ALBUS: a Probabilistic Monitoring Algorithm to Counter Burst-Flood Attacks

Simon Scherrer[1], Jo Vliegen[2], Arish Sateesan[2], Hsu-Chun Hsiao[3], Nele Mentens[3,4], and Adrian Perrig[1]
[1]ETH Zurich [2]KU Leuven [3]National Taiwan University [4]Leiden University

***Abstract*—Modern DDoS defense systems rely on probabilistic monitoring algorithms to identify flows that exceed a volume threshold and should thus be penalized. Commonly, classic sketch algorithms are considered sufficiently accurate for usage in DDoS defense. However, as we show in this paper, these algorithms achieve poor detection accuracy under burst-flood attacks, i.e., volumetric DDoS attacks composed of a swarm of medium-rate sub-second traffic bursts. Under this challenging attack pattern, traditional sketch algorithms can only detect a high share of the attack bursts by incurring a large number of false positives.**

In this paper, we present ALBUS, a probabilistic monitoring algorithm that overcomes the inherent limitations of previous schemes: ALBUS is highly effective at detecting large bursts while reporting no legitimate flows, and therefore improves on prior work regarding both recall and precision. Besides improving accuracy, ALBUS scales to high traffic rates, which we demonstrate with an FPGA implementation, and is suitable for programmable switches, which we showcase with a P4 implementation.

I. Introduction

As distributed denial-of-service (DDoS) attacks continue to plague today's Internet infrastructure, recent research has produced a range of powerful DDoS defense systems. The most prominent examples of such systems include Poseidon [1], Ripple [2], Jaqen [3], COLIBRI [4], and ACC-Turbo [5]. Such DDoS defense systems provide data-plane functionality for *detection* of attack traffic and *mitigation* of the attack, where the defense specifics are configurable by the network operator. In the detection component, probabilistic monitoring algorithms (mostly the CountMin-Sketch [6] and the CountSketch [7]) monitor flows within limited memory, derive approximate flow-size estimates, and report flows that violate a volume threshold set by the network operator. In the mitigation component, the suspicious flows are then blocked, rate-limited, or deprioritized.

To provide effective defense, state-of-the-art DDoS defense systems rely on the accuracy of the built-in monitoring algorithms. This accuracy is essential to construct mitigations that are both comprehensive (restrict all attack traffic) and targeted (minimize the impact on benign traffic). However, we identify a class of attack patterns that disrupt the accuracy of the monitoring algorithms currently used in DDoS defense. In particular, we introduce *burst-flood attacks*, i.e., volumetric DDoS attacks composed of numerous simultaneous bursts, where each burst is sent in a different flow, lasts a few hundred milliseconds, and is only marginally larger than the natural bursts of benign flows (cf. Fig. 1). We demonstrate that these attacks lead to an ugly trade-off when configuring a threshold for common monitoring algorithms: Detecting a large share of attacker bursts comes at the cost of reporting flows that do not actually violate the allowance. In fact, the false reporting of flows can only be eliminated if almost no bursts are reported at all. This poor performance is linked to the regular resets which the sketch algorithms have to perform, as these resets conflict with the arbitrary timing and duration of bursts.

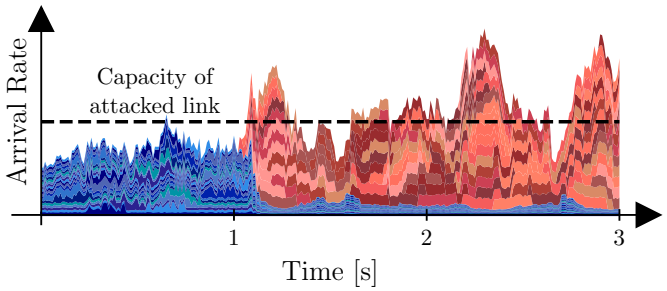


Fig. 1: Burst-flood attack: Multiple attacker flows (in red) send simultaneous bursts. The legitimate flows (in blue) experience packet loss and reduce their sending rate.

Based on these insights, we develop ALBUS (Adaptive Leaky-Bucket Undulation Sensor), a probabilistic monitoring algorithm that substantially improves detection accuracy under burst-flood attacks. Similar to previous algorithms, ALBUS monitors all flows in shared counters to limit memory consumption. Crucially, however, ALBUS does not directly derive flow-volume estimates from these counters (as sketch algorithms do), but only leverages these counters to continuously select the flows that are individually monitored by exact counters. As a result, ALBUS never reports flows that do not violate the configured volume allowance. At the same time, ALBUS consistently detects a high share of excessive bursts by avoiding resets and applying filtering techniques.

Thanks to its design, ALBUS is well-suited for integration into DDoS defense systems, which only allocate limited memory to their monitoring primitives and avoid expensive per-packet processing. We demonstrate ALBUS's suitability in three respects. First, we conduct an extensive accuracy evaluation to demonstrate that ALBUS generally outperforms previous monitoring algorithms in the scenario of interest, given limited memory. Second, we also confirm the processing efficiency of ALBUS with an implementation for a Xilinx FPGA, with a processing capacity of 200 million packets per second. Third, since many DDoS defense systems are implemented in P4 and run on programmable switches [1], [2], [3], [5], we also provide a P4 implementation to support the integration of ALBUS into existing systems.

In summary, this paper makes the following contributions:

Problem description. We characterize burst-flood attacks, demonstrate that commonly used monitoring algorithms provide insufficient defense against such attacks, and elicit requirements that a more effective algorithm must fulfill.

Algorithm design and evaluation. We design the ALBUS algorithm in line with the requirements of burst detection, and analyze the algorithm w.r.t. accuracy, security, and complexity. Our experiments confirm the detection accuracy, the attack robustness, and the processing efficiency.

Implementation. We implement ALBUS on a Xilinx FPGA, thereby demonstrating that the algorithm is suitable for high-speed packet processing, and in P4, thereby confirming the straightforward implementation of ALBUS on modern programmable switches.

II. Problem Statement

A. Background

Burst-flood attacks. Volumetric DDoS attacks attempt to exhaust network resources (e.g., links or servers) through the sheer number of requests from multiple attacker nodes, thereby denying service to users of the targeted resource. The exact structure of attacks can vary in three main ways. First, attacks differ in the network protocol leveraged for the attack, i.e., attack packets take the form various network protocols, e.g., NTP, DNS, ICMP, or bare UDP [8]. Second, attacks differ in the timing of aggregate attack traffic, e.g., pulse-wave DDoS attacks concentrate the attack traffic in bursts lasting a few seconds [9], [10], [5]. Third, attacks differ in the distribution of attack traffic across *flows*, i.e., in how the individual attacker nodes are scheduling and addressing their requests to create the desired aggregate attack traffic.

In this paper, we focus on the third aspect. More precisely, we remain agnostic regarding the content of the attack packets and the shape of the aggregate attack traffic. Instead, we are interested in how attackers can allot the attack traffic to flows in order to evade modern DDoS defense systems. Clearly, attacks using a low number of high-rate long-lived flows may likely be detected rapidly. Therefore, most DDoS attacks today rely on a large number of medium-rate short-lived flows, frequently created by reflection techniques [11], [12], [13]. For brevity, we refer to attacks with such a flow-size distribution as *burst-flood attacks* in this paper.

DDoS defense systems. Recent years have seen a series of proposals for DDoS defense systems, i.e., data-plane applications that try to maintain connectivity for legitimate traffic under DDoS attacks. Since such applications require high forwarding capacity (to process arriving traffic at line rate) and flexibility (to target the defense at the ongoing attack), modern programmable switches (e.g., Tofino [14]) have been a key enabler for most DDoS defense systems. In particular, Poseidon [1], Ripple [2], Jaqen [3] and ACC-Turbo [5] run on programmable switches to both detect and mitigate DDoS attacks. In the detection component, these DDoS defense systems try to identify the flows that are responsible for the excessive load, namely by subjecting flows (or other meaningful aggregates) to monitoring algorithms. In the mitigation component, the forwarding is adapted in order to block, rate-limit, or deprioritize the suspicious flows. Another approach is followed by COLIBRI [4], which is based on source authentication and flow-based reservations. Whenever a flow overuses its reservation, the flow is blocked.

Despite their differences, all these schemes refrain from deterministic per-flow monitoring algorithms (such as NetFlow [15]) to respect fast-memory limits. Instead, all schemes rely on *probabilistic* monitoring algorithms, i.e., algorithms that use limited memory, but only provide a rough estimate of the traffic volume associated with a flow. In particular, the classic CountMin-Sketch [6] and CountSketch [7] are widely used. In this paper, we investigate the effectiveness of these monitoring algorithms under burst-flood attacks, and find that these algorithms are vulnerable to such attacks (cf. Section III).

B. Problem Definition

Threat model. Fundamentally, we consider an adversary that tries to exhaust the capacity of a network resource with high-rate network traffic. We formalize traffic to be composed of *packets*, where each packet p arrives at a certain time t_p, has a certain size s_p, and can be assigned to a *flow* f_p based on properties from the packet header. As a result, the traffic volume sent by flow f in a time window $[t_1, t_2]$ is:

$$\sigma(f, t_1, t_2) = \sum\nolimits_{p \text{ s.t. } f_p = f \ \wedge \ t_p \in [t_1, t_2]} s_p. \tag{1}$$

In our scenario, the packets of each attack flow are concentrated in time, i.e., form a *burst*.

To formalize bursts, we note that a burst is commonly understood as a temporary increase of the sending rate compared to a base rate γ. Hence, if a flow f sends traffic volume $\sigma(f, t_1, t_2) > \gamma(t_2 - t_1)$ during a time window $[t_1, t_2]$, then we consider $b = \sigma(f, t_1, t_2) - \gamma(t_2 - t_1)$ the burst size. A threshold on the burst size b is thus always given in the form of a *flow specification* $\gamma t + \beta$, where $\beta \geq 0$ is the burstiness allowance. Concretely, if a flow sends more traffic than $\gamma(t_2 - t_1) + \beta$ during a time window $[t_1, t_2]$, we consider it *excessively bursty*. This formalization is the classic burstiness definition in networking, reflects queuing dynamics, and is independent of fixed time windows (unlike other burstiness definitions [16]). Moreover, it allows to distinguish excessive burstiness from moderate burstiness; the latter is a feature of almost all Internet traffic.

We extend this definition for our purpose: A burst of *width* w and *overuse ratio* ℓ is a packet sequence of a single flow f, spanning a time window of length w and containing traffic volume $\gamma w + \ell\beta$, i.e., $\sigma(f, t, t + w) = \gamma w + \ell\beta$. The adversary can create bursts of arbitrary size and duration.

Objective. Our goal is to design a monitoring algorithm that reports a set B_r of bursts, which ideally should match the set B of all excessive bursts:

$$B = \{(f, t_1, t_2) \ \mid \ \sigma(f, t_1, t_2) > \gamma(t_2 - t_1) + \beta\}. \tag{2}$$

Hence, our algorithm can be used by DDoS defense systems, which configure monitoring primitives with a threshold.

Metrics. We measure reporting accuracy with two conventional metrics. First, *recall* is the share of allowance-violating bursts that are reported, i.e., $recall = |B_r \cap B|/|B|$. If a burst exceeds the allowance, but is not reported, it constitutes a *false negative* in our setting ($B \setminus B_r$). Second, *precision* is the share of allowance-violating bursts among all reported bursts, i.e., $precision = |B_r \cap B|/|B_r|$. If a burst does not exceed the allowance, but is reported, it constitutes a *false positive* in our context ($B_r \setminus B$). Precision and recall can also be combined into the F_1 score, which is 1 only if $B = B_r$, i.e., $F_1 = 2|B_r \cap B|/(|B_r| + |B|)$.

C. Targeted Deployment Setting

We design an algorithm to be included as a monitoring primitive in existing DDoS defense systems. These DDoS defense systems provide the following relevant functionality:

Flow definition. The exact definition of a flow is at the discretion of the DDoS defense system using our monitoring algorithm, and may depend on the ongoing attack. For example, the DDoS defense system might consider packets belonging to the same flow if they share the destination IP prefix (e.g., in UDP carpet bombing) or the full fivetuple (e.g., in UDP flood) [5]. The monitoring algorithm simply accepts the flow identifier that the incoming packets are tagged with.

Threshold definition. The monitoring algorithm is configured to report flows that exceed a *volume threshold*. This threshold should be defined such that the algorithm reports the exact set of excessively bursty flows. ALBUS accepts thresholds of the form (γ, β) directly matching the allowance, whereas threshold definition for previous algorithms is more involved (cf. §III). Hence, the notion of accuracy in this paper is based on *volume*, i.e., accuracy means reporting the *allowance-violating* flows. Accuracy in terms of *intention*, i.e., reporting *malicious* flows, is thus achieved with an appropriate threshold if malicious flows are identifiable by volume. DDoS defense systems may find such a threshold in various ways:

Signatures. Many DDoS defense systems [1], [2], [3] are based on descriptions of known attacks (signatures), and compare them to current observations. The threshold may be chosen such that the attack flows in the signature violate it.

Calibration with queries. Jaqen [3] allows the network operator to submit *queries*, and reports all flows in the current traffic that match the query, e.g., exceed a volume threshold. For ALBUS, a network operator may perform the following query-based calibration (when not under attack) to find suitable (γ, β): The operator can choose a base rate γ, submit a series of queries with increasing burstiness allowance β, and observe the decreasing share of reported flows. Then, the threshold (γ, β) is appropriate when the number of benign flows violating it has been reduced to an acceptable amount.

Reservations. COLIBRI [4] maintains bandwidth reservations for a guaranteed data rate for individual flows. This data rate then constitutes the allowed base rate γ. The burstiness allowance β is $\beta = (u-1)\gamma w$ if flows with excessive rate $u\gamma$, $u > 1$, are tolerated for at most w seconds.

D. Algorithm Requirements

Given the objective and intended usage of our algorithm as well as the shortcomings of previous algorithms, we identify the following algorithm requirements.

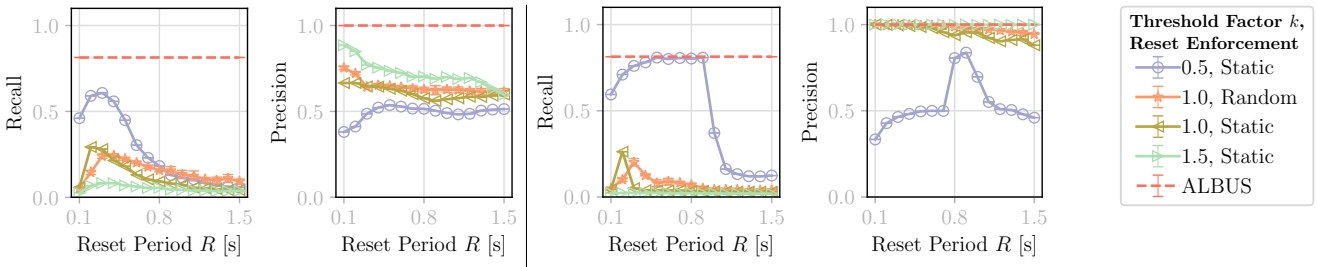
Processing efficiency. A DDoS attacker can maximize the attack damage by targeting network resources of systemic relevance, most importantly routers in the Internet core. Since such routers process billions of packets per second and handle millions of flows concurrently [17], any monitoring algorithm on these routers must conform to narrow constraints on the additional processing complexity. Since the required processing efficiency does not allow main-memory look-ups, processing efficiency also limits admissible memory consumption to the capacity of fast cache or SRAM memory. Hence, individual monitoring of every flow by dedicated counters (as in NetFlow [15]), or approaches that keep time-series data for each flow [18], [19], [20], [21] are impractical in the network core. Hence, DDoS defense systems use probabilistic flow-monitoring algorithms with acceptable memory consumption and low operational overhead per packet.

Accuracy. Despite limited memory, the monitoring algorithm should accurately identify bursts that violate the configured flow specification. In particular, the algorithm should achieve high *recall* and high *precision* simultaneously (cf. Section II-B). A high score in both metrics allows the DDoS defense system to identify and eradicate attack traffic, while limiting punitive actions to flows that actually violate the flow specification. For burst-flood attacks, high accuracy is especially challenging to achieve, because the aggregate attack traffic might be composed of bursts which only marginally exceed the burstiness allowance.

Time-window flexibility. From the experiment in Section III, we deduce that the reliance on discrete time windows constitutes a major impediment to detection accuracy. Attack bursts can be arbitrarily timed and arbitrarily wide, which poses an issue for the numerous monitoring algorithms that operate with a *landmark-window model* [22], [23], [24], [16], [25], [26], [6], [27], [7]. In the landmark-window model, algorithms split time into disjoint intervals and compare traffic measurements to the expectations for the interval. This interval duration is subject to an undesirable trade-off: If the intervals are too long, even a high-volume burst might not exceed the traffic volume allowed for the whole interval if followed by a sending pause; if the measurement intervals are too short, accuracy suffers as statistical noise grows stronger. Furthermore, bursts could be timed to arrive at the transition between two intervals, where measurements are reset. Hence, we aim to design an algorithm without regular resets.

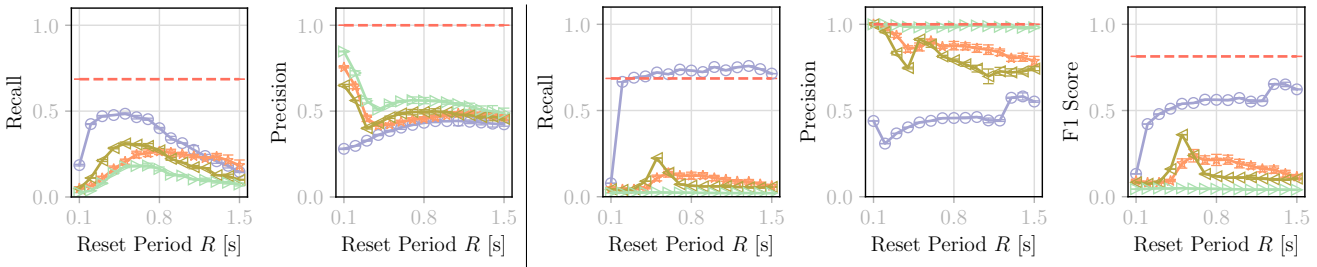
III. Why Another Monitoring Algorithm?

In this section, we evaluate the monitoring algorithms present in current DDoS defense systems under burst-flood attacks. In particular, we conduct a simulation-based experiment to evaluate CountMin-Sketch [6], which is a component of Poseidon [1] and Ripple [2], and CountSketch [7], which is a component of Jaqen [3] (through UnivMon [28]).



(a) CountSketch. (b) CountMin-Sketch.

Fig. 2: Detection performance under burst-flood attack (200ms bursts).



(a) CountSketch. (b) CountMin-Sketch.

Fig. 3: Detection performance under burst-flood attack (500ms bursts).

Experiment. We replay 5 seconds of a CAIDA trace and augment it with a burst flood amouting to 5 Gbps at any point in time. The attack bursts last $w = 200$ ms or $w = 500$ ms, and have an overuse ratio of $\ell = 1.2$ (Experiment details in Section VII-A). The goal of the algorithms (each obtaining 300 KB of memory) is to find all bursts that violate the flow specification of $\gamma = 1$ Mbps and $\beta = 50$ KB, both in the background traffic and the synthesized attack traffic. We measure both recall (share of detected bursts) and precision (share of correctly detected bursts among all reported bursts).

CountMin-Sketch. The CountMin-Sketch relies on a datastructure containing D counter arrays, where each array c_i is associated with a distinct hash function h_i. Upon arrival of a packet of flow f, the counter $h_i(f)$ is increased by the packet size in every counter array c_i. The estimate for a flow volume is then $v_f = \min_{i\in\{1,\dots,D\}} c_i[h_i(f)]$. If v_f exceeds a configured threshold T, the flow is reported. In the following experiment, we apply the CountMin-Sketch using *reset periods*, i.e., time intervals after which the data structure is reset. These reset periods are needed to regularly discard old information about past bursts. Given a reset period R, we consider *static* reset enforcement (used in current DDoS defense systems), where the time R' between resets always equals R, and *randomized* reset enforcement, where the time R' between resets is sampled uniformly at random from $(0, R]$ after every reset. We try to achieve the detection goal by varying the threshold T based on the current reset timer R' and a threshold factor k: $T = k{\cdot}(\gamma \cdot R' + \beta)$. Intuitively, the CountMin-Sketch would identify the attack bursts for $R' = w$ ms and $k = 1$ if the bursts were perfectly aligned with the reset periods; the threshold factor is used to account for temporal misalignment.

CountSketch. The CountSketch is an extension of the CountMin-Sketch. Specifically, the CountSketch contains a second hash function s_i for each counter array c_i, where s_i maps the flow ID f to -1 or $+1$. The counter $h_i(f)$ in array c_i is then increased or decreased by the packet size depending on the value of s_i. The estimate for the volume of flow f is $v_f = \text{median}_{i\in\{1,\dots,D\}}\ s_i(f){\cdot}c_i[h_i(f)]$. The threshold of the CountSketch is varied analogously to the CountMin-Sketch.

Results. Figures 2 and 3 show the experiment results for 200 ms and 500 ms bursts, respectively. From these figures, we make a number of observations. First, threshold factors that lead to good performance on recall lead to poor performance on precision, and vice versa. For example, a low threshold factor achieves high recall, but low precision. This trade-off suggests that the sketches, based on discrete time windows, fail to identify excessively bursty flows in a targeted manner. Second, precision and recall are unpredictably related to the reset period. Moreover, the optimal reset period in terms of a given metric varies between burst widths. Given that burst-flood attacks contain bursts of unknown and varying size, choosing an appropriate threshold is daunting. Third, the limitations of the sketches cannot be overcome by simply randomizing the duration between resets, as such randomization does not yield a clear improvement. Fourth, we have added the performance of our algorithm, ALBUS (ALBUS does not have reset periods). This addition demonstrates that alternative designs to sketches are promising; a detailed comparative evaluation of ALBUS is presented in Section VII.

IV. Design

In this section, we present ALBUS (Adaptive Leaky-Bucket Undulation Sensor), which is our approach to detect excessively bursty flows under the requirements from Section II-D.

Algorithm 1 Leaky-bucket algorithm

function UPDATE(Bucket λ, packet size s, timestamp t)
$d \leftarrow \gamma \cdot (t - \lambda.t)$
$\lambda.c \leftarrow \max(\lambda.c - d, 0) + s$ $\quad\quad$ $\lambda.t \leftarrow t$
if $\lambda.c > \beta$ **then return** True
else return False

As ALBUS is partially based on the leaky-bucket algorithm, we describe this algorithm in Section IV-A. In Section IV-B, we discuss the high-level ideas of ALBUS, and provide a detailed description of the algorithm in Section IV-C.

A. Background: Leaky Buckets

The leaky-bucket (LB) algorithm is a classic approach to rate control in networks, with a history of usage reaching back to ATM networks [29]. We revisit this algorithm for its properties which are perfectly in line with the special requirements of burst detection. In particular, the LB algorithm detects any violation of a flow specification of the from $\gamma t + \beta$ (cf. Section II-B), i.e., reports a flow if and only if the flow sends more traffic than $\gamma w + \beta$ during any time window with duration w. Hence, the LB algorithm by design fulfills all of the requirements from Section II-D, as it is perfectly accurate and flexible with respect to time windows.

Each LB λ contains a count $\lambda.c$ and a timestamp $\lambda.t$. The LB algorithm is akin to a physical leaky bucket, which is filled at a varying rate, leaks at a constant rate (unless it is empty), and overflows if the filling rate exceeds the draining rate for too long (cf. Algorithm 1). Note that the *draining volume* d is of special importance to ALBUS.

B. Design Idea of ALBUS

On a high level, ALBUS applies the LB algorithm to find excessive bursts. Given unlimited memory, each individual flow could be monitored with a dedicated LB. To respect memory constraints, however, we choose to monitor only a subset of flows at any point in time. The challenge thus becomes how to dynamically select the subset of flows individually monitored by dedicated LBs. We observe that this subset selection can be performed on a salient property of bursty flows, namely their *consistent burstiness*: As bursty flows send a large flow volume in a short time, all of their consecutive packets are sent in a bursty manner, i.e., with a rate above allowed rate γ. In contrast, the moderate burstiness of legitimate flows rules out that the allowed rate is persistently exceeded. Instead, moderate bursts are followed by under-utilization of the allowance γ after only a few packets.

This observation inspires the central criterion used for adapting the subset of LB-monitored flows (cf. Fig. 4). Given a set Λ of LBs, a flow is deterministically mapped to an LB λ by a hash on the flow ID f. If this LB λ is not assigned to any other flow, it can operate as a dedicated LB for flow f. Once assigned to an LB, flow f is monitored as long as the *net inflow* (packet size minus draining volume) to the LB is positive. As soon as the net inflow is negative, flow f is evicted from the LB, which can then monitor another flow.

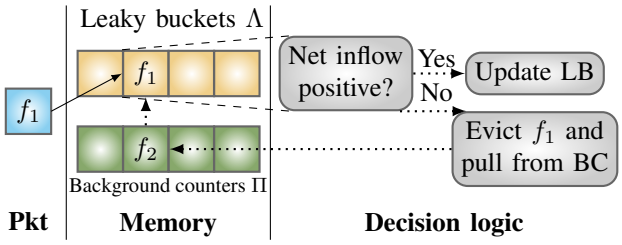


Fig. 4: Basic mechanism of ALBUS.

If reduced to this central mechanism, ALBUS would suffer from an issue in terms of both security and accuracy. In particular, even an extremely bursty flow would never be detected in case another flow already occupies the LB to which the bursty flow is mapped. An attacker could exploit this shortcoming with a *masking attack*: If an attacker can generate two flows that are mapped to the same LB, the attacker could start by sending a moderately bursty flow, thereby occupy the LB, and initiate the excessively bursty flow right afterwards. To mask the excessively bursty flow, the masking flow either has to be bursty within the allowance or not send any packets after it has been assigned to the LB. Regarding accuracy, the assignment of a bursty flow to a free LB is fairly unlikely, as the assignment probability roughly corresponds to the share of the pulsating flow among all packets mapped to a given LB. For a large number of flows or low overuse ratios, this probability may not suffice for swift detection.

To resolve this issue, ALBUS employs a set Π of *background counters* (BCs). Each BC $\pi \in \Pi$ is assigned to an LB $\lambda \in \Lambda$ and is tasked with finding the dominant flow among the flows that are mapped to, but not currently monitored by LB λ. This dominant flow is identified with a probabilistic-decay technique [27], [16]. When an LB is cleared, the flow from the corresponding BC is assigned to the LB directly, and is evicted from the BC. The background counters mitigate the masking attack with regular checks of the BC value. If this value exceeds a *push threshold*, the BC flow immediately replaces the corresponding LB flow. Regarding accuracy, this background counting increases the probability that a bursty flow is assigned to an LB, where it can then be identified as excessively bursty. In fact, background counting significantly boosts the probability of assigning the dominant flow to an LB, and thus the recall of ALBUS (cf. Appendix B).

Finally, ALBUS requires an additional safeguard to evict flows that finish sending while being monitored by an LB. To eventually evict these flows, the timestamp in LBs is regularly checked against a time-out.

C. Description of ALBUS

In the following, we present a more detailed account of the ALBUS algorithm based on Fig. 5. ALBUS operates with both an indexed set Λ of leaky buckets (LBs) and an indexed set Π of background counters (BCs), where the associated LB λ and BC π are at the same index i of the respective sets,

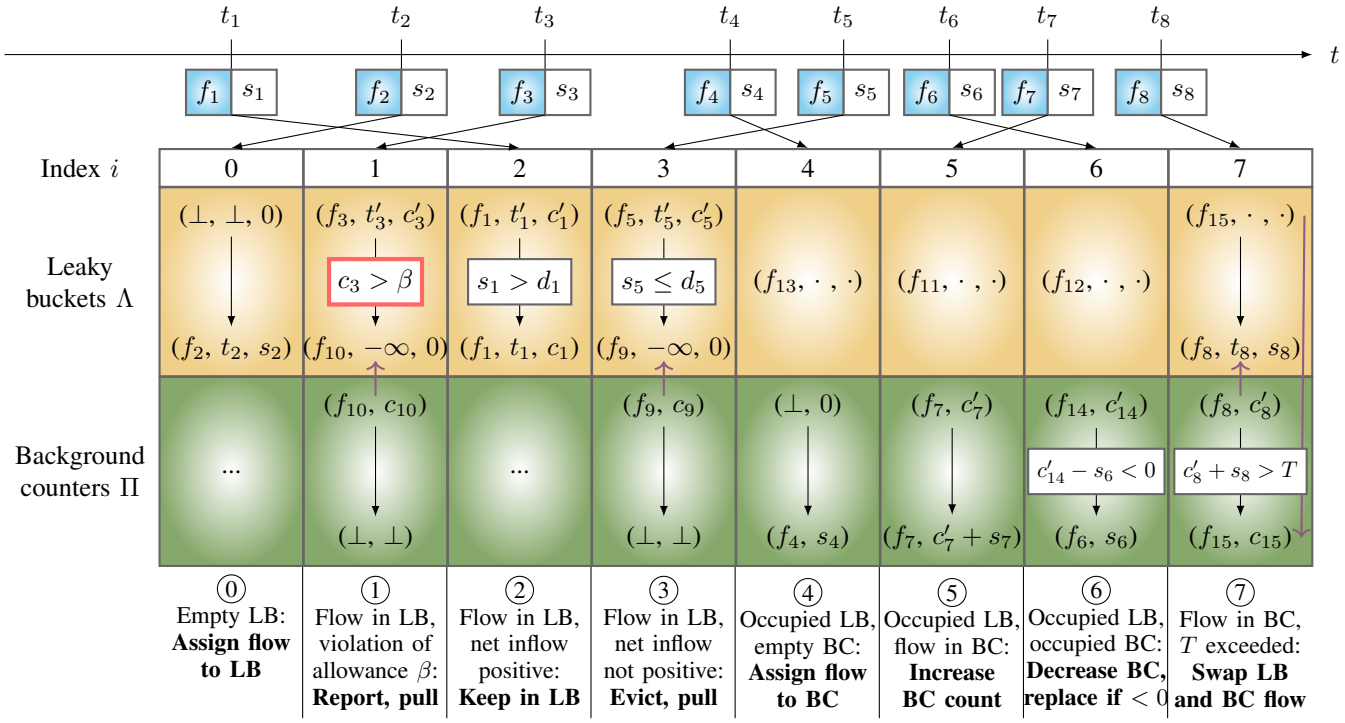


Fig. 5: Detailed overview of case distinctions in ALBUS (Discussion in Section IV-C).

i.e., $\lambda = \Lambda[i]$ and $\pi = \Pi[i]$. For any packet with flow ID f, size s and timestamp t, the index i is determined by a keyed hash function based on the flow ID f. After identifying the pair (λ, π), ALBUS performs the following case distinction.

If LB λ is still empty, flow f is assigned to λ by inserting its flow ID f into λ and setting the LB timestamp $\lambda.t$ to the packet timestamp t and the LB count $\lambda.c$ to the packet size s (cf. Fig. 5 ⓪). If LB λ already monitors flow f, then the leaky-bucket calculation is performed to obtain the drain volume d and the new LB count $\lambda.c$. If $\lambda.c$ exceeds the burstiness allowance β, flow f is reported and replaced by the flow occupying BC π (cf. Fig. 5 ①). If the burstiness allowance is not exceeded, flow $\lambda.f$ is left in LB λ (with updated timestamp $\lambda.t$) if the packet size s is larger than the drain volume d (Fig. 5 ②) or replaced by the flow from π otherwise (③). When pulling flow $\pi.f$ into LB λ, the LB timestamp $\lambda.t$ must be set to $-\infty$, as the timing of the last packet of $\pi.f$ is unknown and the flow must not be overestimated. BC π is cleared after replacement.

So far, we have only considered cases where an update of LB λ is directly possible. However, if another flow occupies LB λ, BC π must be accessed. If BC π is empty, flow f is assigned to π by setting $\pi.f$ to f and $\pi.c$ to packet size s (cf. Fig. 5 ④). If flow f already occupies BC π, the BC count $\pi.c$ is increased by packet size s (cf. Fig. 5 ⑤). If π is already occupied by another flow, ALBUS employs a probabilistic-decay technique: With a configurable probability 0.1^r, the BC count $\pi.c$ is decremented by the current packet size s. If $\pi.c$ is reduced below 0, the current flow f replaces flow $\pi.f$ in BC π (Fig. 5 ⑥). A high r slows BC decrements and BC flow replacements; hence, the parameter r corresponds to the *rigidity* of the background counters.

Finally, the algorithm contains two safeguards, namely safeguards against masking attacks and inactive flows. To counter masking attacks. the BC count $\pi.c$ is checked when increasing $\pi.c$. If this BC count exceeds a configurable threshold T, the current flow $\pi.f$ is swapped with the flow from the corresponding LB (cf. Fig. 5 ⑦). While LB counts and BC counts are not strictly comparable, ALBUS adopts the LB count as the BC count in order not to lose information (A swap in reverse direction might cause false positives). The safeguard against inactive flows corresponds to an occasional check of the LB timestamp $\lambda.t$, which is performed even if the flow ID f of the current packet is not currently assigned to λ. The safeguard procedure checks whether the LB timestamp predates the current time by more than a time-out, and if yes, replaces the flow $\lambda.f$ with the flow $\pi.f$ from the BC. This time-out is chosen such that the LB has certainly been fully drained since the last packet mapped to the LB was sent, i.e., the time-out duration is β/γ.

V. Analysis

A. Security Analysis

ALBUS detects excessively bursty flows which are assigned to an LB for a sufficiently long duration before the bursts ends. To evade detection. the sender of an excessively bursty flow must thus avoid or delay the LB assignment.

Single-flow masking. To completely avoid LB assignment, an adversary could launch a *masking attack*, i.e., occupy an LB with a flow and then send a large traffic burst in another

flow that is mapped to the same LB, speculating that the more damaging flow will not be inserted into the already occupied LB. Masking flows can be flows that either are bursty within acceptable limits (i.e., $\lambda.c < \beta$, but $s > d$ consistently) or do not send any packets while they occupy an LB (so the eviction condition is never evaluated). ALBUS counters such attacks in a two-fold manner. First, ALBUS provides the push-based transfer, i.e., overrides the LB flow with a flow that exceeds threshold T in the BC; hence, an excessively bursty flow might be monitored even if the masking flow is also bursty. Second, the time-out check prevents silent masking flows, as these flows would be quickly evicted.

Multi-flow masking. These safeguards might be circumvented if an attacker creates a high number of flows that map to the same BC as a bursty flow. Then, the probabilistic-decay mechanism might keep the BC count of the bursty flow low (i.e., no LB insertion by threshold violation) or even keep the flow out of the BC (i.e., no LB insertion on time-out). However, our experimental investigation in Section VII confirms that an appropriately configured BC maintains high recall even under a high number of simultaneous bursts.

Outside-LB flooding. Since the adversary is thus unable to keep the excessively bursty flows out of an LB indefinitely, most damage can be caused by sending excess traffic while the bursty flow is not yet assigned to an LB, i.e., if the flow is assigned to a BC or no counter at all. However, this attacker strategy is limited by the lower bounds on the probability of BC assignment and push transfers (Appendix A) Moreover, an attacker cannot observe when the malicious flow enters or exits a BC, and does not know whether the flow is already being monitored by an LB. Hence, the attacker bursts must be sized conservatively to avoid detection. Even if the adversary could pinpoint the exact time at which the flow is transferred from the BC to the LB, this time is brought forward as the attacker burst gets larger and background filtering is sped up.

Reset exploitation. A further weakness of previous algorithms is their reliance on regular resets. These resets allow an attacker to send a burst around the reset time. With this timing, the burst volume is split across two intervals such that the threshold is not exceeded in either interval. Since ALBUS does not rely on fixed time windows or resets, it is not susceptible to this evasion strategy.

Small-burst attack. If the adversary creates an enormous number of flows, each containing small bursts which are also expected from benign flows, it is fundamentally impossible for a volume-based algorithm to identify these attacker flows without also reporting benign flows. This fundamental impossibility applies to previous monitoring algorithms (Count- and CountMin-Sketch) as well as to ALBUS.

To counter such small-burst attacks, functionality provided by the DDoS defense system is required. Depending on the attack, the attacker flows might share some packet attributes which allow to combine the flows into a meaningful, large-volume aggregate. In that case, the DDoS defense system might apply the monitoring algorithm to measure aggregates instead of fivetuple-based flows. In the longer term, small-burst attacks might be effectively countered by DDoS defense systems that restrict flow creation. For example, COLIBRI [4] relies on source authentication and paid per-flow reservations, which limits the number of flows over which attack traffic can be distributed. Given this restriction, the adversary is forced to create medium-sized bursts in each flow, which are then again detectable.

B. Complexity Analysis

The following complexity analysis discusses memory consumption and processing overhead of ALBUS.

Memory consumption. The memory consumption of ALBUS is linear in the number $|\Lambda| = |\Pi|$ of LB-BC pairs, where each LB contains three fields (flow ID, timestamp, and count) and each BC contains two fields (flow ID and count). For a realistic traffic profile, each LB-BC pair can be represented by 16 bytes, which is justified in Appendix C.

This low memory consumption ensures high memory efficiency: Section VII-B suggests that a memory consumption of 300KB (on the order of usual L1/L2 cache sizes) allows ALBUS to outperform previous algorithms (when considering both recall and precision) on a 10 Gbps link [17].

Processing overhead. Unlike its competitor schemes, the processing overhead per packet in ALBUS is independent of detector memory. This independence stems from ALBUS's avoidance of list iterations, which are used in some monitoring algorithms such as BurstSketch (when populating or analyzing the Snapshot component [16]) and EARDet (when looking for an empty counter for flow insertion [30]). Moreover, unlike BurstSketch and EARDet, ALBUS does not require associative arrays (i.e., hashmaps), which are challenging to implement in a lightweight manner in hardware. Moreover, ALBUS requires exactly 1 hash operation, whereas other schemes like Count-Min Sketch [6], CountSketch [7], or BurstSketch [16] use multiple hash operations. Furthermore, the most complex arithmetic computation, namely the LB update, is only performed in the rare case where the flow is monitored by the LB. The lightweight update procedure thus allows high-speed processing, which we confirm with an FPGA implementation using 5 ns per packet (cf. Section VI).

VI. Implementation

First, we implemented ALBUS for an FPGA to demonstrate its suitability for high-speed packet processing (cf. Section VI-A). Second, we implemented ALBUS in P4, showing its suitability for programmable switches (cf. Section VI-B).

A. FPGA Implementation

To demonstrate that ALBUS allows efficient packet processing, we implemented ALBUS for a Xilinx FPGA.

Design. Figure 6 illustrates the design of our ALBUS FPGA implementation. The implementation starts by hashing the flow ID of an incoming packet (e.g., flow fivetuple) with the Xoodoo-NC hash function [31], which is a non-cryptographic hash function based on consecutive Xoodoo permutations [32]. From the 96-bit Xoodoo-NC digest, the

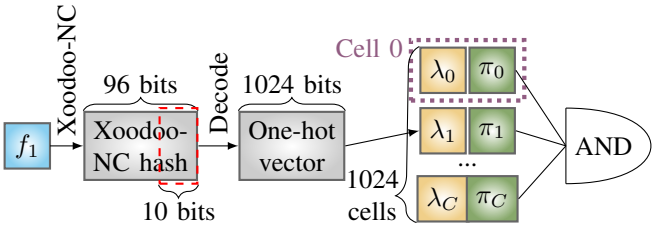


Fig. 6: FPGA implementation design ($C = 1024 - 1$).

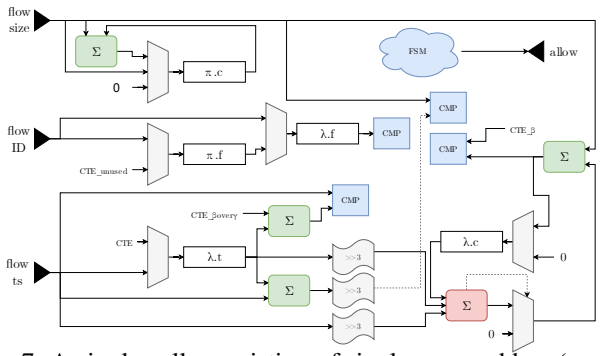


Fig. 7: A single cell, consisting of ripple-carry adders (green), a carry-save adder (red), and the decision-making logic (blue).

implementation decodes the 10 least significant bits into a one-hot vector of 1024 bits. In contrast to a decoding design, a single memory was used. Every binary-encoded address stores a corresponding one-hot encoded value. The single positive bit in this vector then triggers the activation of the corresponding *cell*, i.e., an LB-BC pair (cf. Figure 7).

Resource evaluation. We evaluate our implementation on a Xilinx Virtex UltraScale+ FPGA VCU118 [33] with a dual 112Gbps network interface. On this board, our implementation achieves a clock period of 5.0 nanoseconds, which corresponds to a packet-processing rate of 200 million packets per second for packets of 64 bytes. With these minimum-sized IP packets, this packet rate corresponds to a processing capacity of 102Gbps, which is compatible with the network interface of the FPGA. For a realistic traffic mix, such as the IMIX traffic distribution with an average packet size of 350 bytes [34], the packet rate would allow to process a traffic stream of 560 Gbps, but the network interface then constitutes the bottleneck. Given such a traffic profile, the forwarding switch could asynchronously provide the ALBUS FPGA with pure packet metadata, which amounts to maximum 39 bytes per packet or 62.4 Gbps of inbound traffic at the FPGA network interface.

The resource utilization of the implementation on the Xilinx FPGA amounts to 40% of available look-up tables (LUT), 20% of flip-flops (FF), and 2% of BRAM.

B. P4 Implementation

We created an additional implementation in P4 [35], which allows to embed complex functionality in programmable switches [14]. Our P4 code is available online [36] and can be tested using the virtual V1Model switch [37]. Appendix D presents a conceptual overview of the implementation design.

VII. Evaluation

To compare ALBUS to its competitor algorithms CountMin-Sketch, CountSketch, and BurstSketch under a variety of parameters, we perform a sensitivity analysis of all algorithms in Section VII-B. We also investigate the effectiveness of background filtering in terms of accuracy (Appendix B).

A. Evaluation Set-Up

For our evaluation, we use a simulation framework implemented in Go. This framework replays a CAIDA trace from a 10Gbps link [17] as background traffic. This trace is augmented with a simulated burst-flood attack such that the total rate exceeds the link capacity. These burst-flood attacks vary in the overuse ratio of bursts, the burst duration, and the number of bursts during the observation interval (5 seconds). We measure the detection accuracy of the monitoring algorithms by means of *recall* and *precision* (cf. Section II-B).

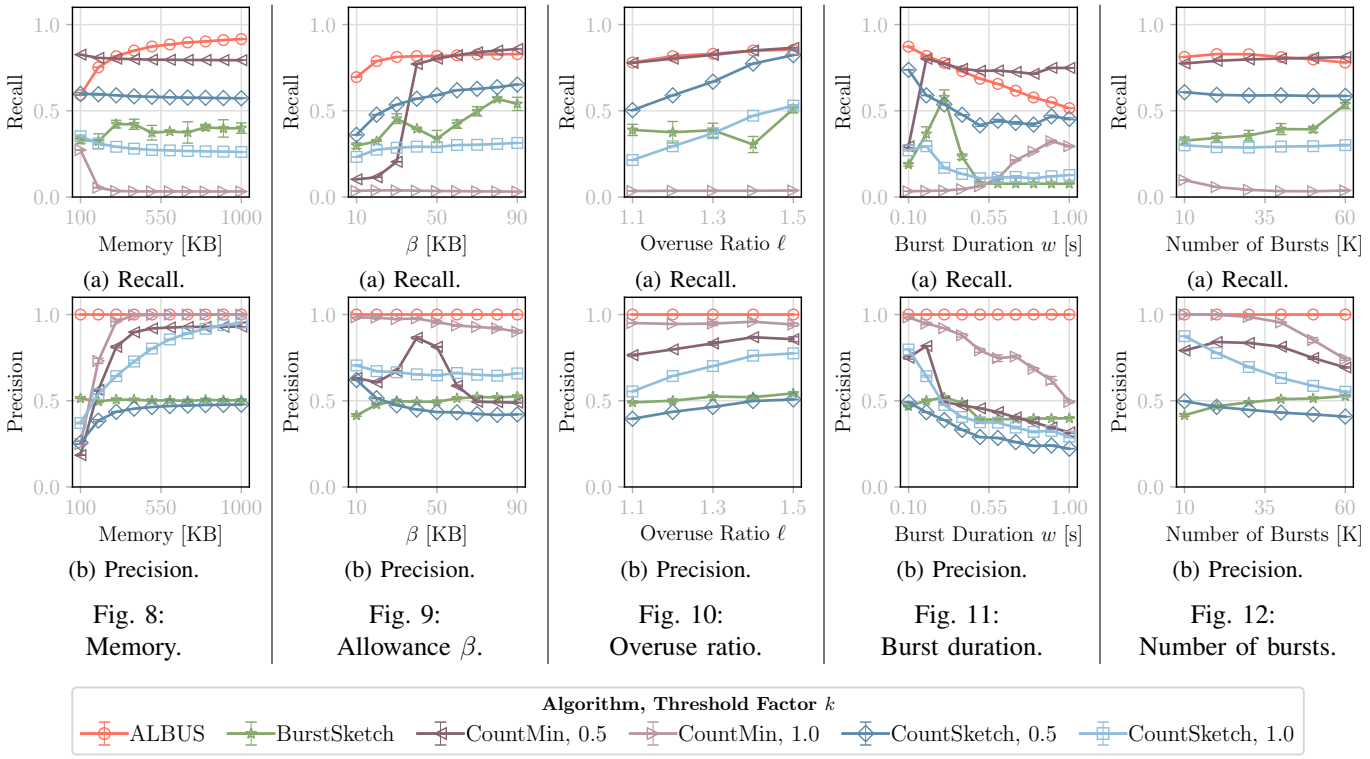
Every experiment in this section is run 6 times. However, since most results are highly stable and standard deviations are small, the error bars are barely visible.

B. Comparative Sensitivity Analysis

In this section, we investigate the detection accuracy of ALBUS in different scenarios, and simultaneously compare it to the monitoring algorithms used in DDoS defense systems, i.e., CountMin-Sketch [6] (used by Poseidon [1] and Ripple [2]), and the CountSketch [7] (used by Jaqen [3]). Since these algorithms are subject to a recall-precision trade-off due to their landmark-window model, they are evaluated for multiple threshold factors (cf. Section III). To distinguish versions of the algorithm with different threshold factors, we write *a(k)* for algorithm *a* with threshold factor *k*. In addition, we also evaluate BurstSketch [16], which is a specialized monitoring algorithm for burst detection.

In the comparative analysis, all detectors are equipped with the same amount of memory. The competitor schemes have been optimally according to the respective papers. ALBUS has been configured with an optimal rigidity of $r = 0$ and $T = 10\text{KB}$ (cf. Appendix B).

Base configuration. The starting point of our sensitivity analysis is a base configuration of experiment parameters. In the base configuration, every algorithm obtains a memory allocation of 300KB, which is on the order of modern L1/L2 cache sizes. All algorithms try to enforce a flow specification with rate $\gamma = 1$ Mbps and burstiness allowance $\beta = 50$ KB, allowing a flow to send at $1 + 0.4/w$ Mbps over w seconds. Only $\sim 1\%$ of background flows from the CAIDA trace violate this specification. The attack in the base configuration contains bursts with an overuse ratio of $\ell = 1.2$ and a burst width of 200 ms, i.e., bursts of rate 3.4 Mbps during 200 ms. To create an aggregate attack rate of around 5 Gbps over 5 seconds, 38,000 bursts are distributed across the observation interval uniformly at random. In the following, we discuss how



(a) Recall. (b) Precision.

Fig. 8: Memory.

(a) Recall. (b) Precision.

Fig. 9: Allowance β.

(a) Recall. (b) Precision.

Fig. 10: Overuse ratio.

(a) Recall. (b) Precision.

Fig. 11: Burst duration.

(a) Recall. (b) Precision.

Fig. 12: Number of bursts.

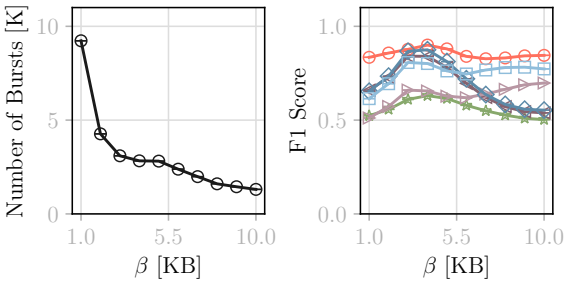
changes in a single parameter affect the detection accuracy of all monitoring algorithms.

Memory. Fig. 8 illustrates how recall and precision are affected as the memory allocation of the monitoring algorithms grows from 100 KB to 1 MB. While CountSketch and CountMin-Sketch can utilize the additional memory to increase precision, their recall surprisingly *decreases* with additional memory. We observe this effect because additional memory is added in the form of additional counters, and thus the number of flows sharing a counter decreases with increasing memory. A low number of flows per counter makes threshold violation less likely, decreasing recall and increasing precision. For BurstSketch, both recall and precision are consistently low, pointing to a mismatch between the detection capabilities of BurstSketch and the objective in the experiment. Finally, ALBUS benefits from additional memory in terms of recall, whereas it achieves perfect precision by design. From a memory allocation of 300KB, ALBUS is superior to all other algorithms regarding both recall and precision. Other algorithms can beat ALBUS in one metric only at the cost of bad performance in the other metric, e.g., at 100 KB, CountMin-Sketch(0.5) outperforms ALBUS regarding recall by 0.2, but only with a precision that is 5 times lower.

Flow specification. In Fig. 9, the burstiness allowance β of the configured flow specification is varied. Hence, this experiment shows how monitoring algorithms behave when the allowed burst volume β grows compared to the allowed base rate γ. In the experiment, recall increases with β for all monitoring algorithms, mostly because the bursts in the simulated attack are dimensioned relative to β, and thus become larger and easier to detect. Precision, however, generally decreases with β for CountSketch and CountMin-Sketch, because the number of false positives stays roughly constant whereas the number of true positives decreases (i.e., fewer specification-violating flows in the background traffic). In general, we observe that ALBUS is uniquely effective at enforcing tight flow specifications, i.e., a low burstiness allowance. Moreover, ALBUS benefits from loosening the specification, whereas the performance of competitor schemes under varying flow specifications is hard to predict.

Overuse ratio. Fig. 10 visualizes the detection performance of the algorithms under varying overuse ratios. Unsurprisingly, higher overuse ratios generally lead to higher recall because larger bursts are easier to detect. Since the number of correctly detected bursts grows compared to the number of incorrectly reported bursts, precision also rises together with the overuse ratio. ALBUS is on par with CountMin-Sketch(0.5) regarding recall, but is consistently more precise.

Burst width. The variation of burst width in Fig. 11 underlines that sketches are inflexible with respect to burst width, i.e., they have a distinct peak in recall around a certain burst width. In contrast, the decreasing recall of ALBUS for growing burst width is due to the constant overuse ratio: As the burst width grows, a constant excess burst volume is distributed over a longer time, lowering the burst rate to which ALBUS is sensitive. Nonetheless, the precision of ALBUS is stable for varying burst width, whereas precision starkly decreases for the sketch algorithms. The reason for this decrease is subtle: As bursts get longer, they more often share counters with other flows, increasing false positives.



(a) Excessive bursts. (b) F1 score.

Fig. 13: CIC-DDoS2019 trace.

Number of bursts. In Fig. 12, the number of bursts in the observation period is grown from 10,000 to 60,000, corresponding to a variation in aggregate attack rate from 1.36 Gbps to 8.16 Gbps. The recall for all detectors is mostly stable across the investigated range. Only BurstSketch achieves a higher recall for higher burst numbers; an inspection of the algorithm suggests that a higher number of flows increases the contention in the first-stage component of BurstSketch, which intensifies filtering such that the second-stage component is less congested and contains more truly large flows. Regarding precision, we again observe that the CountSketch and the CountMin-Sketch perform worse under high load, again because more intense counter sharing leads to more false positives. CountMin-Sketch(0.5), which is on par with ALBUS in terms of recall, is at least 20% less precise.

Attack type. After varying the burst-flood attack, we finally consider a completely different type of attack traffic by using the CIC-DDoS2019 dataset [8]. Specifically, we replay the excerpt of the dataset containing a UDP flood, and instruct all algorithms to detect excessively large bursts for a range of flow specifications. Since the aggregate rate in the CIC-DDoS dataset is far lower than in the CAIDA dataset, we investigate flow specifications for $\gamma = 0.1$ Mbps and $\beta \in [1\text{ KB}, 10\text{ KB}]$. The CIC-DDoS attack traffic is notably different from the burst-flood attacks considered before, as it contains only a few thousand bursts over 24 minutes (cf. Fig. 13a). Nonetheless, ALBUS consistently outperforms remaining algorithms in F_1 score (cf. Fig. 13b), demonstrating the effectiveness of ALBUS beyond burst-flood attacks.

Overall performance. In summary, we note that ALBUS outperforms its competitor algorithms over a wide range of scenarios, considering detector configurations, attack properties, and background traffic (cf. Appendix E). The improvement by ALBUS is especially strong for short bursts and tight flow specifications (i.e., low β compared to γ). While the sketch algorithms can be configured such that they are competitive with ALBUS in one metric, such a configuration leads to poor performance in the other metric. For example, CountMin-Sketch(0.5) is frequently competitive with ALBUS in terms of recall, but is consistently less precise. Conversely, CountMin-Sketch(1.0) achieves competitive precision, but inferior recall.

VIII. Related Work

In this section, we discuss previous probabilistic monitoring algorithms and evaluate their suitability for burst detection.

Since keeping track of each individual flow is impractical, numerous probabilistic flow-monitoring algorithms have been proposed. The previously mentioned CountMin-Sketch [6] (equivalent to multistage filters [22]) and Count-Sketch [7] are the most prominent such algorithms. However, as we have shown in Section III, these algorithms suffer from poor accuracy in the evaluated attacks because they rely on discrete time windows. While the inaccuracy can be partly remedied by retrospective sketch-analysis techniques such as SeqSketch [25], PR-Sketch [26], or LOFT [24], the issues of discrete time windows and false positives remain. Similar issues plague other sketches [38], [39], [16], and top-k detection schemes [23], [27]. EARDet [30] does not rely on resets and yields no false positives, but performs expensive list iterations for every packet. The monitoring algorithm in ACC-Turbo [5] is based on the assumption that attack flows can be correlated based on header information; we make no such assumption.

IX. Conclusion

In this work, we demonstrate that the sketch algorithms used in modern DDoS defense system provide poor detection accuracy under burst-flood attacks, as they fail to report the set of excessively bursty flows without false inclusions. For example, given a flood of bursts that last 500 ms and exceed the allowed burst volume by 20%, the CountMin-Sketch [6] detects either 75% of allowance-violating flows with a false-positive rate of 50%, or 1% of allowance-violating flows with a false-positive rate of 1%, depending on the configuration. The source of this problem is that sketch algorithms need to regularly reset their data structure, which conflicts with arbitrary timing and duration of bursts.

Our algorithm, ALBUS, considerably improves upon previous algorithms by continuously selecting the subset of flows that are precisely monitored within limited memory. Thanks to its reliance on the leaky-bucket algorithm, its avoidance of hard resets, and its leverage of filtering techniques, ALBUS does not falsely report any allowance-conforming flows, but maintains high recall. In our experiments, ALBUS frequently outperforms the other investigated algorithms in *both* precision and recall, and consistently outperforms them in one of these metrics. Regarding processing efficiency, ALBUS consistently avoids design primitives that prevent efficient hardware implementation, and thus enables an FPGA implementation which can process 200 million packets per second. Thanks to its frugality, ALBUS can also be readily implemented in P4, in turn enabling simple integration with modern DDoS defense systems designed for programmable switches.

In summary, ALBUS represents an important complement to the monitoring algorithms in current DDoS defense systems, as ALBUS compensates the weaknesses of these algorithms under pessimal workloads. In future work, we will identify optimal combinations of these traditional monitoring algorithms with ALBUS to achieve comprehensive DDoS mitigation.

REFERENCES

[1] M. Zhang, G. Li, S. Wang, C. Liu, A. Chen, H. Hu, G. Gu, Q. Li, M. Xu, and J. Wu, "Poseidon: Mitigating volumetric DDoS attacks with programmable switches," in *the 27th Network and Distributed System Security Symposium (NDSS 2020)*, 2020.

[2] J. Xing, W. Wu, and A. Chen, "Ripple: A Programmable, Decentralized Link-Flooding Defense Against Adaptive Adversaries," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 3865–3881.

[3] Z. Liu, H. Namkung, G. Nikolaidis, J. Lee, C. Kim, X. Jin, V. Braverman, M. Yu, and V. Sekar, "Jaqen: A High-Performance Switch-Native approach for detecting and mitigating volumetric DDoS attacks with programmable switches," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 3829–3846.

[4] G. Giuliari, D. Roos, M. Wyss, J. A. Garcia-Pardo, M. Legner, and A. Perrig, "Colibri: A Cooperative Lightweight Inter-domain Bandwidth-Reservation Infrastructure," *Proceedings of ACM CoNEXT*, 2021.

[5] A. G. Alcoz, M. Strohmeier, V. Lenders, and L. Vanbever, "Aggregate-based congestion control for pulse-wave DDoS defense," in *Proceedings of the ACM SIGCOMM 2022 Conference*, 2022, pp. 693–706.

[6] G. Cormode and S. Muthukrishnan, "An Improved Data Stream Summary: The Count-Min Sketch and its Applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0196677403001913

[7] M. Charikar, K. Chen, and M. Farach-Colton, "Finding frequent items in data streams," in *International Colloquium on Automata, Languages, and Programming*, 2002.

[8] I. Sharafaldin, A. H. Lashkari, S. Hakak, and A. A. Ghorbani, "Developing realistic distributed denial of service (ddos) attack dataset and taxonomy," in *2019 International Carnahan Conference on Security Technology (ICCST)*. IEEE, 2019, pp. 1–8.

[9] DDoS-GUARD, "Hidden threat of Pulse Wave DDoS attacks," https://ddos-guard.net/en/info/blog-detail/hidden-threat-of-pulse-wave-ddos-attacks, 2019.

[10] I. Zeifman, "Attackers Use DDoS Pulses to Pin Down Multiple Targets," https://www.imperva.com/blog/pulse-wave-ddos-pins-down-multiple-targets/, 2017.

[11] R. Rasti, M. Murthy, N. Weaver, and V. Paxson, "Temporal lensing and its application in pulsing denial-of-service attacks," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 187–198.

[12] C. Rossow, "Amplification hell: Revisiting network protocols for DDoS abuse." in *NDSS*, 2014.

[13] H. Griffioen, K. Oosthoek, P. van der Knaap, and C. Doerr, "Scan, test, execute: Adversarial tactics in amplification DDoS attacks," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 940–954.

[14] A. Agrawal and C. Kim, "Intel Tofino2-A 12.9 Tbps P4-Programmable Ethernet Switch." in *Hot Chips Symposium*, 2020, pp. 1–32.

[15] B. Claise, "Cisco Systems NetFlow Services Export Version 9," RFC 3954 (Informational), Oct. 2004. [Online]. Available: http://www.ietf.org/rfc/rfc3954.txt

[16] Z. Zhong, S. Yan, Z. Li, D. Tan, T. Yang, and B. Cui, "Burstsketch: Finding bursts in data streams," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 2375–2383.

[17] CAIDA, "The CAIDA UCSD Anonymized Internet Traces - Oct. 18th." 2018. [Online]. Available: http://www.caida.org/data/passive/passive_dataset.xml

[18] Y. Chen and K. Hwang, "Collaborative detection and filtering of shrew DDoS attacks using spectral analysis," *Journal of Parallel and Distributed Computing*, vol. 66, no. 9, pp. 1137–1151, 2006.

[19] C. Fu, Q. Li, M. Shen, and K. Xu, "Realtime robust malicious traffic detection via frequency domain analysis," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 3431–3446.

[20] X. Luo, R. K. Chang *et al.*, "On a new class of pulsing denial-of-service attacks and the defense." in *NDSS*, 2005.

[21] C.-W. Chang, S. Lee, B. Lin, and J. Wang, "The taming of the shrew: Mitigating low-rate tcp-targeted attack," *IEEE Transactions on Network and Service Management*, vol. 7, no. 1, pp. 1–13, 2010.

[22] C. Estan and G. Varghese, "New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice," *ACM Transactions on Computer Systems*, vol. 21, no. 3, pp. 270–313, 2003.

[23] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, "Heavy-hitter detection entirely in the data plane," in *Proceedings of the Symposium on SDN Research*. ACM, 2017, pp. 164–176.

[24] S. Scherrer, C.-Y. Wu, Y.-H. Chiang, B. Rothenberger, D. E. Asoni, A. Sateesan, J. Vliegen, N. Mentens, H.-C. Hsiao, and A. Perrig, "Low-Rate Overuse Flow Tracer (LOFT): An Efficient and Scalable Algorithm for Detecting Overuse Flows," in *Proceedings of the Symposium on Reliable Distributed Systems (SRDS)*, 2021.

[25] Q. Huang, S. Sheng, X. Chen, Y. Bao, R. Zhang, Y. Xu, and G. Zhang, "Toward nearly-zero-error sketching via compressive sensing," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, 2021, pp. 1027–1044.

[26] S. Sheng, Q. Huang, S. Wang, and Y. Bao, "PR-Sketch: monitoring per-key aggregation of streaming data with nearly full accuracy," *Proceedings of the VLDB Endowment*, vol. 14, no. 10, pp. 1783–1796, 2021.

[27] T. Yang, H. Zhang, J. Li, J. Gong, S. Uhlig, S. Chen, and X. Li, "HeavyKeeper: An Accurate Algorithm for Finding Top-k Elephant Flows," *IEEE/ACM Transactions on Networking*, vol. 27, no. 5, pp. 1845–1858, 2019.

[28] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, "One sketch to rule them all: Rethinking network flow monitoring with UnivMon," in *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016, pp. 101–114.

[29] G. Niestegge, "The 'leaky bucket'policing method in the atm (asynchronous transfer mode) network," *International Journal of Digital & Analog Communication Systems*, vol. 3, no. 2, pp. 187–197, 1990.

[30] H. Wu, H.-C. Hsiao, and Y.-C. Hu, "Efficient large flow detection over arbitrary windows: An algorithm exact outside an ambiguity region," in *Proceedings of the 2014 Conference on Internet Measurement Conference (IMC)*. ACM, 2014, pp. 209–222.

[31] A. Sateesan, J. Vliegen, J. Daemen, and N. Mentens, "Novel Bloom filter algorithms and architectures for ultra-high-speed network security applications," in *2020 23rd Euromicro Conference on Digital System Design (DSD)*. IEEE, 2020, pp. 262–269.

[32] J. Daemen, S. Hoffert, G. Van Assche, and R. Van Keer, "Xoodoo cookbook." *IACR Cryptol. ePrint Arch.*, vol. 2018, p. 767, 2018.

[33] Xilinx, "Xilinx Virtex UltraScale+ FPGA VCU118 Evaluation Kit," https://www.xilinx.com/products/boards-and-kits/vcu118.html, 2021.

[34] C. P. Popoviciu, E. Levy-Abegnoli, and P. Grossetete, "Network Performance Considerations: Coexistence of IPv4 and IPv6," https://bit.ly/3BHWIIc, 2007.

[35] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[36] Anon., "ALBUS P4 Code," 2022. [Online]. Available: https://www.dropbox.com/s/0su0713hq6opkuy/albus_p4.zip?dl=0

[37] Barefoot Networks Inc., "V1Model Switch," 2021. [Online]. Available: https://github.com/p4lang/p4c/blob/main/p4include/v1model.p4

[38] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, "One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon," in *ACM SIGCOMM*, 2016.

[39] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, "Elastic sketch: Adaptive and fast network-wide measurements," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. ACM, 2018, pp. 561–575.

[40] R. S. Boyer and J. S. Moore, "Mjrty—a fast majority vote algorithm," in *Automated Reasoning*. Springer, 1991, pp. 105–117.

[41] A. Rao, A. Legout, Y.-s. Lim, D. Towsley, C. Barakat, and W. Dabbous, "Network characteristics of video streaming traffic," in *Proceedings of the seventh conference on emerging networking experiments and technologies*, 2011, pp. 1–12.

APPENDIX

A. Accuracy Analysis

The detection accuracy of ALBUS is measured by its ability to avoid false positives and false negatives. Since the LB algorithm is perfectly accurate in that sense, all inaccuracy stems from the dynamic selection of the LB-monitored flows.

False positives. ALBUS preserves the zero false positives of the LB algorithm. By definition from Section II-B, a flow violates a flow specification $\gamma t + \beta$ if there exists a time window with length w in which the flow volume exceeds $\gamma w + \beta$. If ALBUS reports a flow, this flow was monitored during a time window with length w' and had a volume of more than $\gamma w' + \beta$. Hence, since ALBUS reports a flow only if it encountered a time window in which the volume allowance is exceeded, ALBUS never reports flows which do not violate the flow specification, i.e., ALBUS has zero false positives.

False negatives. To formally characterize the occurrence of false negatives in ALBUS, we consider a burst of a flow f with width w, overuse ratio ℓ, and shape $v_f(t)$, a fluid approximation of the volume sent until time $t \in [0, w]$ after the burst start. Hence, it holds that $v_f(0) = 0$ and $v_f(w) = \gamma w + \ell\beta$. Moreover, we consider bursts with $v'_f(t) := d/dt\ v_f(t) \geq \gamma\ \forall t \in [0, w]$, as any burst can be decomposed into shorter bursts for which this property holds. As a result, it holds that $v(t) \geq \gamma t$ for all $t \in [0, w]$. If such a burst is assigned to an LB in ALBUS at time t', ALBUS observes the burst volume $v_f(w) - v_f(t')$, and reports the flow if the observed volume exceeds $\gamma(w - t') + \beta$. Hence, ALBUS detects the excessively bursty flow if the flow is assigned to an LB at time t', where

$$v_f(t') < \gamma t' + (\ell - 1)\beta. \tag{3}$$

Since $v'_f(t) > \gamma$ for all t, there exists a unique time $t^* \in (0, w)$ such that for all $t' \in [0, t^*)$, the detection condition in Eq. (3) holds. Hence, t^* denotes the time from which detection is not achieved anymore. ALBUS provides two options how a flow might be assigned to an LB before time t^*.

The first possibility for assigning the excessively bursty flow f to LB λ is *push-based*: If a time t' exists such that f is assigned to the BF π and the push threshold is exceeded by the BC count $\pi.c(t') > T$, flow f is inserted into LB λ. With the rigidity $r = 0$ used in our experiments, flow f can only reach such a high BC count if it outweighs the aggregate traffic volume of all other flows $g \in F_\pi$, $g \neq f$, mapped to π. Let $v_g(t')$ be the volume sent by flow g between the burst start of flow f (i.e., $t = 0$) and time t'. Importantly, the BC count $\pi.c(0)$ of the flow $g' := \pi.f(0)$ is subsumed into $v_{g'}$. In formal terms, the detection probability $p(f)$ of flow f is 1 (i.e., detection is guaranteed) under the following condition:

$$\begin{aligned} p(f) = 1 \iff \exists t' \in [0, t^*).\ v_f(t') > T + \sum_{\substack{g \in F_\pi. \\ g \neq f}} v_g(t') \\ =: P_{push}(f) \end{aligned} \tag{4}$$

Second, even if a push is not possible, detection may still be possible via the *pull-based* approach, i.e., if the LB λ is cleared at a time $t' \in [0, t^*)$ and flow f occupies BC π at that time, i.e., $\pi.f(t') = f$. To characterize the probability of $\pi.f(t') = f$, we need to consider all flows $g \in F_\pi$ mapped to BC π. Given such flow volumes $\{v_g(t)\}_{g \in F_\pi}$ for any t, we note that the probabilistic-decay technique is equivalent to the Majority algorithm [40] for rigidity $r = 0$, which we use in our experiments. This algorithm is guaranteed to find the majority item in arbitrary streams, if such an element exists. If no majority item exists, the algorithm outputs item f with a probability that corresponds to the volume of item f compared to the volume of all other items in the stream. Applied to our setting, the probability that the bursty flow f occupies BC π at time t' is given as follows:

$$\mathrm{P}\left[\pi.f(t') = f\right] = p_M(v_f(t'), t'), \tag{5}$$

$$\text{where } p_M(v, t') = \min\left(1,\ \frac{v}{\sum_{g \in F_\pi.\ g \neq f} v_g(t')}\right). \tag{6}$$

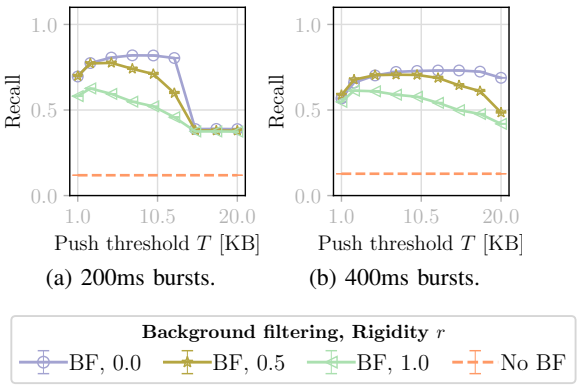
Notably, multiple LB-clearing moments may provide opportunity for a pull, i.e., the occupancy probability above is relevant at all moments $\{t'_i\}_{i \in \mathbb{N}, i \leq \tau}$, $t'_i \in [0, t^*)$, $\tau \geq 0$. The cumulative probability of LB pulls with subsequent detection is thus:

$$p_{pull}(f) = \sum_{\substack{i \in \mathbb{N} \\ i \leq \tau}} p_M(v_f(t'_i), t'_i) \prod_{\substack{j \in \mathbb{N} \\ j < i}} (1 - p_M(v_f(t'_j), t'_j)) \tag{7}$$

In summary, the detection probability $p(f)$ is given by:

$$p(f) = \begin{cases} 1 & \text{if } P_{push}(f), \\ p_{pull}(f) & \text{otherwise,} \end{cases} \tag{8}$$

which is only zero if $\neg P_{push}(f)$ and $\tau = 0$ (i.e., the LB λ is never cleared). Therefore, even if the adversary manages to prevent LB clearings with a masking attack, the attack damage in this scenario is bounded because P_{push} becomes true for high enough attack rates.



(a) 200ms bursts. (b) 400ms bursts.

Fig. 14: Background-filtering evaluation.

B. Evaluation of Background Filtering

In this section, we test the background filter on its effects on detector accuracy. Apart from its safeguard function, the background filter also aims at singling out bursty flows that are worth monitoring for the leaky buckets. To test whether

the background filter lives up to this expectation, we repeat the base-configuration experiment from Section VII-B, but vary the background0filtering configuration. In particular, we evaluate ALBUS with and without background filtering, with varying rigidity r and with varying push threshold T. For the version without background filtering, the memory formerly allocated to background counters is instead allocated to additional leaky buckets to hold the total memory allocation constant. The recall results of the evaluation are shown in Fig. 14 (Perfect precision is consistently achieved by design).

Background filtering in general. We observe that background filtering boosts recall considerably. All investigated background-filtering configurations achieve substantially higher recall than ALBUS without background filtering. Hence, the background filter is a vital part of ALBUS.

Push threshold. The results confirm the expected trade-off: If the push threshold is too high, recall decreases because bursty flows fail to evict the current LB-monitored flow. Conversely, if the push threshold is too low, flows evict each other from the LB before any bursty flow is detected, also harming recall. Moreover, both the optimal T and the recall considerably depend on the rigidity r, where we have found the minimum rigidity $r = 0$ to be optimal. Given the varying burst width in Figs. 14a and 14b, we observe that the push threshold should be set low if short bursts are to be detected, e.g., not above 12 KB if bursts of duration 200 ms should be detected. However, we also observe that a low push threshold does not strongly compromise the ability of ALBUS to detect longer bursts. For example, if the push threshold is set to 10 KB to optimize recall of 200 ms bursts, the recall of 400 ms bursts is still near-optimal.

C. Memory Analysis

The following analysis justifies the claim of 16 bytes per LB-BC pair in Section V-B.

LB/BC flow ID. Empirical data suggests a concurrency of around 10 million flows for 1 Tbps of forwarding capacity [17], which implies that a field size of 3 bytes is sufficient to distinguish the expected number of flows.

LB timestamp. 4 bytes are sufficient to accommodate nanosecond timestamps with a timestamp-reset period of around 4.3 seconds. Given two timestamps in known order, the correct difference between the two timestamps can be calculated if these timestamps differ by less than the timestamp-reset period, even if an overflow takes place between the two timestamps. In a realistic environment, flows that send no packet for more than 4.3 seconds are not considered bursty and are evicted by the time-out safeguard before this maximum difference is exceeded.

LB count. The LB count field must be able to record the maximum burstiness volume β. If operating with a rate allowance of $\gamma = 1$ Mbps and tolerating flows that send at most 5 times rate γ during 100 milliseconds, the burstiness allowance β corresponds to 50 kilobytes, which can in turn be counted by fields of 2 bytes.

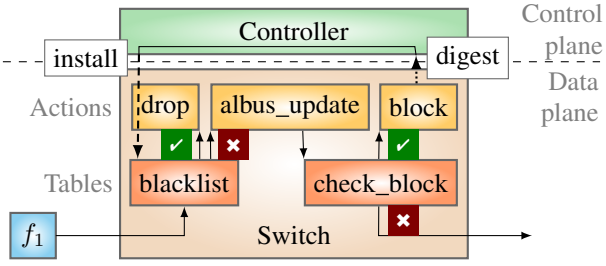


Fig. 15: P4 implementation design.

BC count. The BC count field must be able to record probabilistically decaying flow volume up to the push threshold T. This threshold is maximal in the (artificial) case $r = \infty$, where the BC count is never decremented. The threshold T in this maximum case limits the acceptable flow volume during the time w' in which another flow is in the LB. Assuming again that the algorithm targets flows sending at more than 5 times the allowed rate for $w = 100$ milliseconds, and that flows occupy a LB for at most 100ms ($w' = w$), a reasonable choice for the threshold T in the maximum case $r = \infty$ is $5\gamma w = 62.5$ kilobytes, which can also be counted with a 2-byte field.

In summary, 14 bytes are needed for an LB-BC combination, which we increase to 16 bytes for alignment. This low memory consumption ensures high memory efficiency: Section VII-B suggests that a memory consumption of 300KB (on the order of usual L1/L2 cache sizes) allows ALBUS to outperform previous algorithms (when considering both recall and precision) on a 10 Gbps link [17].

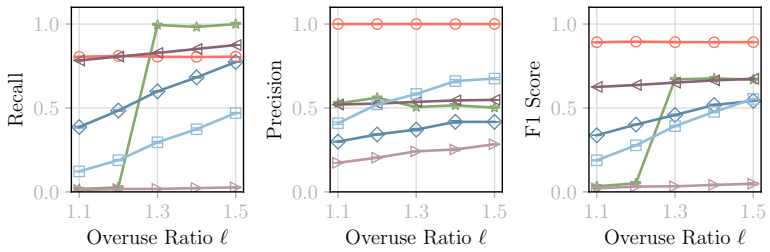
D. P4 Implementation Details

We created an additional implementation in P4 [35], which allows to embed complex functionality in programmable switches [14]. Our P4 code is available online [36] and can be tested using the virtual V1Model switch [37].

While ALBUS is meant to be used as a monitoring primitive within sophisticated DDoS defense systems (mostly also implemented in P4), we have created a stand-alone implementation of the algorithm for testing (cf. Fig. 15). First, an incoming packet is matched against the blacklist, which is embodied in a match-action table. If the blacklist matches the flow attributes in the packet, the packet is dropped; otherwise, the ALBUS algorithm is applied with an action containing the ALBUS control flow. If the packet flow is excessively bursty, the ALBUS update sets a metadata flag in the packet. If a subsequent match-action table matches this flag, blacklisting is triggered by sending the flow attributes to the controller via a digest operation (Digestion is not possible within the ALBUS update action). The controller then installs a rule including the malicious-flow attributes into the blacklist match-action table.

E. Evaluation on Synthetic Trace

In the CAIDA trace used in Section VII, most background flows are short and small, i.e., only consist of a few packets within one second. To evaluate a different scenario, we



(a) Recall. (b) Precision. (c) F1 score.

Fig. 16: Evaluation on synthetic background traffic.

synthesize background traffic with relatively large and long-lived background flows, i.e., each flow in the background traffic sends exactly at the allowed rate $\gamma = 1$ Mbps during the whole experiment. Such traffic can be present on links delivering a high amount of streamed video [41]. Using this background traffic, we again vary the overuse ratio of the bursts in the attack, leading to the results in Fig. 16. While the recall results are largely similar to Fig. 10a (same experiment with CAIDA traffic), BurstSketch has perfect recall on high overuse ratios, again because the long-lived background flows increase contention in the first stage of BurstSketch. ALBUS achieves a slightly lower recall than CountMin-Sketch(0.5) on high overuse ratios, but substantially outperforms it in both precision and F1 score, an aggregation of recall and precision (Fig. 16c).