aBBRate: Automating BBR Attack Exploration Using a Model-Based Approach

Anthony Peterson
Northeastern University

Endadul Hoque
Syracuse University

Samuel Jero
Purdue University

David Choffnes
Northeastern University

Cristina Nita-Rotaru
Northeastern University

Abstract

BBR is a new congestion control algorithm proposed by Google that builds a model of the network path consisting of its bottleneck bandwidth and RTT to govern its sending rate rather than packet loss (like CUBIC and many other popular congestion control algorithms). Loss-based congestion control has been shown to be vulnerable to acknowledgment manipulation attacks. However, no prior work has investigated how to design such attacks for BBR, nor how effective they are in practice. In this paper we systematically analyze the vulnerability of BBR to acknowledgement manipulation attacks. We create the first detailed BBR finite state machine and a novel algorithm for inferring its current BBR state at runtime by passively observing network traffic. We then adapt and apply a TCP fuzzer to the Linux TCP BBR v1.0 implementation. Our approach generated 30,297 attack strategies, of which 8,859 misled BBR about actual network conditions. From these, we identify 5 classes of attacks causing BBR to send faster, slower or stall. We also found that BBR is immune to acknowledgment burst, division and duplication attacks that were previously shown to be effective against loss-based congestion control such as TCP New Reno.

1 Introduction

BBR (Bottleneck Bandwidth and Round-trip propagation time) is a new congestion control algorithm for TCP [24] and QUIC [25] proposed by Google in 2016. BBR is motivated by how commonly deployed *loss-based* congestion control algorithms inaccurately rely on packet loss as the primary signal for network congestion, often leaving networks underutilized or highly congested. This inaccuracy occurs because in today's networks, the relationship between packet loss and network congestion has become disjoint due to varying switch buffer sizes. Instead, BBR is *model-based*, as it creates a model of the network by periodically estimating the available bottleneck bandwidth `BtlBw` and round-trip propagation delay `RTprop`, which are used to govern the rate packets are sent into the network and the maximum amount of data allowed in-transit.

Prior work [29, 30, 32, 36] showed how loss-based congestion control algorithms (*e.g.*, New Reno, CUBIC) designed for TCP are prone to *acknowledgment manipulation attacks*, where an adversary exploits the semantics of acknowledgments to mislead the sender (*i.e.*, the victim) of a flow about network congestion. These attacks are possible because TCP headers are unencrypted and have no authentication mechanism other than a random initial sequence number which may be observed or predicted by *on-path* [29] or *off-path* [7, 23, 35] attackers, respectively. While at first it may appear BBR is less prone to such attacks, as it relies on a different congestion control approach, its estimation of `BtlBw` and `RTprop` is based on received acknowledgments. The impact of such attacks can not be easily assessed from existing attacks against loss-based congestion control, because BBR follows a different algorithm for adjusting its sending rate. Given BBR is implemented for TCP [8], the underlying protocol for much of the Internet traffic, and being deployed on YouTube and Google.com [9], studying BBR security and its vulnerability to acknowledgment manipulation attacks is critical.

In this work, we discover and analyze acknowledgment manipulation attacks targeting the Linux TCP BBR congestion control implementation, a popular implementation of BBR. We use a protocol-fuzzing approach to systematically inject at runtime maliciously modified acknowledgment packets that target the core mechanism of BBR: the estimation of `BtlBw` and `RTprop`. In order to achieve this, we adapt TCPwn[1], a TCP congestion control protocol fuzzer, to automatically find vulnerabilities targeting BBR. TCPwn attack strategies are defined by tuplets that dictate which type of acknowledgment manipulation attack to execute when the sender is in a certain congestion control state. TCPwn uses the model of the congestion control algorithm to map all theoretically possible attack paths to actual attack strategies. It then uses a state inference algorithm by observing network traffic to discern when

[1] https://github.com/samueljero/TCPwn

to inject the counterfeit acknowledgments. Since TCPwn supports only loss-based congestion control algorithms, we derive a finite state machine for BBR by consulting documentation [9, 16, 17], presentations [10–15] and source code [8]. We additionally develop a new algorithm for inferring the current BBR state in real-time based on network traffic alone, and integrate it with TCPwn.

Using this approach, we automatically generated and executed 30,297 attack strategies from both *off-path and on-path attackers*, of which 8,859 caused BBR to send data at abnormal rates: 14 caused a faster sending rate, 4,025 caused a slower sending rate and 4,820 caused a stalled connection (*i.e.*, the flow did not complete). All of these successful attacks originated from an *on-path attacker* with read/write access to the flow. Attacks causing slower/stalled sending performance could be used by an adversary to throttle other flows—leading to poor performance for victim flows and possibly making more bandwidth available to the attacker's flows. Those causing faster sending performance could be used by a destructive adversary to increase network congestion, leading to unfairness, poor quality of service and congestion collapse. Attacks causing stall connections are a form of denial of service attacks difficult to detect as the connection is active and data is being sent, but with no progress for the flow itself. We summarize our contributions as follows:

- We derive the first *state machine model* for BBR and use it to demonstrate that BBR is vulnerable to acknowledgement manipulation attacks.
- We derive an algorithm for estimating the current BBR state in real-time by observing network traffic.
- We adapt a TCP congestion control fuzzer, TCPwn, to BBR using the our newly derived BBR state machine and inference algorithm to automatically generate and execute 30,297 automatically attack strategies.
- We identify 5 classes of acknowledgement manipulation attacks from *on-path attakers* against BBR that cause faster, slower and stalled sending rates. We did not find effective attacks from *off-path attackers*. To the best of our knowledge, we are the first to discover and evaluate attacks on BBR.
- We analyze how BBR distinctly reacts to these attacks, in comparison with other congestion control algorithms. We also found that BBR is immune to acknowledgment burst, division and duplication attacks that were previously shown to be effective against loss-based congestion control such as TCP New Reno.

2 Vulnerability of BBR to Attacks

We now describe BBR, derive a model for it, and show how an attacker can exploit the model to create attacks.

2.1 BBR Overview

BBR is motivated by how *loss-based* congestion control algorithms such as CUBIC and New Reno assume packet loss implies network congestion, which is not always the case. As a result, sending behavior is adjusted based on signals possibly unrelated to actual congestion, leading to network under utilization and excessive queue delay (bufferbloat).

Instead of relying solely on packet loss to infer congestion, BBR is *model-based* meaning congestion is inferred primarily by two properties of the network path: its bottleneck bandwidth `BtlBw` and round-trip propagation delay `RTprop`. BBR paces its sending rate proportionally to `BtlBw` and aims for at least one BDP = `BtlBw` × `RTprop` worth of data in-flight for full utilization. At any given time, its sending rate is limited by two factors: the congestion window `cwnd`, or `pacing_rate` = `pacing_gain` × `BtlBw` that defines inter-packet spacing. Pacing, first proposed by Zhang *et al.* [39], aims to reduce burstiness and in some situations, offers improved fairness and throughput [2]. BBR caps `cwnd` to 2 × BDP to overcome delays in received acknowledgments, which would otherwise cause BBR to underestimate the bottleneck bandwidth [10]. Obtaining an accurate and up-to-date model of the network path is essential to BBR's effectiveness, and thus is updated on every new acknowledgement.

Algorithm 1 Delivery rate samples [17] are computed to estimate the bottleneck bandwidth. For each new ACK, the average ACK rate is computed between when a data segment is sent to when an acknowledgment is explicitly received for it. Delivery rates are capped by the send rate as data should not arrive at the receiver faster than it is transmitted.

```
    Input: A data segment P and a BBR connection C.
    Output: The delivery rate sample
1: function ComputeDeliveryRateSample(P, C)
2:     data_acked = C.delivered - P.delivered
3:     ack_elapsed = C.delivered_time - P.delivered_time
4:     send_elapsed = P.sent_time - P.first_sent_time
5:     ack_rate = data_acked / ack_elapsed
6:     send_rate = data_acked / send_elapsed
7:     delivery_rate = min(ack_rate, send_rate)
8:     return delivery_rate
9: end function
```

2.2 Estimating the Network Path Model

Accurate measurements of `BtlBw` and `RTprop` are obtained sequentially at different times and network conditions because the network conditions required to obtain accurate measurements of each parameter interfere with each others measurements. At mutually exclusive times, BBR adjusts its sending rate so the network conditions are met for each parameter. For `BtlBw`, the sending rate is increased to discover available bottleneck bandwidth while for `RTprop`, the congestion window is reduced to 4 packets. Note that increasing sending rate to measure `BtlBw` may create queues which would create inaccurate `RTprop` measurements. Decreasing the sending rate to measure RTprop would not allow available bandwidth to be discovered.

Bottleneck Bandwidth. The bottleneck bandwidth is estimated by employing a max filter that retains the maximum observed delivery rate sample over the past 10 round-trips. A delivery rate sample is computed on each new ACK, which is shown in Algorithm 1. Delivery rate samples represent the average acknowledgment rate between when a data segment is transmitted to when an acknowledgment is received for that segment. Delivery rate samples are only computed for the exact packet it acknowledges. This is because factors such as delayed acknowledgment can cause delivery rates to be overestimated. These samples are used primarily to estimate the rate at which data is arriving at the receiver, which is naturally capped by the bottleneck bandwidth. Delivery rate samples only reflect the actual bottleneck bandwidth when BBR sends at a rate that matches or exceeds capacity, which is accomplished by periodically increasing its sending rate 25% faster than the current BtlBw.

Round-Trip Propagation Delay. BBR estimates RTprop using a min filter that retains the minimum observed round-trip time sample over the past 10 seconds. Round-trip samples are measured by computing the elapsed time between when a flight of data is sent to when it is acknowledged. Accurately measuring the RTT presents a challenge because packets queued in switch buffers cause increased and inaccurate RTT samples. To overcome this, BBR drains switch queues by periodically limiting its sending behavior. After measuring BtlBw, it is entirely possible the bottleneck is already saturated, causing queue build-up. To mitigate this, BBR sends 25% slower than the current estimated BtlBw immediately after measuring the bottleneck link. BBR also reduces cwnd to 4 packets every 10 seconds to update RTprop, which we describe in the following section in greater detail.

Rate Limiting. BBR attempts to detect token-bucket policers (TBPs) as they can cause data to be sent faster than the token drain rate, leading to high packet loss. In networks with such TBPs, it is common to see bursts of throughput before tokens are exhausted, after which packets are dropped. Due to BBR's long-lived BtlBw max filter, the burst rate would cause the estimated bottleneck bandwidth to be greater than the token drain rate, leading to high packet loss for as long as 10 round-trips. BBR detects TBPs when there is significant packet loss and consistent throughput, after which it paces its sending rate to the estimated token drain rate for 48 RTTs.

2.3 A State Machine for BBR

To systematically analyze BBR, we derive a finite-state machine (FSM) for it. To the best of our knowledge no such model has been published, so we empirically developed our own through documentation [9, 16, 17], presentations [10–15] and source code [8]. In Figure 1, we illustrate our BBR FSM and describe its variables and events in Table 1.

BBR employs several similar mechanisms to traditional congestion control algorithms, which we model in Appendix A.3. When a flow first begins, BBR uses a mechanism to quickly discover the available bandwidth (*i.e.* slow start). Afterwards, BBR paces its sending rate at the estimated bandwidth (*i.e.* congestion avoidance) while simultaneously probing the network for available bandwidth and updating its network path model: BtlBw and RTprop. Even though packet loss is not at BBR's core, BBR includes mechanisms to handle such cases. Finally, BBR includes methods for detecting and accounting for token-bucket policers, as they can allow traffic bursts until tokens run up, making BBR to send too quickly causing packet loss. The states of our BBR FSM are:

Startup. Similar to slow start, Startup is the first state BBR enters and aims to quickly discover the available bottleneck bandwidth by doubling its sending rate on each round-trip. Startup transitions into Drain when either cwnd reaches ssthresh or if three consecutive delivery rate samples show less than a 25% increases over the last, indicating the bottleneck bandwidth has been reached.

Drain. This state aims to drain queues that were likely created during Startup. Those queues are reduced in a single round-trip by sending data at $\ln(2)/2 \approx 0.34$ times the rate before entering this state, after which ProbeBW is entered.

ProbeBW. Similar to congestion avoidance, ProbeBW aims to pace the sending rate at the estimated bottleneck bandwidth, achieve fairness, and probe for additional bandwidth with low queuing delay. These are accomplished using gain cycling where the pacing_gain cycles through a set of eight phases: $[5/4,\ 3/4\ , 1, 1, 1, 1, 1, 1]$ where each phase lasts one RTprop. In the first phase, BBR sends 25% faster than BtlBw to probe for additional bandwidth. In the second phase, BBR sends 25% slower than BtlBw to drain any queues created in the last phase and to achieve fairness with other flows. In the remaining phases, BBR sends equal to BtlBw; the target operating point.

ProbeRTT. The goal of this state is to obtain a recent and accurate measurement of RTprop. Since queue delay increases the measured RTprop, ProbeRTT explicitly backs off from the network in order to drain any queues. This way, the min RTprop filter can capture a RTprop measurement without queues. ProbeRTT is entered if 10 seconds have elapsed since RTprop was last updated, and lasts for 200 ms; long enough to overlap with other flow's ProbeRTT states such that queues are fully drained.

Recovery. This state is entered when data has been lost and exits once all outstanding data when Recovery was entered has been acknowledged. Upon entry, cwnd is set to the amount of in-flight data and resets to $2 \times$ BDP upon exit.

Exponential Backoff. This state is entered upon a retransmission timeout indicating lost data due to no new acknowledgments for several RTTs. The lost segment is retransmited with a doubled timeout time; exponentially backing off from the network. Once an acknowledgment is re-

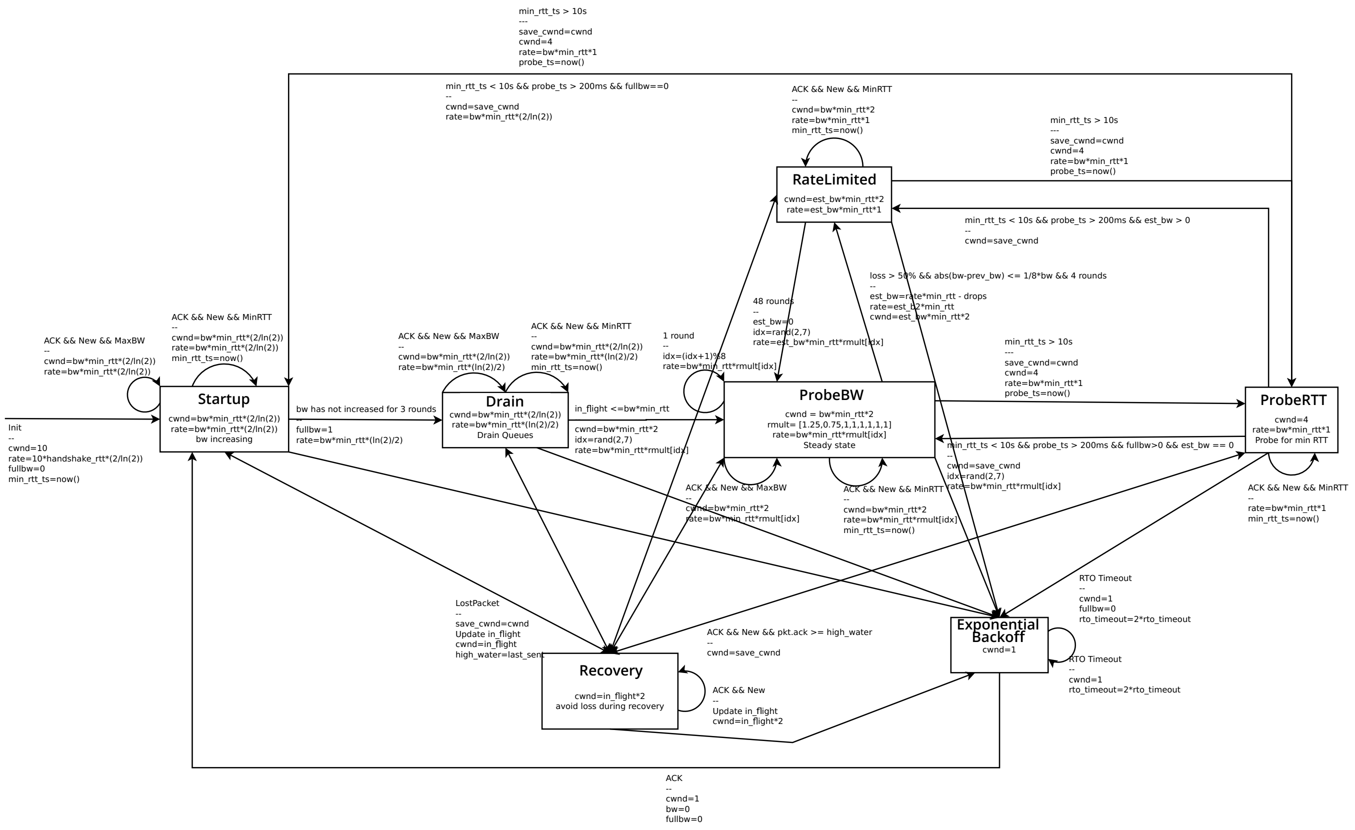


Figure 1: TCP BBR finite-state machine, see Table 1 for variable descriptions.

ceived, the current model is discarded and Startup is entered.

Rate Limited. This state is entered when a token-bucket policer is detected on the network, as these can lead to high amounts of packet loss. This state is entered when the packet loss-to-delivered ratio is greater than 20%, but the throughput remains steady. BBR sets `BtlBw` to the estimate token bucket drain rate and sustains this for 48 round-trips.

3 Automated Attack Exploration in BBR

In order to systematically examine vulnerabilities of TCP BBR implementation for the Linux kernel TCP stack [8], we apply a TCP congestion control fuzzer, TCPWN. Below we describe the attacker model and the changes we had to make to TCPWN in order to apply it to BBR.

3.1 Attacker Model

We focus on manipulation attacks in the implementation of BBR, where the attacker targets to mislead the sender's congestion control about the current network condition. These attacks are conducted through maliciously crafted acknowledgment packets, which can result in either increasing or decreasing the throughput of the target flow(s) and in stalled TCP connection.

We support the following acknowledgement-based malicious actions: ACK duplication, ACK stepping (several acknowledgments are dropped and then several let through in a cycle), ACK bursting (acknowledgments are sent in bursts), optimistic ACK (acknowledge highest byte, dropping duplicates), delayed ACK (delay acknowledgments for a fixed amount of time), limited ACK (prevent acknowledgment numbers from increasing), stretch ACK (forward only every nth ACK. injecting off-path duplicate acknowledgments, injecting off-path offset acknowledgments, and injecting off-path incrementing acknowledgments.

In order to achieve its goals, the attacker applies an *attack strategy*, which is defined as a sequence of acknowledgment-based malicious actions and the corresponding sender states when each action is conducted. We focus on TCP flows with bulk data transfers because they are widespread, and the effect of the conducted attacks is easy to measure.

We assume that the attacker is interested in causing BBR to send faster than usual, slower, or stall, and these attacks are meant to affect servers, clients, or the provider of a bottleneck link. In the case of sending faster, the goal of the attack can be to waste/exhaust bandwidth resources, worsening performance for all other clients of the server and/or shared bottleneck link. In the case of sending slower, the goal is to target individual connections for performance degrada-

Table 1: Descriptions of variables unique to the BBR finite-state machine (left), and its events (right).

Variable	Description
`bw`	maximum measured bottleneck bandwidth.
`bw_est`	estimated token-bucket drain rate.
`fullbw`	boolean indicating when pipe is filled.
`idx`	current index into `rmult`.
`min_rtt`	minimum measured RTT.
`min_rtt_ts`	timestamp `min_rtt` was measured.
`probe_ts`	timestamp ProbeRTT was entered.
`rate`	current pace data is sent.
`rmult`	array containing the 8 `pacing_gain` phases.

Event	Description
`ACK`	recipient of an acknowledgment packet, representing the highest correct byte received.
`MaxBW`	new maximum bottleneck bandwidth is observed.
`MinRTT`	new minimum RTT sample is observed.
`New`	new acknowledgment received, acknowledging previously outstanding data.
`LostPacket`	TCP packet loss event (3 duplicate ACKs).
`RTO Timeout`	outstanding data has not been acknowledged for many RTTs.

tion, which could selectively cause a service provider's quality to be poor (e.g., low resolution video streaming) and/or make more bottleneck bandwidth available for other competing flows. In stall attack, the goal is to disrupt communication between endpoints indefinitely, without causing an error from the transport layer to propagate to the application that is using it, effectively causing a denial of service.

3.2 Modifying TCPWN for BBR

We leverage TCPWN [29], a recent *open-source* platform designed to automatically find manipulation attacks in TCP congestion control implementations. We chose TCPWN because it does not require the source code of the congestion control implementation, and is designed specifically for TCP congestion control implementations.

At the core, TCPWN employs a network protocol fuzzer to find acknowledgment-based manipulation attacks against TCP congestion control implementations. Instead of applying random fuzzing, TCPWN guides the fuzzer using a model-guided technique, where the model is represented as a finite state machine (FSM) that captures the main functionality of several TCP congestion control algorithms.

For fuzzing an actual implementation of TCP congestion control in its native environment, TCPWN utilizes virtualization and proxy-based attack injection. To be effective, these attacks must be executed at the right time during execution, and therefore TCPWN monitors network packets exchanged to infer the current state of the sender in real-time.

While TCPWN is amenable to TCP congestion control algorithms, it assumes that the algorithm is a *loss-based* model based on TCP New Reno. Thus, we can not directly apply it to BBR, as the models are substantially different. We leverage our own BBR FSM (see Figure 1) to feed it as an input to TCPWN to generate *abstract* attack strategies, each of which specifies a vulnerable path in the FSM that the attacker can exploit. Each transition on a vulnerable path dictates the network condition that the attacker needs to trigger to mount the desired attack.

While there are several ways to trigger the necessary network conditions, TCPWN takes the abstract strategies and converts them into *concrete* attack strategies consisting of *basic acknowledgement-packet-level actions* (*e.g.*, send duplicate ACKs). During fuzzing, the attack injector applies these actions in particular states of the FSM. Although the generation of attack strategies is fully automated, TCPWN requires us to provide a manually crafted mapping between network conditions and basic actions because the mapping relies on domain knowledge about the underlying model (in our case, the BBR FSM).

Another change we had to make is changing the state inference algorithm. TCPWN needs to know what is the state of the sender in order to inject attacks in the states specified by the attack strategy. The state inference available in TCPWN cannot infer BBR's states because the algorithm expects the underlying model (*i.e.*, FSM) to be based on TCP New Reno. Hence, we develop a new state inference algorithm for BBR to infer the sender's state from network traffic alone (§ 3.3).

3.3 State Inference for BBR

We present a novel algorithm to infer the current state of the sender in real-time by passively observing network traffic, presented for completeness in Appendix. Our algorithm operates by computing flow metrics on each round-trip and comparing metrics across intervals to determine BBR's state. We compute metrics on each round-trip because BBR sustains a constant sending behavior for at least one round-trip.

When our algorithm starts, we begin a round-trip by recording the first data packet's sequence number and end when it is acknowledged. During round-trips, we collect flow metrics and compute average throughput, re-transmission count, number of data packets sent when the round completes. We then update the inferred BBR state on each new round by computing metrics across round-trips. For BBR, the most revealing metric about its current state is change in average throughput across round-trips. On each round, we compute the throughput *ratio* since the last round. For example, if the current and last rounds had average throughputs of 30 and 20 Mbit/sec respectively, then the *ratio* would equal 1.5.

We infer Startup if throughput has increase significantly since the last round-trip. Drain is primarily inferred if the current state is Startup and we notice throughput has not been increasing. ProbeBW is inferred if $1.4 > ratio > 0.6$, which allows ProbeBW to be inferred during phases 1 and 2 of gain cycling. We also infer ProbeBW when BBR transitions out of

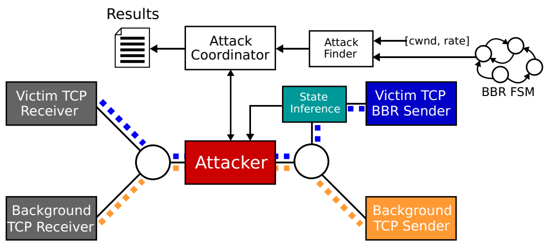


Figure 2: TCPWN testing environment.

Drain indicated by a significant increase in throughput. We infer ProbeRTT when we observe only 4-5 data packets in the last round, resulting from `cwnd` = 4 packets to drain queues, and exit after 10 data packets have been sent. Recovery is inferred when re-transmitted segments have been observed and exits when the highest data sequence (when Recovery was entered) is acknowledged. Exponential-backoff is inferred when the estimated RTO has elapsed since the last data packet. Lastly, we infer RateLimited when more than 16 round-trips have passed and there has been little variance in average throughput.

We infer Exponential-backoff, Recovery and Drain without waiting for a round-trip to complete, as these can be entered at anytime during the flow.

4 Experimental Results

In this section we describe and analyze our discovered attacks on BBR congestion control. We first describe the testing environment used. We then describe how we analyze and classified the attacks. Lastly, we discuss and illustrate the discovered attack classes in detail.

4.1 Experiment Setup

Environment. Our testing environment, shown in Figure 2, consists of four virtual machines running Ubuntu Linux 17.10 sharing a virtual dumbell network topology. We limit the bottleneck bandwidth to 100Mbits/sec with a 500 packet queue and a 10 ms. end-to-end latency between either end of the topology. We configure the virtual network with reasonably low latency and high bandwidth, allowing us to isolate the impact of attack strategies on BBR in a "friendly" environment.

For each attack strategy, two TCP flows are instantiated, a victim and background flow, each transferring an identical 100MB file over HTTP. The attacker is located between either end of the topology. The victim flow uses a Linux TCP BBR sender, whose flow is injected with attacks where the background flow is not targeted. We use `tcpdump` to measure both flow's performance, captured between the senders and the bottleneck. The victim flow is measured to understand the impact of the attack, and the background flow is measured to investigate the impact of competing flows during attack, *e.g.*, to determine if there was collateral damage.

Attacks in the wild. For an attack to be effective in the wild, an attacker will need to be able to recognize that TCP flows are indeed using BBR and to be able to be on the path (all the attacks we found were from on-path attackers). An attacker can determine that TCP is using BBR for congestion control by examining the startup phase of a target connection. Specifically, during startup BBR doubles its send rate every round, even with loss, until the bottleneck bandwidth estimate is relatively stable. So an on-path attacker could drop a single packet early in the connection startup to see if the TCP sender exits exponential growth. If so, this is probably loss-based congestion control, i.e. not BBR. Being on path can be accomplished by compromising a router along the path or inserting ones self into the path by ARP spoofing or similar attacks. Cross-traffic could also impede the effectiveness of the attack. We conducted a few experiments where we varied the number of cross-traffic flows in an attempt to gauge the impact of such traffic on our attacks. In particular, we varied the number of background CUBIC flows from 1 to 32 while executing each of our attacks. We repeated each of these scenarios 10 times and found that our attacks continue to be effective even with this significant level of background traffic (see Table 3).

Attack strategies. We used TCPWN to execute 30,297 attack strategies for manipulating BBR's sending rate. After each attack strategy is executed, it is classified into one of four categories: faster, slower, stalled (data transfer did not complete), and benign (no attack was detected). These categories represent the BBR's respectful *sending rate* performance. The attack categorization algorithm is the same as the one used in [29] and is included in Appendix B1.

Attack analysis. After all 30,297 attack strategies were executed, 8,859 were flagged as potential attacks: 14 faster, 4,025 slower and 4,820 stalled. We initially focused on extracting the strategies that were most effective at manipulating BBR's sending rate in each category. To identify which attack strategies were most effective at impacting BBR's performance, we grouped attack strategies in each category (ignoring attack specific parameters) and sorted each by average sending rate.

While this allowed us to understand which exact strategies were most effective, the limitation with this method is that it does not reveal if any subset of actions is more effective over others. Take for instance the following attack strategy that hypothetically affects sending rate performance:
`[(StateA,Action1),(StateB,Action2),(StateC,Action3)]`.
While it is true that this attack strategy affects sending rate performance, the above method does not indicate if performing Action2 in StateB was *necessary* for causing it. To find which actions were most effective, we generated all possible attack action subset combinations for each category and sorted them by their occurrence in the original attack strategies. This allowed us to see which attack actions the

Table 2: Descriptions of discovered attack classes targeting Linux TCP BBR congestion control.

No.	Attack	Attacker	Description	Result
1	Optimistic ACK	On/off-path	Acknowledge the highest sent sequence number before it is received, hiding all losses. This causes an overestimated `BtlBw` and for data to be send earlier/faster than otherwise.	Faster
2	Delayed ACK	On-path	Delay acknowledgments from reaching the sender from the receiver for a fixed amount of time, causing `BtlBw` to be underestimated and data to be sent at a slower pace.	Slower
3	Repeated RTO	On-path	For the entire flow, prevent new data from being acknowledged causing a RTO. Optimistically acknowledge the lost segment causing Startup to be entered. This causes substantial amounts of time to be wasted (not sending data) during periods before each RTO.	Slower
4	RTO stall	On-path	Prevent new data from ever being acknowledged causing a RTO and exponential backoff to never exit. This causes the connection to stall as no new data will ever be sent.	Stalled
5	Sequence Number Desync	On/off-path	Acknowledge the highest sent sequence number before it is received, hiding all losses causing sequence numbers to desynchronize. Induce RTO causing exponential backoff to be entered. For each re-transmission, the receiver replies with a lower acknowledgement number than the sender expects, causing the lost segment to never be acknowledged and exponential backoff to never exit.	Stalled

Table 3: Avg. throughput (in **Mbps**) of target BBR flow during attacks with varying numbers of CUBIC flows as cross-traffic

	Background Flows					
Attack	1	2	4	8	16	32
None	51.7	57.9	21.0	14.3	6.7	3.9
Attack 1	287.4	236.3	102.4	78.7	85.5	76.8
Attack 2	3.6	4.2	4.7	4.0	4.8	14.2
Attack 3	3.1	0.6	4.7	1.1	0.9	0.3
Attack 4	0.7	1.5	0.1	0.06	0.02	0.01
Attack 5	7.0	0.7	0.06	0.03	0.02	0.01

above method were most effective in each category.

For attack strategies causing faster send rates, the optimistic acknowledgment attack appeared in 100% of the strategies. For attack strategies causing slower send rates, executing the delayed acknowledgment attack in ProbeBW appeared in 53% of the strategies, while attacks causing RTOs and quickly acknowledging data in exponential backoff appeared in 47% of the strategies. As for the attack strategies causing a stalled connection, attacks causing RTOs and preventing data from being acknowledged in exponential backoff appeared in 89% of the strategies. Finally, attacks that optimistically acknowledged lost data appeared in 11% of the strategies.

We use time/sequence graphs (TSGs) in Figure 3 and 4 to understand why these attacks affect BBR's sending rate on a per-ACK basis. In each TSG, the *x*-axis is time and the *y*-axis is sent/acknowledged bytes from the BBR sender's (victim's) point-of-view. We use TSGs to also understand how BBR flows targeted by these attacks compare to benign flows (flows without attacks taking place) and how they affect background flows that share the network concurrently. Figure 5 shows the impact of executing each attack 100 times, using a CDF of the average send rate for the victim flow in each execution. Note that most curves are nearly vertical, indicating that each attack had a high probability of affecting the sending rate.

4.2 Discovered Attacks on BBR

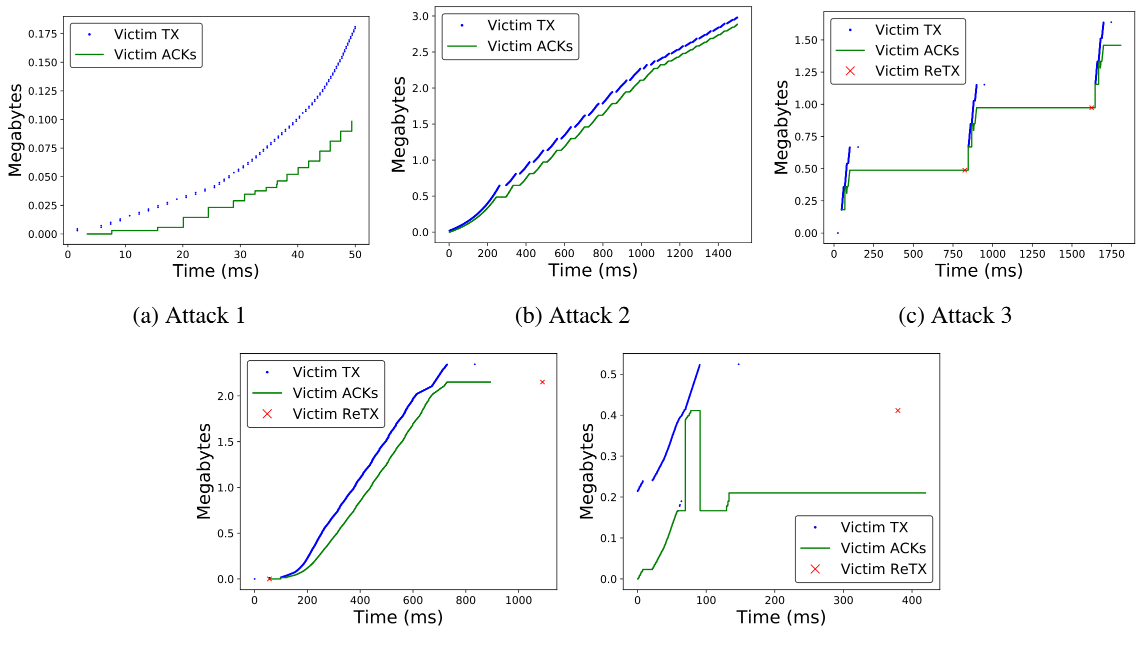
Attack 1 – Optimistic Acknowledgments. A faster sending rate is caused by optimistically acknowledging *only* new data sequences sent by the sender, without sending duplicates,

Table 4: The optimistic acknowledgment attack causes BBR to increase its sending rate by 25% every 8 round-trips. In this example, this attack effectively cuts the perceived RTT of 20 ms in half.

Time (ms)	Mbit/sec	Time (ms)	Mbit/sec
0	6.0	560	28.6
80	7.5	640	35.7
160	9.4	720	44.7
240	11.7	800	55.9
320	14.6	880	69.8
400	18.3	960	87.3
480	22.8	1040	109.1

effectively causing BBR to overestimate the bottleneck bandwidth. Figure C1a shows how this attack modifies acknowledgments from the receiver to mislead BBR about network conditions. The attacker records the highest observed data sequence number sent by the sender and modifies acknowledgment numbers from the receiver such that they acknowledge the highest sequence. If the modified acknowledgment would send a duplicate acknowledgment, then it is dropped. This results in packet loss (indicated by duplicate acknowledgments) to be hidden. An off-path attacker who is able to predict the sender's sequence number and the receiver's acknowledgment rate would be able to achieve the same affect on a victim flow by maliciously injecting acknowledgments such that they acknowledge new data sent by the sender.

A faster sending rate (see Figure 3a) is a byproduct of how BBR aggressively probes for additional bandwidth by sending 25% faster than `BtlBw` for 1/8 RTTs. This attack causes the acknowledgment rate to reflect the increased sending rate. This is reflected in the delivery rate samples (see Algorithm 1) causing the estimated bottleneck bandwidth to increase by 25% as well. BBR maintains the increased sending rate for the next 8 round-trips until it sends 25% faster again on the next probing phase. Surprisingly, we discovered that this attack alone does not cause the sending rate to increase. Even though this attack caused the acknowledgment rate to increase (due to a shortened `ack_elapsed`), delivery rate samples are capped by the sending rate. It is not until BBR probes for bandwidth when this attack becomes effective, meaning it is self-induced. This attack also halves `RTprop` because data is acknowledged sooner causing BBR to cycle through the 8 gain phases twice



(a) Attack 1 (b) Attack 2 (c) Attack 3

(d) Attack 4 (e) Attack 5

Figure 3: Time-sequence graphs illustrate how each attack manipulates acknowledgments to achieve a faster, slower or stalled sending rate. The blue lines represent data being sent by TCP BBR (victim) and the green represents acknowledgments being received from the attacker.

as fast. In our testing environment, this attack caused BBR to increase its sending rate from 6 Mbit/sec [11] to 800 Mbit/sec in less than 2 seconds! Table 4 shows how BBR's sending rate exponentially grows in our testing environment.

Attack 2 – Delayed Acknowledgments. A slower sending rate (see Figure 3b) is caused by delaying acknowledgment packets from reaching the sender for a fixed amount of time, causing the bottleneck bandwidth to be underestimated. Interestingly, this attack caused BBR's sending rate to grow inversely to the optimistic acknowledgments attack. Figure 6 shows how BBR's sending rate grows in $O(\frac{1}{n \cdot \ln(\text{delay})}) = O(\frac{1}{n})$ time (derivative of $\log_{\text{delay}} n$). In general, longer delays (that do not cause the re-transmission timer to expire) caused BBR to decrease its sending rate quicker. The amount of data sent over time is not to be confused with the rate of change of its sending rate, hence the derivation.

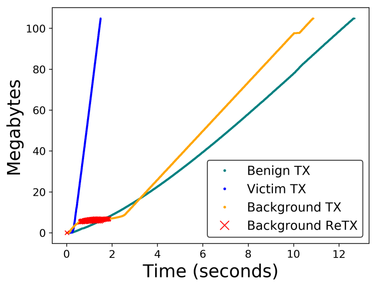
This attack causes BBR's sending rate to sequentially decrease over time because when this attack first starts, the sender experiences an initial delay in acknowledgments (equal to the however long ACKs are delayed for) after which acknowledgments arrive at their natural rate. This causes the sender to stop sending new data until the ACKs after the delay arrive. Since the attacker stops sending data, this causes the sender to experience another delay in acknowledgment packets after one RTT. This creates a pattern where the sender experiences delays in acknowledgments every RTT, meaning regardless when delivery rate samples are taken, the plateaus in acknowledgments cause the delivery rate samples to always be less than the current `BtlBw`. This implies BBR will never increase its sending rate because as older `BtlBw` estimates expire after 10 RTTs, it is replaced with the decreased delivery rate samples due to this attack.

Surprisingly, BBR's probing phase does not mitigate the effect this attack has on the sending rate. One would think that when BBR probes for bandwidth, the delivery rate samples computed during probing would be large enough to surpass the decreased delivery rate samples. This would be true, however because BBR drains queues immediately following probing, the delivery rate samples take during probing are still less than the current `BtlBw`. If BBR sent at a steady pace for at least 1 RTT in between probing and draining, then this attack would not result in its sending rate to decay.

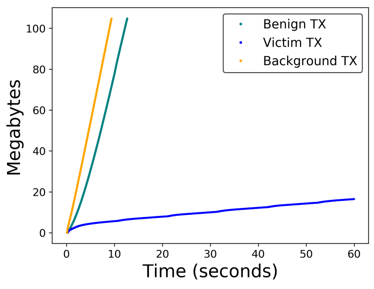
Figure 6 shows how this attack is less effective on congestion control that uses AIMD such as TCP New Reno. Although this attack is still effective on New Reno, its sending rate maintains a constant decreased rate rather than decaying.

It should be noted that this attack does not require TCP header information to be modified to be effective, as packets are only delayed. This implies QUIC [25], Google's experimental transport layer protocol that uses encrypted headers, using BBR can be targeted by this attack.

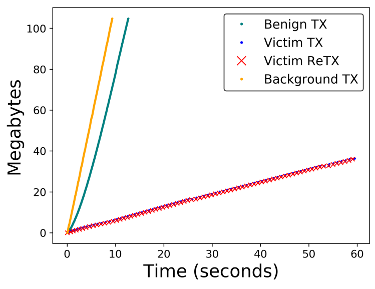
Attack 3 – Repeated RTO. A slower sending rate is caused by an attacker who allows small amounts of data to be sent in between repeated re-transmission timeouts. This causes a slower sending rate because the sender does not send any new data until the RTO expires, and this cycle repeats



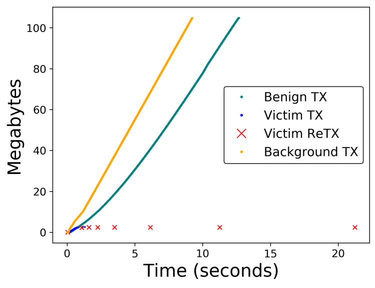
(a) Attack 1



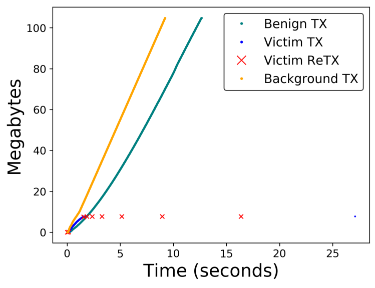
(b) Attack 2



(c) Attack 3



(d) Attack 4



(e) Attack 5

Figure 4: To understand how the victim flow for each attack compares to the background flow, these time-sequence graphs illustrate each attack is carried out during the entirety of a flow (100 MB transfer). For testing, we limit flows to 60 seconds. The orange and blue lines represent the victim and background flow's cumulative data transfer over time (executed concurrently). The green line represents the benign flow with no attacks taking place (executed as a separate experiment). In attack 1, the background flow in unable to obtain bandwidth until the victim flow completes. In attack 2, 3, 4 and 5, the background flow obtains greater bandwidth due to the victim sending slower or stalling.

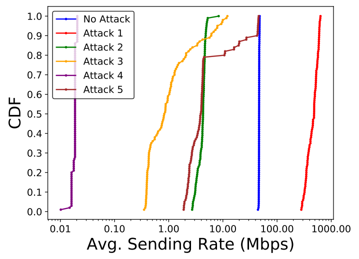


Figure 5: Each CDF represents the distribution of the avg. sending rate for each attack class executed 100 times. The victim BBR flow shares the network with an identical benign background CUBIC flow.

throughout the duration of the transmission.

This attack begins by causing the sender to re-transmission timeout, which is achieved by preventing new data from being acknowledged. There are four acknowledgment-based manipulation actions that were found to cause this: dropping, limiting, stretching and delaying acknowledgments. Dropping acknowledgments simply consists of preventing ACKs from being delivered to the sender. By limiting acknowledgments, acknowledgment numbers are such that they equal $min(ack,\ limit)$. Stretching acknowledgments consists of forwarding only every n^{th} acknowledgment to the receiver. Lastly, delaying acknowledgments (also used in attack 2) consists of delaying acknowledgments from reaching the sender.

After the re-transmission timeout is achieved from one of the above methods, the sender enters exponential backoff and the attacker optimistically acknowledges some data and repeats the process. The purpose here is to quickly acknowledge the lost segment in order to cause BBR enter Startup. When BBR transitions into Startup, its network path model (`BtlBw` and `RTprop`) is discarded, meaning the model must be rediscovered after each RTO. This attack prevents BBR from obtaining an opportunity to send data anywhere near the optimal operating point, resulting in decreased throughput. Instead, data is sent in bursts with lengthy idling in between.

Figure 3c illustrates these bursts and how the connection idles until the timeout. At 0 ms, BBR is in Startup, and the attacker begins to drop, limit, stretch or delay ACKs. As a result, the sender stops sending data because `in_flight` has reached `cwnd`. When the timeout occurs around 800 ms, the attacker optimistically acknowledges the lost segment, causing Startup to be reentered, after which the attack repeats. In Figure 4c, the victim flow can be seen experiencing timeouts throughout the entire flow (indicated by the red x-markers), allowing the background flow to obtain more bandwidth.

Interestingly, the work in [29] reported a similar attack resulted in a faster sending rate in several cases. They note

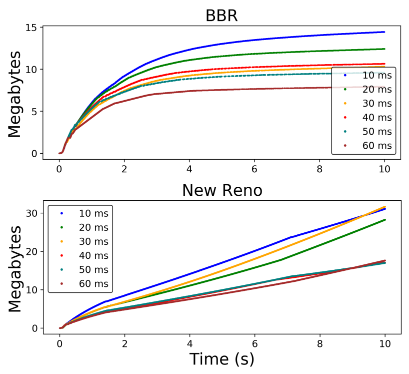


Figure 6: To understand how delayed ACKs affect the sending rate of BBR and New Reno, this figure illustrates several time-sequence graphs, each with a distinct delay time. Each line represents the cumulative amount of data sent during the connection. Due to different underlying techniques, delayed acknowledgments cause BBR's sending rate to decay over time, where New Reno maintains a rate.

how the idle periods in between timeouts is outweighed by repeatedly entering slow start (where `cwnd` doubles on each ACK). This was not the case in our work, however, most likely due to how we induce timeouts the very instant BBR enters Startup, resulting in less time for the sending rate to double.

Attack 4 – RTO Stall. In this attack, a stalled connection is caused by an attacker who causes BBR to enter exponential-backoff, and prevents it from ever exiting. The attack begins by causing the sender to timeout by dropping, limiting, stretching or delaying acknowledgments. After the timeout and when exponential backoff is entered, the attacker prevents any new data from being acknowledged, by limiting or dropping acknowledgments. This causes BBR to permanently remain in exponential backoff because the re-transmitted segment will never be acknowledged. Additionally, no new data will ever be sent because `in_flight` reached `cwnd`, effectively stalling the connection. The lost segment will be re-transmitted 15 times (Linux default) with a doubled timeout time in between each re-transmit. On the 16th re-transmission, the TCP connection would be torn down by the sender (at least 15 minutes, 24 seconds from the first re-transmission).

In Figure 3d, the connection stalls around 700 ms after only sending about 2.5 MB. In Figure 4d, the background flow is able to obtain greater bandwidth made available by the victim stalling. It is important to note that this attack is highly flexible as it can be applied at any time during a connection and is not limited to a specific state or time.

Attack 5 – Sequence Number Desync. In this attack, a stalled connection is caused by an attacker who optimistically acknowledges lost data causing sequence numbers between the sender and receiver to de-synchronize. This attack works by acknowledging a lost segment (that was not actually delivered to the receiver). The primary reason why this attack causes a stalled connection is because the sender is unable to re-transmit the lost segment because it was removed from the "re-transmission queue". The TCP write queue retains segments until they have been acknowledged. As segments are acknowledged, they are discarded from the queue in order to free memory.

Next, the sender transmits the next data segments, which will be delivered out-of-order from the receiver's point of view, meaning the receiver will respond by acknowledging the highest correct data segment received so far. Since the out-of-order segment the sender sends cannot be acknowledged, it will keep being re-transmitted eventually causing three duplicate acknowledgments to be sent by the receiver. When this occurs, the sender will try to re-transmit the lost segment but cannot because it has been removed from the re-transmission queue.

In Figure 3e, the connection stalls because while the receiver sends duplicate acknowledgments (around the 0.2 MB mark), the sender keeps re-transmitting the same segment (around the 0.4 MB mark) due to RTOs. Since the receiver cannot receive that segment because it is out-of-order, it cannot acknowledge it, causing a stalled connection. In Figure 4e, the background flow can be seen slight increasing its sending rate because when the victim's connection stalled, more bandwidth is made available, allowing the background flow to send faster. This attack was discovered in [29] for TCP New Reno which also resulted in a stalled connection.

Although this attack is most effective from an on-path attacker, this attack could be achieved by an off-path attacker who successfully learns the victim flow's sequence number state. If an off-path attacker successfully crafts and injects an acknowledgment packet acknowledging a lost segment, then a stalled connection would result.

4.3 Ineffective Attacks Against BBR

Below we describe how some previously known attacks against congestion control were ineffective against BBR.

Acknowledgment Bursts. In this attack, the attacker accumulates n acknowledgment packets from the receiver before forwarding them to the sender in a single burst. In [29], this attack caused New Reno to send data in bursts because TCP is ACK-clocked meaning its sending behavior closely mimics the acknowledgment behavior. In BBR, acknowledgment bursts do not cause data to be sent in bursts because even though the delivery rate samples computed for the first $n-1$ ACKs in the burst are deflated, the delivery rate sample for the n^{th} (last) ACK in the burst is no different than without an attack. Figure D2a illustrates why this is the case. ACKs 1 and 2 arrive later than normal, meaning `ack_elapsed` is increased, which deflates their delivery rate samples. However, ACK 3 arrives normally, even with the attack taking place, meaning its delivery rate sample is unchanged. Since the delivery rate samples for the first $n-1$ ACKs are always less than the n^{th} (last) ACK, the ACKs arrive at the same time and larger delivery rate samples take precedence, this attack does not impact BBR's `BtlBw` estimate.

Acknowledgment Division. In this attack, a single ACK acknowledging m bytes is divided into n valid ACKs each acknowledging roughly m/n bytes. In [36], this attack caused `cwnd` to grow n times as fast because for each ACK, `cwnd` increased by one segment. Depending on how the attacker injected the divided ACKs, this attack had no effect on BBR's sending rate for different reasons. If the attacker sent the divided ACKs at the same time as the valid ACK (the ACK being divided), then the attack would not be effective for the exact same reason as to why the acknowledgment burst attack did not affect BBR's sending rate (Section 4.3). If the attacker sent the divided ACKs at the same time as the last ACK (the ACK before the one that is being divided), then the ACK rate would be clamped by BBR's current sending rate. If the attacker performed this during BBR's probing phase, then it would be identical to attack 1 where the sender's sending rate is increased. If the attacker evenly spaced each divided ACK, then the ACK rate of the divided ACKs would be no different than the ACK rate without the attack.

Duplicate Acknowledgments. In this attack, n duplicate acknowledgments are injected for every acknowledgment packet from the receiver. In congestion control schemes such as CUBIC and New Reno that use packet loss to detect congestion, when ≥ 3 duplicate acknowledgments were injected, Fast Recovery was entered to re-transmit the lost segment. This caused a decreased sending rate because upon entering Fast Recovery, the congestion window is halved causing data to be sent at a slower rate. Although BBR is not loss-based, it still includes a mechanism for dealing with packet loss (by detecting duplicate acknowledgments) by entering a Recovery state. The reason this attack is not effective against BBR is because BBR does not backoff from the network upon packet loss. Instead, BBR sets `cwnd` to `in_flight` and re-transmits the lost segment until all outstanding data when Recovery was entered is acknowledged.

5 Defenses

Most of our attacks rely on being able to modify acknowledgement information in TCP packets. The best defense against these attacks is to encrypt or authenticate this information. QUIC, a new transport protocol initially developed by Google but currently being standardized by the IETF, takes this approach. Unfortunately, adding this kind of authentication to TCP is impractical due to backwards compatibility issues. Similarly, prior work [39] has suggested adding a nonce to TCP acknowledgements to prevent optimistic ACK attacks. This suffers from similar backwards compatibility issues. Finally, some attacks, like the Delayed ACK attack, require only the ability to delay/reorder packets and appear to be inherent in trying to infer model parameters from delivered packets.

6 Related Work

Congestion Control Attacks. The work in [36] demonstrates how a misbehaving receiver can undermine congestion control making senders sent data at a faster pace without compromising reliability. It is shown how TCP is susceptible to divided, duplicate and optimistic acknowledgment attacks.

Much work has gone into *off-path* attackers who have write-only access to a flow. Sequence numbers can be predicted [5, 7, 18, 23, 33–35] to inject malicious content into a victim's connection. The work in [35] shows how sequence numbers can be leaked to unprivileged, on-device malware to coordinate with an off-path attacker, yielding connection hijacking in under one second. The work in [5] aims for better initial sequence number generation to make it more difficult for off-path attackers to succeed.

Protocol Fuzzing. Program analysis by automatically generating inputs has long been used to test for security, robustness and reliability. Instead of generating random inputs, the work in [22] takes an approach by generating relevant tests tailored to all possible source code paths. Similar approaches have been used for network protocol analysis. MAX [32] discovers attacks in network protocols however requires source code to be annotated where vulnerabilities are likely to exist, yielding thorough manual analysis. This motivated model-guided testing [19, 20, 29] where a protocol's state machine is used to discover relevant attacks which has been applied to a variety protocols. KiF [1], SNOOZE [4] and SNAKE [28] all take model-guided approaches to discover relevant and effective attack strategies in network protocols. TCPwn [29] takes a model-guided approach for discovering acknowledgement-based manipulation attacks in TCP congestion control implementations. As discussed in Section 3, TCPwn can not be directly applied to BBR.

7 Conclusion

We identified 5 classes of attacks from on-path attackers that caused BBR to send data a high, slow and stalled rates. We found that due to how BBR multiplicatively probes for bandwidth, an attacker who optimistically acknowledges data caused BBR to increase its sending rate by 13x in under 1 second. We showed that the combination of gain cycling and delayed acknowledgments by an attacker caused BBR to sequentially decrease its sending rate. We also showed that an attacker that prevented new data from being acknowledged caused re-transmission timeouts and for BBR to reset and rediscover the network path model each time. We also identified two attacks that stall data transmission: an attacker who prevents new data from being acknowledged and an attacker that optimistically acknowledges lost data causing sequence numbers to desynchronize. Finally, we show how the burst, divide and duplicate acknowledgment attacks against prior congestion control schemes are not effective against BBR.

References

[1] Humberto J. Abdelnur, Radu State, and Olivier Festor. KiF: A Stateful SIP Fuzzer. In *Proceedings of the 1st International Conference on Principles, Systems and Applications of IP Telecommunications*, pages 47–56, New York, NY, USA, 2007. ACM.

[2] Amit Aggarwal, Stefan Savage, and Thomas Anderson. Understanding the Performance of TCP Pacing. *Proceedings - IEEE INFOCOM*, 01 2000.

[3] Jong Suk Ahn, Peter B. Danzig, Zhen Liu, and Limin Yan. Evaluation of TCP Vegas: Emulation and Experiment. *SIGCOMM Comput. Commun. Rev.*, 25(4):185–195, October 1995.

[4] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer, and G. Vigna. SNOOZE: Toward a stateful network protocol fuzzer. In *International Conference on Information Security*, pages 343–358. 2006.

[5] S. Bellovin. Defending Against Sequence Number Attacks. https://tools.ietf.org/html/rfc1948, 1996.

[6] Lawrence S. Brakmo, Sean W. O'malley, and Larry L. Peterson. Tcp vegas: New techniques for congestion detection and avoidance. In *In SIGCOMM*, 1994.

[7] Yue Cao, Zhiyun Qian, Zhongjie Wang, Tuan Dao, Srikanth V. Krishnamurthy, and Lisa M. Marvel. Off-path TCP Exploits: Global Rate Limit Considered Dangerous. In *Proceedings of the 25th USENIX Conference on Security Symposium*, SEC'16, pages 209–225, Berkeley, CA, USA, 2016. USENIX Association.

[8] N. Cardwell, J. Priyaranjan, E. Dumazet, K. Yang, D. Miller, and Y. Seung. Linux TCP BBR. https://git.kernel.org/pub/scm/linux/kernel/git/davem/net-next.git/tree/net/ipv4/tcp_bbr.c, 2018.

[9] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. BBR: Congestion-Based Congestion Control. *ACM Queue*, 2016.

[10] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. BBR Congestion Control. https://www.ietf.org/proceedings/97/slides/slides-97-iccrg-bbr-congestion-control-02.pdf, November 2016.

[11] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. BBR Congestion Control: An Update. https://www.ietf.org/proceedings/98/slides/slides-98-iccrg-an-update-on-bbr-congestion-control-00.pdf, March 2017.

[12] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, Ian Swett, Jana Iyengar, Victor Vasiliev, and Van Jacobson. BBR Congestion Control: IETF 100 Update: BBR in shallow buffers. https://datatracker.ietf.org/meeting/100/materials/slides-100-iccrg-a-quick-bbr-update-bbr-in-shallow-buffers, November 2017.

[13] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, Ian Swett, Jana Iyengar, Victor Vasiliev, and Van Jacobson. BBR Congestion Control: IETF 99 Update. https://www.ietf.org/proceedings/99/slides/slides-99-iccrg-iccrg-presentation-2-00.pdf, July 2017.

[14] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, Ian Swett, Jana Iyengar, Victor Vasiliev, Priyaranjan Jha, Yousuk Seung, and Van Jacobson. BBR Congestion Control Work at Google: IETF 101 Update. https://datatracker.ietf.org/meeting/101/materials/slides-101-iccrg-an-update-on-bbr-work-at-google-00, March 2018.

[15] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, Ian Swett, Jana Iyengar, Victor Vasiliev, Priyaranjan Jha, Yousuk Seung, Kevin Yang, Matt Mathis, and Van Jacobson. BBR Congestion Control Work at Google: IETF 101 Update. https://datatracker.ietf.org/meeting/102/materials/slides-102-iccrg-an-update-on-bbr-work-at-google-00, July 2018.

[16] Neal Cardwell, Yuchung Cheng, Soheil Hassas Yeganeh, and Van Jacobson. BBR Congestion Control. https://tools.ietf.org/id/draft-cardwell-iccrg-bbr-congestion-control-00.html, 2017.

[17] Neal Cardwell, Yuchung Cheng, Soheil Hassas Yeganeh, and Van Jacobson. Delivery Rate Estimation. https://tools.ietf.org/html/draft-cheng-iccrg-delivery-rate-estimation-00, 2018.

[18] Weiteng Chen and Zhiyun Qian. Off-Path TCP Exploit: How Wireless Routers Can Jeopardize Your Secrets. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1581–1598, Baltimore, MD, 2018. USENIX Association.

[19] C. Cho, D. Babic, P. Poosankam, K. Chen, E. Wu, and D. Song. MACE: Model-inference-Assisted Concolic Exploration for Protocol and Vulnerability Discovery. In *USENIX Conference on Security*, 2011.

[20] Joeri de Ruiter and Erik Poll. Protocol State Fuzzing of TLS Implementations. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 193–206, Washington, D.C., 2015. USENIX Association.

[21] Defense Advanced Research Projects Agency. Transmission Control Protocol. https://tools.ietf.org/html/rfc793, 1981.

[22] Joe W. Duran and Simeon Ntafos. A report on random testing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 179–183, Piscataway, NJ, USA, 1981. IEEE Press.

[23] Yossi Gilad and Amir Herzberg. Off-path Attacking the Web. In *Proceedings of the 6th USENIX Conference on Offensive Technologies*, WOOT'12, pages 5–5, Berkeley, CA, USA, 2012. USENIX Association.

[24] University of Southern California Information Sciences Institute. Transmission Control Protocol. https://tools.ietf.org/html/rfc793, 1981.

[25] J. Iyengar, Ed and Fastly and M. Thomas, Ed and Mozilla. QUIC: A UDP-Based Multiplexed and Secure Transport. https://tools.ietf.org/html/draft-ietf-quic-transport-18, 2019.

[26] Van Jacobson. Congestion Avoidance and Control. *ACM SIGCOMM Computer Communication Review*, 18(4):314–329, 1988.

[27] R. Jain. A delay-based approach for congestion avoidance in interconnected heterogeneous computer networks. *SIGCOMM Comput. Commun. Rev.*, 19(5):56–71, October 1989.

[28] S. Jero, H. Lee, and C. Nita-Rotaru. Leveraging State Information for Automated Attack Discovery in Transport Protocol Implementations. In *IEEE/IFIP International Conference on Dependable Systems and Networks*, 2015.

[29] Samuel Jero, Endadul Hoque, David Choffnes, Alan Mislove, and Cristina Nita-Rotaru. Automated Attack Discovery in TCP Congestion Control Using a Model-guided Approach. In *Proc. of Network & Distributed System Security Symposium (NDSS)*, 2018.

[30] Laurent Joncheray. A simple active attack against TCP. In *USENIX Security Symposium*, 1995.

[31] L Kleinrock. Power and deterministic rules of thumb for probabilistic problems in computer communications. 01 1979.

[32] Nupur Kothari, Ratul Mahajan, Todd Millstein, Ramesh Govidan, and Madanlal Musuvathi. Finding Protocol Manipulation Attacks. In *SIGCOMM*, pages 26–37, 2011.

[33] Robert T. Morris. A Weakness in the 4.2BSD Unix TCP/IP Software, 1985.

[34] Z. Qian and Z. M. Mao. Off-path TCP Sequence Number Inference Attack - How Firewall Middleboxes Reduce Security. In *2012 IEEE Symposium on Security and Privacy*, pages 347–361, May 2012.

[35] Zhiyun Qian, Z. Morley Mao, and Yinglian Xie. Collaborative TCP sequence number inference attack: how to crack sequence number under a second. In *ACM Conference on Computer and Communications Security*, 2012.

[36] Stefan Savage, Neal Cardwell, David Wetherall, and Tom Anderson. TCP Congestion Control with a Misbehaving Receiver. *ACM SIGCOMM Computer Communication Review*, 29(5), 1999.

[37] V. Jacobsen and LBL and R. Braden and ISI. TCP Extensions for Long-Delay Paths. https://tools.ietf.org/html/rfc1072, October 1988.

[38] David X. Wei, Cheng Jin, Steven H. Low, and Sanjay Hegde. FAST TCP: Motivation, Architecture, Algorithms, Performance. *IEEE/ACM Trans. Netw.*, 14(6):1246–1259, December 2006.

[39] Lixia Zhang, Scott Shenker, and Daivd D. Clark. Observations on the Dynamics of a Congestion Control Algorithm: The Effects of Two-way Traffic. *SIGCOMM Comput. Commun. Rev.*, 21(4):133–147, August 1991.

A Background on Congestion Control

A.1 Congestion Control Overview

Congestion control determines whether to send a segment of data based on information inferred about a network path between endpoints. A primary goal is to saturate the bottleneck link (maintaining high utilization) while avoiding congestion collapse (due to sending faster than the bottleneck link can support). The bottleneck link is saturated when the total amount of data in-flight equals the path's bandwidth-delay product (BDP) which represents the maximum amount of in-flight data the network [31] can process without dropping packets. A path's BDP is dynamic and typically computed as the product of the bottleneck link's maximum bandwidth and the path's round-trip time without queue delay [37]. Congestion control is also tasked with avoiding congestion collapse and achieving fairness with other flows sharing the network. Accomplishing these goals is challenging because networks are unpredictable: links vary in bandwidth capacity and are shared anywhere between few to millions of hosts such as the global Internet. While several congestion control algorithms have been developed, most adhere to the same basic principles first described by Jacobson in 1988 [26]. Below we describe the main goals of a congestion control algorithm.

Discovering the target sending rate. The target sending rate is one that achieves high throughput and avoids congestion. The sending rate is dictated by a per-connection variable known as the congestion window `cwnd` which governs the maximum amount of unacknowledged data allowed in-transit. When a connection first starts, congestion control performs *slow start* to quickly discover the available bandwidth of the link. Afterwards, *congestion avoidance* is performed whereby data is sent conservatively while slowly probing the network for available bandwidth.

Inferring congestion. Congestion control must use signals from the network to infer congestion as an indicator to "back off", or reduce its load on the network. The most popular paradigms have been loss-based congestion control and delay-based congestion control. Loss-based congestion control has dominated the Internet since its creation and uses packet loss as signal of congestion. Packet loss occurs when switch buffers along a network path fill to capacity and are left with no choice but to discard, *i.e.*, drop, incoming packets. Delay-based congestion control compares predicted and actual round-trip time (RTT) samples to signal congestion. These signals, be they packet loss or RTT, govern the rate at which data is sent into the network.

Achieving fairness with competing flows. Since networks today are shared by several end-hosts, congestion control aims to share the limited resources of a bottleneck link evenly across all flows. Coexisting congestion control algorithms may not be fair to each other due to differing probing and backoff mechanisms. For example, delay and loss-based congestion control do not operate well together. Delay-based congestion control has been shown to reduce its congestion window much earlier than loss-based [3], resulting in unfair bandwidth allocation. Because of this, delay-based congestion control is not commonly used in today's networks.

A.2 Congestion Signal

Loss-based congestion control uses two signals to detect packet loss: re-transmission timeouts (RTOs) and duplicate acknowledgment packets. Delay-based congestion control leverages changes in RTT samples to infer congestion. All methods leverage feedback from the receiver (*i.e.* acknowledgment packets) to detect congestion.

Re-transmission timeout. RTOs ensure data delivery when there is no feedback from the receiver. Each time a data segment is sent, a timer starts and expires if the segment has not been acknowledged after a certain amount of time (usually several RTTs). Each time the timer expires, the data segment is re-transmitted and the timer restarts with a doubled timeout time. The initial timeout time is typically a function of RTT samples, gathered over the duration of a connection. This event can indicate congestion so severe that acknowledgments cannot be delivered in a sufficient amount of time.

Duplicate acknowledgments. When an out-of-order data segment is received, a receiver will ignore its payload and reply acknowledging the last correctly received data byte. If a data segment becomes lost in-transit, then the following in-transit segments will be received out-of-order, causing the receiver to send several duplicate acknowledgments. When a sender receives multiple (*e.g.*, three) duplicate acknowledgments in a row, congestion is inferred. This event signifies less severe network congestion because despite the loss of a data packet, acknowledgments are still able to be received by the sender.

Delayed acknowledgments. Delay-based congestion control, used by TCP Vegas [6], FAST TCP [38] and CARD [27], takes a *proactive* approach for detecting congestion rather than loss-based which takes a *reactive* approach after congestion already occurs. The advantage of delay over loss-based congestion control is that it detects the onset of congestion as switch buffers grow instead of waiting until they have filled for packet loss to occur. Delay-based congestion control infers congestion by comparing actual throughput to expected throughput. If actual throughput is significantly less than expected throughput, then the network is inferred to be congested.

Adapting to congestion. Congestion control adapts the sending rate based on the above congestion signals, typically accomplished by adjusting `cwnd` based on congestion severity. The most common approach for adjusting `cwnd` is the additive increase/multiplicative decrease (AIMD) scheme where `cwnd` linearly increases on each new ACK to probe for available bandwidth until a congestion event occurs, where `cwnd` exponentially decreases to "back off" from the network. When an RTO occurs, `cwnd` resets to 1 segment as this usually implies major changes in network conditions. Upon three duplicate acknowledgments, `cwnd` halves as this implies small amounts of packet loss. Fairness is achieved because flows back off at different times, allowing others to occupy the available bandwidth. The limitation of AIMD is while it can attain its fair share of bottleneck bandwidth, it backs off shortly after its discovery. The AIMD scheme is a high-level approach which varies depending on implementation.

A.3 A State Machine for Congestion Control

Acknowledgment packets. Every byte of data is associated with a unique sequence number [21] that increments by 1 with each additional data byte. A packet containing data is accompanied by a sequence number, which represents the sequence of the first byte of the data. Sequence numbers allow data segments to be easily reassembled and acknowledged by receivers. Acknowledgment packets are sent from the receiver to explicitly inform the sender about the highest correctly received data byte thus far. They also implicitly inform the sender about current network conditions. An acknowledgment packet acknowledging sequence X implies all bytes up to but not including X have been correctly received. This allows the sender to either re-transmit lost segments or send new data. Receivers typically send one acknowledgment packet for every two received data packets.

Slow Start. This is the first state a connection enters and aims to quickly discover the available bandwidth of the network before congestion avoidance is entered. Since network capacities today span several orders of magnitude, this state performs an exponential search for the available bandwidth. When slow start is entered, `cwnd` starts at 1 MSS and doubles on every round-trip until either congestion is detected or `cwnd` reaches the target rate, the slow start threshold, or `ssthresh`. The slow start threshold defines the upper limit for `cwnd` growth while in slow start, which is initially set to the receive window, or `rwnd`.

Congestion Avoidance. The goal of this state is to avoid congestion by sending data at the estimated available bandwidth while slowly increasing `cwnd` to probe for available bandwidth. On every round-trip, `cwnd` increases by 1 MSS until congestion is detected: a RTO timeout or the recipient of three duplicate acknowledgments.

Fast Recovery. This state is entered from any other state when three duplicate acknowledgments are received. This event indicates less severe congestion because while it may indicate lost packets, it also indicates the network is at least able transmit acknowledgments from the receiver. This state aims to quickly recover from the lost packets by halving `cwnd` and re-transmitting the last unacknowledged data segment. Fast recovery returns to congestion avoidance once all unacknowledged data before fast recovery was entered has been acknowledged.

Exponential Backoff. This state is entered when a re-transmission timeout (RTO) expires which infers major changes in network conditions. Re-transmission timers begin when data segments are first sent and expire when a certain amount of time has elapsed without the segment being acknowledged (usually several RTTs). Each time an RTO timeout expires, the segment is re-transmitted and the timer restarts with a doubled timeout time. This results in the sender exponentially backing-off from the network by allowing more time to elapse before it re-transmits the lost segment in response to the perceived network congestion. This repeats until an acknowledgment is received after which `ssthresh` becomes half of `cwnd`, `cwnd` restarts from 1 MSS and exponential-backoff transitions into slow start.

Algorithm B1 Attack strategy categorization algorithm: 20 benign experiments are first executed to obtain a baseline average and standard deviation. Since each strategy transfers the same 100MB file, an strategy is categorized as a function of its total transfer time and the baseline average and standard deviation transfer time.

```
    Input: Strategy execution metrics
    Output: The category of the strategy
 1: function CATEGORIZEATTACKSTRATEGY($s$)
 2:     if $s.Time > (s.TimeAvg + 2 * TimeStddev)$ then
 3:         return SLOWER
 4:     else if $s.Time < (TimeAvg + 2 * TimeStddev)$ then
 5:         if $s.SentData >= (0.7*100\text{MB})$ then
 6:             return FASTER
 7:         else
 8:             return STALLED
 9:         end if
10:     else
11:         return BENIGN
12:     end if
13: end function
```

B Attack Strategy Categorization

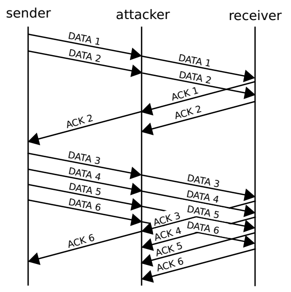
Attack strategy categorization algorithm is shown in Algorithm B1.

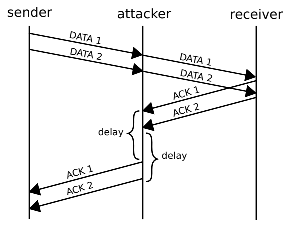
C Illustrations of Malicious Actions Used by our Attacks Strategies against BBR

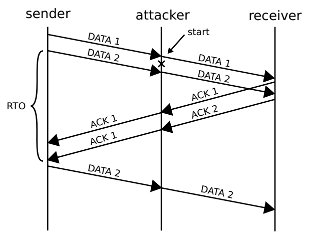
D Illustrations of Ineffective Attacks against BBR

E BBR State Inference Algorithm

For completeness, we present in details the BBR state inference algorithm below.


(a) Optimistic ACK: acknowledge highest byte, dropping duplicates.


(b) Delayed ACK: delay acknowledgments for a fixed amount of time.


(c) Limited ACK: prevent ACK numbers from increasing.

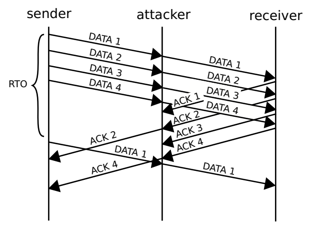
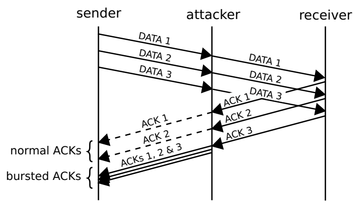
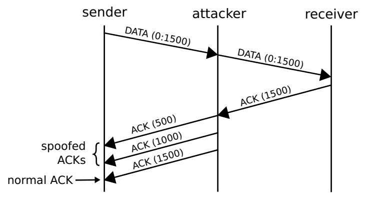

(d) Stretch ACKs: forward only every n^{th} ACK. In this example, $n = 2$.

Figure C1: Time lines of acknowledgment-based manipulation actions used in our attack strategies.


(a) ACK burst: send n ACKs in a single burst. In this example, $n = 3$.


(b) Divided ACKs: ACK m bytes using n ACKs, each acknowledging m/n bytes.

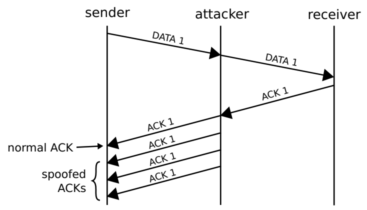

(c) Duplicate ACKs: inject n duplicate ACKs. In this example, $n = 3$.

Figure D2: Time lines of acknowledgment-based manipulation actions that were previously known to be effective against TCP congestion control, but were ineffective against BBR.

Algorithm E2 BBR state inference algorithm

```
function OnNewBBRPacket(packet)
    newInt = CheckInterval(packet) /* a round-trip has passed */
    if dataPackets > 0 and RTOPassedForDataPacket() then
        priorState = currentState
        currentState = EXPONENTIAL_BACKOFF
        ComputeIntervalMetrics()
    else if retransmissions > 0 and not currentState == RECOVERY then
        priorState = currentState
        currentState = RECOVERY
        /* leave recovery when high water is ACK'd */
        highWater = highestDataSequence
    else if currentState == RECOVERY then
        ratio = ThroughputRatioSinceLastRoundTrip()
        if totalAcked ≥ highWater then
            currentState = priorState /* return to previous state */
            SetRetransmissionCount(0)
        else if newInt and priorState == STARTUP and 0.7 > ratio > 0.1 then
            priorState = DRAIN
        end if
    else if currentState == PROBERTT and dataPackets > 10 then
        currentState = priorState
    else if newInt then
        currentState = UpdateBBRNewInterval()
    else if dataPackets ≥ 10 then
        /* check for drain on frequent basis */
        ratio = ThroughputRatioSinceLastRoundTrip()
        if currentState == STARTUP and 0.7 > ratio > 0.1 then
            priorState = currentState
            return DRAIN
        end if
    end if
end function

function UpdateBBRNewInterval()
    ratio = ThroughputRatioSinceLastRoundTrip()
    dataPackets = GetDataPacketCount()
    nonIncrease = NonIncreaseIntervalCount()
    if 6 > dataPackets > 3 then /* small amount of data in-flight */
        return PROBERTT
    else if currentState == DRAIN and ratio > 1.4 then
        priorState = currentState
        return PROBEBW
    else if not currentState == PROBEBW and ratio > 1.4 then
        priorState = currentState
        return STARTUP
    else if currentState == STARTUP and 0.7 > ratio > 0.1 then
        priorState = currentState
        return DRAIN
    else if currentState == STARTUP and nonIncrease > 10 then
        priorState = currentState
        return DRAIN
    else if intervalCount > 16 and throughputVariance < 100 then
        priorState = currentState
        return RATE_LIMIT
    else if not currentState == STARTUP and 1.4 > ratio > 0.6 then
        priorState = currentState
        return PROBEBW
    else
        return currentState
    end if
end function

function CheckInterval(packet)
    if not inInterval and IsDataPacket(packet) then
        ClearIntervalData()
        ackMark = packet.sequenceNumber
        inInterval = true
    end if
    if inInterval and packet.ackNumber ≥ ackMark then
        inInterval = false
        ComputeIntervalMetrics()
        return true
    end if
    return false
end function
```