Reverse-Engineering Congestion Control Algorithm Behavior

Margarida Ferreira, Ranysha Ware, Yash Kothari,
Inês Lynce, Ruben Martins, Akshay Narayan,
Justine Sherry

Carnegie Mellon University

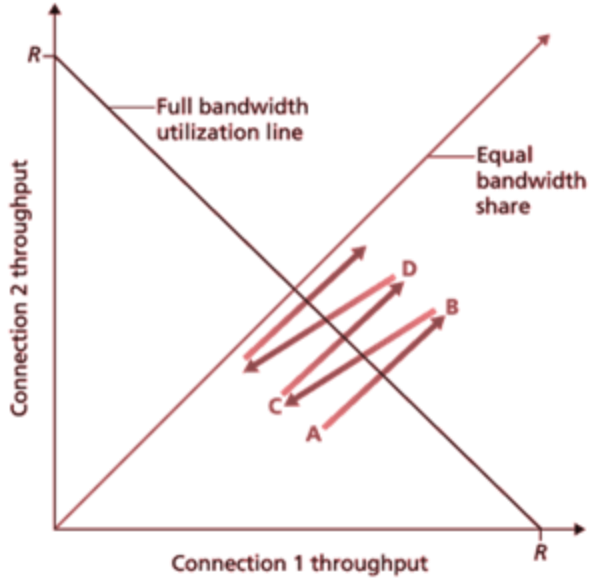
Carnegie Mellon Portugal

TÉCNICO LISBOA

BROWN



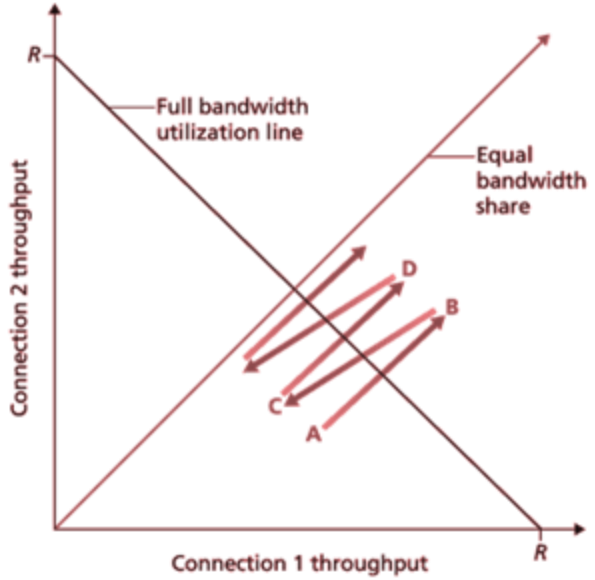
Congestion Control Algorithms (CCAs) affect every Internet connection

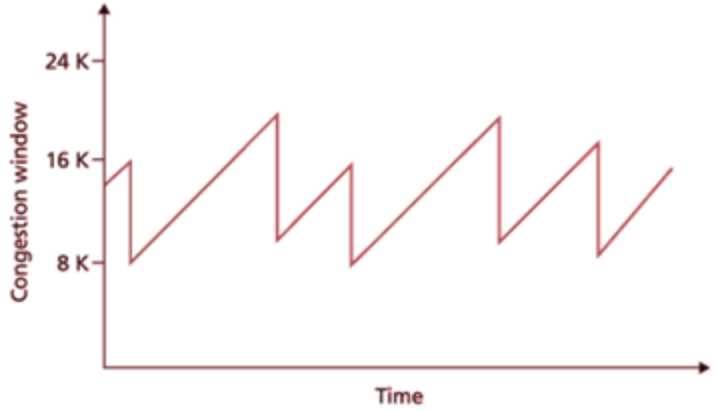


Fairness: whether competing applications share network bandwidth fairly

Figures from Kurose, J. F., & Ross, K. W. (2001). Computer networking: A top-down approach featuring the Internet

Congestion Control Algorithms (CCAs) affect every Internet connection



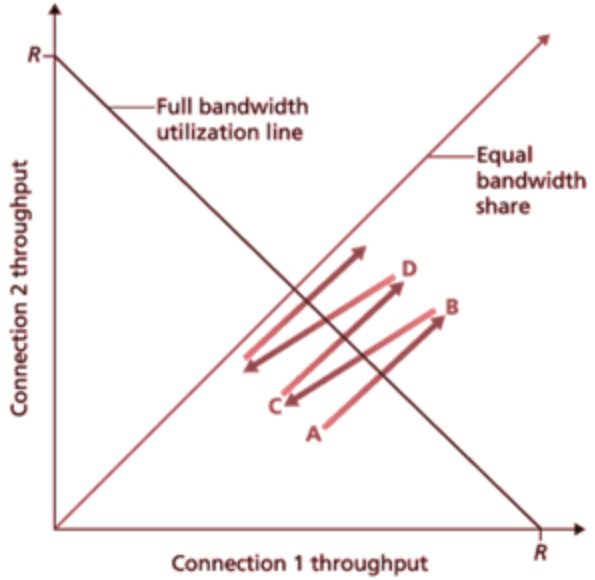


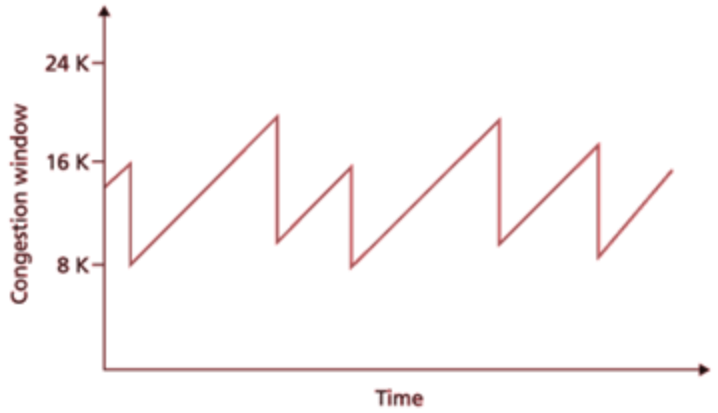
Fairness: whether competing applications share network bandwidth fairly

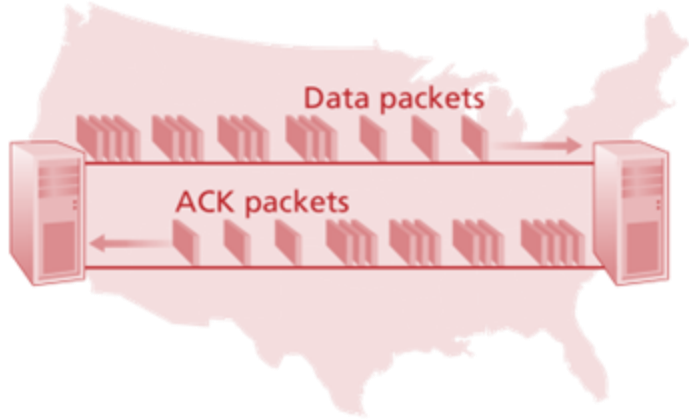
Stability: how stable bandwidth allocations are (or whether performance oscillates)

Figures from Kurose, J. F., & Ross, K. W. (2001). Computer networking: A top-down approach featuring the Internet

Congestion Control Algorithms (CCAs) affect every Internet connection







Fairness: whether competing applications share network bandwidth fairly

Stability: how stable bandwidth allocations are (or whether performance oscillates)

Utilization: whether network links are utilized efficiently

Figures from Kurose, J. F., & Ross, K. W. (2001). Computer networking: A top-down approach featuring the Internet

Congestion Control Algorithms (CCAs) affect every Internet connection

33

Beyond Jain's Fairness Index: Setting the Bar For The Deployment of Congestion Control Algorithms

Ranysha Ware, Carnegie Mellon University, rware@cs.cmu.edu
Matthew K. Mukerjee, Nefeli Networks, mukerjee@nefeli.io
Srinivasan Seshan, Carnegie Mellon University, srini@cs.cmu.edu
Justine Sherry, Carnegie Mellon University, sherry@cs.cmu.edu

ABSTRACT

The Internet community faces an explosion in new congestion control algorithms such as Copa, Sprout, PCC, and BBR. In this paper, we discuss considerations for deploying new algorithms on the Internet. While past efforts have focused on achieving 'fairness' or 'friendliness' between new algorithms and deployed algorithms, we instead advocate for an approach centered on quantifying and limiting harm caused by the new algorithm on the status quo. We argue that a harm-based approach is more practical, more future proof, and handles a wider range of quality metrics than traditional notions of fairness and friendliness.

ACM Reference Format:
Ranysha Ware, Matthew K. Mukerjee, Srinivasan Seshan, and Justine Sherry. 2019. Beyond Jain's Fairness Index: Setting the Bar For The Deployment of Congestion Control Algorithms. In *The 18th ACM Workshop on Hot Topics in Networks (HotNets '19), November 13–15, 2019, Princeton, NJ, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3365609.3365855

1 INTRODUCTION

In recent years, the networking research community has generated an explosion of new congestion control algorithms (CCAs) [1, 2, 5, 6, 25–27], many of which are being explored by Internet content providers [4, 19]. This state of affairs brings the community back to an age-old question: what criteria do we use to decide whether a new congestion control algorithm is acceptable to deploy on the Internet? Without a standard *deployment threshold*, we are left without foundation to argue whether a service provider's new algorithm is or is not overly-aggressive.

A deployment threshold concerns inter-CCA phenomena, not intra-CCA phenomena. Rather than analyzing the outcomes between a collection of flows, all using some CCA α, we need to analyze what happens when a new CCA α is deployed on a network with flows using some legacy CCA β. Is α's impact on the status quo is acceptable?

Our community has traditionally analyzed inter-CCA competition in two ways, which we refer to as 'fairness' and 'mimic

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
HotNets'19, November 14-15, 2019, Princeton NJ, USA
© 2019 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-7020-2/19/1…$15.00
https://doi.org/10.1145/3365609.3365855

Axiomatizing Congestion Control

DORON ZARCHY, Hebrew University of Jerusalem
RADHIKA MITTAL, University of Illinois at Urbana-Champaign
MICHAEL SCHAPIRA, Hebrew University of Jerusalem
SCOTT SHENKER, UC Berkeley, ICSI

The overwhelmingly large design space of congestion control protocols, along with the increasingly diverse range of application environments, makes evaluating such protocols a daunting task. Simulation and experiments are very helpful in evaluating the performance of designs in *specific* contexts, but give limited insight into the more general properties of these schemes and provide no information about the *inherent* limits of congestion control designs (such as, which properties are simultaneously achievable and which are mutually exclusive). In contrast, traditional theoretical approaches are typically focused on the design of protocols that

Toward Formally Verifying Congestion Control Behavior

Venkat Arun*, Mina Tahmasbi Arashloo†, Ahmed Saeed*, Mohammad Alizadeh*, Hari Balakrishnan*
MIT CSAIL* and Cornell University†
Email: ccac@mit.edu Website: https://projects.csail.mit.edu/ccac

ABSTRACT

The diversity of paths on the Internet makes it difficult for designers and operators to confidently deploy new congestion control algorithms (CCAs) without extensive real-world experiments, but such capabilities are not available to most of the networking community. And even when they are available, understanding why a CCA under-performs by trawling through massive amounts of statistical

performance (e.g., by giving applications poor ratings or finding alternatives). Performance matters not only in the mean, but also in the tail statistics. In response, the research community and industry have developed numerous innovative methods to improve congestion control, because CCAs determine when packets are sent and determine transport performance [3, 5, 14, 18, 19, 36, 49, 50, 52, 54].

A key problem in CCA development is *evaluation*: how can developers, operators, and the networking community gain confidence in any given proposal? Real-world network paths exhibit a wide range of complex behaviors due to token-bucket filters, rate limiters, traffic shapers, network-layer packet schedulers with various artifacts, link-layer schedulers that vary link rates, physical-layer vagaries, link-layer acknowledgment (ACK) aggregation, higher-layer ACK compression or aggregation, delayed ACKs, and more. It is impossible even for seasoned engineers to contemplate the composition of every "weird" thing that could happen along a path, much less model or simulate these behaviors faithfully.

The process of evaluating and gaining confidence with a CCA today involves some combination of simulation [1, 2], prototype implementation with tests on a modest number of emulated [13, 26, 39] and real-world paths [53, 54], and, in some cases, empirical analysis via controlled A/B tests at large content providers. Simulations and small-scale tests are invaluable in the design and refinement stages, but provide little confidence about performance on the trillions of real-world paths.

If one has access to servers at a large content provider, then A/B tests are feasible where a new CCA can be tried on a fraction of the users to compare its performance with another scheme. If the measured results of the new CCA compare well, it increases confidence in its behavior, but still does not guarantee that it will perform well in all scenarios. Moreover, as is likely, the new CCA will not perform better in the A/B tests for all users. The aggregate results of an A/B test may hide significant weaknesses that arise in certain cases. When such cases are identified, understanding the behavior of a CCA requires going a massive data analysis, which may be futile because the operator might not have visibility into the network conditions that led to poor performance. We also note that most of the community does not work at a "hyperscaler" with access to such a live-testing infrastructure, yet has good ideas that deserve serious consideration.

In this paper, we propose initial steps to mitigate these issues. We have developed the Congestion Control Anxiety Controller (CCAC), pronounced "seek-ack" or "see-cack". CCAC uses formal verification to prove certain properties of CCAs. With CCAC, a user can (1) express a CCA in first-order logic, (2) specify hypotheses about the CCA for the tool to prove, and (3) test the hypothesis in the context of the expressed CCA running in a customizable, built-in path model. The user's ingenuity is useful in expressing the CCA and using CCAC to propose and iterate on useful hypotheses, while CCAC will prove the hypothesis correct or find insightful

Experimental Evaluation of BBR Congestion Control

Mario Hock, Roland Bless, Martina Zitterbart
Karlsruhe Institute of Technology
Karlsruhe, Germany
E-Mail: mario.hock@kit.edu, bless@kit.edu, zitterbart@kit.edu

***Abstract*—BBR is a recently proposed congestion control. Instead of using packet loss as congestion signal, like many currently used congestion controls, it uses an estimate of the available bottleneck link bandwidth to determine its sending rate. BBR tries to provide high link utilization while avoiding to create queues in bottleneck buffers. The original publication of BBR shows that it can deliver superior performance compared to CUBIC TCP in some environments. This paper provides an independent and extensive experimental evaluation of BBR at higher speeds. The experimental setup uses BBR's Linux kernel 4.9 implementation and typical data rates of 10 Gbit/s and 1 Gbit/s at the bottleneck link. The experiments vary the flows' round-trip times, the number of flows, and buffer sizes at the bottleneck. The evaluation considers throughput, queuing delay, packet loss, and fairness. On the one hand, the intended behavior of BBR could be observed with our experiments. On the other hand, some severe inherent issues such as increased queuing delays, unfairness, and massive packet loss were also detected. The paper provides an in-depth discussion of BBR's behavior in different experiment setups.**

I. Introduction

Congestion control protects the Internet from persistent overload situations. Since its invention and first Internet-wide introduction congestion control has evolved a lot [1], but is still a topic of ongoing research [10], [15]. In general, congestion control mechanisms try to determine a suitable amount of data to transmit at a certain point in time in order to utilize the available transmission capacity, but to avoid a persistent overload of the network. The bottleneck link is fully utilized if the amount of *inflight data* $D^{inflight}$ matches exactly the bandwidth delay product $bdp = b_r \cdot RTT_{min}$, where b_r is the available bottleneck data rate (i.e., the smallest data rate along a network path between two TCP end systems) and RTT_{min} is the minimal round-trip time (without any queuing delay). A fundamental difficulty of congestion control is to calculate a suitable amount of inflight data without exact knowledge of the current bdp. Usually, acknowledgments as feedback help to create estimates for the bdp. If $D^{inflight}$ is larger than bdp, the bottleneck is overloaded, and any excess data is filled into a buffer at the bottleneck link or dropped if the buffer capacity is exhausted. If this overload situation persists the bottleneck becomes congested. If $D^{inflight}$ is smaller than bdp,

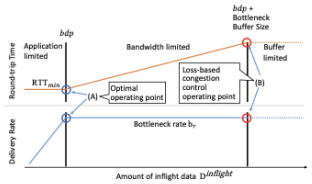


Fig. 1: Congestion control operating points: delivery rate and round-trip time vs. amount of inflight data, based on [5]

to completely fill the available buffer capacity at a bottleneck link, since most buffers in network devices still apply a tail drop strategy. A filled buffer implies a *large queuing delay* that adversely affects everyone's performance on the Internet: the inflicted latency is unnecessarily high. This also highly impacts interactive applications (e.g., Voice-over-IP, multiplayer online games), which often have stringent requirements to keep the one way end-to-end delay below 100 ms. Similarly, many transaction-based applications suffer from high latencies.

Recently, BBR was proposed by a team from Google [5] as new congestion control. It is called "congestion-based" congestion control in contrast to loss-based or delay-based congestion control. The fundamental difference in their mode of operation is illustrated in fig. 1 (from [5]), which shows round-trip time and delivery rate in dependence of $D^{inflight}$ for a single sender at a bottleneck. If the amount of inflight data $D^{inflight}$ is just large enough to fill the available bottleneck link capacity (i.e., $D^{inflight} = bdp$), the bottleneck link is fully utilized and the queuing delay is still zero or close to zero. This is the optimal operating point (A), because the bottleneck link is already fully utilized at this point. If the amount of inflight data is increased any further, the bottleneck buffer gets filled with the excess data. The delivery rate, however, does not

Stability Analysis of Explicit Congestion Control Protocols

Hamsa Balakrishnan, Nandita Dukkipati, Nick McKeown and Claire J. Tomlin
Stanford University, Stanford, CA 94305
Email: {hamsa, nanditad, nickm, tomlin}@stanford.edu

Abstract

Much recent attention has been devoted to analyzing the stability of congestion control algorithms, in the context of TCP modifications (e.g., TCP/RED [10], [15], FAST [17]) and new protocols (e.g., XCP [21], RCP [8], TeXCP [20]). The control-theoretic framework used in most previous work is linear systems theory. The analyses assume that the system can be well approximated by linearization, and the linearization is then used to derive conditions for stability using techniques based on the Bode or Nyquist criteria.

We show that linearization is *not* a good approximation when the queue lengths are close to zero. Because the goal of several congestion control algorithms is to keep queue lengths small, the linearization turns out to be the most inaccurate precisely in the realm in which a good algorithm would hope to operate. We show, in the context of explicit congestion control protocols like XCP and RCP, that the stability region derived from traditional Nyquist analysis is not an accurate representation of the actual stability region. Using XCP as an example, we then show that modeling the congestion control algorithm as a switched linear control system with time delay, and using new Lyapunov stability conditions can provide sound and more general sufficient conditions for stability than previously derived. For piecewise linear systems with time-delay, the proposed conditions guarantee global stability. We show that the proposed framework can be used to analyze the stability of congestion control protocols in the presence of heterogeneous delays.

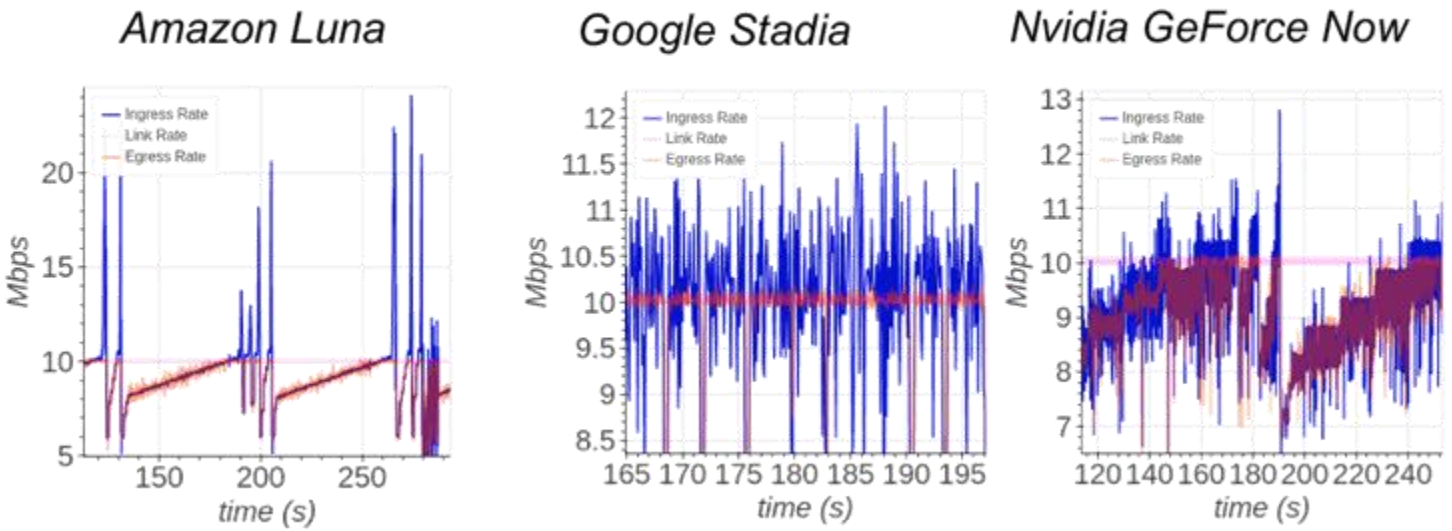
Stanford University Department of Aeronautics and Astronautics Report: SUDAAR No. 776, September 9, 2005. This research was supported by an NSF Career award (ECS-9985072). H. Balakrishnan was supported by a Stanford Graduate Fellowship.

Companies are using new proprietary CCAs for different applications

- Video streaming
- Online gaming
- Videoconferencing



Steady State at Low Bandwidth



D. Caban, D. Ray and S.Seshan. Understanding Congestion Control for Cloud Game Streaming. CMU REU 2020

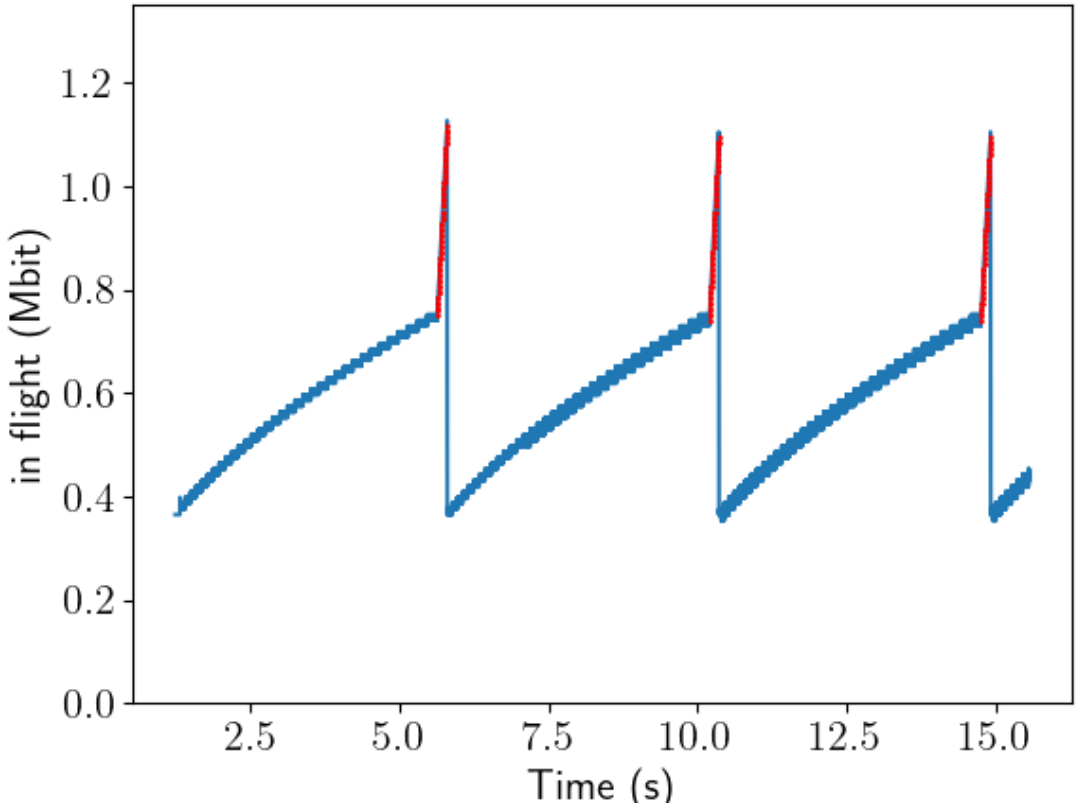
CCAs are implemented in thousands of lines of code in the kernel



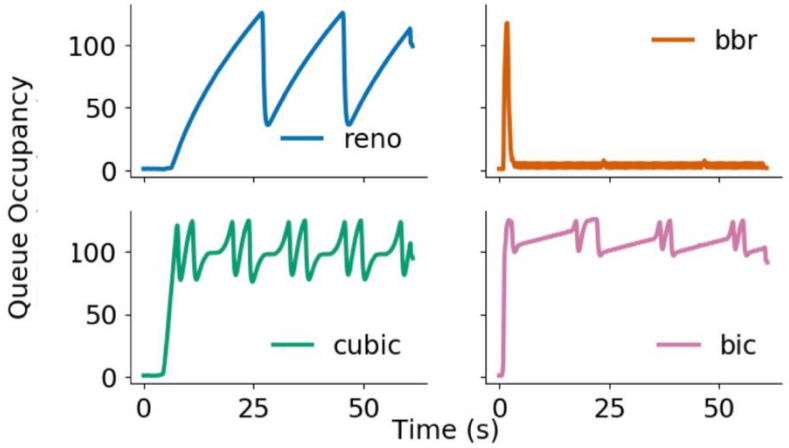
```
        /* Initial outgoing SYN's get put onto the write_queue
         * just like anything else we transmit.  It is not
         * true data, and if we misinform our callers that
         * this ACK acks real data, we will erroneously exit
         * connection startup slow start one packet too
         * quickly.  This is severely frowned upon behavior.
         */
        if (likely(!(scb->tcp_flags & TCPHDR_SYN))) {
                flag |= FLAG_DATA_ACKED;
        } else {
                flag |= FLAG_SYN_ACKED;
                tp->retrans_stamp = 0;
        }

        if (!fully_acked)
                break;

        tcp_ack_tstamp(sk, skb, ack_skb, prior_snd_una);

        next = skb_rb_next(skb);
        if (unlikely(skb == tp->retransmit_skb_hint))
                tp->retransmit_skb_hint = NULL;
        if (unlikely(skb == tp->lost_skb_hint))
                tp->lost_skb_hint = NULL;
        tcp_highest_sack_replace(sk, skb, next);
        tcp_rtx_queue_unlink_and_free(skb, sk);
}

if (!skb)
        tcp_chrono_stop(sk, TCP_CHRONO_BUSY);

if (likely(between(tp->snd_up, prior_snd_una, tp->snd_una)))
        tp->snd_up = tp->snd_una;

if (skb) {
        tcp_ack_tstamp(sk, skb, ack_skb, prior_snd_una);
        if (TCP_SKB_CB(skb)->sacked & TCPCB_SACKED_ACKED)
                flag |= FLAG_SACK_RENEGING;
}

if (likely(first_ackt) && !(flag & FLAG_RETRANS_DATA_ACKED)) {
        seq_rtt_us = tcp_stamp_us_delta(tp->tcp_mstamp, first_ackt);
        ca_rtt_us = tcp_stamp_us_delta(tp->tcp_mstamp, last_ackt);

        if (pkts_acked == 1 && last_in_flight < tp->mss_cache &&
            last_in_flight && !prior_sacked && fully_acked &&
            sack->rate->prior_delivered + 1 == tp->delivered &&
            !(flag & (FLAG_CA_ALERT | FLAG_SYN_ACKED))) {
                /* Conservatively mark a delayed ACK. It's typically
                 * from a lone runt packet over the round trip to
                 * a receiver w/o out-of-order or CE events.
                 */
                flag |= FLAG_ACK_MAYBE_DELAYED;
        }
}
if (sack->first_sackt) {
        sack_rtt_us = tcp_stamp_us_delta(tp->tcp_mstamp, sack->first_sackt);
        ca_rtt_us = tcp_stamp_us_delta(tp->tcp_mstamp, sack->last_sackt);
}
rtt_update = tcp_ack_update_rtt(sk, flag, seq_rtt_us, sack_rtt_us,
                                ca_rtt_us, sack->rate);

if (flag & FLAG_ACKED) {
        flag |= FLAG_SET_XMIT_TIMER;  /* set TLP or RTO timer */
        if (unlikely(icsk->icsk_mtup.probe_size &&
                     !after(tp->mtu_probe.probe_seq_end, tp->snd_una))) {
                tcp_mtup_probe_success(sk);
        }

        if (tcp_is_reno(tp)) {
                tcp_remove_reno_sacks(sk, pkts_acked, ece_ack);

                /* If any of the cumulatively ACKed segments was
                 * retransmitted, non-SACK case cannot confirm that
                 * progress was due to original transmission due to
                 * lack of TCPCB_SACKED_ACKED bits even if some of
                 * the packets may have been never retransmitted.
                 */
                if (flag & FLAG_RETRANS_DATA_ACKED)
                        flag &= ~FLAG_ORIG_SACK_ACKED;
        } else {
                int delta;

                /* Non-retransmitted hole got filled? That's reordering */
                if (before(reord, prior_fack))
                        tcp_check_sack_reordering(sk, reord, 0);

                delta = prior_sacked - tp->sacked_out;
                tp->lost_cnt_hint -= min(tp->lost_cnt_hint, delta);
        }
} else if (skb && rtt_update && sack_rtt_us >= 0 &&
           sack_rtt_us > tcp_stamp_us_delta(tp->tcp_mstamp,
                                            tcp_skb_timestamp_us(skb))) {
        /* Do not re-arm RTO if the sack RTT is measured from data sent
         * after when the head was last (re)transmitted. Otherwise the
         * timeout may continue to extend in loss recovery.
         */
        flag |= FLAG_SET_XMIT_TIMER;  /* set TLP or RTO timer */
}

if (icsk->icsk_ca_ops->pkts_acked) {
        struct ack_sample sample = { .pkts_acked = pkts_acked,
                                     .rtt_us = sack->rate->rtt_us,
                                     .in_flight = last_in_flight };

        icsk->icsk_ca_ops->pkts_acked(sk, &sample);
}

#if FASTRETRANS_DEBUG > 0
        WARN_ON((int)tp->sacked_out < 0);
        WARN_ON((int)tp->lost_out < 0);
        WARN_ON((int)tp->retrans_out < 0);
        if (!tp->packets_out && tcp_is_sack(tp)) {
                icsk = inet_csk(sk);
                if (tp->lost_out) {
                        pr_debug("Leak l=%u %d\n",
                                 tp->lost_out, icsk->icsk_ca_state);
                        tp->lost_out = 0;
                }
                if (tp->sacked_out) {
                        pr_debug("Leak s=%u %d\n",
                                 tp->sacked_out, icsk->icsk_ca_state);
                        tp->sacked_out = 0;
                }
                if (tp->retrans_out) {
                        pr_debug("Leak r=%u %d\n",
                                 tp->retrans_out, icsk->icsk_ca_state);
                        tp->retrans_out = 0;
                }
        }
#endif
        return flag;
}

static void tcp_ack_probe(struct sock *sk)
{
        struct inet_connection_sock *icsk = inet_csk(sk);
        struct sk_buff *head = tcp_send_head(sk);
        const struct tcp_sock *tp = tcp_sk(sk);

        /* Was it a usable window open? */
        if (!head)
                return;
        if (!after(TCP_SKB_CB(head)->end_seq, tcp_wnd_end(tp))) {
                icsk->icsk_backoff = 0;
                icsk->icsk_probes_tstamp = 0;
                inet_csk_clear_xmit_timer(sk, ICSK_TIME_PROBE0);
                /* Socket must be waked up by subsequent tcp_data_snd_check().
                 * This function is not for random using!
                 */
        } else {
                unsigned long when = tcp_probe0_when(sk, TCP_RTO_MAX);

                when = tcp_clamp_probe0_to_user_timeout(sk, when);
                tcp_reset_xmit_timer(sk, ICSK_TIME_PROBE0, when, TCP_RTO_MAX);
        }
}

static inline bool tcp_ack_is_dubious(const struct sock *sk, const int flag)
{
        return !(flag & FLAG_NOT_DUP) || (flag & FLAG_CA_ALERT) ||
                inet_csk(sk)->icsk_ca_state != TCP_CA_Open;
}

/* Decide wheather to run the increase function of congestion control. */
static inline bool tcp_may_raise_cwnd(const struct sock *sk, const int flag)
{
        /* If reordering is high then always grow cwnd whenever data is
         * delivered regardless of its ordering. Otherwise stay conservative
         * and only grow cwnd on in-order delivery (RFC5681). A stretched ACK w/
         * new SACK or ECE mark may first advance cwnd here and later reduce
         * cwnd in tcp_fastretrans_alert() based on more states.
         */
        if (tcp_sk(sk)->reordering > sock_net(sk)->ipv4.sysctl_tcp_reordering)
                return flag & FLAG_FORWARD_PROGRESS;

        return flag & FLAG_DATA_ACKED;
}

/* The "ultimate" congestion control function that aims to replace the rigid
 * cwnd increase and decrease control (tcp_cong_avoid,tcp_*cwnd_reduction).
 * It's called toward the end of processing an ACK with precise rate
 * information. All transmission or retransmission are delayed afterwards.
 */
static void tcp_cong_control(struct sock *sk, u32 ack, u32 acked_sacked,
                             int flag, const struct rate_sample *rs)
{
        const struct inet_connection_sock *icsk = inet_csk(sk);

        if (icsk->icsk_ca_ops->cong_control) {
                icsk->icsk_ca_ops->cong_control(sk, rs);
                return;
        }

        if (tcp_in_cwnd_reduction(sk)) {
                /* Reduce cwnd if state mandates */
                tcp_cwnd_reduction(sk, acked_sacked, rs->losses, flag);
        } else if (tcp_may_raise_cwnd(sk, flag)) {
                /* Advance cwnd if state allows */
                tcp_cong_avoid(sk, ack, acked_sacked);
        }
        tcp_update_pacing_rate(sk);
}

/* Check that window update is acceptable.
 * The function assumes that snd_una<=ack<=snd_next.
 */
static inline bool tcp_may_update_window(const struct tcp_sock *tp,
                                         const u32 ack, const u32 ack_seq,
                                         const u32 nwin)
{
        return  after(ack, tp->snd_una) ||
                after(ack_seq, tp->snd_wl1) ||
                (ack_seq == tp->snd_wl1 && nwin > tp->snd_wnd);
}

/* If we update tp->snd_una, also update tp->bytes_acked */
static void tcp_snd_una_update(struct tcp_sock *tp, u32 ack)
{
        u32 delta = ack - tp->snd_una;

        sock_owned_by_me((struct sock *)tp);
        tp->bytes_acked += delta;
        WRITE_ONCE(tp->rcv_nxt, seq);
}

/* Update our send window.
 *
 * Window update algorithm, described in RFC793/RFC1122 (used in linux-2.2
 * and in FreeBSD. NetBSD's one is even worse.) is wrong.
 */
static int tcp_ack_update_window(struct sock *sk, const struct sk_buff *skb, u32 ack,
                                 u32 ack_seq)
{
        struct tcp_sock *tp = tcp_sk(sk);
        int flag = 0;
        u32 nwin = ntohs(tcp_hdr(skb)->window);

        if (likely(!tcp_hdr(skb)->syn))
                nwin <<= tp->rx_opt.snd_wscale;

        if (tcp_may_update_window(tp, ack, ack_seq, nwin)) {
                flag |= FLAG_WIN_UPDATE;
                tcp_update_wl(tp, ack_seq);

                if (tp->snd_wnd != nwin) {
                        tp->snd_wnd = nwin;

                        /* Note, it is the only place, where
                         * fast path is recovered for sending TCP.
                         */
                        tp->pred_flags = 0;
                        tcp_fast_path_check(sk);

                        if (!tcp_write_queue_empty(sk))
                                tcp_slow_start_after_idle_check(sk);

                        if (nwin > tp->max_window) {
                                tp->max_window = nwin;
                                tcp_sync_mss(sk, inet_csk(sk)->icsk_pmtu_cookie);
                        }
                }
        }

        tcp_snd_una_update(tp, ack);

        return flag;
}

static bool __tcp_oow_rate_limited(struct net *net, int mib_idx,
                                   u32 *last_oow_ack_time)
{
        if (*last_oow_ack_time) {
                s32 elapsed = (s32)(tcp_jiffies32 - *last_oow_ack_time);

                if (0 <= elapsed && elapsed < net->ipv4.sysctl_tcp_invalid_ratelimit) {
                        NET_INC_STATS(net, mib_idx);
                        return true;    /* rate-limited: don't send yet! */
                }
        }

        *last_oow_ack_time = tcp_jiffies32;

        return false;   /* not rate-limited: go ahead, send dupack now! */
}

/* Return true if we're currently rate-limiting out-of-window ACKs and
 * thus shouldn't send a dupack right now. We rate-limit dupacks in
 * response to out-of-window SYNs or ACKs to mitigate ACK loops or DoS
 * attacks that send repeated SYNs or ACKs for the same connection. To
 * do this, we do not send a duplicate SYNACK or ACK if the remote
 * endpoint is sending out-of-window SYNs or pure ACKs at a high rate.
 */
bool tcp_oow_rate_limited(struct net *net, const struct sk_buff *skb,
                          int mib_idx, u32 *last_oow_ack_time)
{
        /* Data packets without SYNs are not likely part of an ACK loop. */
        if ((TCP_SKB_CB(skb)->seq != TCP_SKB_CB(skb)->end_seq) &&
            !tcp_hdr(skb)->syn)
```




When we cannot observe CCA behavior from the implementation, we can use packet traces



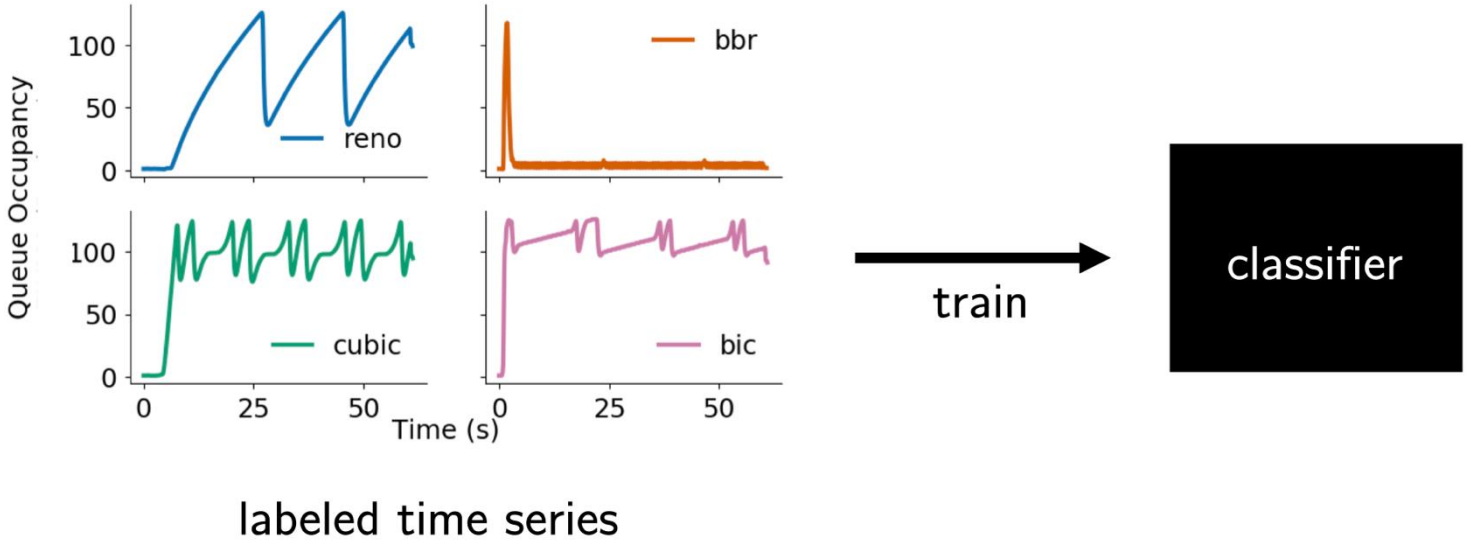
One way to uncover CCAs in the wild: Classification



labeled time series

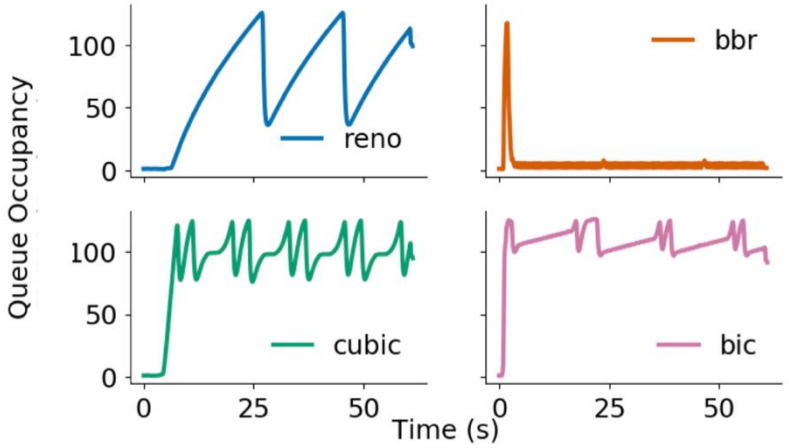
R Ware, A A Philip, N Hungria, Y Kothari, J Sherry, and S Seshan. CCAnalyzer: An Efficient and Nearly-Passive Congestion Control Classifier. In SIGCOMM 2024.
Ayush Mishra, Lakshay Rastogi, Raj Joshi, and Ben Leong. Keeping an Eye on Congestion Control in the Wild with Nebby. In SIGCOMM 2024.

One way to uncover CCAs in the wild: Classification



R Ware, A A Philip, N Hungria, Y Kothari, J Sherry, and S Seshan. CCAnalyzer: An Efficient and Nearly-Passive Congestion Control Classifier. In SIGCOMM 2024.
Ayush Mishra, Lakshay Rastogi, Raj Joshi, and Ben Leong. Keeping an Eye on Congestion Control in the Wild with Nebby. In SIGCOMM 2024.

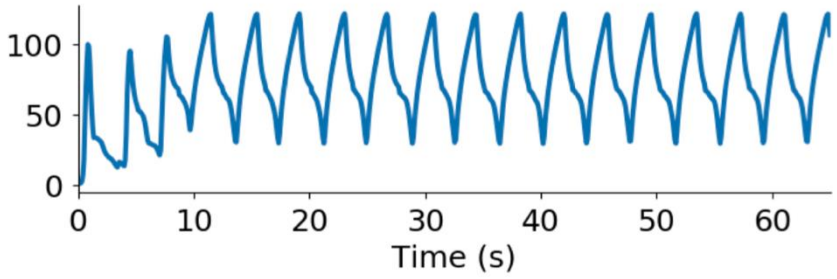
One way to uncover CCAs in the wild: Classification



labeled time series

train

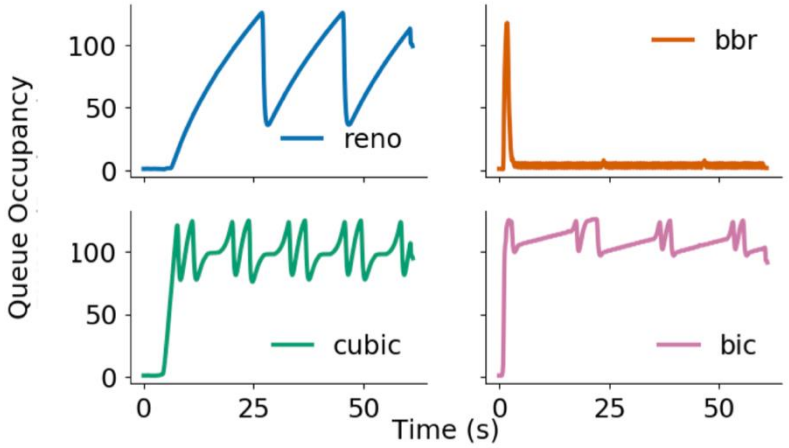
classifier



unlabeled time series

R Ware, A A Philip, N Hungria, Y Kothari, J Sherry, and S Seshan. CCAnalyzer: An Efficient and Nearly-Passive Congestion Control Classifier. In SIGCOMM 2024.
Ayush Mishra, Lakshay Rastogi, Raj Joshi, and Ben Leong. Keeping an Eye on Congestion Control in the Wild with Nebby. In SIGCOMM 2024.

One way to uncover CCAs in the wild: Classification



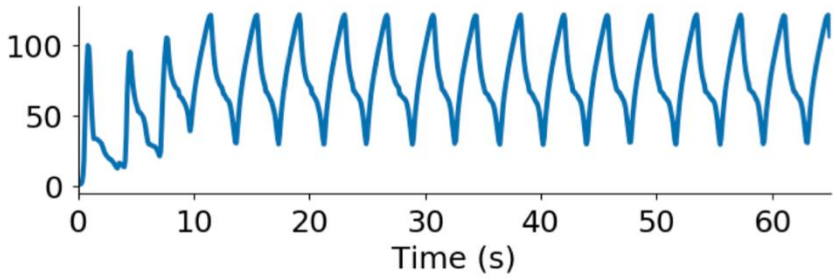
labeled time series

train

classifier

classify

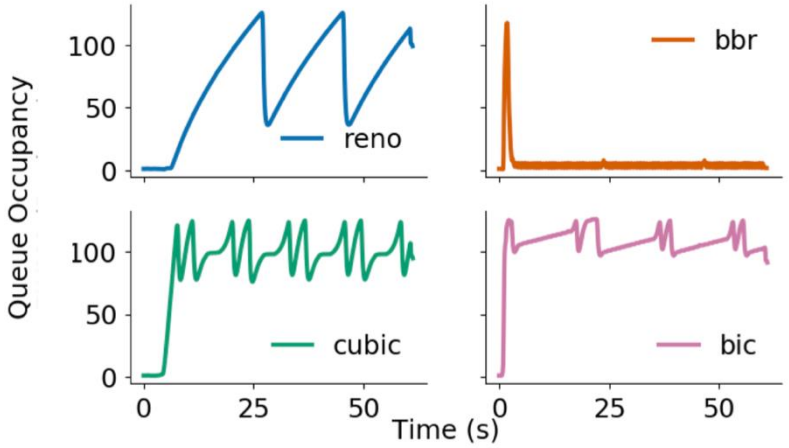
predicted label or "unknown"



unlabeled time series

R Ware, A A Philip, N Hungria, Y Kothari, J Sherry, and S Seshan. CCAnalyzer: An Efficient and Nearly-Passive Congestion Control Classifier. In SIGCOMM 2024.
Ayush Mishra, Lakshay Rastogi, Raj Joshi, and Ben Leong. Keeping an Eye on Congestion Control in the Wild with Nebby. In SIGCOMM 2024.

One way to uncover CCAs in the wild: Classification



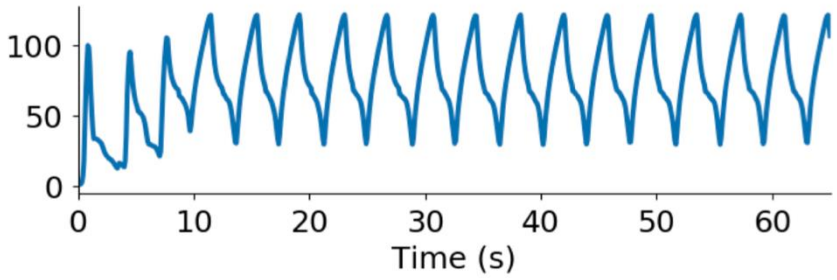
labeled time series

train

classifier

classify

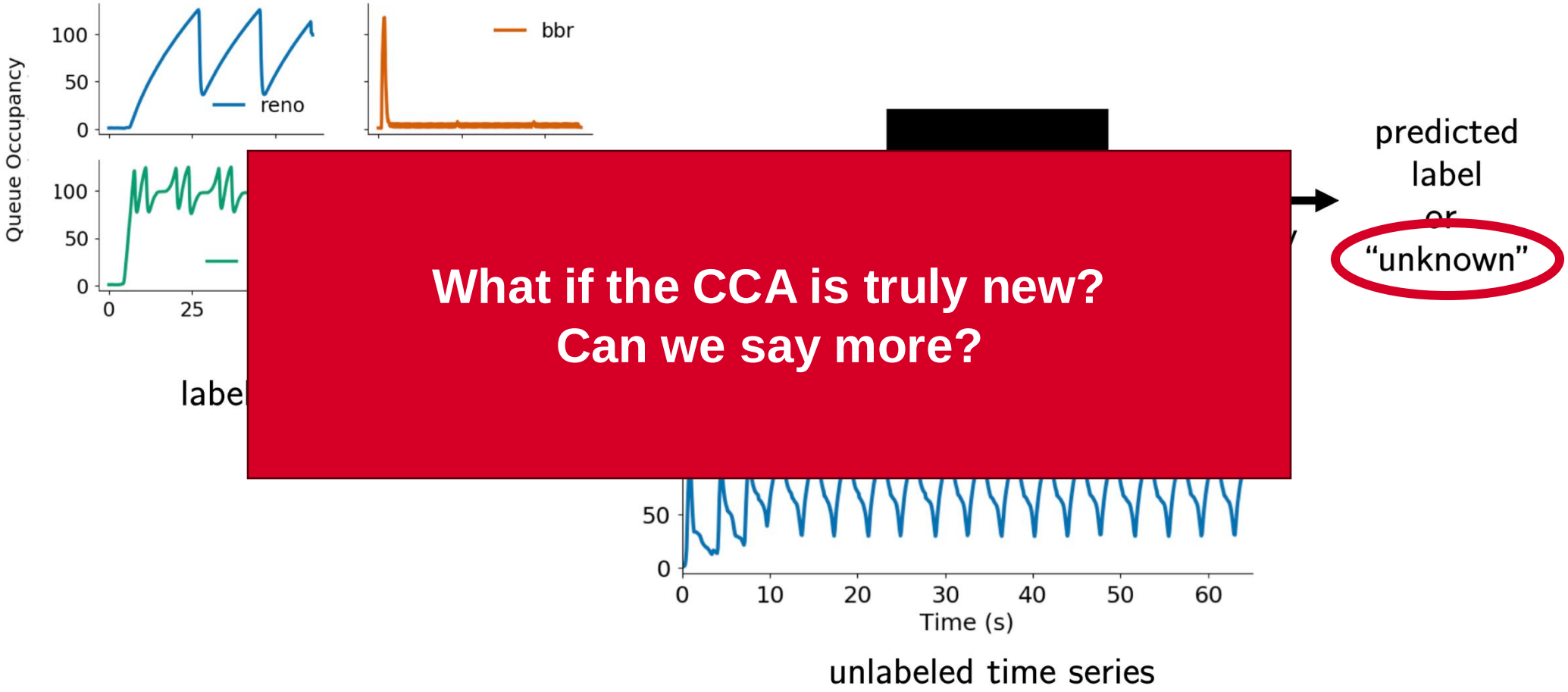
predicted label or "unknown"



unlabeled time series

R Ware, A A Philip, N Hungria, Y Kothari, J Sherry, and S Seshan. CCAnalyzer: An Efficient and Nearly-Passive Congestion Control Classifier. In SIGCOMM 2024.
Ayush Mishra, Lakshay Rastogi, Raj Joshi, and Ben Leong. Keeping an Eye on Congestion Control in the Wild with Nebby. In SIGCOMM 2024.

One way to uncover CCAs in the wild: Classification



R Ware, A A Philip, N Hungria, Y Kothari, J Sherry, and S Seshan. CCAnalyzer: An Efficient and Nearly-Passive Congestion Control Classifier. In SIGCOMM 2024.
Ayush Mishra, Lakshay Rastogi, Raj Joshi, and Ben Leong. Keeping an Eye on Congestion Control in the Wild with Nebby. In SIGCOMM 2024.

Generate simple implementations of CCAs from packet traces showing their behavior



Generate simple implementations of CCAs from packet traces showing their behavior

✓ ease the analysis of known CCAs



Generate simple implementations of CCAs from packet traces showing their behavior

- ✓ ease the analysis of known CCAs
- ✓ enable the analysis of unknown CCAs



Abagnale uses program synthesis to reverse engineer CCAs



Most congestion control code is boilerplate



```
        tcp_rate_skb_delivered(sk, skb, sack->rate);

        /* Initial outgoing SYN's get put onto the write_queue
         * just like anything else we transmit.  It is not
         * true data, and if we misinform our callers that
         * this ACK acks real data, we will erroneously exit
         * connection startup slow start one packet too
         * quickly.  This is severely frowned upon behavior.
         */
        if (likely(!(scb->tcp_flags & TCPHDR_SYN))) {
                flag |= FLAG_DATA_ACKED;
        } else {
                flag |= FLAG_SYN_ACKED;
                tp->retrans_stamp = 0;
        }

        if (!fully_acked)
                break;

        tcp_ack_tstamp(sk, skb, ack_skb, prior_snd_una);

        next = skb_rb_next(skb);
        if (unlikely(skb == tp->retransmit_skb_hint))
                tp->retransmit_skb_hint = NULL;
        if (unlikely(skb == tp->lost_skb_hint))
                tp->lost_skb_hint = NULL;
        tcp_highest_sack_replace(sk, skb, next);
        tcp_rtx_queue_unlink_and_free(skb, sk);
}

if (!skb)
        tcp_chrono_stop(sk, TCP_CHRONO_BUSY);

if (likely(between(tp->snd_up, prior_snd_una, tp->snd_una)))
        tp->snd_up = tp->snd_una;

if (skb) {
        tcp_ack_tstamp(sk, skb, ack_skb, prior_snd_una);
        if (TCP_SKB_CB(skb)->sacked & TCPCB_SACKED_ACKED)
                flag |= FLAG_SACK_RENEGING;
}

if (likely(first_ackt) && !(flag & FLAG_RETRANS_DATA_ACKED)) {
        seq_rtt_us = tcp_stamp_us_delta(tp->tcp_mstamp, first_ackt);
        ca_rtt_us = tcp_stamp_us_delta(tp->tcp_mstamp, last_ackt);

        if (pkts_acked == 1 && last_in_flight < tp->mss_cache &&
            last_in_flight && !prior_sacked && fully_acked &&
            sack->rate->prior_delivered + 1 == tp->delivered &&
            !(flag & (FLAG_CA_ALERT | FLAG_SYN_ACKED))) {
                /* Conservatively mark a delayed ACK. It's typically
                 * from a lone runt packet over the round trip to
                 * a receiver w/o out-of-order or CE events.
                 */
                flag |= FLAG_ACK_MAYBE_DELAYED;
        }
}
if (sack->first_sackt) {
        sack_rtt_us = tcp_stamp_us_delta(tp->tcp_mstamp, sack->first_sackt);
        ca_rtt_us = tcp_stamp_us_delta(tp->tcp_mstamp, sack->last_sackt);
}
rtt_update = tcp_ack_update_rtt(sk, flag, seq_rtt_us, sack_rtt_us,
                                ca_rtt_us, sack->rate);

if (flag & FLAG_ACKED) {
        flag |= FLAG_SET_XMIT_TIMER;  /* set TLP or RTO timer */
        if (unlikely(icsk->icsk_mtup.probe_size &&
                     !after(tp->mtu_probe.probe_seq_end, tp->snd_una))) {
                tcp_mtup_probe_success(sk);
        }

        if (tcp_is_reno(tp)) {
                tcp_remove_reno_sacks(sk, pkts_acked, ece_ack);

                /* If any of the cumulatively ACKed segments was
                 * retransmitted, non-SACK case cannot confirm that
                 * progress was due to original transmission due to
                 * lack of TCPCB_SACKED_ACKED bits even if some of
                 * the packets may have been never retransmitted.
                 */
                if (flag & FLAG_RETRANS_DATA_ACKED)
                        flag &= ~FLAG_ORIG_SACK_ACKED;
        } else {
                int delta;

                /* Non-retransmitted hole got filled? That's reordering */
                if (before(reord, prior_fack))
                        tcp_check_sack_reordering(sk, reord, 0);

                delta = prior_sacked - tp->sacked_out;
                tp->lost_cnt_hint -= min(tp->lost_cnt_hint, delta);
        }
} else if (skb && rtt_update && sack_rtt_us >= 0 &&
           sack_rtt_us > tcp_stamp_us_delta(tp->tcp_mstamp,
                                            tcp_skb_timestamp_us(skb))) {
        /* Do not re-arm RTO if the sack RTT is measured from data sent
         * after when the head was last (re)transmitted. Otherwise the
         * timeout may continue to extend in loss recovery.
         */
        flag |= FLAG_SET_XMIT_TIMER;  /* set TLP or RTO timer */
}

if (icsk->icsk_ca_ops->pkts_acked) {
        struct ack_sample sample = { .pkts_acked = pkts_acked,
                                     .rtt_us = sack->rate->rtt_us,
                                     .in_flight = last_in_flight };

        icsk->icsk_ca_ops->pkts_acked(sk, &sample);
}

#if FASTRETRANS_DEBUG > 0
        WARN_ON((int)tp->sacked_out < 0);
        WARN_ON((int)tp->lost_out < 0);
        WARN_ON((int)tp->retrans_out < 0);
        if (!tp->packets_out && tcp_is_sack(tp)) {
                icsk = inet_csk(sk);
                if (tp->lost_out) {
                        pr_debug("Leak l=%u %d\n",
                                 tp->lost_out, icsk->icsk_ca_state);
                        tp->lost_out = 0;
                }
                if (tp->sacked_out) {
                        pr_debug("Leak s=%u %d\n",
                                 tp->sacked_out, icsk->icsk_ca_state);
                        tp->sacked_out = 0;
                }
                if (tp->retrans_out) {
                        pr_debug("Leak r=%u %d\n",
                                 tp->retrans_out, icsk->icsk_ca_state);
                        tp->retrans_out = 0;
                }
        }
#endif
        return flag;
}

static void tcp_ack_probe(struct sock *sk)
{
        struct inet_connection_sock *icsk = inet_csk(sk);
        struct sk_buff *head = tcp_send_head(sk);
        const struct tcp_sock *tp = tcp_sk(sk);

        /* Was it a usable window open? */
        if (!head)
                return;
        if (!after(TCP_SKB_CB(head)->end_seq, tcp_wnd_end(tp))) {
                icsk->icsk_backoff = 0;
                icsk->icsk_probes_tstamp = 0;
                inet_csk_clear_xmit_timer(sk, ICSK_TIME_PROBE0);
                /* Socket must be waked up by subsequent tcp_data_snd_check().
                 * This function is not for random using!
                 */
        } else {
                unsigned long when = tcp_probe0_when(sk, TCP_RTO_MAX);

                when = tcp_clamp_probe0_to_user_timeout(sk, when);
                tcp_reset_xmit_timer(sk, ICSK_TIME_PROBE0, when, TCP_RTO_MAX);
        }
}

static inline bool tcp_ack_is_dubious(const struct sock *sk, const int flag)
{
        return !(flag & FLAG_NOT_DUP) || (flag & FLAG_CA_ALERT) ||
                inet_csk(sk)->icsk_ca_state != TCP_CA_Open;
}

/* Decide wheather to run the increase function of congestion control. */
static inline bool tcp_may_raise_cwnd(const struct sock *sk, const int flag)
{
        /* If reordering is high then always grow cwnd whenever data is
         * delivered regardless of its ordering. Otherwise stay conservative
         * and only grow cwnd on in-order delivery (RFC5681). A stretched ACK w/
         * new SACK or ECE mark may first advance cwnd here and later reduce
         * cwnd in tcp_fastretrans_alert() based on more states.
         */
        if (tcp_sk(sk)->reordering > sock_net(sk)->ipv4.sysctl_tcp_reordering)
                return flag & FLAG_FORWARD_PROGRESS;

        return flag & FLAG_DATA_ACKED;
}

/* The "ultimate" congestion control function that aims to replace the rigid
 * cwnd increase and decrease control (tcp_cong_avoid,tcp_*cwnd_reduction).
 * It's called toward the end of processing an ACK with precise rate
 * information. All transmission or retransmission are delayed afterwards.
 */
static void tcp_cong_control(struct sock *sk, u32 ack, u32 acked_sacked,
                             int flag, const struct rate_sample *rs)
{
        const struct inet_connection_sock *icsk = inet_csk(sk);

        if (icsk->icsk_ca_ops->cong_control) {
                icsk->icsk_ca_ops->cong_control(sk, rs);
                return;
        }

        if (tcp_in_cwnd_reduction(sk)) {
                /* Reduce cwnd if state mandates */
                tcp_cwnd_reduction(sk, acked_sacked, rs->losses, flag);
        } else if (tcp_may_raise_cwnd(sk, flag)) {
                /* Advance cwnd if state allows */
                tcp_cong_avoid(sk, ack, acked_sacked);
        }
        tcp_update_pacing_rate(sk);
}

/* Check that window update is acceptable.
 * The function assumes that snd_una<=ack<=snd_next.
 */
static inline bool tcp_may_update_window(const struct tcp_sock *tp,
                                         const u32 ack, const u32 ack_seq,
                                         const u32 nwin)
{
        return  after(ack, tp->snd_una) ||
                after(ack_seq, tp->snd_wl1) ||
                (ack_seq == tp->snd_wl1 && nwin > tp->snd_wnd);
}

/* If we update tp->snd_una, also update tp->bytes_acked */
static void tcp_snd_una_update(struct tcp_sock *tp, u32 ack)
{
        u32 delta = ack - tp->snd_una;

        sock_owned_by_me((struct sock *)tp);
        tp->bytes_acked += delta;
        WRITE_ONCE(tp->rcv_nxt, seq);
}

/* Update our send window.
 *
 * Window update algorithm, described in RFC793/RFC1122 (used in linux-2.2
 * and in FreeBSD. NetBSD's one is even worse.) is wrong.
 */
static int tcp_ack_update_window(struct sock *sk, const struct sk_buff *skb, u32 ack,
                                 u32 ack_seq)
{
        struct tcp_sock *tp = tcp_sk(sk);
        int flag = 0;
        u32 nwin = ntohs(tcp_hdr(skb)->window);

        if (likely(!tcp_hdr(skb)->syn))
                nwin <<= tp->rx_opt.snd_wscale;

        if (tcp_may_update_window(tp, ack, ack_seq, nwin)) {
                flag |= FLAG_WIN_UPDATE;
                tcp_update_wl(tp, ack_seq);

                if (tp->snd_wnd != nwin) {
                        tp->snd_wnd = nwin;

                        /* Note, it is the only place, where
                         * fast path is recovered for sending TCP.
                         */
                        tp->pred_flags = 0;
                        tcp_fast_path_check(sk);

                        if (!tcp_write_queue_empty(sk))
                                tcp_slow_start_after_idle_check(sk);

                        if (nwin > tp->max_window) {
                                tp->max_window = nwin;
                                tcp_sync_mss(sk, inet_csk(sk)->icsk_pmtu_cookie);
                        }
                }
        }

        tcp_snd_una_update(tp, ack);

        return flag;
}

static bool __tcp_oow_rate_limited(struct net *net, int mib_idx,
                                   u32 *last_oow_ack_time)
{
        if (*last_oow_ack_time) {
                s32 elapsed = (s32)(tcp_jiffies32 - *last_oow_ack_time);

                if (0 <= elapsed && elapsed < net->ipv4.sysctl_tcp_invalid_ratelimit) {
                        NET_INC_STATS(net, mib_idx);
                        return true;    /* rate-limited: don't send yet! */
                }
        }

        *last_oow_ack_time = tcp_jiffies32;

        return false;   /* not rate-limited: go ahead, send dupack now! */
}

/* Return true if we're currently rate-limiting out-of-window ACKs and
 * thus shouldn't send a dupack right now. We rate-limit dupacks in
 * response to out-of-window SYNs or ACKs to mitigate ACK loops or DoS
 * attacks that send repeated SYNs or ACKs for the same connection. To
 * do this, we do not send a duplicate SYNACK or ACK if the remote
 * endpoint is sending out-of-window SYNs or pure ACKs at a high rate.
 */
bool tcp_oow_rate_limited(struct net *net, const struct sk_buff *skb,
                          int mib_idx, u32 *last_oow_ack_time)
{
        /* Data packets without SYNs are not likely part of an ACK loop. */
        if ((TCP_SKB_CB(skb)->seq != TCP_SKB_CB(skb)->end_seq) &&
            !tcp_hdr(skb)->syn)
```




Most congestion control code is boilerplate



```
tcp_rate_skb_delivered(sk, skb, sack->rate);

/* Initial outgoing SYN's get put onto the write_queue
 * just like anything else we transmit.  It is not
 * true data, and if we misinform our callers that
 * this ACK acks real data, we will erroneously exit
 * connection startup slow start one packet too
 * quickly.  This is severely frowned upon behavior.
 */
if (likely(!(scb->tcp_flags & TCPHDR_SYN))) {
        flag |= FLAG_DATA_ACKED;
} else {
        flag |= FLAG_SYN_ACKED;
        tp->retrans_stamp = 0;
}

if (!fully_acked)
        break;

tcp_ack_tstamp(sk, skb, ack_skb, prior_snd_una);

next = skb_rb_next(skb);
if (unlikely(skb == tp->retransmit_skb_hint))
        tp->retransmit_skb_hint = NULL;
if (unlikely(skb == tp->lost_skb_hint))
        tp->lost_skb_hint = NULL;
tcp_highest_sack_replace(sk, skb, next);
tcp_rtx_queue_unlink_and_free(skb, sk);
}

if (!skb)
        tcp_chrono_stop(sk, TCP_CHRONO_BUSY);

if (likely(between(tp->snd_up, prior_snd_una, tp->snd_una)))
        tp->snd_up = tp->snd_una;

if (skb) {
        tcp_ack_tstamp(sk, skb, ack_skb, prior_snd_una);
        if (TCP_SKB_CB(skb)->sacked & TCPCB_SACKED_ACKED)
                flag |= FLAG_SACK_RENEGING;
}

if (likely(first_ackt) && !(flag & FLAG_RETRANS_DATA_ACKED)) {
        seq_rtt_us = tcp_stamp_us_delta(tp->tcp_mstamp, first_ackt);
        ca_rtt_us = tcp_stamp_us_delta(tp->tcp_mstamp, last_ackt);

        if (pkts_acked == 1 && last_in_flight < tp->mss_cache &&
            last_in_flight && !prior_sacked && fully_acked &&
            sack->rate->prior_delivered + 1 == tp->delivered &&
            !(flag & (FLAG_CA_ALERT | FLAG_SYN_ACKED))) {
                /* Conservatively mark a delayed ACK. It's typically
                 * from a lone runt packet over the round trip to
                 * a receiver w/o out-of-order or CE events.
                 */
                flag |= FLAG_ACK_MAYBE_DELAYED;
        }
}
if (sack->first_sackt) {
        sack_rtt_us = tcp_stamp_us_delta(tp->tcp_mstamp, sack->first_sackt);
        ca_rtt_us = tcp_stamp_us_delta(tp->tcp_mstamp, sack->last_sackt);
}
rtt_update = tcp_ack_update_rtt(sk, flag, seq_rtt_us, sack_rtt_us,
                                ca_rtt_us, sack->rate);

if (flag & FLAG_ACKED) {
        flag |= FLAG_SET_XMIT_TIMER;  /* set TLP or RTO timer */
        if (unlikely(icsk->icsk_mtup.probe_size &&
                     !after(tp->mtu_probe.probe_seq_end, tp->snd_una))) {
                tcp_mtup_probe_success(sk);
        }

        if (tcp_is_reno(tp)) {
                tcp_remove_reno_sacks(sk, pkts_acked, ece_ack);

                /* If any of the cumulatively ACKed segments was
                 * retransmitted, non-SACK case cannot confirm that
                 * progress was due to original transmission due to
                 * lack of TCPCB_SACKED_ACKED bits even if some of
                 * the packets may have been never retransmitted.
                 */
                if (flag & FLAG_RETRANS_DATA_ACKED)
                        flag &= ~FLAG_ORIG_SACK_ACKED;
        } else {
                int delta;

                /* Non-retransmitted hole got filled? That's reordering */
                if (before(reord, prior_fack))
                        tcp_check_sack_reordering(sk, reord, 0);

                delta = prior_sacked - tp->sacked_out;
                tp->lost_cnt_hint -= min(tp->lost_cnt_hint, delta);
        }
} else if (skb && rtt_update && sack_rtt_us >= 0 &&
           sack_rtt_us > tcp_stamp_us_delta(tp->tcp_mstamp,
                                            tcp_skb_timestamp_us(skb))) {
        /* Do not re-arm RTO if the sack RTT is measured from data sent
         * after when the head was last (re)transmitted. Otherwise the
         * timeout may continue to extend in loss recovery.
         */
        flag |= FLAG_SET_XMIT_TIMER;  /* set TLP or RTO timer */
}

if (icsk->icsk_ca_ops->pkts_acked) {
        struct ack_sample sample = { .pkts_acked = pkts_acked,
                                     .rtt_us = sack->rate->rtt_us,
                                     .in_flight = last_in_flight };

        icsk->icsk_ca_ops->pkts_acked(sk, &sample);
}

#if FASTRETRANS_DEBUG > 0
        WARN_ON((int)tp->sacked_out < 0);
        WARN_ON((int)tp->lost_out < 0);
        WARN_ON((int)tp->retrans_out < 0);
        if (!tp->packets_out && tcp_is_sack(tp)) {
                icsk = inet_csk(sk);
                if (tp->lost_out) {
                        pr_debug("Leak l=%u %d\n",
                                 tp->lost_out, icsk->icsk_ca_state);
                        tp->lost_out = 0;
                }
                if (tp->sacked_out) {
                        pr_debug("Leak s=%u %d\n",
                                 tp->sacked_out, icsk->icsk_ca_state);
                        tp->sacked_out = 0;
                }
                if (tp->retrans_out) {
                        pr_debug("Leak r=%u %d\n",
                                 tp->retrans_out, icsk->icsk_ca_state);
                        tp->retrans_out = 0;
                }
        }
#endif
        return flag;
}

static void tcp_ack_probe(struct sock *sk)
{
        struct inet_connection_sock *icsk = inet_csk(sk);
        struct sk_buff *head = tcp_send_head(sk);
        const struct tcp_sock *tp = tcp_sk(sk);

        /* Was it a usable window open? */
        if (!head)
                return;
        if (!after(TCP_SKB_CB(head)->end_seq, tcp_wnd_end(tp))) {
                icsk->icsk_backoff = 0;
                icsk->icsk_probes_tstamp = 0;
                inet_csk_clear_xmit_timer(sk, ICSK_TIME_PROBE0);
                /* Socket must be waked up by subsequent tcp_data_snd_check().
                 * This function is not for random using!
                 */
        } else {
                unsigned long when = tcp_probe0_when(sk, TCP_RTO_MAX);

                when = tcp_clamp_probe0_to_user_timeout(sk, when);
                tcp_reset_xmit_timer(sk, ICSK_TIME_PROBE0, when, TCP_RTO_MAX);
        }
}

static inline bool tcp_ack_is_dubious(const struct sock *sk, const int flag)
{
        return !(flag & FLAG_NOT_DUP) || (flag & FLAG_CA_ALERT) ||
                inet_csk(sk)->icsk_ca_state != TCP_CA_Open;
}

/* Decide wheather to run the increase function of congestion control. */
static inline bool tcp_may_raise_cwnd(const struct sock *sk, const int flag)
{
        /* If reordering is high then always grow cwnd whenever data is
         * delivered regardless of its ordering. Otherwise stay conservative
         * and only grow cwnd on in-order delivery (RFC5681). A stretched ACK w/
         * new SACK or ECE mark may first advance cwnd here and later reduce
         * cwnd in tcp_fastretrans_alert() based on more states.
         */
        if (tcp_sk(sk)->reordering > sock_net(sk)->ipv4.sysctl_tcp_reordering)
                return flag & FLAG_FORWARD_PROGRESS;

        return flag & FLAG_DATA_ACKED;
}

/* The "ultimate" congestion control function that aims to replace the rigid
 * cwnd increase and decrease control (tcp_cong_avoid,tcp_*cwnd_reduction).
 * It's called toward the end of processing an ACK with precise rate
 * information. All transmission or retransmission are delayed afterwards.
 */
static void tcp_cong_control(struct sock *sk, u32 ack, u32 acked_sacked,
                             int flag, const struct rate_sample *rs)
{
        const struct inet_connection_sock *icsk = inet_csk(sk);

        if (icsk->icsk_ca_ops->cong_control) {
                icsk->icsk_ca_ops->cong_control(sk, rs);
                return;
        }

        if (tcp_in_cwnd_reduction(sk)) {
                /* Reduce cwnd if state mandates */
                tcp_cwnd_reduction(sk, acked_sacked, rs->losses, flag);
        } else if (tcp_may_raise_cwnd(sk, flag)) {
                /* Advance cwnd if state allows */
                tcp_cong_avoid(sk, ack, acked_sacked);
        }
        tcp_update_pacing_rate(sk);
}

/* Check that window update is acceptable.
 * The function assumes that snd_una<=ack<=snd_next.
 */
static inline bool tcp_may_update_window(const struct tcp_sock *tp,
                                         const u32 ack, const u32 ack_seq,
                                         const u32 nwin)
{
        return  after(ack, tp->snd_una) ||
                after(ack_seq, tp->snd_wl1) ||
                (ack_seq == tp->snd_wl1 && nwin > tp->snd_wnd);
}

/* If we update tp->snd_una, also update tp->bytes_acked */
static void tcp_snd_una_update(struct tcp_sock *tp, u32 ack)
{
        u32 delta = ack - tp->snd_una;

        sock_owned_by_me((struct sock *)tp);
        tp->bytes_acked += delta;
        WRITE_ONCE(tp->rcv_nxt, seq);
}

/* Update our send window.
 *
 * Window update algorithm, described in RFC793/RFC1122 (used in linux-2.2
 * and in FreeBSD. NetBSD's one is even worse.) is wrong.
 */
static int tcp_ack_update_window(struct sock *sk, const struct sk_buff *skb, u32 ack,
                                 u32 ack_seq)
{
        struct tcp_sock *tp = tcp_sk(sk);
        int flag = 0;
        u32 nwin = ntohs(tcp_hdr(skb)->window);

        if (likely(!tcp_hdr(skb)->syn))
                nwin <<= tp->rx_opt.snd_wscale;

        if (tcp_may_update_window(tp, ack, ack_seq, nwin)) {
                flag |= FLAG_WIN_UPDATE;
                tcp_update_wl(tp, ack_seq);

                if (tp->snd_wnd != nwin) {
                        tp->snd_wnd = nwin;

                        /* Note, it is the only place, where
                         * fast path is recovered for sending TCP.
                         */
                        tp->pred_flags = 0;
                        tcp_fast_path_check(sk);

                        if (!tcp_write_queue_empty(sk))
                                tcp_slow_start_after_idle_check(sk);

                        if (nwin > tp->max_window) {
                                tp->max_window = nwin;
                                tcp_sync_mss(sk, inet_csk(sk)->icsk_pmtu_cookie);
                        }
                }
        }

        tcp_snd_una_update(tp, ack);

        return flag;
}

static bool __tcp_oow_rate_limited(struct net *net, int mib_idx,
                                   u32 *last_oow_ack_time)
{
        if (*last_oow_ack_time) {
                s32 elapsed = (s32)(tcp_jiffies32 - *last_oow_ack_time);

                if (0 <= elapsed && elapsed < net->ipv4.sysctl_tcp_invalid_ratelimit) {
                        NET_INC_STATS(net, mib_idx);
                        return true;    /* rate-limited: don't send yet! */
                }
        }

        *last_oow_ack_time = tcp_jiffies32;

        return false;   /* not rate-limited: go ahead, send dupack now! */
}

/* Return true if we're currently rate-limiting out-of-window ACKs and
 * thus shouldn't send a dupack right now. We rate-limit dupacks in
 * response to out-of-window SYNs or ACKs to mitigate ACK loops or DoS
 * attacks that send repeated SYNs or ACKs for the same connection. To
 * do this, we do not send a duplicate SYNACK or ACK if the remote
 * endpoint is sending out-of-window SYNs or pure ACKs at a high rate.
 */
bool tcp_oow_rate_limited(struct net *net, const struct sk_buff *skb,
                          int mib_idx, u32 *last_oow_ack_time)
{
        /* Data packets without SYNs are not likely part of an ACK loop. */
        if ((TCP_SKB_CB(skb)->seq != TCP_SKB_CB(skb)->end_seq) &&
            !tcp_hdr(skb)->syn)
```




Most congestion control code is boilerplate

```
        tcp_rate_skb_delivered(sk, skb, sack->rate);

        /* Initial outgoing SYN's get put onto the write_queue
         * just like anything else we transmit.  It is not
         * true data, and if we misinform our callers that
         * this ACK acks real data, we will erroneously exit
         * connection startup slow start one packet too
         * quickly.  This is severely frowned upon behavior.
         */
        if (likely(!(scb->tcp_flags & TCPHDR_SYN))) {
                flag |= FLAG_DATA_ACKED;
        } else {
                flag |= FLAG_SYN_ACKED;
                tp->retrans_stamp = 0;
        }

        if (!fully_acked)
                break;

        tcp_ack_tstamp(sk, skb, ack_skb, prior_snd_una);

        next = skb_rb_next(skb);
        if (unlikely(skb == tp->retransmit_skb_hint))
                tp->retransmit_skb_hint = NULL;
        if (unlikely(skb == tp->lost_skb_hint))
                tp->lost_skb_hint = NULL;
        tcp_highest_sack_replace(sk, skb, next);
        tcp_rtx_queue_unlink_and_free(skb, sk);
}

if (!skb)
        tcp_chrono_stop(sk, TCP_CHRONO_BUSY);

if (likely(between(tp->snd_up, prior_snd_una, tp->snd_una)))
        tp->snd_up = tp->snd_una;

if (skb) {
        tcp_ack_tstamp(sk, skb, ack_skb, prior_snd_una);
        if (TCP_SKB_CB(skb)->sacked & TCPCB_SACKED_ACKED)
                flag |= FLAG_SACK_RENEGING;
}

if (likely(first_ackt) && !(flag & FLAG_RETRANS_DATA_ACKED)) {
        seq_rtt_us = tcp_stamp_us_delta(tp->tcp_mstamp, first_ackt);
        ca_rtt_us = tcp_stamp_us_delta(tp->tcp_mstamp, last_ackt);

        if (pkts_acked == 1 && last_in_flight < tp->mss_cache &&
            last_in_flight && !prior_sacked && fully_acked &&
            sack->rate->prior_delivered + 1 == tp->delivered &&
            !(flag & (FLAG_CA_ALERT | FLAG_SYN_ACKED))) {
                /* Conservatively mark a delayed ACK. It's typically
                 * from a lone runt packet over the round trip to
                 * a receiver w/o out-of-order or CE events.
                 */
                flag |= FLAG_ACK_MAYBE_DELAYED;
        }
}
if (sack->first_sackt) {
        sack_rtt_us = tcp_stamp_us_delta(tp->tcp_mstamp, sack->first_sackt);
        ca_rtt_us = tcp_stamp_us_delta(tp->tcp_mstamp, sack->last_sackt);
}
rtt_update = tcp_ack_update_rtt(sk, flag, seq_rtt_us, sack_rtt_us,
                                ca_rtt_us, sack->rate);

if (flag & FLAG_ACKED) {
        flag |= FLAG_SET_XMIT_TIMER;  /* set TLP or RTO timer */
        if (unlikely(icsk->icsk_mtup.probe_size &&
                     !after(tp->mtu_probe.probe_seq_end, tp->snd_una))) {
                tcp_mtup_probe_success(sk);
        }

        if (tcp_is_reno(tp)) {
                tcp_remove_reno_sacks(sk, pkts_acked, ece_ack);

                /* If any of the cumulatively ACKed segments was
                 * retransmitted, non-SACK case cannot confirm that
```

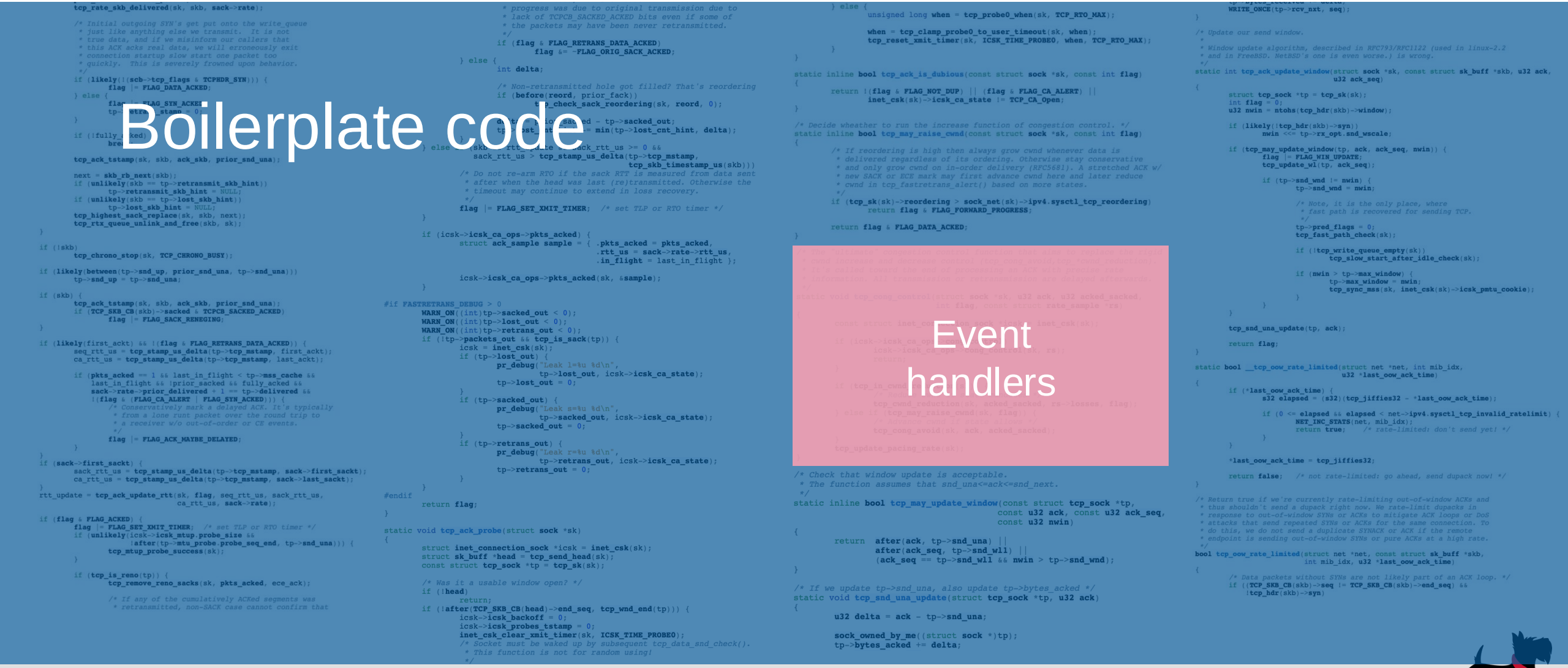
```
                         * progress was due to original transmission due to
                         * lack of TCPCB_SACKED_ACKED bits even if some of
                         * the packets may have been never retransmitted.
                         */
                        if (flag & FLAG_RETRANS_DATA_ACKED)
                                flag &= ~FLAG_ORIG_SACK_ACKED;
                } else {
                        int delta;

                        /* Non-retransmitted hole got filled? That's reordering */
                        if (before(reord, prior_fack))
                                tcp_check_sack_reordering(sk, reord, 0);

                        delta = prior_sacked - tp->sacked_out;
                        tp->lost_cnt_hint -= min(tp->lost_cnt_hint, delta);
                }
        } else if (skb && rtt_update && sack_rtt_us >= 0 &&
                   sack_rtt_us > tcp_stamp_us_delta(tp->tcp_mstamp,
                                                    tcp_skb_timestamp_us(skb))) {
                /* Do not re-arm RTO if the sack RTT is measured from data sent
                 * after when the head was last (re)transmitted. Otherwise the
                 * timeout may continue to extend in loss recovery.
                 */
                flag |= FLAG_SET_XMIT_TIMER;  /* set TLP or RTO timer */
        }

        if (icsk->icsk_ca_ops->pkts_acked) {
                struct ack_sample sample = { .pkts_acked = pkts_acked,
                                             .rtt_us = sack->rate->rtt_us,
                                             .in_flight = last_in_flight };

                icsk->icsk_ca_ops->pkts_acked(sk, &sample);
        }

#if FASTRETRANS_DEBUG > 0
        WARN_ON((int)tp->sacked_out < 0);
        WARN_ON((int)tp->lost_out < 0);
        WARN_ON((int)tp->retrans_out < 0);
        if (!tp->packets_out && tcp_is_sack(tp)) {
                icsk = inet_csk(sk);
                if (tp->lost_out) {
                        pr_debug("Leak l=%u %d\n",
                                 tp->lost_out, icsk->icsk_ca_state);
                        tp->lost_out = 0;
                }
                if (tp->sacked_out) {
                        pr_debug("Leak s=%u %d\n",
                                 tp->sacked_out, icsk->icsk_ca_state);
                        tp->sacked_out = 0;
                }
                if (tp->retrans_out) {
                        pr_debug("Leak r=%u %d\n",
                                 tp->retrans_out, icsk->icsk_ca_state);
                        tp->retrans_out = 0;
                }
        }
#endif
        return flag;
}

static void tcp_ack_probe(struct sock *sk)
{
        struct inet_connection_sock *icsk = inet_csk(sk);
        struct sk_buff *head = tcp_send_head(sk);
        const struct tcp_sock *tp = tcp_sk(sk);

        /* Was it a usable window open? */
        if (!head)
                return;
        if (!after(TCP_SKB_CB(head)->end_seq, tcp_wnd_end(tp))) {
                icsk->icsk_backoff = 0;
                icsk->icsk_probes_tstamp = 0;
                inet_csk_clear_xmit_timer(sk, ICSK_TIME_PROBE0);
                /* Socket must be waked up by subsequent tcp_data_snd_check().
                 * This function is not for random using!
                 */
```

```
        } else {
                unsigned long when = tcp_probe0_when(sk, TCP_RTO_MAX);

                when = tcp_clamp_probe0_to_user_timeout(sk, when);
                tcp_reset_xmit_timer(sk, ICSK_TIME_PROBE0, when, TCP_RTO_MAX);
        }
}

static inline bool tcp_ack_is_dubious(const struct sock *sk, const int flag)
{
        return !(flag & FLAG_NOT_DUP) || (flag & FLAG_CA_ALERT) ||
                inet_csk(sk)->icsk_ca_state != TCP_CA_Open;
}

/* Decide wheather to run the increase function of congestion control. */
static inline bool tcp_may_raise_cwnd(const struct sock *sk, const int flag)
{
        /* If reordering is high then always grow cwnd whenever data is
         * delivered regardless of its ordering. Otherwise stay conservative
         * and only grow cwnd on in-order delivery (RFC5681). A stretched ACK w/
         * new SACK or ECE mark may first advance cwnd here and later reduce
         * cwnd in tcp_fastretrans_alert() based on more states.
         */
        if (tcp_sk(sk)->reordering > sock_net(sk)->ipv4.sysctl_tcp_reordering)
                return flag & FLAG_FORWARD_PROGRESS;

        return flag & FLAG_DATA_ACKED;
}

/* The "ultimate" congestion control function that aims to replace the rigid
 * cwnd increase and decrease control (tcp_cong_avoid,tcp_*cwnd_reduction).
 * It's called toward the end of processing an ACK with precise rate
 * information. All transmission or retransmission are delayed afterwards.
 */
static void tcp_cong_control(struct sock *sk, u32 ack, u32 acked_sacked,
                             int flag, const struct rate_sample *rs)
{
        const struct inet_connection_sock *icsk = inet_csk(sk);

        if (icsk->icsk_ca_ops->cong_control) {
                icsk->icsk_ca_ops->cong_control(sk, rs);
                return;
        }

        if (tcp_in_cwnd_reduction(sk)) {
                /* Reduce cwnd if state mandates */
                tcp_cwnd_reduction(sk, acked_sacked, rs->losses, flag);
        } else if (tcp_may_raise_cwnd(sk, flag)) {
                /* Advance cwnd if state allows */
                tcp_cong_avoid(sk, ack, acked_sacked);
        }
        tcp_update_pacing_rate(sk);
}

/* Check that window update is acceptable.
 * The function assumes that snd_una<=ack<=snd_next.
 */
static inline bool tcp_may_update_window(const struct tcp_sock *tp,
                                         const u32 ack, const u32 ack_seq,
                                         const u32 nwin)
{
        return  after(ack, tp->snd_una) ||
                after(ack_seq, tp->snd_wl1) ||
                (ack_seq == tp->snd_wl1 && nwin > tp->snd_wnd);
}

/* If we update tp->snd_una, also update tp->bytes_acked */
static void tcp_snd_una_update(struct tcp_sock *tp, u32 ack)
{
        u32 delta = ack - tp->snd_una;

        sock_owned_by_me((struct sock *)tp);
        tp->bytes_acked += delta;
```

Event handlers

```
        WRITE_ONCE(tp->rcv_nxt, seq);
}

/* Update our send window.
 *
 * Window update algorithm, described in RFC793/RFC1122 (used in linux-2.2
 * and in FreeBSD. NetBSD's one is even worse.) is wrong.
 */
static int tcp_ack_update_window(struct sock *sk, const struct sk_buff *skb, u32 ack,
                                 u32 ack_seq)
{
        struct tcp_sock *tp = tcp_sk(sk);
        int flag = 0;
        u32 nwin = ntohs(tcp_hdr(skb)->window);

        if (likely(!tcp_hdr(skb)->syn))
                nwin <<= tp->rx_opt.snd_wscale;

        if (tcp_may_update_window(tp, ack, ack_seq, nwin)) {
                flag |= FLAG_WIN_UPDATE;
                tcp_update_wl(tp, ack_seq);

                if (tp->snd_wnd != nwin) {
                        tp->snd_wnd = nwin;

                        /* Note, it is the only place, where
                         * fast path is recovered for sending TCP.
                         */
                        tp->pred_flags = 0;
                        tcp_fast_path_check(sk);

                        if (!tcp_write_queue_empty(sk))
                                tcp_slow_start_after_idle_check(sk);

                        if (nwin > tp->max_window) {
                                tp->max_window = nwin;
                                tcp_sync_mss(sk, inet_csk(sk)->icsk_pmtu_cookie);
                        }
                }
        }

        tcp_snd_una_update(tp, ack);

        return flag;
}

static bool __tcp_oow_rate_limited(struct net *net, int mib_idx,
                                   u32 *last_oow_ack_time)
{
        if (*last_oow_ack_time) {
                s32 elapsed = (s32)(tcp_jiffies32 - *last_oow_ack_time);

                if (0 <= elapsed && elapsed < net->ipv4.sysctl_tcp_invalid_ratelimit) {
                        NET_INC_STATS(net, mib_idx);
                        return true;    /* rate-limited: don't send yet! */
                }
        }

        *last_oow_ack_time = tcp_jiffies32;

        return false;   /* not rate-limited: go ahead, send dupack now! */
}

/* Return true if we're currently rate-limiting out-of-window ACKs and
 * thus shouldn't send a dupack right now. We rate-limit dupacks in
 * response to out-of-window SYNs or ACKs to mitigate ACK loops or DoS
 * attacks that send repeated SYNs or ACKs for the same connection. To
 * do this, we do not send a duplicate SYNACK or ACK if the remote
 * endpoint is sending out-of-window SYNs or pure ACKs at a high rate.
 */
bool tcp_oow_rate_limited(struct net *net, const struct sk_buff *skb,
                          int mib_idx, u32 *last_oow_ack_time)
{
        /* Data packets without SYNs are not likely part of an ACK loop. */
        if ((TCP_SKB_CB(skb)->seq != TCP_SKB_CB(skb)->end_seq) &&
            !tcp_hdr(skb)->syn)
```

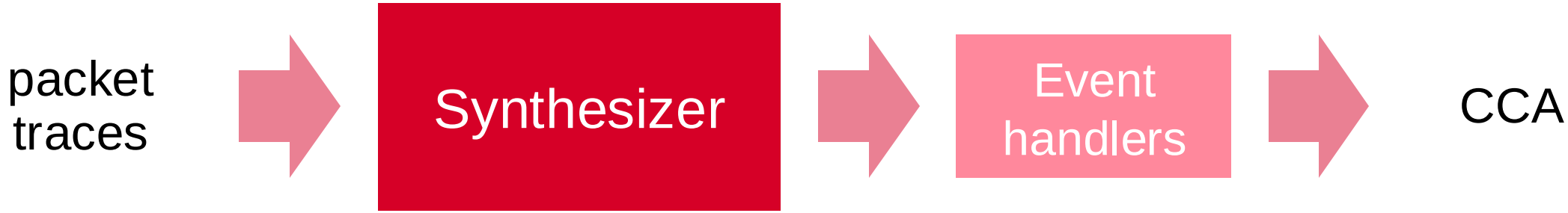
Most congestion control code is boilerplate



Abagnale uses program synthesis to reverse engineer CCAs



Abagnale reverse engineers CCAs by synthesizing event handlers



CCAs are modeled as a set of handler functions

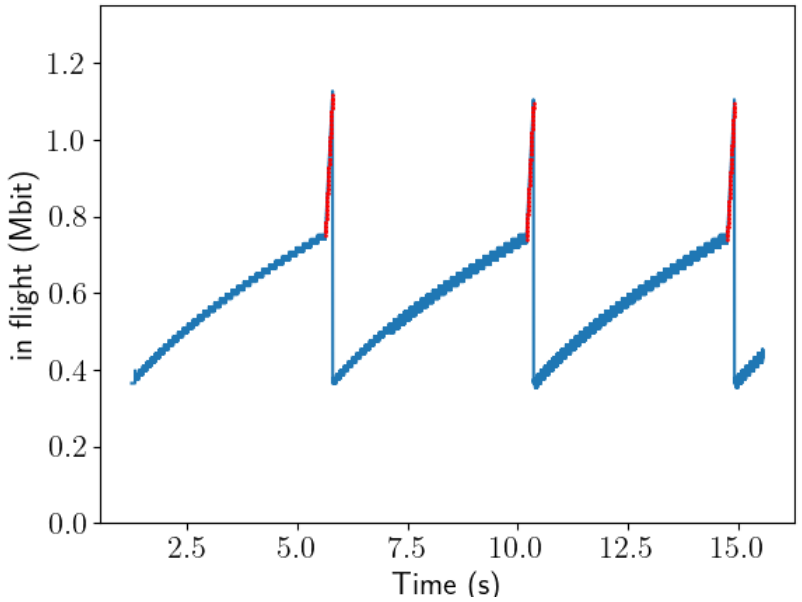
$$h(cwnd_i , signals_i) = cwnd_{i+1}$$

The output of each execution of each handler is used as input to the next execution

$$h(cwnd_0 , signals_0) = cwnd_1$$
$$h(cwnd_1 , signals_1) = cwnd_2$$
$$h(cwnd_2 , signals_2) = cwnd_3$$
$$h(cwnd_3 , signals_3) = cwnd_4$$
$$h(cwnd_4 , signals_4) = cwnd_5$$
$$...$$
$$h(cwnd_{n-1} , signals_{n-1}) = cwnd_n$$

The behavior in the trace is the result of successive execution of the handlers

TCP Reno

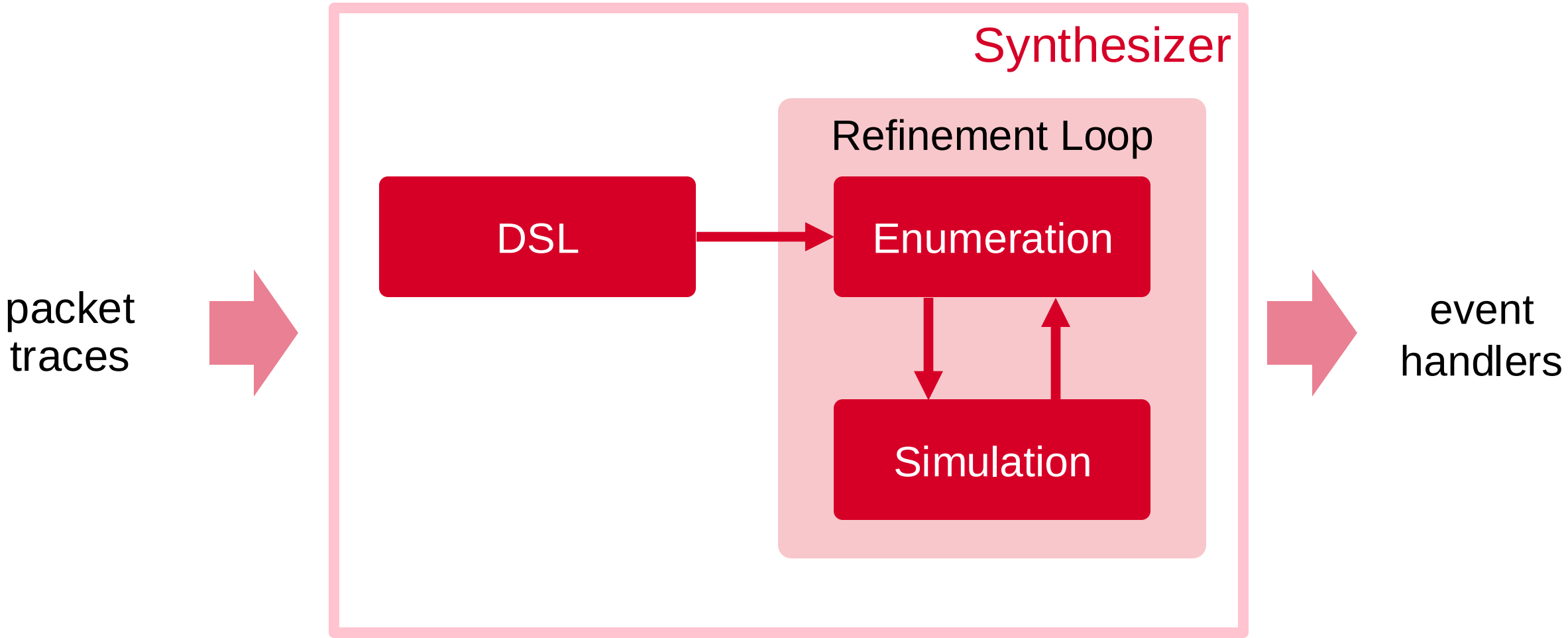


h: cwnd + MSS * acked-bytes / cwnd

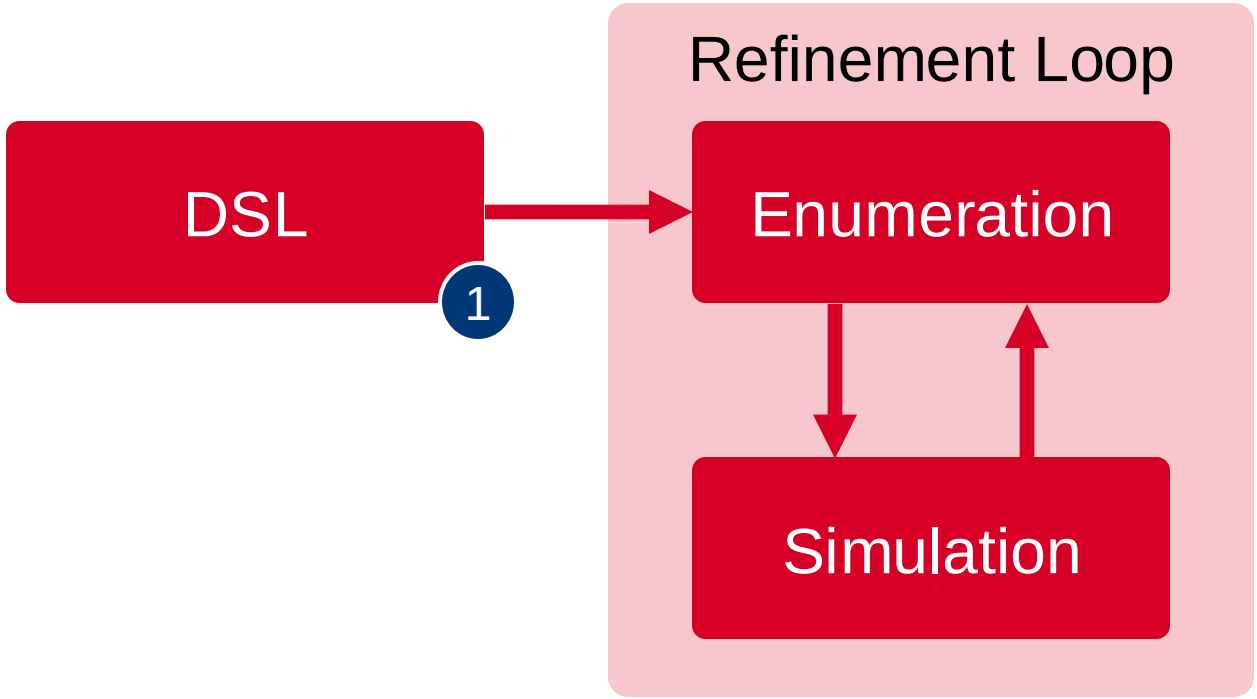
Abagnale



Abagnale's synthesis pipeline



Abagnale's DSL defines the search space



Domain-Specific Language (DSL)

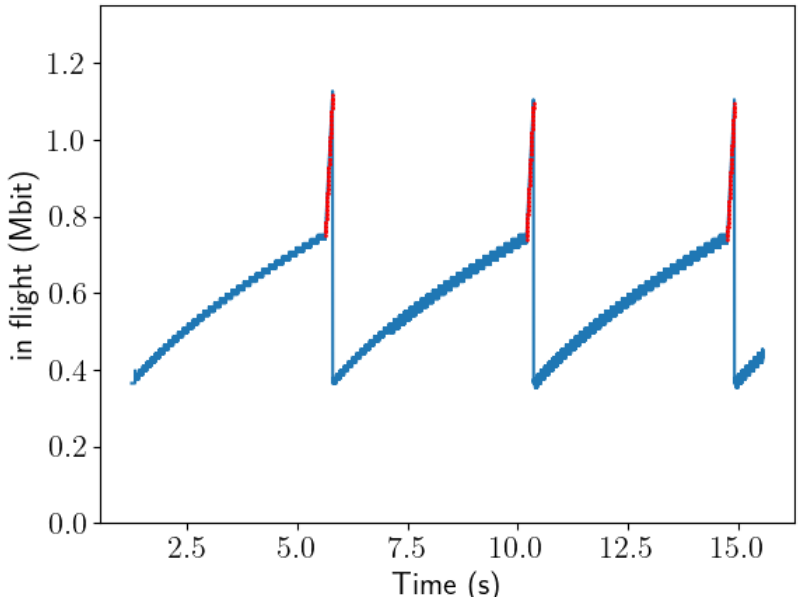
The DSL includes:

- The **congestion signals** that can be used as inputs
  ex: cwnd, MSS, acked-bytes, time-since-loss, RTT, min-RTT, ack-rate, …
- The **operators** that can be used to combine them
  ex: +, –, /, *, if-then-else, <, >, …
- Numerical **constants** c_1, c_2, c_3, …



Handlers are compositions of DSL components

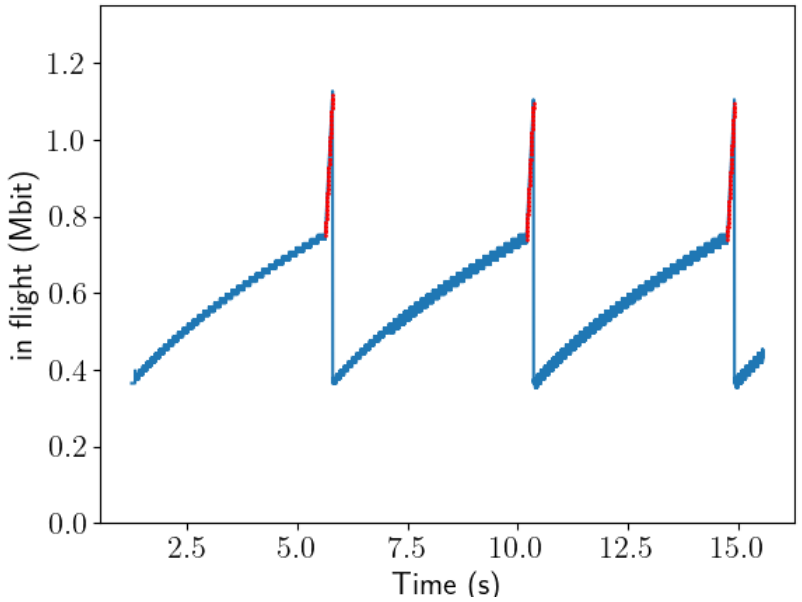
TCP Reno



h: cwnd + MSS * acked-bytes / cwnd

Handlers are compositions of DSL components

TCP Reno

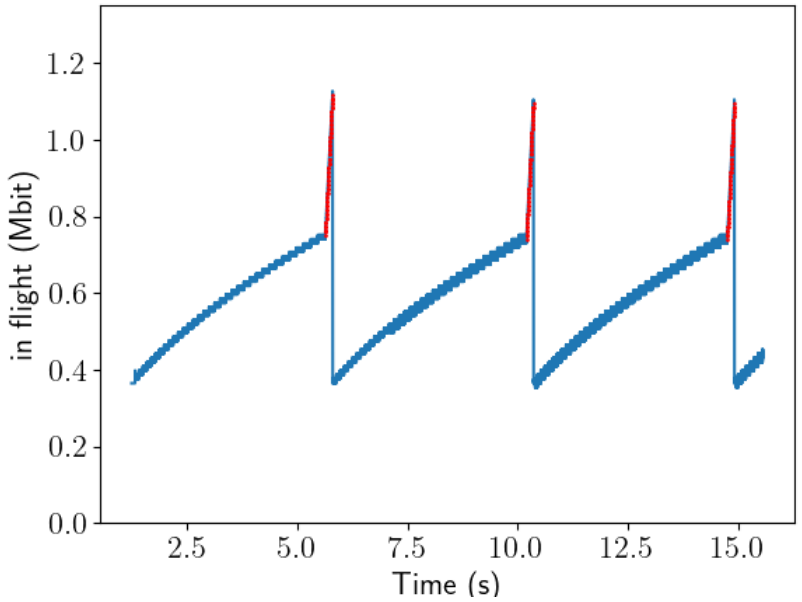


h: cwnd + MSS * acked-bytes / cwnd

3 operators

Handlers are compositions of DSL components

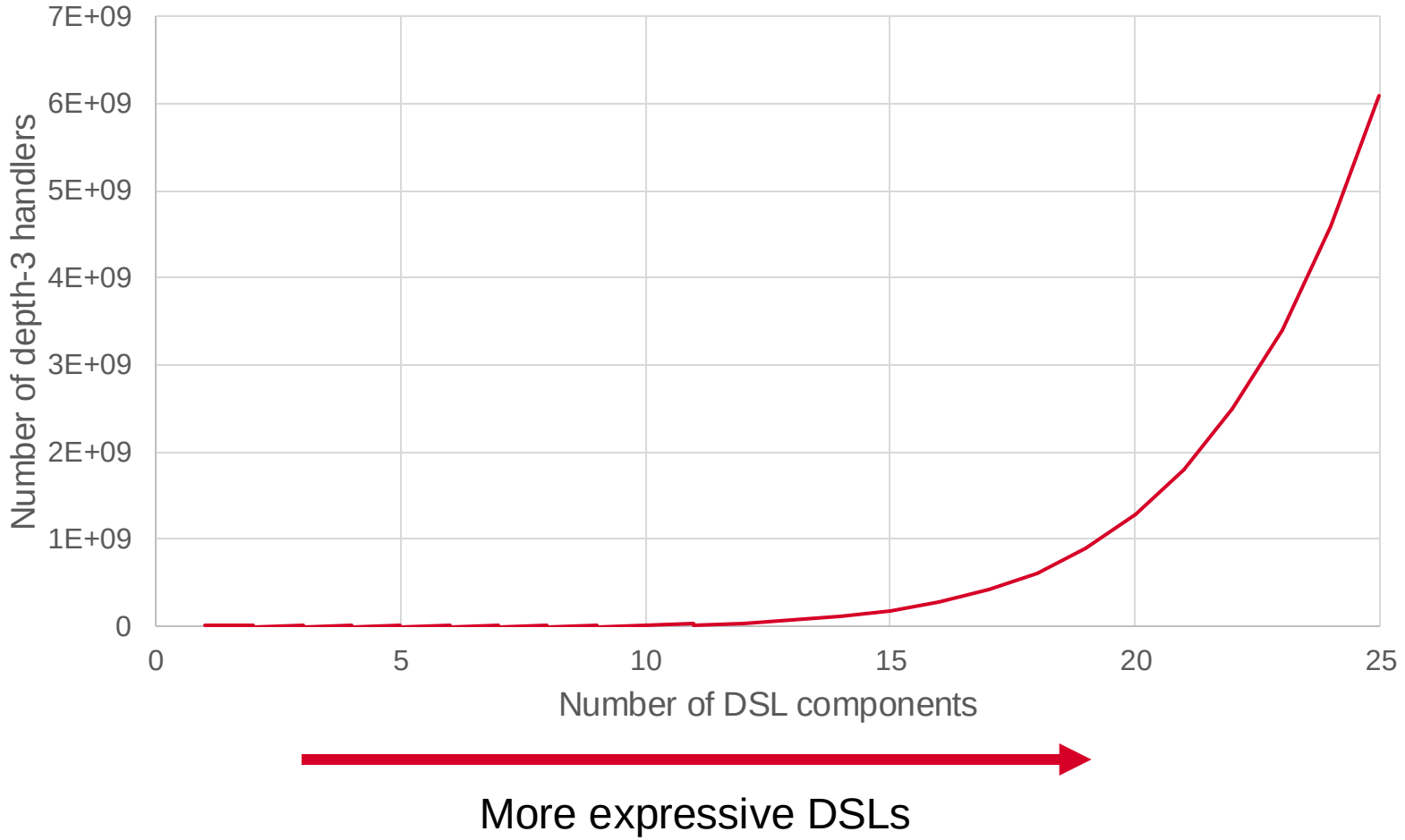
TCP Reno



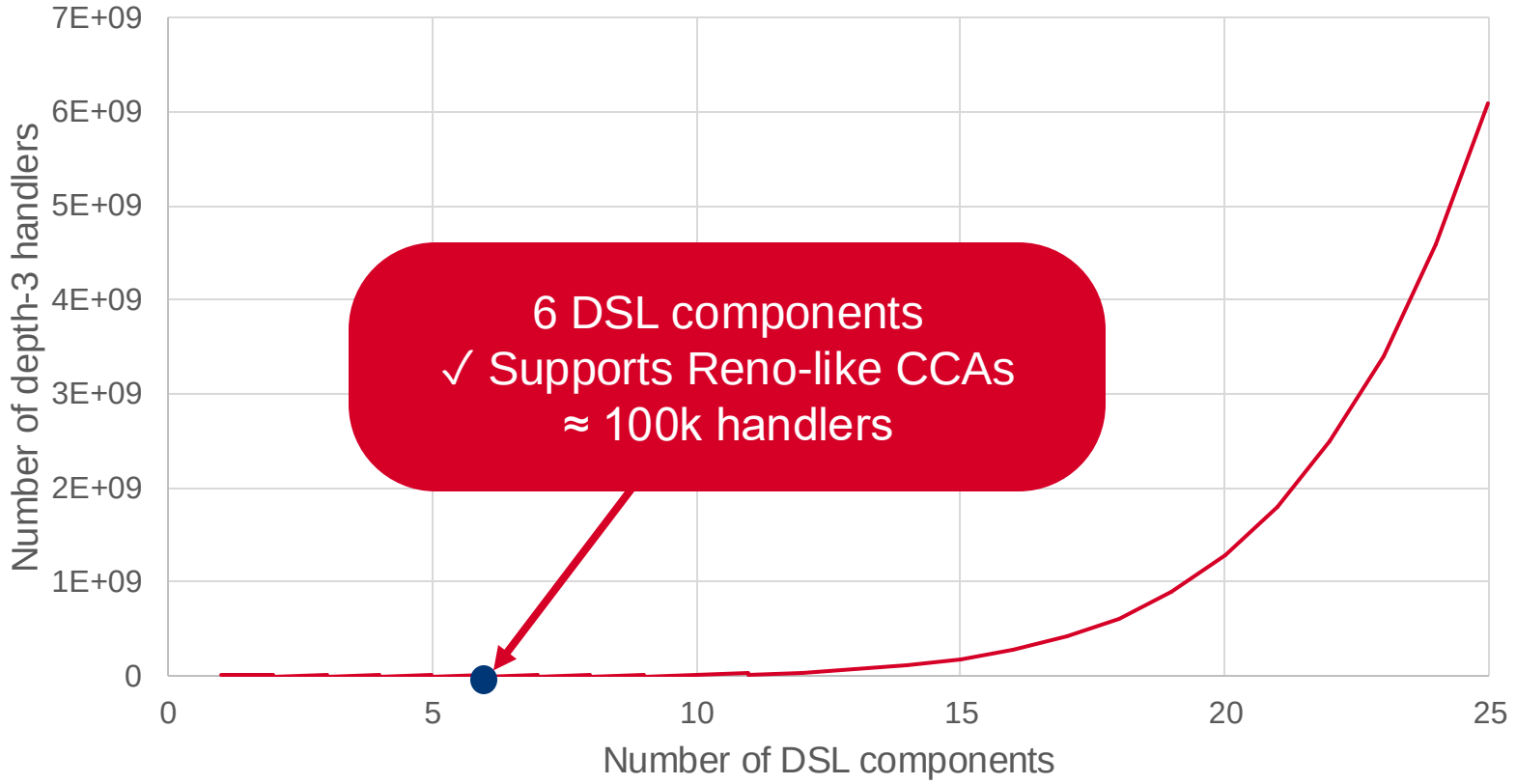
h: cwnd + MSS * acked-bytes / cwnd

3 congestion signals + 3 operators

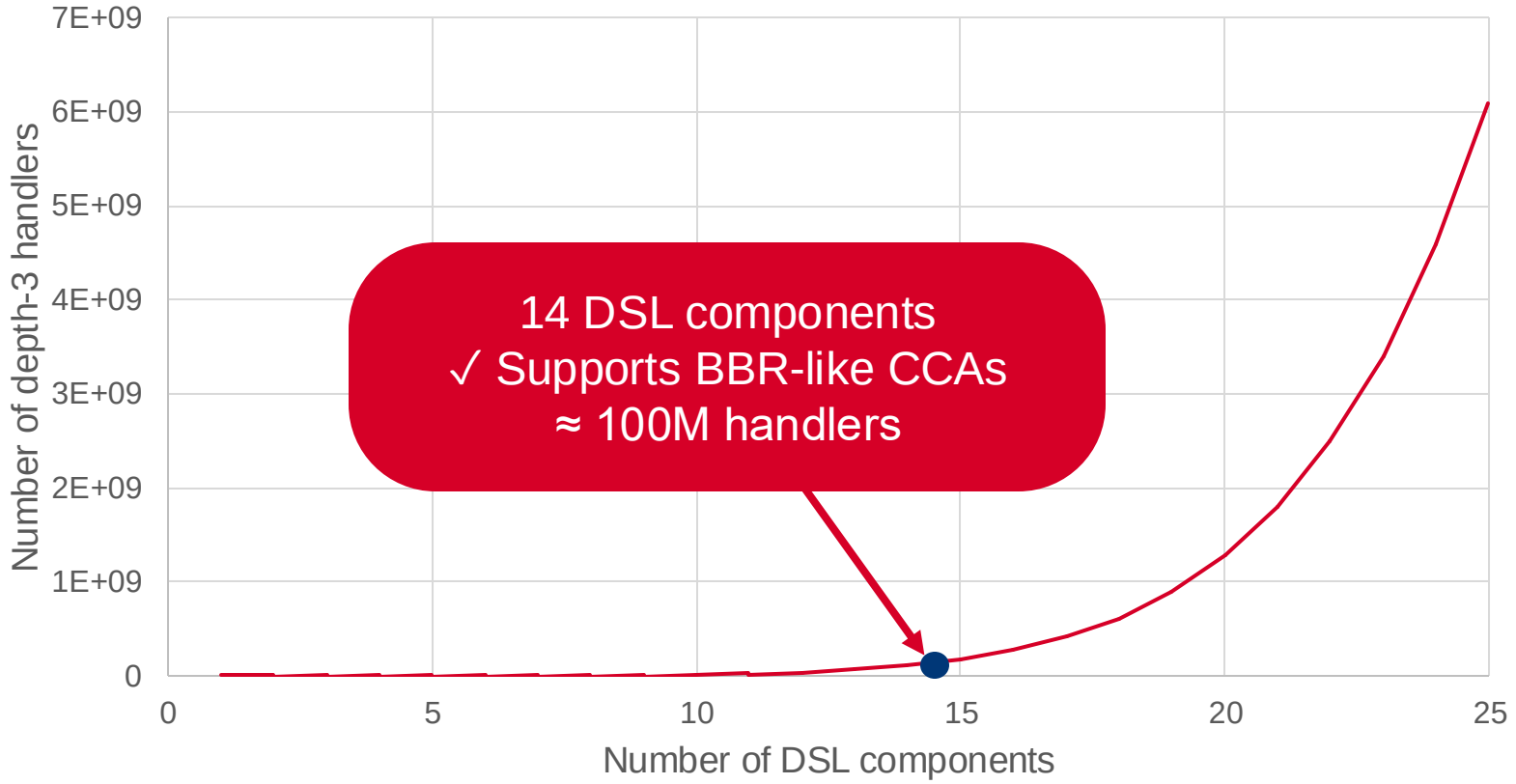
The search space grows exponentially with the number of DSL components



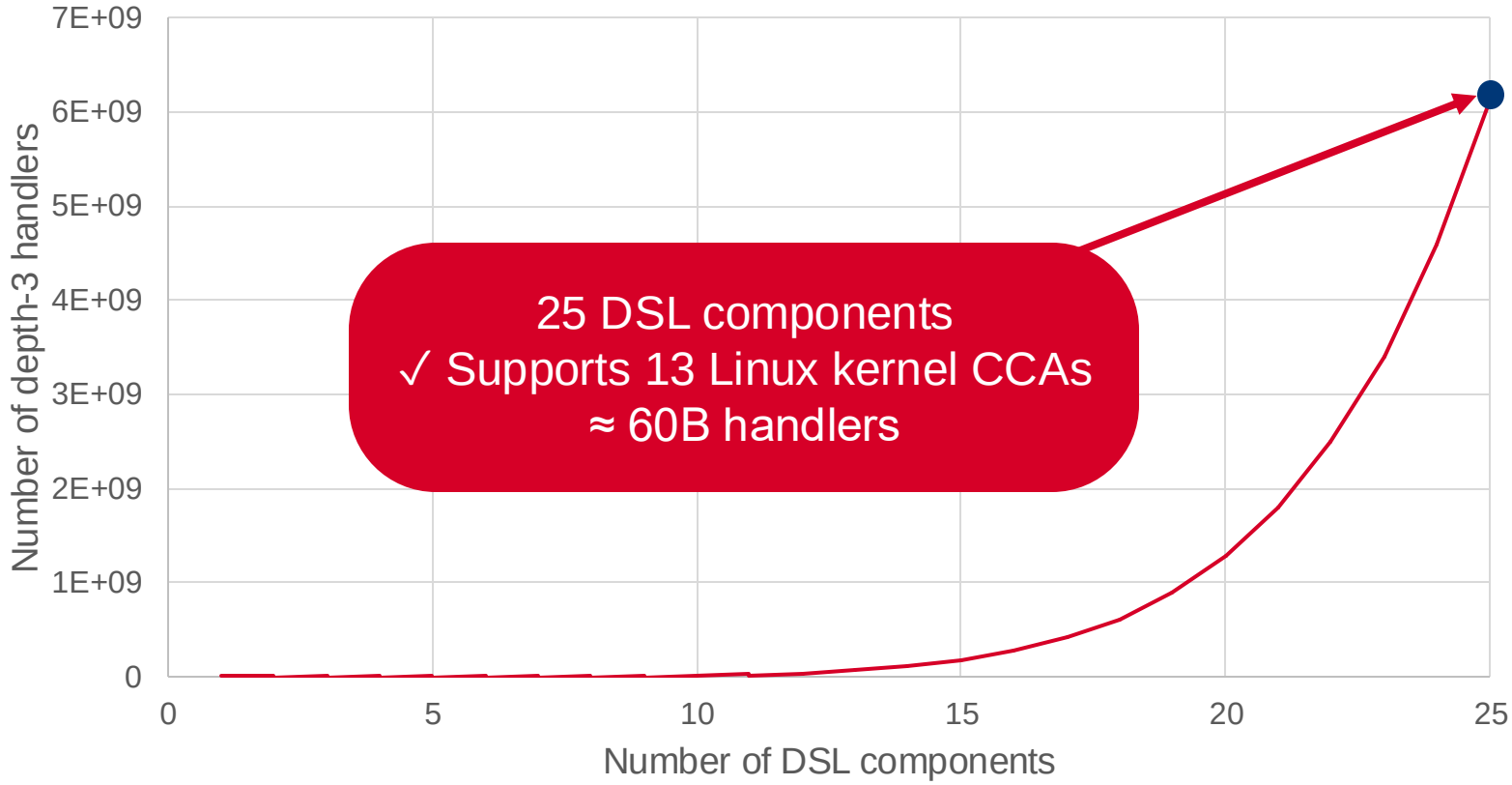
The search space grows exponentially with the number of DSL components



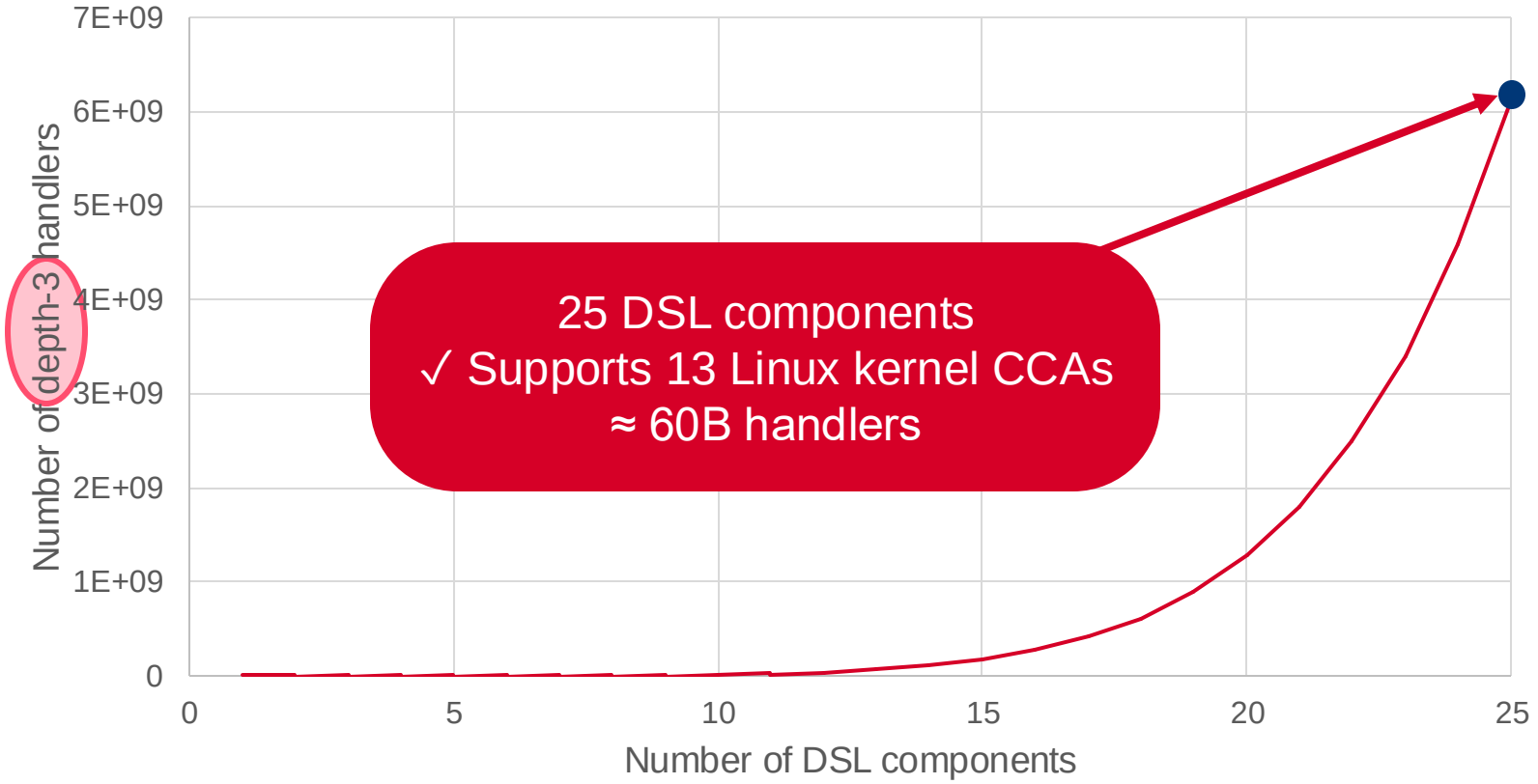
The search space grows exponentially with the number of DSL components



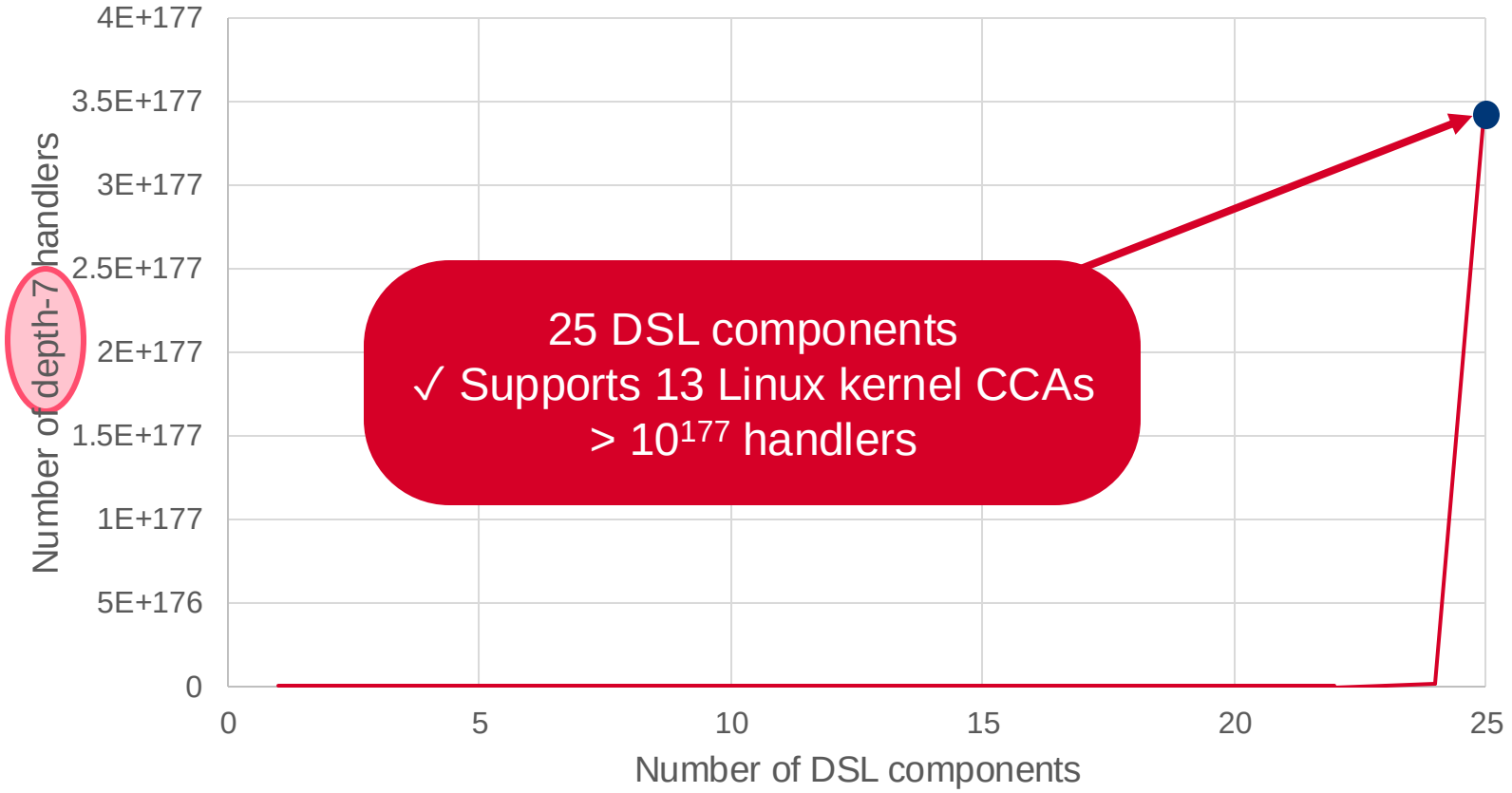
The search space grows exponentially with the number of DSL components



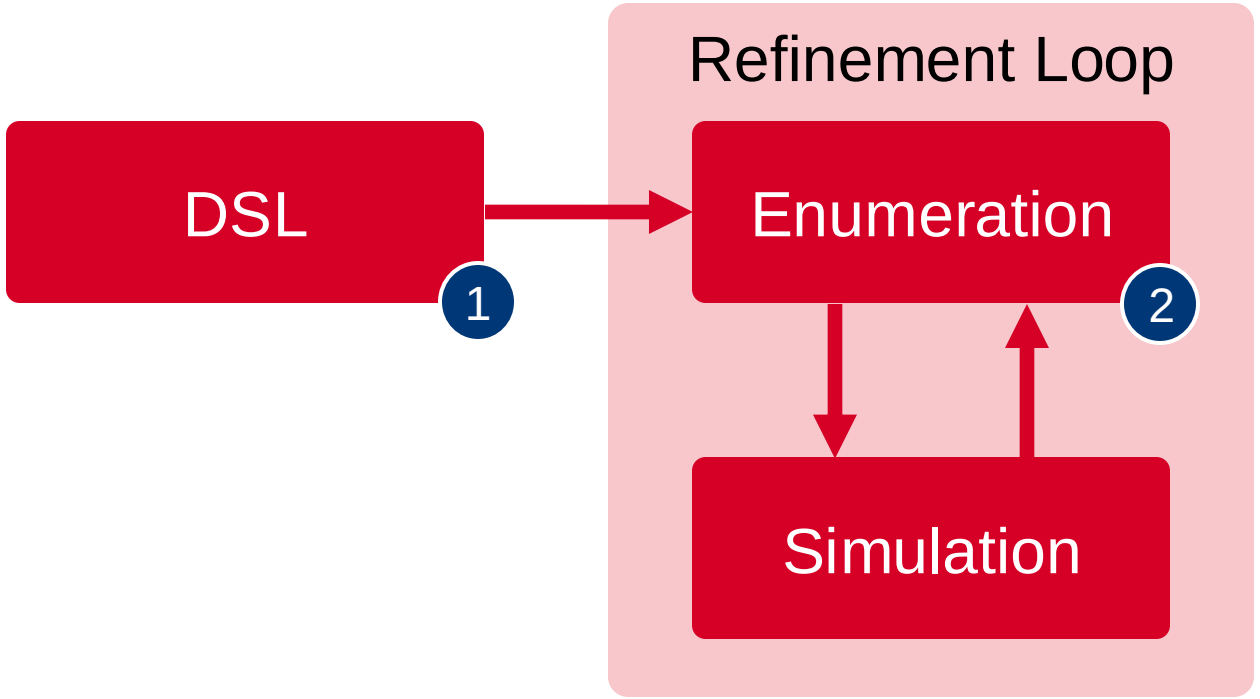
The search space grows exponentially with the number of DSL components



The search space grows exponentially with the number of DSL components



Abagnale's enumeration traverses the search space



Solver-based pruning removes 99.9999% of the search space

Solver-based pruning removes 99.9999% of the search space

Solver-based pruning removes all handlers that

- do not type-check,
- do not unit-check,
- are algebraically equivalent to other handlers
- would never increase or never decrease the signal they are computing
- …

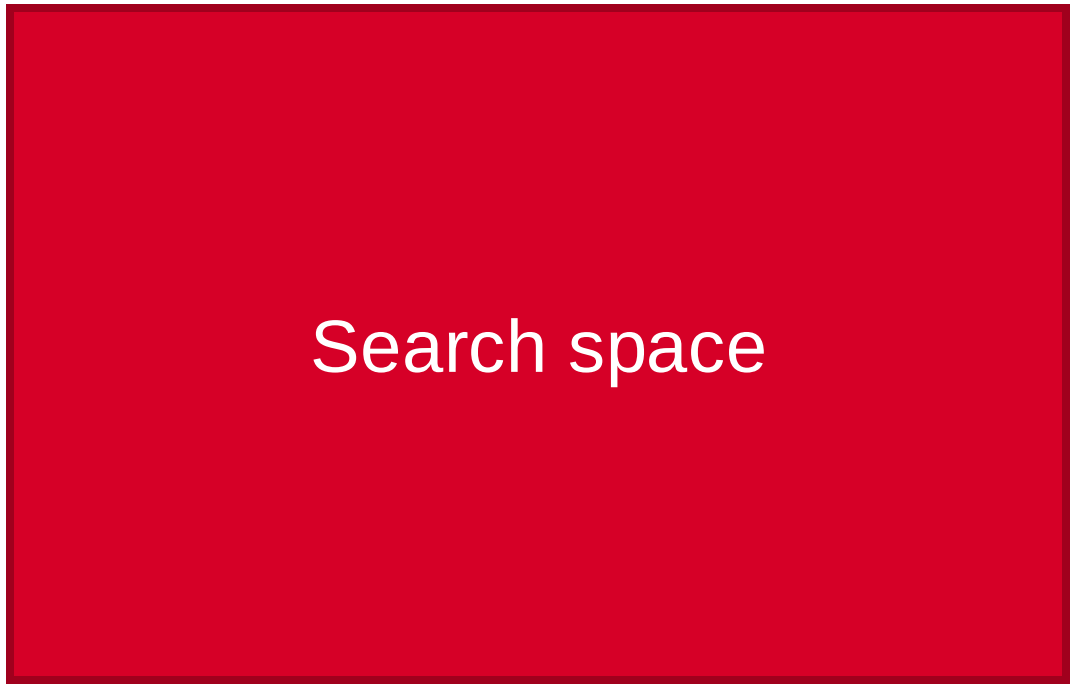
Solver-based pruning removes 99.9999% of the search space

Solver-based pruning removes all handlers that

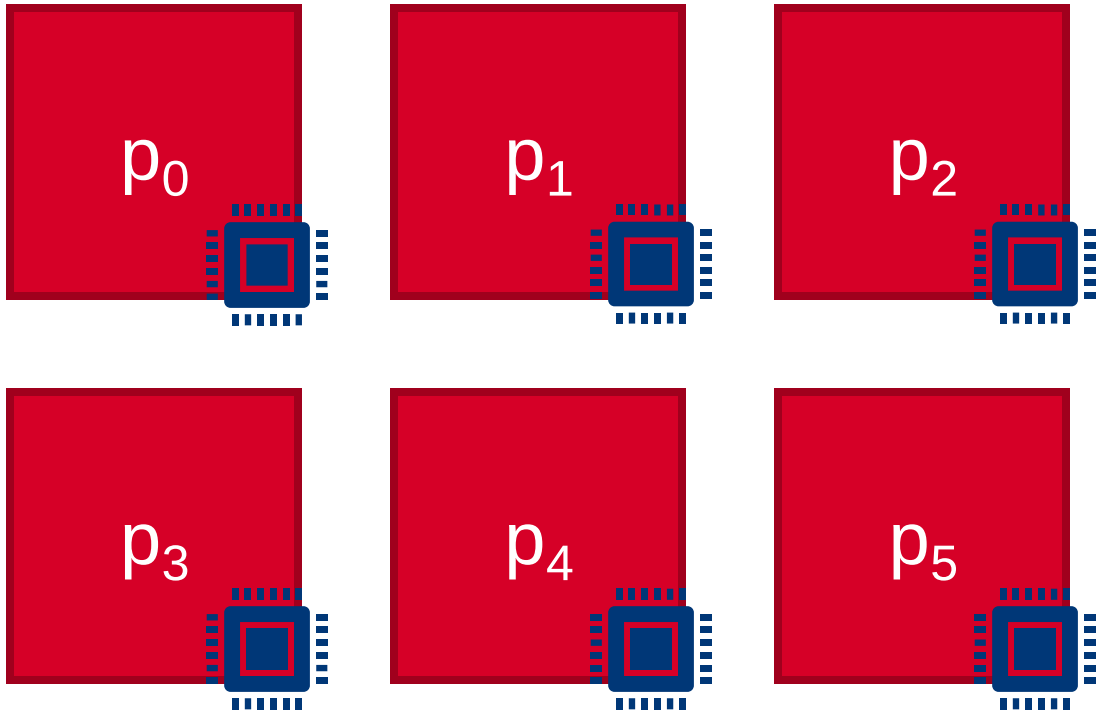
- do not type-check,
- do not unit-check,
- are algebraically equivalent to other handlers
- would never increase or never decrease the signal they are computing
- …

it still leaves >100k handlers in the search space to be explored

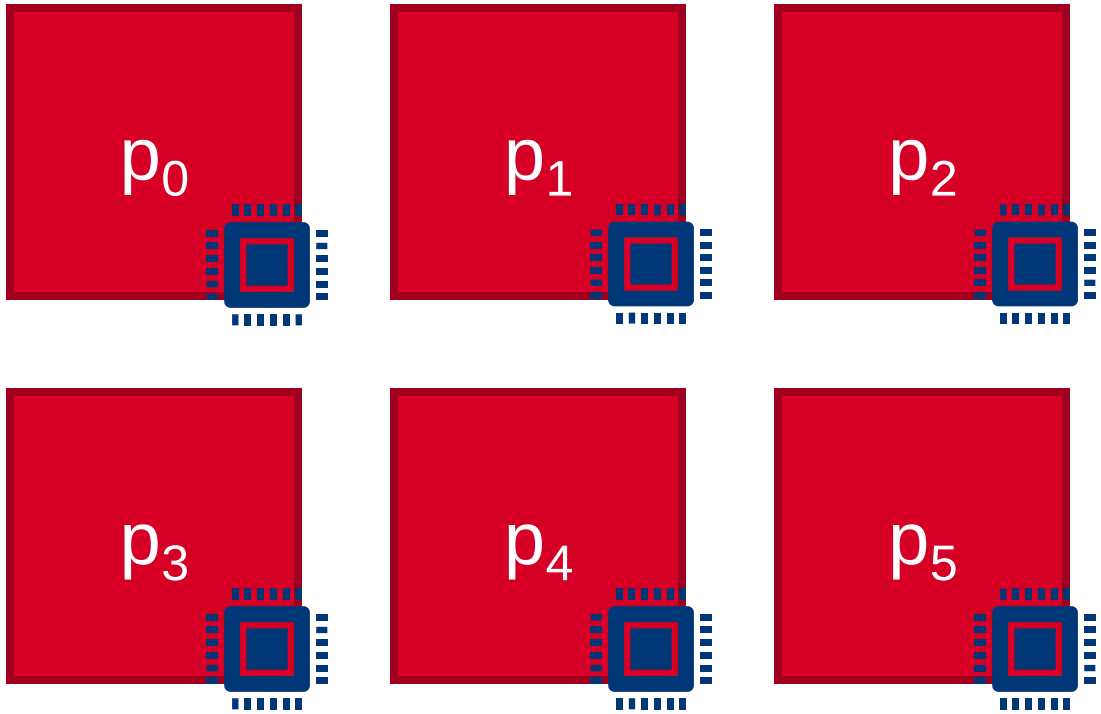
Partition the space to parallelize the search



Partition the space to parallelize the search



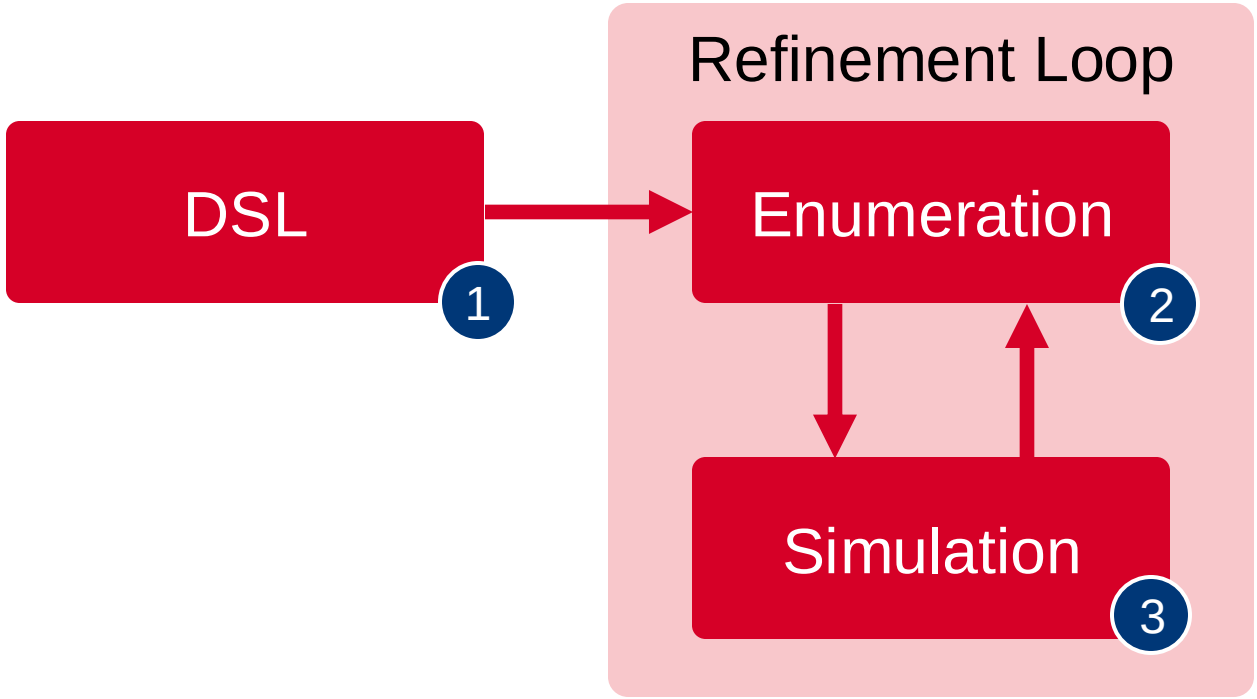
Partition the space to parallelize the search



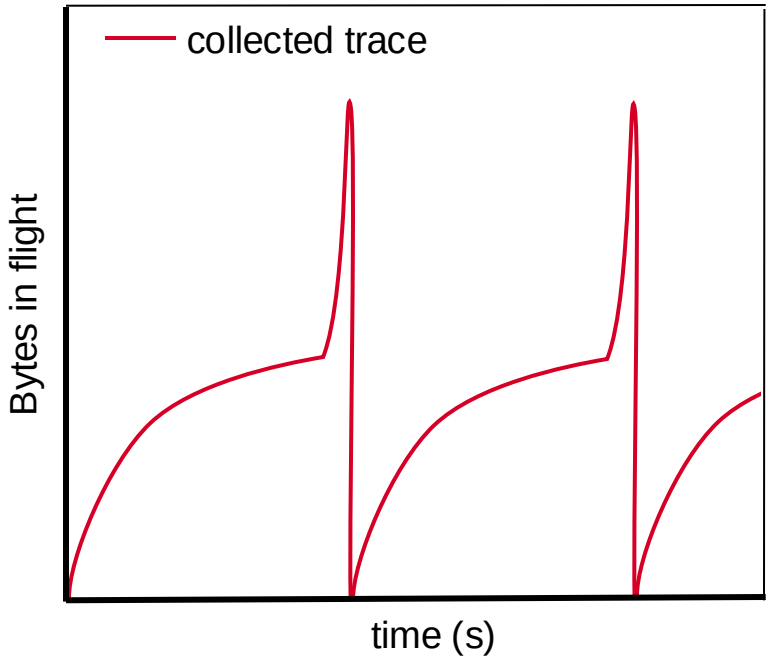
We partition the search space such that parts:

- are disjoint
- can be encoded in the enumerator

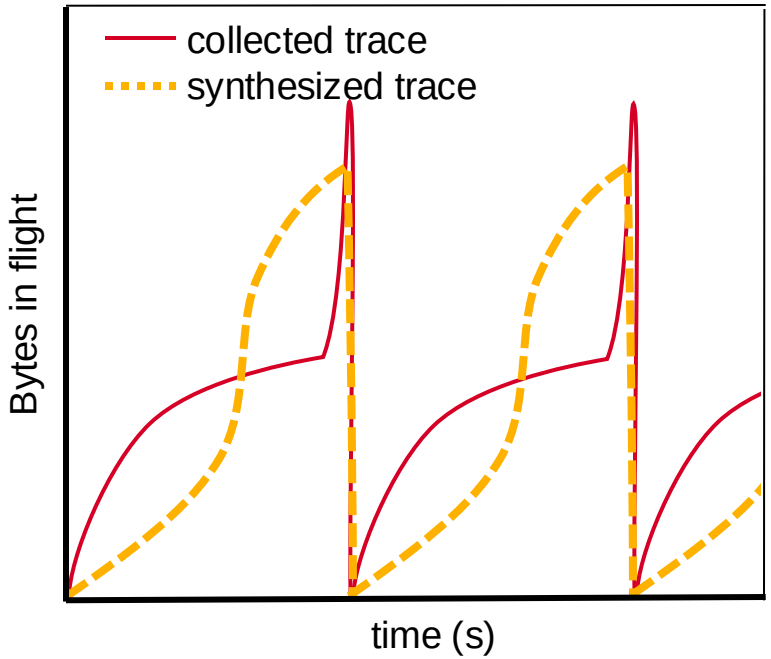
Abagnale



We simulate each candidate CCA in the same conditions that we collected the trace

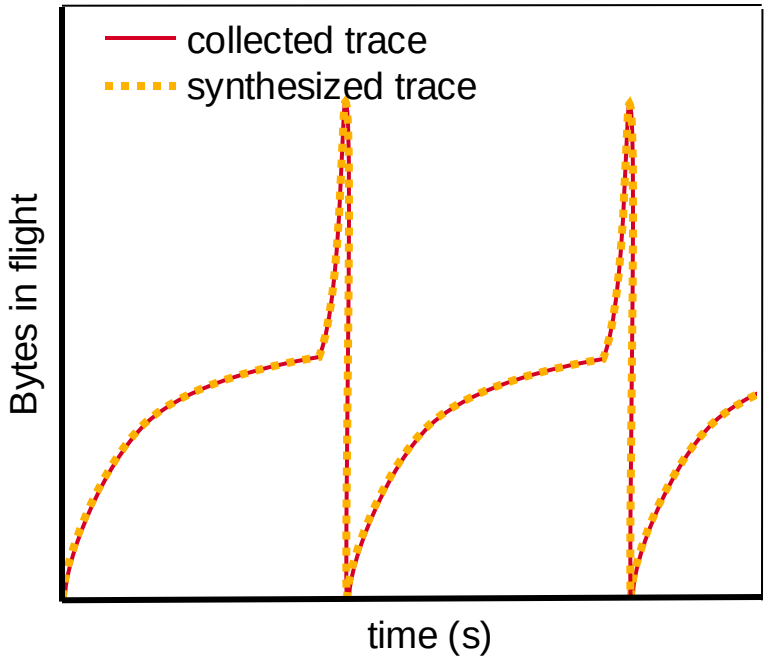


We simulate each candidate CCA in the same conditions that we collected the trace



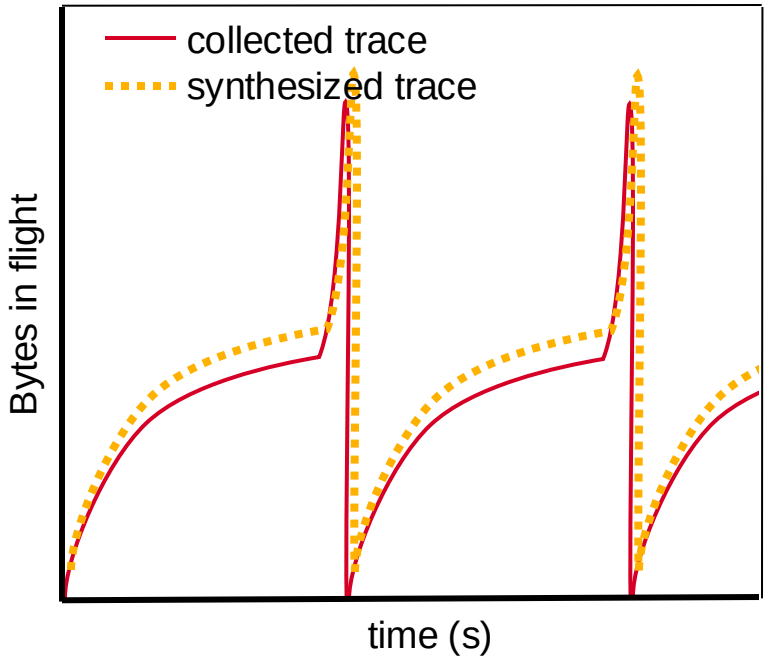
We get a second trace, the synthesized trace, and we can compare them.

We will never find a CCA that exactly matches a noisy trace



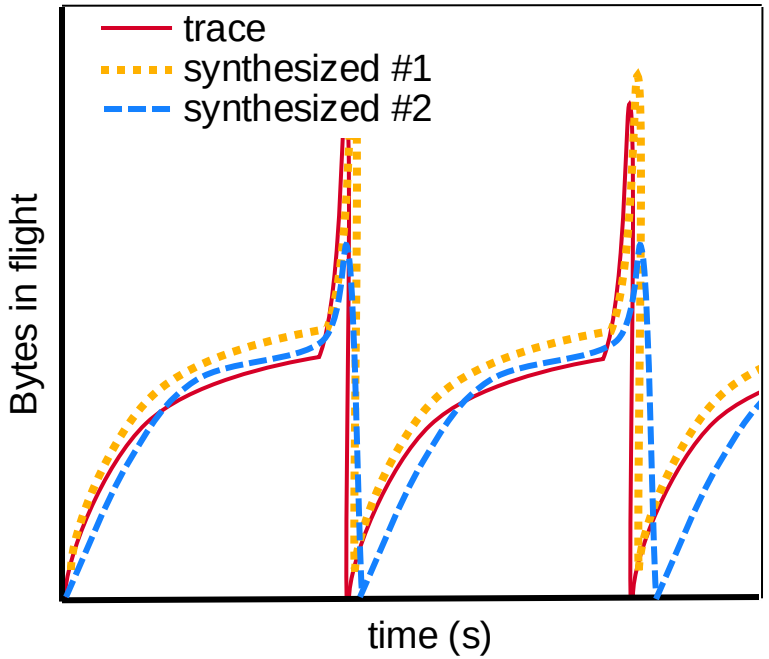
Unrealistic!

We will never find an exact match, so we look for an *approximate* match



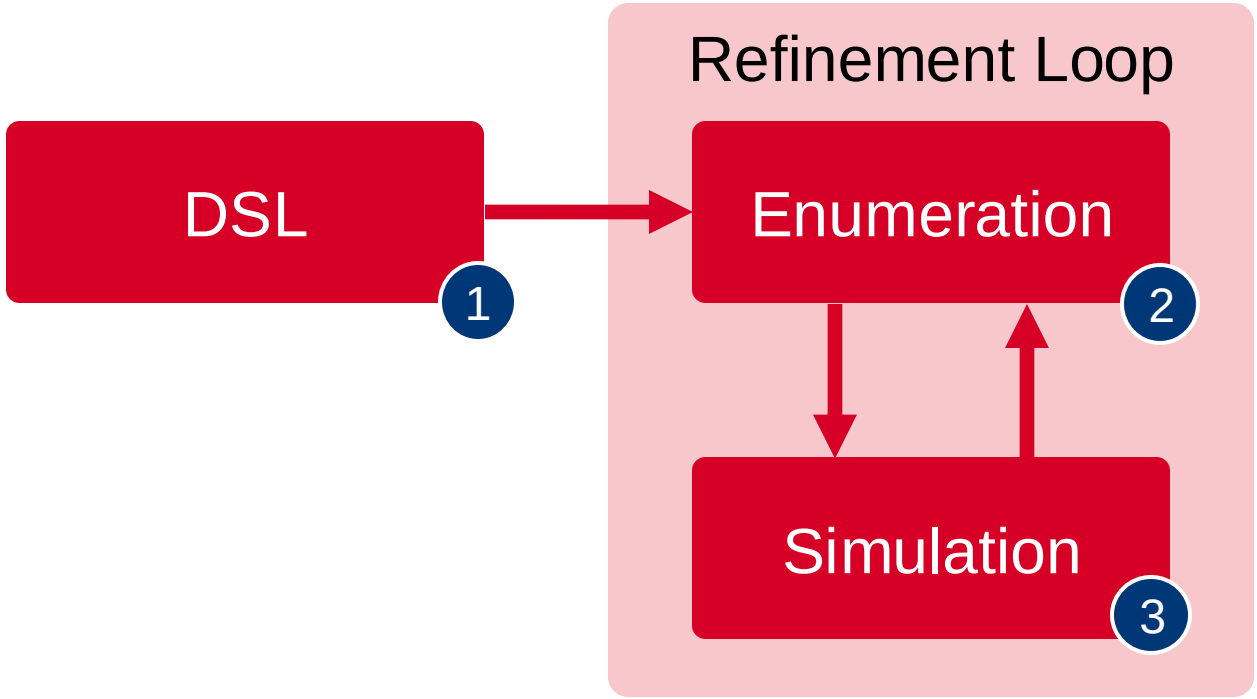
We look at the *distance* between the synthesized and the collected traces

We will never find an exact match, so we look for an *approximate* match



We look at the *distance* between the synthesized and the collected traces, and select the CCA handler with the minimum distance

Abagnale



Evaluation

Carnegie Mellon University

Evaluation overview

We compare the semantics of Abagnale's synthesized handler with a handwritten version of the handler finetuned by a domain expert

Evaluation overview

We compare the semantics of Abagnale's synthesized handler with a handwritten version of the handler finetuned by a domain expert

We evaluate Abagnale in

- 13 Linux kernel CCAs

Evaluation overview

We compare the semantics of Abagnale's synthesized handler with a handwritten version of the handler finetuned by a domain expert

We evaluate Abagnale in

- 13 Linux kernel CCAs
- 7 unknown CCAs implemented by students as part of a class

Evaluation overview

We compare the semantics of Abagnale's synthesized handler with a handwritten version of the handler finetuned by a domain expert

We evaluate Abagnale in

- 13 Linux kernel CCAs
- 7 unknown CCAs implemented by students as part of a class

Abagnale finds semantically correct handlers for "Reno-like" CCAs, and semantically proximate handlers for "Vegas-like" and BBR.

Evaluation overview

We compare the semantics of Abagnale's synthesized handler with a handwritten version of the handler finetuned by a domain expert

We evaluate Abagnale in

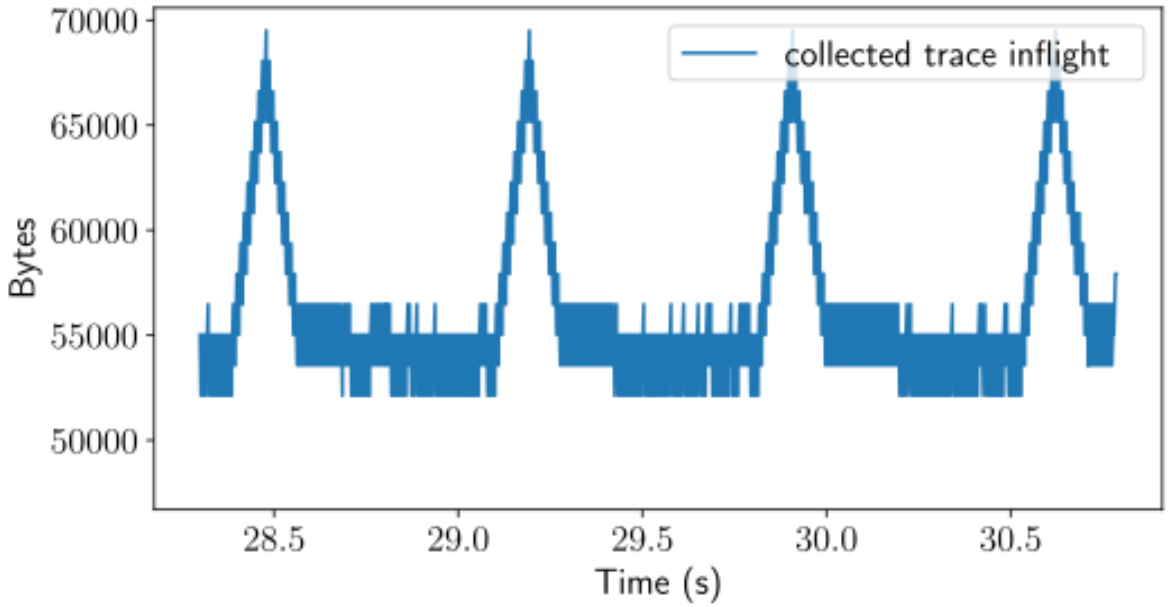
- 13 Linux kernel CCAs
- 7 unknown CCAs implemented by students as part of a class

Abagnale finds semantically correct handlers for "Reno-like" CCAs, and semantically proximate handlers for "Vegas-like" and BBR.

See complete evaluation in the paper!



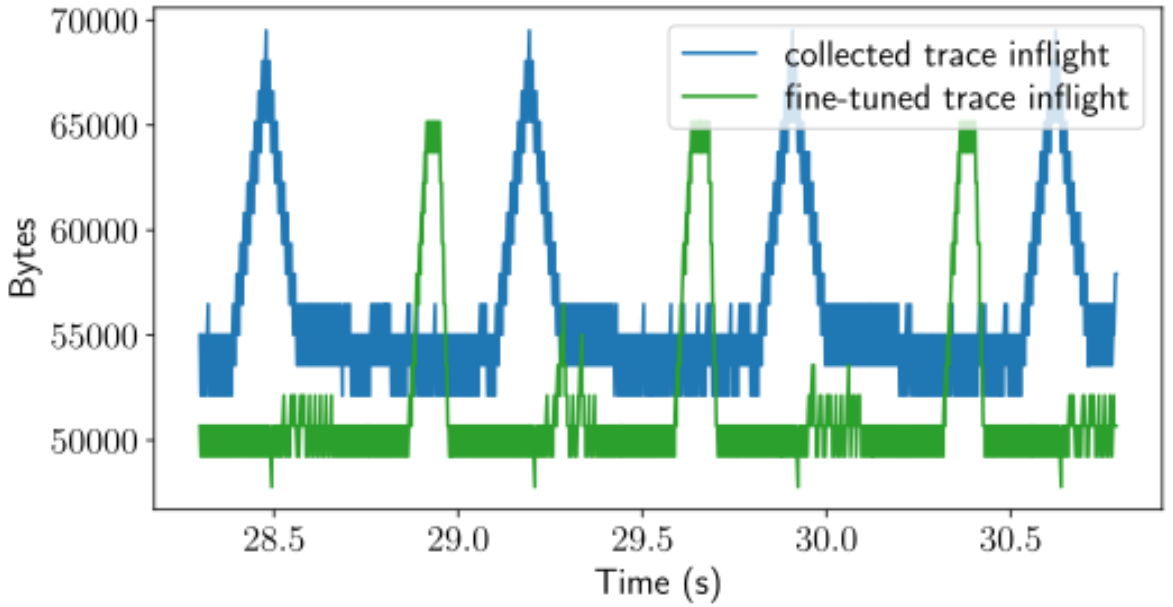
BBR: Abagnale's synthesized handler with BBR traces mimics PROBE_BW pulses, but with a different trigger



BBR: Abagnale's synthesized handler with BBR traces mimics PROBE_BW pulses, but with a different trigger

Fine-tuned win-ack handler for BBR

$$\textit{win-ack} = ((\text{rtts_since_loss}\%8.0 = 0)\ ?\ 2.6 : 2.05) \cdot (\text{min_rtt} \cdot \text{ack_rate})$$



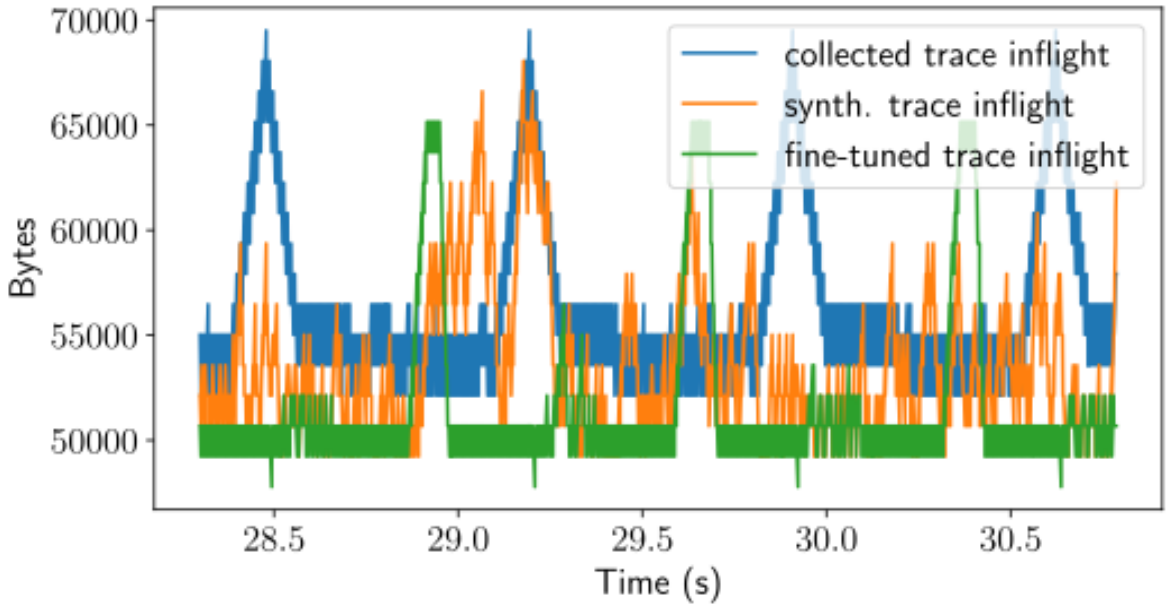
BBR: Abagnale's synthesized handler with BBR traces mimics PROBE_BW pulses, but with a different trigger

Synthesized win-ack handler for BBR

$$\textit{win-ack} = (2) \cdot \text{ack_rate} \cdot \text{min_rtt} + ((\text{cwnd}\%2.7 = 0)\ ?\ 2.05 \cdot \text{cwnd} : \text{mss})$$

Fine-tuned win-ack handler for BBR

$$\textit{win-ack} = ((\text{rtts_since_loss}\%8.0 = 0)\ ?\ 2.6 : 2.05) \cdot (\text{min_rtt} \cdot \text{ack_rate})$$



Reverse-Engineering Congestion Control Algorithm Behavior

Abagnale outputs *simple* implementations of Congestion Control Algorithms from packet traces showing their behavior

- domain-specific strategies allow us to narrow the search space
- we capture the behavior of 13 CCAs from the Linux kernel without any prior knowledge

Margarida Ferreira

margarida@cmu.edu

marghrid.github.io



Carnegie Mellon University