity [12]. However, they differ to pulse-wave DDoS attacks in that they are low-rate and target TCP vulnerabilities.

Conventional-DDoS defenses. Poseidon [55], Jaqen [31], and Ripple [51] use programmable switches to mitigate DDoS attacks. Poseidon proposes a system-level solution to orchestrate traffic to pre-defined defenses in dedicated servers. Jaqen and Ripple are both signature-based, and rely on a network-wide view of attack signals to deploy pre-configured defenses. *ACC-Turbo* runs autonomously in a switch (not requiring network orchestration nor switch reprogramming), achieves fast reaction times and covers unknown attacks. Bohatei [20] uses network function virtualization to adapt the placement, scale and the type of pre-defined defenses depending on the detected attacks. SPIFFY [25] detects link-flooding attacks by actively modifying the bandwidth available, and analyzing the traffic reaction. Kitsune [35] trains an ensemble of autoencoders to identify anomalies in network traffic. Euclid [18] uses statistical analysis to detect attacks from the network. While these solutions achieve high accuracy, they execute drastic mitigation policies such as filtering. *ACC-Turbo* uses programmable scheduling to mitigate attacks directly from the network with reduced collateral damage.

Scheduling-based DDoS defenses. DDoS-Shield [39] uses scheduling to mitigate DDoS attacks. While *ACC-Turbo* faces pulse-wave DDoS attacks on the network, DDoS-Shield targets application-layer attacks from the end-host. In [29], a DDoS defense is proposed which uses two priority queues, and assigns suspected flows to the low-priority queue. Suspected flows are identified based on the difference of harmonic means of the incoming packets. *ACC-Turbo* uses a broader number of priority queues to enable finer-grained decisions, and accommodates more complex DDoS attacks.

11 CONCLUSIONS

We presented *ACC-Turbo*, the *first* aggregate-based congestion control mechanism that mitigates pulse-wave DDoS attacks by running at line-rate on commodity hardware. *ACC-Turbo* leverages two key insights: online clustering—to effectively identify (possibly unknown) attack pulses; and programmable scheduling—to safely deprioritize traffic according to its maliciousness. We implemented *ACC-Turbo* in P4 and deployed it on programmable hardware. *ACC-Turbo* can mitigate pulse-wave DDoS attacks in almost real time.

REFERENCES

[1] 2017. Intel Tofino. http://barefootnetworks.com/products/brief-tofino/. (2017).

[2] 2018. Netbench. http://github.com/ndal-eth/netbench. (2018).

[3] 2019. Hidden threat of Pulse Wave DDoS attacks. https://ddos-guard.net/en/info/blog-detail/hidden-threat-of-pulse-wave-ddos-attacks. (2019).

[4] Link 11. 2020. The Evolution of DDoS Reflection Amplification Vectors: a Chronology. https://www.link11.com/en/blog/threat-landscape/ddos-reflection-amplification-vectors-chronology/. (2020).

[5] Akamai. 2018. Memcached-fueled 1.3 Tbps Attacks. https://blogs.akamai.com/2018/03/memcached-fueled-13-tbps-attacks.html. (2018).

[6] Akamai. 2020. Prolexic Routed: Protect Your Entire Application Infrastructure Against Large, Complex DDoS Attacks. https://www.akamai.com/uk/en/multimedia/documents/product-brief/prolexic-routed-product-brief.pdf. (2020).

[7] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J. Alex Halderman, Luca Invernizzi, Michalis Kallitsis, Deepak Kumar, Chaz Lever, Zane Ma, Joshua Mason, Damian Menscher, Chad Seaman, Nick Sullivan, Kurt Thomas, and Yi Zhou. 2017. Understanding the Mirai Botnet. In *USENIX Security*. Vancouver, Canada.

[8] Steinthor Bjarnason. 2018. DDoS Defences in the Terabit Era: Attack Trends, Carpet Bombing. https://blog.apnic.net/2018/12/04/ddos-defences-in-the-terabit-era-attack-trends-carpet-bombing/. (2018).

[9] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. 2014. P4: Programming Protocol-Independent Packet Processors. *ACM SIGCOMM CCR* (2014).

[10] CAIDA. 2018. Anonymized Traces from CAIDA Equinix NYC Internet Data Collection Monitor-B, 2018. https://www.caida.org/data/passive/trace_stats/nyc-B/2018/. (2018).

[11] Moses Charikar, Chandra Chekuri, Tomás Feder, and Rajeev Motwani. 2004. Incremental Clustering and Dynamic Information Retrieval. In *SIAM JoC*.

[12] Yu Chen, Kai Hwang, and Yu-Kwong Kwok. 2005. Collaborative Defense Against Periodic Shrew DDoS Attacks in Frequency Domain. *ACM TISSEC* (2005).

[13] Ilya V. Chugunkov, Leonid O. Fedorov, Bela Sh. Achmiz, and Zarina R. Sayfullina. 2018. Development of the Algorithm for Protection Against DDoS-attacks of Type Pulse Wave. In *IEEE EIConRus*.

[14] Catalin Cimpanu. 2019. 'Carpet-bombing' DDoS Attack Takes Down South African ISP for an Entire Day. https://www.zdnet.com/article/carpet-bombing-ddos-attack-takes-down-south-african-isp/. (2019).

[15] Catalin Cimpanu. 2020. FBI Warns of New DDoS Attack Vectors: CoAP, WS-DD, ARMS, and Jenkins. https://www.zdnet.com/article/fbi-warns-of-new-ddos-attack-vectors-coap-ws-dd-arms-and-jenkins/. (2020).

[16] Cloudflare. 2014. Technical Details Behind a 400Gbps NTP Amplification DDoS Attack. https://blog.cloudflare.com/technical-details-behind-a-400gbps-ntp-amplification-ddos-attack/. (2014).

[17] Cloudflare. 2017. Cloudflare Advanced DDoS Protection. https://www.cloudflare.com/media/pdf/cloudflare-whitepaper-ddos.pdf. (2017).

[18] Alexandre da Silveira Ilha, Ângelo Cardoso Lapolli, Jonatas Adilson Marques, and Luciano Paschoal Gaspary. 2020. Euclid: A Fully In-Network, P4-based Approach for Real-Time DDoS Attack Detection and Mitigation. *IEEE TNSM* (2020).

[19] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. 2015. Moongen: A Scriptable High-Speed Packet Generator. In *ACM IMC*. Tokyo, Japan.

[20] Seyed K Fayaz, Yoshiaki Tobioka, Vyas Sekar, and Michael Bailey. 2015. Bohatei: Flexible and Elastic DDoS Defense. In *USENIX Security*. Washington, DC, USA.

[21] Albert Gran Alcoz, Alexander Dietmüller, and Laurent Vanbever. 2020. SP-PIFO: Approximating Push-In First-Out Behaviors using Strict-Priority Queues. In *NSDI*.

[22] Sudipto Guha, Adam Meyerson, Nina Mishra, Rajeev Motwani, and Liadan O'Callaghan. 2003. Clustering Data Streams: Theory and Practice. In *IEEE TKDE*.

[23] Tiago Heinrich, Rafael R Obelheiro, and Carlos Alberto Maziero. 2021. New Kids on the DRDoS Block: Characterizing Multiprotocol and Carpet Bombing Attacks.. In *PAM*.

[24] Intel. [n. d.]. Tofino 3 Intelligent Fabric Processor. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-3-brief.html. ([n. d.]).

[25] Min Suk Kang, Virgil D Gligor, Vyas Sekar, et al. 2016. SPIFFY: Inducing Cost-Detectability Tradeoffs for Persistent Link-Flooding Attacks. In *NDSS Symposium*. San Diego, CA, USA.

[26] Simon Kassing, Asaf Valadarsky, Gal Shahaf, Michael Schapira, and Ankit Singla. 2017. Beyond Fat-trees Without Antennae, Mirrors, and Disco-balls. In *ACM SIGCOMM*. Los Angeles, CA, USA.

[27] Ali Kheradmand. 2020. Automatic Inference of High-Level Network Intents by Mining Forwarding Patterns. In *ACM SOSR*. San Jose, CA, USA.

[28] Oleg Kupreev, Ekaterina Badovskaya, and Alexander Gutnikov. 2020. Kaspersky DDoS reports. DDoS attacks in Q1 2020. https://securelist.com/ddos-attacks-in-q1-2020/96837/. (2020).

[29] Chu-Hsing Lin, Jung-Chun Liu, Hsun-Chi Huang, and Tsung-Che Yang. 2008. Using Adaptive Bandwidth Allocation Approach to Defend DDoS Attacks. In *IEEE MUE*. Busan, Korea.

[30] Thomas Lintemuth, Patrick Hevesi, and Sushil Aryal. 2020. Solution Comparison for DDoS Cloud Scrubbing Centers. In *Gartner*.

[31] Zaoxing Liu, Georgios Nikolaidis, Jeongkeun Lee, Changhoon Kim, Xin Jin, Vladimir Braverman, Minlan Yu, and Vyas Sekar. 2021. Jaqen: A High-Performance Switch-Native Approach for Detecting and Mitigating Volumetric DDoS Attacks with Programmable Switches. In *USENIX Security*. Virtual.

[32] Ratul Mahajan, Steven M. Bellovin, Sally Floyd, John Ioannidis, Vern Paxson, and Scott Shenker. 2002. Controlling High Bandwidth Aggregates in the Network. *SIGCOMM CCR* (2002).

[33] Christopher D. Manning, Hinrich Schütze, and Prabhakar Raghavan. 2008. Introduction to Information Retrieval (Chapter 16.3). In *Cambridge University Press*.

[34] Damian Menscher. 2019. Practical Solutions for Amplification Attacks. https://pc.nanog.org/static/published/meetings//NANOG76/daily/day_2.html. (2019).

[35] Yisroel Mirsky, Tomer Doitshman, Yuval Elovici, and Asaf Shabtai. 2018. Kitsune: An Ensemble of Autoencoders for Online Network Intrusion Detection. In *NDSS Symposium*. San Diego, CA, USA.

[36] Edgecore Networks. [n. d.]. 400 GbE Data Center Spine Switch Bare-Metal Hardware. https://stordirect.com/wp-content/uploads/woocommerce-products-data/product_documents/Edgecore_AS9516-32D_datasheet.pdf. ([n. d.]).

[37] Matthew Prince. 2013. The DDoS That Almost Broke the Internet. https://blog.cloudflare.com/the-ddos-that-almost-broke-the-internet/. (2013).

[38] Radware. 2021. DefensePro: Advanced DDoS Defense and Attack Mitigation. https://www.radware.com/products/defensepro/. (2021).

[39] Supranamaya Ranjan, Ram Swaminathan, Mustafa Uysal, and Edward W. Knightly. 2006. DDoS-Resilient Scheduling to Counter Application Layer Attacks Under Imperfect Detection. In *IEEE INFOCOM*. Barcelona, Spain.

[40] Tara Seals. 2017. Pulse-Wave DDoS Attacks Mark a New Tactic in Q2. https://www.infosecurity-magazine.com/news/pulsewave-ddos-attacks-mark-q2/. (2017).

[41] Iman Sharafaldin, Arash Habibi Lashkari, Saqib Hakak, and Ali A Ghorbani. 2019. Developing Realistic Distributed Denial of Service (DDoS) Attack Dataset and Taxonomy. In *ICCST*.

[42] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. 2016. Programmable Packet Scheduling at Line Rate. In *ACM SIGCOMM*. Florianopolis, Brazil.

[43] Jared M. Smith and Max Schuchard. 2018. Routing Around Congestion: Defeating DDoS Attacks and Adverse Network Conditions via Reactive BGP Routing. In *IEEE S&P*. San Francisco, CA, USA.

[44] Robin Sommer and Vern Paxson. 2010. Outside the Closed World: On Using Machine Learning for Network Intrusion Detection. In *IEEE S&P*. Oakland, CA, USA.

[45] Proton Team. 2015. Guide to DDoS Protection. https://protonmail.com/blog/ddos-protection-guide/. (2015).

[46] Proton Team. 2015. Message Regarding the ProtonMail DDoS Attacks. https://protonmail.com/blog/protonmail-ddos-attacks/. (2015).

[47] Alethea Toh. 2022. Azure DDoS Protection—2021 Q3 and Q4 DDoS attack trends. https://azure.microsoft.com/en-gb/blog/azure-ddos-protection-2021-q3-and-q4-ddos-attack-trends/. (2022).

[48] Muoi Tran, Min Suk Kang, Hsu-Chun Hsiao, Wei-Hsuan Chiang, Shu-Po Tung, and Yu-Su Wang. 2019. On the Feasibility of Rerouting-Based DDoS Defenses. In *IEEE S&P*. San Diego, CA, USA.

[49] Jai Vijayan. 2017. 'Pulse Wave' DDoS Attacks Emerge As New Threat. https://www.darkreading.com/attacks-breaches/pulse-wave-ddos-attacks-emerge-as-new-threat-/d/d-id/1329657. (2017).

[50] Ron Winward. 2017. Mirai Inside of an IoT Botnet. https://www.nanog.org/news-stories/nanog-tv/top-talks/mirai-inside-iot-botnet/. (2017).

[51] Jiarong Xing, Wenqing Wu, and Ang Chen. 2021. Ripple: A Programmable, Decentralized Link-Flooding Defense Against Adaptive Adversaries. In *USENIX Security*.

[52] Rui Xu and Don Wunsch. 2008. Clustering (Chapter 5.2.1). In *John Wiley & Sons*.

[53] Zhuolong Yu, Chuheng Hu, Jingfeng Wu, Xiao Sun, Vladimir Braverman, Mosharaf Chowdhury, Zhenhua Liu, and Xin Jin. 2021. Programmable Packet Scheduling with a Single Queue. In *ACM SIGCOMM*. Virtual Event, USA.

[54] Igal Zeifman. 2017. Attackers Use DDoS Pulses to Pin Down Multiple Targets. https://www.imperva.com/blog/pulse-wave-ddos-pins-down-multiple-targets/. (2017).

[55] Menghao Zhang, Guanyu Li, Shicheng Wang, Chang Liu, Ang Chen, Hongxin Hu, Guofei Gu, Qi Li, Mingwei Xu, and Jianping Wu. 2020. Poseidon: Mitigating Volumetric DDoS Attacks with Programmable Switches. In *NDSS Symposium*. San Diego, CA, USA.

A TODAY'S DDOS ARE STILL AGGREGATES

One may reasonably wonder whether (pulse-wave) DDoS attacks today still fit into the definition of aggregates. Undeniably, Internet traffic has evolved a lot in the last twenty years and DDoS attacks have become more intricate. While attack durations have drastically reduced [47], most attacks are still formed by well-defined traffic subsets, thus being characterizable by aggregates [23, 28].

For instance, the most famous botnet-based attack to date is probably the Mirai attack in 2016. Mirai included several flooding attacks such as UDP flood, SYN flood, ACK flood, or HTTP flood [7]. Each such attack generated highly-similar packets [50]: for instance, all SYN-flood packets had the same size, protocol, flags, and originated from the same IP subnets.

Amplification attacks, such as the NTP-based attack on Cloudflare (2014), DNS-based attack on Google (2017), or the Memcached-based attack against GitHub (2018), can be generally characterized by unusually-large packets sourcing from a specific port, using the same protocol, and originating from a common subset of IP subnets [5, 16].

Finally, previous examples of allegedly link-flooding attacks [37, 45] (defined as such in [25]) also fit our observation. Indeed, attackers employed (i) "DNS responses of 3000 bytes and TCP reflections targeting specific addresses of the victim IXP" [37], and (ii) "a number of amplification vectors such as NTP-reflection, UDP floods, TCP, and ICMP floods" [45].

B ACC'S PARAMETERS

Name	Definition	Value
K	Sustained-congestion period	$2s$
p_{high}	Sustained-congestion droprate	0.1
p_{target}	Target droprate	0.05
k	Exponential-moving-average interval for rate estimation	$0.1s$
Sessions	Maximum number of allowed rate-limiting sessions	5
Release Time	Minimum time required for an aggregate to be released after rate-limiting starts	$10s$
Free Time	Minimum time required for an aggregate to be released after it is detected to "behave"	$20s$
Cyc. Time	Time to revisit the aggregate	$5s$
Init. Time	Time to revisit the aggregate in the initial phase	$0.5s$

Table 4: List of the main ACC parameters, excluding RED and rate-limiting configurations.

C *ACC-TURBO* CLUSTERING ALGORITHM

Algorithm 1 ACC-Turbo Clustering Algorithm

Require: p: New packet, $\boldsymbol{min}$, $\boldsymbol{max}$: Initial ranges

procedure CLUSTERING
  for all p: incoming packet **do**
    for all $c_i \in C$ **do**
      $d(p, c_i) \leftarrow$ COMPUTEDISTANCE(p, c_i)
  $c_{selected} \leftarrow c_0$ ▷ Initialize selected cluster
  $d_{min} \leftarrow d(p, c_0)$ ▷ Initialize min. distance
  for all $c_i \in C,\ i \neq 0$ **do**
    if $d(p, c_i) < d_{min}$ **then**
      $d_{min} \leftarrow d(p, c_i)$
      $c_{selected} \leftarrow c_i$ ▷ Select cluster
  if $d(p, c_{selected}) > 0$ **then**
    $\boldsymbol{min}, \boldsymbol{max} \leftarrow$ UPDATECLUSTER$(p, c_{selected})$

function COMPUTEDISTANCE(p, c_i)
  $d(p, c_i) \leftarrow 0$ ▷ Initialize distance
  for $f \in \mathcal{F}$ **do** ▷ Iterate over all features
    $d_f(p, c_i) \leftarrow 0$
    if f is ordinal **then**
      if $p_f < min_f(c_i)$ **then**
        $d_f(p, c_i) \leftarrow min_f(c_i) - p_f$
      if $p_f > max_f(c_i)$ **then**
        $d_f(p, c_i) \leftarrow p_f - max_f(c_i)$
      $d(p, c_i)$ += $d_f(p, c_i)$ ▷ Aggregate distances
    else
      if $p_f \notin f(c_i)$ **then**
        $d_f(p, c_i) \leftarrow 1$
  return $d(p, c_i)$

function UPDATECLUSTER$(p, c_{selected})$
  for $f \in \mathcal{F}$ **do** ▷ Iterate over all features
    if f is ordinal **then** ▷ Update ranges
      if $p_f < min_f(c_{selected})$ **then**
        $min_f(c_{selected}) \leftarrow p_f$
      if $p_f > max_f(c_{selected})$ **then**
        $max_f(c_{selected}) \leftarrow p_f$
    else ▷ Update feature-value set
      if $p_f \notin f(c_{selected})$ **then**
        $f(c_{selected}) = f(c_{selected}) \cup p_f$