Reverse-Engineering Congestion Control Algorithm Behavior

Margarida Ferreira[1,2], Ranysha Ware[1], Yash Kothari[1], Inês Lynce[2], Ruben Martins[1], Akshay Narayan[3], Justine Sherry[1]
[1] CMU, [2] INESC-ID/IST U. Lisboa, [3] Brown University

Abstract

The rise of proprietary and novel congestion control algorithms (CCAs) opens questions about the future of Internet utilization, latency, and fairness. However, fully analyzing how novel CCAs impact these properties requires understanding the inner workings of these algorithms. We thus aim to reverse-engineer deployed CCAs' behavior from collected packet traces to facilitate analyzing them. We present Abagnale, a program synthesis pipeline that helps users automate the reverse-engineering task. Using Abagnale, we discover simple expressions capturing the behavior of 9 of the 16 CCAs distributed with the Linux kernel and analyze 7 CCAs from a graduate networking course.

CCS Concepts

• **Networks** → **Transport protocols**;

Keywords

Congestion Control

ACM Reference Format:
Margarida Ferreira, Ranysha Ware, Yash Kothari, Inês Lynce, Ruben Martins, Akshay Narayan, Justine Sherry. 2024. Reverse-Engineering Congestion Control Algorithm Behavior . In *Proceedings of the 2024 ACM Internet Measurement Conference (IMC '24), November 4–6, 2024, Madrid, Spain.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3646547.3688443

1 Introduction

Analyzing congestion control algorithms (CCAs) is vital to our understanding of Internet traffic stability, fairness, and performance. For example, past analyses have shown that AIMD approaches such as Reno will converge to fair bandwidth shares [17] and that Google's BBRv1 will converge to unfair bandwidth shares in many scenarios [62]. Recent work has further introduced model checking to CCAs [2], providing the ability to prove performance guarantees for input CCAs given complex edge case scenarios. Of course, these analysis techniques rely on access to a CCA's implementation.

Unfortunately, implementations of CCAs are not universally available. Especially given the rise of user-space CCA implementations [42, 52], it is easier than ever to develop and deploy a proprietary CCA without revealing its implementation details. Indeed, researchers have both observed [10, 50] and publicly claimed [64] proprietary CCA implementations. Further, even with access to a CCA's implementation, many such implementations contain datapath-specific implementation details that obscure their behavior [52], making it difficult for analysts to determine their nature.

Thus, we seek to facilitate CCA analyses by *synthesizing* simple implementations directly from packet traces. These simple implementations make the analysis of CCAs with known (but complex) implementations *easier*, while making analysis of unknown CCAs *possible*.

Generating a program based on a series of observed, example outputs is a form of *program synthesis* [30, 31] (specifically, a class of synthesis called programming-by-example, or 'PBE'). General-purpose PBE remains out of scope for the state of the art: today, the most effective PBE tools are specialized to a particular domain or language framework. We scope our design to synthesize *classically-designed congestion control algorithms*. We formalize what it means for a CCA to be 'classically designed' in §3.3, but at a high level, this means that it can be constructed using a domain-specific language derived from the set of existing Linux Kernel supported CCAs. For example, we aim to design a tool that could reverse engineer TCP Westwood if it were brand new to the CCA landscape, since Westwood can be constructed using the same language as TCP Reno. Since most novel CCAs on the Internet today are variants or extensions of classical algorithms (§5), 'classically designed' algorithms are a useful category of algorithms to specialize in. Importantly, however, this design excludes from our scope CCAs with non-deterministic behavior (including those using machine learning techniques).

The reason that program synthesizers target domain-specific regimes is tractability. At their core, all synthesizers frame a search space using a domain-specific language (DSL) defining the 'set of all possible programs' and aim to find a needle in this haystack: a program that, given the pre-specified inputs, produces the pre-specified outputs. Constraining the size of the DSL makes the haystack smaller. Nonetheless, even a constrained haystack is still large: even when constrained to CCAs following a 'classical' DSL, there are 10^{150} possible programs to explore.

To meet this challenge, we design Abagnale, a program synthesis pipeline that utilizes domain knowledge to produce approximate expressions representing CCAs in our scope. Abagnale cuts down on the intractability of synthesizing CCAs by taking an uncommon approach to program synthesis: formulating the problem as an *optimization* problem (in which a numerical objective is maximized or minimized) rather than a *decision* problem (in which a logical formula is 'satisfied' or 'unsatisfied'). *Our insight is that we can evaluate candidate programs in the search space based on a measurable distance between the visible CWND of the candidate CCA in simulation and the observed CWND of the true CCA in the wild. We then formulate our procedure to select a program that minimizes this distance.*

IMC '24, November 4–6, 2024, Madrid, Spain

ACM ISBN 979-8-4007-0592-2/24/11.
https://doi.org/10.1145/3646547.3688443

Our observation that CCAs *are* amenable to an optimization formulation provides cross-cutting gains across four dimensions of the synthesis process:

Evaluation: When we have a candidate CCA proposed by the synthesizer, how can we evaluate whether it matches the expected behavior of the ground truth CCA? A key challenge with measurements of ground truth CCAs 'in the wild' is the presence of noise in our measurements of the true CWND: our observation of the CWND may be incomplete due to our measurement vantage point, packets may be dropped or delayed, *etc.* An optimization formulation allows us to accept candidate CCAs that are close, but imperfect matches to what is observed, compensating for this noise.

DSL Formulation: We aim to identify any CCA that fits the DSL defined by 'classical' CCAs. Nonetheless, this search space still remains intractably large. Here, we leverage the fact that CCAs typically fall into 'families' (*e.g.*, Westwood is in the Reno family; Veno is in the Vegas family.) Hence, we break the DSL into sub-DSLs per CCA family. When given a new ground-truth CCA trace, we use existing classification techniques to constrain the search to a sub-DSL containing only operators and values for that CCA family.

Program Search: Finally, the process of searching for candidate programs is greatly aided by the presence of a measurable function to declare which programs are 'closer' to the correct solution and which are 'further'. We perform several key optimizations here, including breaking down the search further across sub-DSL 'buckets' and parallelizing search across these buckets [9]. To prioritize which buckets to search first, we sample a few candidate CCAs from the bucket and evaluate their distance to the ground-truth measurements; we then prioritize search in the buckets with closer measurements over CCAs with further measurements.

Goal: We emphasize that, with Abagnale, we do not seek to produce the *precise* CCA implementation that produced the behavior observed in a given set of packet traces. Rather, our goal is to produce a succinct representative expression that captures a CCA's core behavior. Even this limited goal is an ambitious step relative to both modern CCAs as well as state-of-the-art program synthesis techniques. Thus, Abagnale is an initial step towards understanding unknown CCAs rather than the final word. For example, many of the most advanced congestion controllers today incorporate machine learning or statistical techniques, and reverse engineering these remains out of reach for Abagnale or any other existing technique. Similarly, as we discuss in §3, Abagnale cannot discover hidden state variables in CCAs that affect their externally visible trace behavior.

Key Results: Despite these challenges, as we show in §5, Abagnale produces a closed-form expression that approximates BBR without using state variables to maintain the pulse state, as most existing implementations do. Further, Abagnale produces expressions matching those fine-tuned by a domain expert with knowledge of the CCA in question for 9 out of 16 CCAs distributed with the Linux kernel. Of the remaining 7 CCAs, 2 (LP and HTCP) miss conditional modes of operation, 2 (HighSpeed and CDG) are out of our scope due to their use of non-determinism and out-of-DSL operators, 1 (Cubic) exposes a limitation in our SMT-encoded search space constraints, and the last (BIC) has an expression depth too deep to find within Abagnale's time-bound. Additionally, we find (§6) that Abagnale is able to efficiently discard large and irrelevant portions of the search space from contention in all evaluated cases. In contrast, prior work on Mister880 [23] cannot synthesize any algorithm other than NewReno (measured without noise) and cannot handle noisy traces at all.

2 Motivation and Background

In this section, we first discuss why reverse engineering congestion control algorithms (CCAs) from packet traces is useful (§2.1) and then discuss why existing state-of-the-art approaches to synthesizing programs from examples are insufficient to synthesize CCAs (§2.2).

2.1 Why reverse-engineer CCAs?

Today's Internet presents an unprecedented explosion in congestion control diversity. Although NewReno and Cubic were often assumed to be the only players in the past, recent studies show tremendous diversity in CCA deployments, with one 2019 study reporting 6 algorithms with deployments across 2% or more of servers [50]. There is also an incredible amount of experimentation: in 2017, Google silently rolled out BBR'1.1' without fanfare; in 2019, Netflix deployed a custom variant of NewReno using its RACK stack in FreeBSD; cloud gaming providers today continue to develop bespoke proprietary CCAs [46].

Many researchers predict CCA diversity will increase in future years. First, with the rise of user-space networking stacks such as HTTP/3 (QUIC), modifying CCA code will become easier for developers who no longer have to delve into kernel space to make modifications. Second, application developers are beginning to see gains from "bespoke" CCAs designed in a way that is specifically tailored to their application. Hence, we have seen proposals for CCAs tailored to video streaming [26, 53] or cloud gaming [56]. Novel algorithms may furthermore be proprietary, with companies unlikely to share the 'secret' behind their applications' competitive network performance.

The explosion in CCA diversity has important implications for many of the fundamental design goals of the Internet. For example, new CCAs may improve or harm any of capacity utilization, the Internet's fairness landscape, average latency, or burstiness. *Hence, it is no surprise that in recent years researchers have invested significant attention towards characterizing the explosion in CCA diversity.*

CCA Classifiers: Most attempts to characterize the Internet CCA landscape focus on *classifying* CCAs: identifying whether a given Internet service is using a particular published CCA. State-of-the-art examples from this literature include Gordon [50], Inspector Gadget [28], and others [54, 59, 63].

Most CCA classifiers connect to a server under investigation and attempt to model or measure the CCAs visible congestion window (visible CWND): the number of outstanding bytes in flight, over time. They then use some classification algorithm (e.g., decision tree, neural network) to match this time series of CWNDs to known, ground-truth observations of existing CCAs (typically some subset of the 16 default CCAs available in the Linux kernel). Classifiers can neither provide insight into these unknown CCAs (other than

that they are unknown), nor provide any insight into known CCAs (and indeed can also mis-classify known CCAs).

The Impact of Unknown CCAs: As discussed above, CCAs determine crucial properties of the Internet's performance, such as link utilization, fairness, burstiness/variability, and latency. One useful way to understand unknown CCAs is performing measurement either "in the wild" [19] or in testbed environments [3, 50, 55, 58]. While these experiments *can* illuminate useful, empirical properties of the observed CCAs under study, they remain limited. Without knowledge of the underlying algorithm, it is not possible to *prove* bounds or guarantees about the algorithm's behavior [2]. It is also not possible to diagnose *why* a particular pessimal behavior is happening or make recommendations how to fix it: empirical tests simply discover *that* something is wrong. *Hence, we argue that to truly understand the impact of novel CCAs on the Internet performance landscape, it is crucial to understand the algorithm behind each and every CCA.*

2.2 Program Synthesis for Reverse Engineering

The process of generating a program based on its observed inputs and outputs is, by definition, a form of *program synthesis*. A large literature of programming-by-example (PBE) tools follows the blueprint of taking observed inputs and outputs and generating a program that maps from input to output. However, existing PBE tools cannot reverse engineer congestion control algorithms for two key reasons: **statefulness** and **noise**.

Statefulness: Prior work has proposed many specialized PBE synthesizers to solve practical problems such as data structure transformations [24], spreadsheet data manipulation [30], data preparation tasks [21], and applications to computer networks [14, 60, 67]. There are also general PBE synthesizers [47] that take any DSL and a set of examples as input and produce a program that satisfies the examples. To produce this program, synthesizers use machine learning [5, 15, 38, 43], constraint solvers [37, 61], or some combination thereof [16, 20].

Unfortunately, not all program outputs are merely the result of a stateless operation over visible input variables, and CCAs are one such case. For example, the output of TCP NewReno after a loss is not simply $\frac{1}{2}$, it is $\frac{1}{2} \times CWND$, the previously held congestion window. Each timestep of the CCA's progress depends on *both* the inputs observed (losses, packets ACKed, measured RTTs) *as well as the existing state of the system.* Because most PBE synthesizers cannot model this state, they cannot synthesize CCAs.

Noise: Mister880 is a prototype CCA synthesizer [23]. Like us, the authors aim to use Mister880 to uncover the underlying algorithms of novel CCAs over the Internet. However, Mister880 formulates the synthesis as a *decision problem* (as do the vast majority of program synthesizers): it is only capable of modeling candidate synthesized programs as 'correct' or 'incorrect' and has no flexibility for programs that are 'close' to the correct solution but which do not perfectly replicate the observed behavior of the ground truth implementation. This matters because in the Internet, packet traces from a given CCA are *noisy*. Measurements of the ground-truth CWND may reflect the vantage point of the trace measurement; there may be packet delays or jitter that are unobserved; there may be unexpected timeouts or losses that are observed at the sender but not to our measurements, *etc.*. Hence, even if we had an exact copy of the ground truth system, it is not possible to guarantee that our measurements of the system 'in the wild' and our measurements of the system in our testbed will be truly identical. In this setting, Mister880 would discard even the correct algorithm as incorrect.

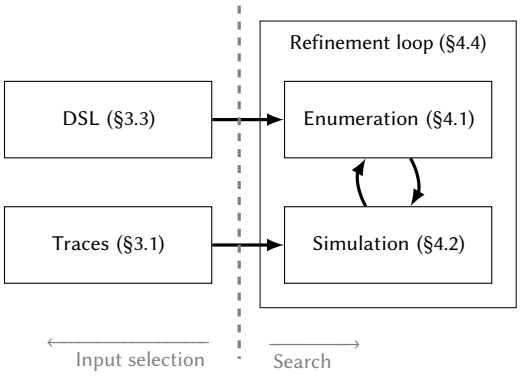


Figure 1: Abagnale system overview

3 Model and Inputs

In the following sections, we describe Abagnale, the first program synthesizer to take on (a) stateful programs and (b) noisy input data. Abagnale's stateful model of CCA behavior is similar to the model used by Mister880, but its formulation of program synthesis as an *optimization* problem rather than a *decision problem* is an entirely different formulation.

Abagnale uses the measured outputs of candidate synthesized CCAs to compute a 'distance' between the proposed CCA and the ground-truth measurement; candidate CCAs with a lower distance are considered better than those with a higher distance. Because our goal is to minimize the distance, but not necessarily bring it to 0, Abagnale can produce algorithms which almost match the observed behavior of the ground truth CCA, thus accounting for noise.

We find that formulating the synthesis problem as an optimization procedure allows us to make further improvements to our synthesis procedure, including making the search for the best candidate CCA more efficient and making the search space of candidate CCAs smaller (§4.4).

Model: In general, a CCA contains multiple state variables–*e.g.* congestion window or link capacity estimate–that determine its behavior. The CCA reacts to multiple *events*–*e.g.* the arrival of an acknowledgment or the determination of a packet loss. A comprehensive model of CCAs would thus determine expressions, or *handlers*, to update each state variable upon the occurrence of each event. With Abagnale, we focus on a specific but important sub-problem: synthesizing an expression to update the congestion window upon an acknowledgment's arrival. While we believe Abagnale's technique generalizes to synthesizing expressions to update other known state variables for other events, we do not evaluate such scenarios in this paper. We also leave synthesizing expressions to update a CCA's packet pacing rate to future work. A further generalization might consider *unknown*, or hidden, state variables, that affect the CCA's behavior. We do not address these cases with

Abagnale and leave them to future work. However, we note that Abagnale's model can in some cases (*e.g.* BBR, see §5.2) nevertheless synthesize handlers that approximate the CCA's behavior despite not modeling hidden state variables.

Fig.1 shows an overview of Abagnale's synthesis process. In §4, we discuss how Abagnale searches a space of possible programs to identify candidate CCAs to replicate the ground truth CCA. Recall that this search space is intractably large; without both optimizations and approximations we discuss, the synthesis process does not succeed. However, before discussing the search process, we must first identify a method to determine whether a candidate CCA has replicated a ground truth CCA (§3.1); second, we must determine what space of possible programs to search (§3.3).

3.1 Evaluating Candidate CCAs

Like CCA classifiers [28, 50, 63], we measure the observable CWND and other signals (RTT, packet rate, *etc.*) over time from a packet trace. Similarly to Mister880 [23], we execute each candidate handler function in simulation given the same events and inputs observed for the ground-truth CCA. For each packet received in the collected trace, we execute the candidate handler function, and, based on resulting CWND value, decide whether to send the next packet. Once this is done for the whole trace, we have a second time-series of the CWND produced by that handler. We call the trace resulting from this simulation the *synthesized trace*. Unlike Mister880 [23], we compare two candidate CCAs by computing the *distance* (§4.3) between the two CWND traces. A handler is a better candidate than another handler if it has a lower distance to the ground truth trace. Using a distance measure rather than assuming the best handler will produce identical outputs to the ground truth CCA allows us to handle *measurement noise* – e.g., unobserved losses or jitter between our vantage point and the server. In addition, using a distance measure allows us to use three optimizations that tune Abagnale's exploration of the search space; we explain how we generate candidate handlers in §4.2, how we select a distance metric in §4.3, and finally how we use these pieces to explore the search space in §4.4.

3.2 Trace Collection

An additional necessity in evaluating candidate handler functions is ensuring that we have *representative traces* for that handler function. We need a wide range of measurements of the ground truth CCA which capture the CCA's behaviors under varying conditions and events, otherwise we risk 'overfitting' to one particular trace and set of conditions (for example, we might return a handler that simply returns a constant CWND, the trace's BDP) [57]. To avoid this, we provide Abagnale with a *diverse* set of testing environments in order to observe more behaviors from the ground truth CCA while also doing so in a way that does not result in *too much data* for Abagnale to take in.

To achieve trace diversity, we use a controlled testbed from a prior study [62] to configure a virtual network with RTTs ranging between 10 to 100ms and bandwidth between 5 and 15Mbps. Collecting traces representing a wide range of conditions enables Abagnale to choose better candidate handlers. Indeed, when we attempt to synthesize Cubic based on traces, Abagnale fails to find a correct function when only given traces from any one configuration of RTT and bandwidth: it is only when we provide traces from a range of settings that Abagnale correctly synthesizes a Cubic function.

$$
\begin{aligned}
\textit{cong-signal} &: \textit{mss} \mid \textit{acked-bytes} \mid \textit{time-since-loss} \\
&\mid \textit{rtt} \mid \textit{min-rtt} \mid \textit{max-rtt} \mid \textit{ack-rate} \mid \textit{rtt-gradient} \\
\textit{num} &: \textit{cwnd} \mid \textit{cong-signal} \mid \textit{constant} \\
&\mid \textit{num} + \textit{num} \mid \textit{num} - \textit{num} \mid \textit{num} \cdot \textit{num} \mid \frac{\textit{num}}{\textit{num}} \\
&\mid \textit{bool}\ ?\ \textit{num} : \textit{num} \mid \textit{num}^3 \mid \sqrt[3]{\textit{num}} \\
\textit{bool} &: \textit{num} < \textit{num} \mid \textit{num} > \textit{num} \mid \textit{num}\ \%\ \textit{num} = 0
\end{aligned}
$$

Listing 1: Non-colored elements are in the base Reno-DSL, teal elements are extensions for the Cubic-DSL, olive elements are extensions for the rate/delay-DSL.

While providing a large number of traces improves fidelity, evaluating the distance function as described above in §3.1 requires a fixed amount of work per packet in each trace. Thus, evaluating every packet of every trace is too costly. Instead, in Abagnale we first split flow traces into trace segments corresponding to periods between loss events. We infer loss events by searching for instances of triple-duplicate-ACKs. Abagnale increases the number of trace segments considered per candidate expression in each iteration of its refinement loop (§4.4). Given a number of trace segments to consider in an iteration, Abagnale first randomly selects half the desired number of trace segments. For each of these sampled segments t, Abagnale then selects the remaining un-picked segment with the highest distance from t. This trace segment selection strategy makes it more likely to sample a diverse set of trace segments representing many network conditions; this in turn helps Abagnale avoid handlers that "over-fit" to specific traces.

3.3 DSL Curation

Finally, Abagnale takes as input a *domain-specific language* (DSL) from which it will produce an expression to match the input packet traces. Abagnale supports a large set of congestion signals and variables, based on prior work on frameworks for developing CCAs [52, 65]. This high expressivity comes at a cost: including all known signals and combinations of signals in the DSL would make the search space intractably large. Instead, we provide as input to Abagnale a CCA-family-specific DSL which uses only a subset of signals and operations. Of course, the user could use different DSLs in separate Abagnale invocations on the same traces.

Indeed, groups of CCAs often use similar signals (e.g., Reno/Westwood/etc, BBR/Vegas/Veno). Hence, we only include these signals if called for in a chosen 'sub-DSL.' For example, the 'rate/delay' sub-DSL includes signals for the RTT and ACK rate, used by CCAs like BBR, Vegas, and Veno, but not by CCAs like Reno or Cubic. We use existing CCA classifiers to hint which sub-DSL Abagnale should use for a given set of traces. We show in §6.3 that this strategy picks DSLs similar to those we would have chosen manually.

Listing 1 shows how input DSLs can vary. Almost all useful DSLs will include the black-colored elements (*e.g.*, arithmetic operators), while the user may choose to exclude uncommon operators such as cube-root. Some of the operators are derived from others, *e.g.*,

reno-inc	$\frac{\text{ACKed}\times\text{MSS}}{\text{CWND}}$
vegas-diff	$\frac{(\text{RTT}-\text{minRTT})\times\text{ack-rate}}{\text{MSS}}$
htcp-diff	$\frac{\text{RTT}-\text{minRTT}}{\text{maxRTT}}$
RTTs-since-loss	$\frac{\text{time-since-loss}}{\text{RTT}}$

Table 1: Pre-defined macros used in Abagnale's DSLs

we offer a built-in EWMA operation since it is commonly used in CCA evaluation. While we could ask Abagnale to 'discover' the EWMA operation during the synthesis process, we have found that encoding common macros in the DSL enables Abagnale to more effectively identify fruitful candidate expressions. Table 1 lists the macros used to simplify CCAs' commonly used expressions in Abagnale's DSLs. *reno-inc* is Reno's CWND increment of one MSS per sent packet. *vegas-diff* is Vegas's estimation of the difference between the expected and the actual sending rate [8]. *htcp-diff* is the variation in RTT, as used by H-TCP [44]. *RTTs-since-loss* is the time since the last loss event scaled by the current RTT estimate, as used by BBR [12].

4 Exploring the Search Space

Once the input DSL and traces are defined, Abagnale explores the resulting search space. The main challenge is the search space size: even when considering a sub-DSL, the search space would remain intractable if traversed naively. To cope with the size of the search space, we adopt the following four key techniques:

DSL constraints. First, we place constraints on the members of the DSL we will *enumerate* (*i.e.* consider to be candidate CCAs) (§4.1).

Constant sampling. Second, candidate CCAs will contain constant values, and the correct setting of those values is susceptible to trace noise, so rather than considering each setting separately, we consider a random sampling of constant assignments as a single set (§4.2).

Bucketization. Third, we devise a divide-and-conquer approach that splits the search space into partitions; searching through these partitions independently is faster than searching the entire space as a whole (§4.4).

Bucket prioritization. Fourth, we identify a bucketing metric that allows us to consider entire buckets of CCAs *as a whole* and prioritize which buckets to explore deeper into (by splitting them into sub-buckets) (§4.4).

4.1 DSL Enumeration

Our search space is composed of all the trees of operations (*i.e.*, abstract syntax tree, or AST) that can be built by combining the DSL components. These ASTs may not correspond to concrete event handlers, since they may have nodes assigned the DSL component *constant* that do not receive a final value until the simulation phase (§4.2). Until then, we call these incomplete handlers *sketches*. Possible sketches from this DSL include the Reno (equation 1) and Vegas (equation 2) sketches. The olive-colored elements represent the components specific to the Vegas-DSL, and c_1, c_2, c_3, c_4, c_5 represent undefined constants. Figure 2 visualizes searching an AST within the Reno-DSL.

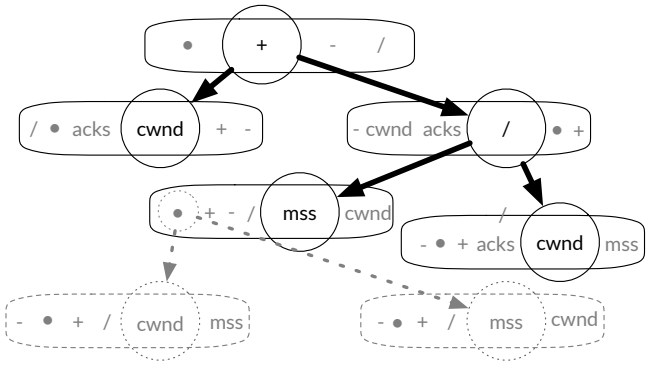


Figure 2: Visualizing the search through an AST.

$$cwnd \mathrel{+}= mss \cdot \frac{acked}{cwnd} \tag{1}$$

$$cwnd \mathrel{+}= \left(\frac{cwnd}{min\text{-}rtt} - ack\text{-}rate < c_1\right) \; ? \; c_3 \cdot mss : \left(\frac{cwnd}{min\text{-}rtt} - ack\text{-}rate > c_2\right) \; ? \; c_4 \cdot mss : c_5 \tag{2}$$

The number of sketches we can build from DSL components is infinite – we could simply keep growing the expression tree – so we limit the search space by limiting the maximum depth of the AST. For a fixed depth, the number of possible sketches grows exponentially with the number of DSL components. This makes the search space very large: if we consider trees of maximum depth 7 with the 25 components of the DSL in Listing 1, the correct sketch is one out of a universe of $\approx 10^{150}$.[1]

Dealing with large search space sizes is common in synthesis literature, so we start by leveraging techniques from previous work. First, we rely on an SMT formula to extract from this space only sketches that type-check [22, 32, 47]. We also specify that sketches should not be arithmetically simplifiable using the sympy [49] library. Second, we impose CCA-specific constraints: the output should have the correct units (in this case bytes) and should not monotonically decrease (since any reasonable CCA must grow the window at some point). We iteratively query an SMT solver to explore the search space with the resulting formula. Each solution to the SMT formula is a sketch with the desired properties. After obtaining a sketch, we can ask the SMT solver for a different sketch by adding a constraint that blocks the previous solution.

4.2 Concretizing Enumerated Sketches

The sketches the enumeration process returns can have unassigned constants, e.g., $c1$, $c2$, ... in equation 2. To evaluate a candidate sketch, we first need to produce a concrete handler function with no unassigned constants from the sketch. One way to concretize the constants in a sketch is to try different concrete number values for each constant. The problem with this combinatorial search is that the set of concrete handlers associated with a single sketch grows exponentially: the number of ways we can assign k variables with n values is k^n. So, for example, the Vegas sketch has 5 unassigned

[1] The number of atoms in our universe is $\approx 10^{79}$, so the sketch's universe is significantly larger.

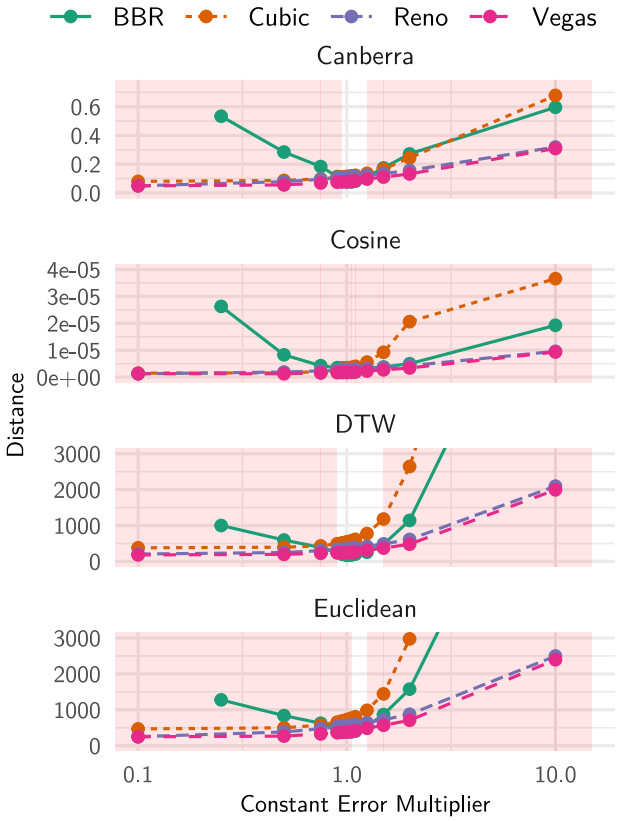


Figure 3: Comparison of distance metrics' tolerance to error in constant values, for traces from the BBR CCA. Red-shaded regions indicate that a synthesized CCA other than BBR had a smaller distance to the traces. Note the x-axis, showing the amount of error we introduce to the fine-tuned constant values, is in log-scale.

variables, so if we consider 10 different values for each, we will get ≈10 million handlers for just one sketch out of millions. We want these constants to be able to take any real value, but solving a real-valued optimization problem for each sketch is prohibitively slow. Rather, Abagnale focuses on identifying promising sketches with *approximate* concretization. That is, we limit the values constants can take to a small set of values observed in known CCAs to estimate a sketch's distance from a trace fragment. This strategy makes our approach incomplete, i.e., there are handlers that we will not explore, dependent on the predefined values chosen for the constants. However, in our experiments, this did not prevent Abagnale from returning useful sketches. After Abagnale returns a handler, it is possible to evaluate this handler's sketch using a broader set of constant values.

4.3 Selecting a Distance Metric

How should Abagnale determine whether a candidate handler matches a set of traces? Because of our optimization formulation of the synthesis problem, we require a distance metric. However, there are various methods for computing the distance between two traces.

Importantly, as described above, Abagnale cannot exhaustively evaluate all assignments of constant values since doing so would make the search space too large. As a result, it is important to select a distance metric that tolerates error to the greatest extent possible.

In Figure 3, we show how four distance metrics respond to errors in handlers' constant values. We use packet traces corresponding to BBR,[2] and we calculate the distances using in-DSL expressions for BBR, Cubic, Reno, and Vegas written by a domain expert. We introduced a fixed amount of multiplicative error, from 0.1 to 10, to each constant in each handler, and measure the resulting handler's distance from the trace. We then determine for each amount of error whether the correct CCA's handler remained the closest to the trace. If another CCA's handler was closer using that distance metric, we shade the background in red. We observe that the Dynamic Time Warping (DTW) [6] distance remains correct for the widest range of constant error. This distance metric is alignment-based; *i.e.* it seeks to correct for temporal shifts between curves. Unfortunately, the DTW distance metric is significantly more expensive to compute than Euclidean distance. For Abagnale, we find that in most cases, DTW's improved amenability to constant error is worth the additional runtime, and configure Abagnale to use it unless otherwise described.

4.4 Guiding the Search

Even when using curated sub-DSLs and the above techniques, Abagnale could not synthesize CCAs that use more complex DSLs. We tackled this problem by (1) splitting the search space into disjoint subspaces, and (2) using our distance metric (§3.1) to prioritize subspaces that are more likely to generate CCAs with lower distance to the input traces.

Partitioning the search space. Partitioning the space facilitates parallelization by allowing Abagnale to use a specialized solver invocation per bucket and performing the search across buckets in parallel [9]. This makes enumeration significantly faster not only because it can take advantage of multiple cores, but also because each solver is searching over a smaller space. Recall that the solver query grows every time we query the solver for a new sketch since we must exclude all previously returned sketches to get a new one and that solver execution time grows rapidly with query size. Thus, smaller sub-search spaces can use smaller queries, which is faster. We call each of these sub-search spaces a *bucket*.

Bucketing metric. How should we divide the space into buckets? We must ensure that each sketch in the DSL belongs to a single bucket according to a *bucket discriminator*: a metric we can express in the solver query which will cause it to only enumerate sketches in the bucket corresponding to the metric's value.

We want to choose a metric such that two sketches in the same bucket share not only some *structural* similarity so that we can easily encode it into a dedicated solver but also share some *behavioral* similarity. Provided *behavioral* similarity between the sketches in the same bucket, we can sample N sketches from all the buckets at the beginning, simulate them using the procedure described in §4.2, and assign each bucket a score based on how close the traces in the sample were to the desired behavior. Using these scores, we

[2]When selecting a primary distance metric, we additionally evaluated other CCAs' traces and other distance metrics, but we elide those results for brevity.

Algorithm 1 Abagnale's refinement Loop

```
1:  procedure SynthesisLoop(DSL, buckets, N, k)
2:      while buckets not exhausted do
3:          for all bucket ∈ buckets do                    ▷ in parallel
4:              samples ← enumerate N sketches from bucket
5:              distances ← distance for each sketch ∈ bucket
6:              bucket-score ← min(distances)
7:          end for
8:          buckets ← only-top-k(buckets, bucket-scores)
9:          N ← N · 2^3
10:         k ← k/2
11:     end while
12: end procedure
```

order the buckets from most promising to least promising, drop the least promising buckets, and repeat this loop with an increased size of samples N and a reduced number of buckets.

We considered four different bucketing metrics: (1) fixing the operations of the nodes of the first 3 levels of the tree, (2) limiting the subset of DSL operators (addition, multiplication, power, etc) the sketch can use, (3) limiting the subset of congestion signals and state variables (delay gradient, minimum RTT, time since the last loss, etc) the sketch can use, and (4) fixing *all* DSL elements the sketch can use (so, a combination of (2) and (3)). We found that limiting the subset of DSL operators (addition, multiplication, power, etc.) the sketch can use–option (2) above–provided the best results: this metric is easy to integrate into the enumeration process and allows Abagnale to have an independent SMT solver for each bucket and enforce the bucket's metric value without much overhead in the formula.

Search prioritization. We use our bucketing metric to guide the search. Abagnale uses a refinement loop as shown in Algorithm 1. The algorithm takes as input the DSL, and each bucket's discriminator. It also takes the initial values of N, the number of samples we will consider from each bucket and k, the number of buckets that are retained to the next iteration. In each loop iteration, we sample N sketches from each bucket (line 4). In line 5, we run the simulation procedure on each of them and save the best distance that a concrete handler built from that sketch can achieve. Then each bucket is assigned a score in line 6, equal to the minimum of these distances. Having all scores for all buckets, we sort them in line 6 from most promising to least promising, which corresponds to increasing values of their score. Then, in line 8, we refine the search space by selecting only the most promising buckets to be further explored. *only-top-k* will return the subset of the buckets whose scores are lower than or equal to the k-th bucket score. This means that, if there are no ties, k buckets are retained to the next iteration of the loop. Before going on to the next iteration of the loop, we update N and k. Now that we know we are looking into the more promising subset of all buckets, we want to dig deeper into each one to find as good a handler as possible, so N, the sample size for each bucket, is increased by 8 times in l.9. As we get deeper into each bucket, we increase the trust in our scoring, so we want to get more and more conservative in how many buckets keep in the search. In l.10 we update k to half its previous value. Since we expect each iteration to evaluate fewer handlers, we can afford to compute distances using more traces, so we also increase the number of distinct traces being used by two. Abagnale repeats this loop until either (1) there is one bucket left, in which case we exhaustively enumerate it and return the best handler within it, or (2) N grows larger than the size of the largest bucket still in consideration, which means all buckets have already been exhaustively enumerated. During the whole loop duration, Abagnale stores the lowest distance handler it has found thus far, so if the user interrupts the loop (e.g., with a timeout), Abagnale will return that handler.

5 Results

We now show Abagnale's synthesized expressions across two sets of packet traces. The first set of traces corresponds to the 16 CCAs with implementations distributed with the Linux kernel: BBR [12, 40], Cubic [33], Vegas [8], Reno [36], BIC [66], CDG [35], HighSpeed, H-TCP [44], Hybla [11], Illinois [45], LowPriority [41], NV [7], Scalable [39], Veno [27], Westwood [48], and YeAH [4]. These CCAs are implemented as Linux kernel modules in ~50–500 lines of C. The second set of CCAs is a publicly available dataset of novel CCAs written by students at a US university as part of a graduate-level networking class. These CCAs are implemented in between 50–150 lines of C++.

Abagnale produces arithmetically simple expressions—*i.e.* with a maximum AST depth of 5, which is significantly simpler than the original implementations—for all these CCAs.

Implementation. To synthesize these expressions, we implemented Abagnale on Python 3.11.7. We ran all experiments using Intel Xeon Gold 6226R with 256GB of RAM, Intel Xeon E5-2630 v2 with 64GB of RAM, and Intel Xeon Silver 4110 CPUs with 64GB of RAM, with different numbers of cores and RAM. We used Z3 [18] version 4.8.10 for all SMT queries. Since scoring handlers (§4.3) is a parallelizable task, we used Ray [51] to distribute the synthesis tasks among cores across different machines. For every experiment, we explored different depths of the same DSL on different parallel machines (we evaluate the impact of DSL depth in §6.3). We ran all synthesis tasks to completion (*i.e.*, until Abagnale returned a result); in all cases, this took less than 48 hours per depth per CCA.

5.1 Results Overview

We show a summary of CCAs we attempted to synthesize in Table 2. Each row in the table refers to an analysis of traces derived from a single CCA, identified in the first column. In the second column we show the expression Abagnale synthesizes, as well as the sum of distances between the synthesized traces and the respective collected traces. Note that these expressions use only the default constant values listed in §6.1, but we arithmetically simplify the expressions where possible for readability. Abagnale computes these distance values shown over the trace segments used to synthesize each CCA. Since Abagnale synthesizes different CCAs using different sets of traces, the distance values shown for these handlers are not comparable across CCAs, i.e., across rows. Within the same row, the difference between the synthesized handler distance and the fine-tuned handler distance gives us an idea of how close the behavior of these two handlers is. In rows where the synthesized handler distance is the same as, or very close to, the fine-tuned handler distance (e.g., BBR, Reno, Scalable, LP, Hybla, HTCP, Illinois, Vegas, Veno), Abagnale outputs a handler that closely matches the behavior of the fine-tuned handler in the

CCA	Synthesized cwnd-ack handler	DTW distance	Fine-tuned cwnd-ack handler	DTW distance
BBR	$2 \times \text{ack-rate} \times \text{minRTT} + \{\text{CWND}\%2.7 = 0\}\ ?\ 2.05 \times \text{CWND} : \text{MSS}$	195.21	$\text{minRTT} \times \text{ack-rate} \times (\{\textit{RTTs-since-loss}\%8 = 0\}\ ?\ 2.6 : 2.05)$	143.08
Reno	$\text{CWND} + .7 \times \textit{reno-inc}$	18.84	$\text{CWND} + .7 \times \textit{reno-inc}$	18.84
Westwood	$\text{CWND} + \textit{reno-inc}$	86.99	$\text{CWND} + .68 \times \textit{reno-inc}$	12.72
Scalable	$\text{CWND} + .37 \times \textit{reno-inc}$	26.25	$\text{CWND} + .37 \times \textit{reno-inc}$	26.25
LP	$\text{CWND} + .68 \times \textit{reno-inc}$	18.2	$\text{CWND} \times (\{\textit{htcp-diff} > .5\}\ ?\ .5 : 1) + .68 \times \textit{reno-inc}$	18.2
Hybla	$\text{CWND} + 8 \times \text{RTT} \times \textit{reno-inc}$	35.77	$\text{CWND} + 8 \times \text{RTT} \times \textit{reno-inc}$	35.77
HTCP	$\text{CWND} + \textit{reno-inc}$	56.24	$\text{CWND} + \textit{reno-inc} \times \{\textit{htcp-diff} < .25\}\ ?\ 1 : .2$	54.53
Illinois	$\text{CWND} + 1.3 \times \textit{reno-inc}$	397.99	$\text{CWND} + .3 \times \textit{reno-inc} + 5 \times \textit{reno-inc} \times \textit{htcp-diff}$	467.81
Vegas	$\text{CWND} + \{\textit{vegas-diff} < 1\}\ ?\ .7 \times \textit{reno-inc}\ :\ 0$	24.36	$\text{CWND} + \{\textit{vegas-diff} < 1\}\ ?\ .7 \times \textit{reno-inc} : \{\textit{vegas-diff} > 5\}\ ?\ -.7 \times \textit{reno-inc} : 0$	20.21
Veno	$\text{CWND} + \textit{reno-inc} \times \{\textit{vegas-diff} < .7\}\ ?\ .35 : .16$	9.26	$\text{CWND} + \textit{reno-inc} \times (\{\textit{vegas-diff} < .7\}\ ?\ .35 : .16)$	9.26
NV	$\text{CWND} + \{\textit{vegas-diff} < 1\}\ ?\ .7 \times \textit{reno-inc}\ :\ 0$	58.1	$\text{CWND} + \{\textit{vegas-diff} > 1\}\ ?\ .7 \times \textit{reno-inc} : \{\textit{vegas-diff} > 5\}\ ?\ -.7 \times \textit{reno-inc} : 0$	479.39
YeAH	$\text{CWND} + \textit{reno-inc} \times \{\textit{vegas-diff} > 5\}\ ?\ .3\ :\ 1$	33.41	$\text{CWND} + \textit{reno-inc} \times \{\textit{vegas-diff} > 5\}\ ?\ .3 : 1$	33.41
Cubic	$\text{CWND} + \text{time-since-loss}^3$	3580.67	$\text{wmax} + (8 \times \text{time-since-loss} - \sqrt[3]{(.24 \times \text{wmax})})^3$	41.74
Student 1	88	196.06	–	–
Student 2	$\left\{\frac{\textit{vegas-diff}}{\text{minRTT}} < 5\right\}\ ?\ \text{CWND} + \text{MSS}\ :\ \text{MSS}$	12203.07	–	–
Student 3	$.8 \times \frac{\text{ACKed}}{\text{minRTT}}$	7698.63	–	–
Student 4	MSS	217.56	–	–
Student 5	$2 \times \text{MSS}$	32.69	–	–
Student 6	$\frac{\textit{cwnd}+150\times\text{MSS}}{\text{delay-gradient}}$	24406.14	–	–
Student 7	$\text{CWND} + \frac{2\times\text{ACKed}}{\text{RTT}}$	17541.93	–	–

Table 2: Results of running Abagnale on different input traces. The first column, "CCA" shows the ground truth, i.e., the algorithm that was running when the set of traces used for this task was collected. The second column shows Abagnale's output cwnd-ack handler expression, and the sum of DTW distances between synthesized traces computed with this handler and the respective ground truth traces. The third column shows a domain-expert's attempt at handwriting a cwnd-ack handler expression from the source code of the respective CCA, as well as the sum of DTW distances computed with these handlers.

traces used for synthesis. When the synthesized handler distance is much higher than that of the fine-tuned handler (e.g., Cubic), Abagnale's refinement loop was unable to select the correct bucket for exploration, and the fine-tuned handler was never evaluated.

Before running Abagnale on the collected traces, we run a CCA Classifier, Gordon [50], on the same CCA. Gordon establishes multiple connections to the server, and classifies each connection as running one of its known CCAs (BBR, Cubic, BIC, HTCP, Scalable, YeAH, Vegas/Veno, Reno, Illinois, and Westwood), or as "Unknown". Table 3 shows Gordon's output. If Gordon determines that a majority of the test connections match a single CCA, we list that CCA in the table. If Gordon only matches a minority of the connections, we report that output in parentheses. Finally, when we ran Gordon on New Vegas ("NV" in the table), it classified all of the connections as "Unknown", so we report this result as "Unknown".

The CCAs from the class project dataset are implemented with a UDP transport, which Gordon does not support. Thus, for these, we run CCAnalyzer [63]. As expected (since these are all novel algorithms), CCAnalyzer outputs "Unknown" for all algorithms. Since this classifier uses a distance metric to compare with its known algorithms, we can also ask it for the closest known algorithms to the trace behavior. CCAnalyzer reported CDG and Vegas as the closest CCAs to all the student's algorithms but one, for which it reported Vegas and Scalable. As before, we use these classifier results to pick the DSLs we run Abagnale with.

Fine-Tuned Handlers: We emphasize that it is not Abagnale's goal to reproduce the CCA implementation that resulted in the collected packet trace; as described above, these implementations can comprise hundreds of lines of code, covering both logic irrelevant to the CCA's behavior such as custom congestion signal measurement logic and edge cases that Abagnale does not attempt to capture.

CCA	Classifier output
BBR	**BBR**
Reno	**Reno**
Westwood	Vegas
Scalable	**Scalable**
LP	Unknown (Vegas)
Hybla	BBR
HTCP	**HTCP**
Illinois	**Illinois**
Vegas	**Vegas**
Veno	YeAH
NV	Unknown
YeAH	**YeAH**
Cubic	**Cubic**
Student 1	Unknown (CDG, Vegas)
Student 2	Unknown (CDG, Vegas)
Student 3	Unknown (Scalable, Vegas)
Student 4	Unknown (CDG, NV)
Student 5	Unknown (CDG, Vegas)
Student 6	Unknown (CDG, Vegas)
Student 7	Unknown (CDG, Vegas)

Table 3: Result of running a classifier (Gordon [50] for the Kernel algorithms, or CCAnalyzer [63] for the students algorithms) for the CCA. The CCA name in parenthesis after "Unknown" is the CCA that the classifier identified as closest, despite the output being Unknown. We color classifier outputs blue if correct and red if incorrect.

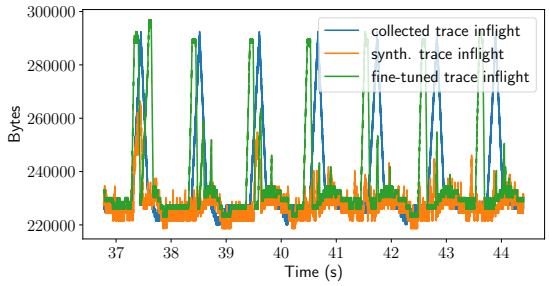
Thus, rather than using the implementation that generated the trace as our ground truth, we use fine-tuned versions of the synthesized handlers. To write these fine-tuned handlers, we used the synthesized expression as a starting point and use domain knowledge of the CCAs' implementations and original publications to write a handler with the same depth and within the same DSL that captures the CCA's behavior. It is easy to miss implementation details and it is hard to simplify the computations in the code in the form of a single handler, so these handlers are also not a perfect match of the CCA's behavior. In fact, as can be seen in Table 2, some fine-tuned handlers have longer distance to the collected traces than the respective synthesized handler. This happens when, analyzing the original CCA's implementation, we expected the CCA to closer to what we show in the fine-tuned handler, yet we were unable to adjust the expression to match the observed traces better than the synthesized handler. We show the fine-tuned handler for each CCA in the third column of Table 2. We used these fine-tuned handlers to understand the handlers Abagnale produces more deeply. We show an evaluation of Abagnale's accuracy relative to these fine-tuned handlers in §6.2.

5.2 BBR

BBR [12]'s core behavior consists of periodic pulses that probe for additional bandwidth, called "PROBE_BW" mode. In most implementations, these pulses are controlled by a state variable which determines whether the sending rate and congestion window are set

Synthesized win-ack handler for BBR (DTW distance 92.0):
win-ack = $(2) \cdot \text{ack\_rate} \cdot \text{min\_rtt} + ((\text{cwnd}\%2.7 = 0)\ ?\ 2.05 \cdot \text{cwnd} : \text{mss})$

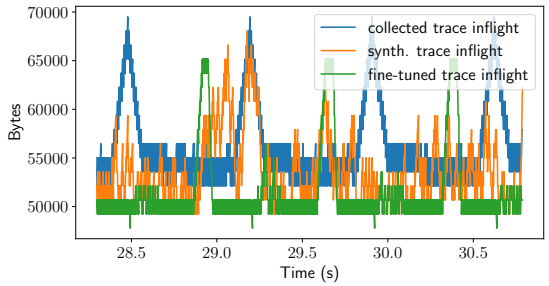
Fine-tuned win-ack handler for BBR (DTW distance 73.9):
win-ack = $((\text{rtts\_since\_loss}\%8.0 = 0)\ ?\ 2.6 : 2.05) \cdot (\text{min\_rtt} \cdot \text{ack\_rate})$



(a) A trace in which the fine-tuned handler achieves a lower distance than the synthesized one.

Synthesized win-ack handler for BBR (DTW distance 44.9):
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 (DTW distance 75.5):
win-ack = $((\text{rtts\_since\_loss}\%8.0 = 0)\ ?\ 2.6 : 2.05) \cdot (\text{min\_rtt} \cdot \text{ack\_rate})$



(b) A trace for which the fine-tuned handler achieves a higher distance than the synthesized one.

Figure 4: Even though the handler handwritten by a domain expert based on the BBR kernel implementation is a better visual match to the collected BBR trace, the synthesized trace with random "spikes" has a lower distance for some traces.

above, below, or at the estimated bottleneck rate. Of course, Abagnale does not support hidden state variables and can only produce closed-form expressions. In this case, Abagnale produces pulsing behavior in a different way: if the CWND is an even number, it sets the CWND to $8 \times CWND$, and otherwise uses $2.15 \times \text{minRTT} \times \text{ack-rate}$. This expression captures BBR's "CWND gain" feature that seeks to maintain a standing queue [62]. By periodically increasing the CWND beyond this value, the handler will achieve the same probing property as BBR, since if the true bottleneck bandwidth is a higher value than ack-rate, then the ack-rate will increase correspondingly.

Digging deeper into this synthesized handler, we compare its fidelity to the fine-tuned handler (which uses $rtt\_since\_loss\ \%\ 8 == 0$ to implement pulses) in Figure 4. Indeed, in Figure 4a we see that this fine-tuned handler achieves a lower DTW distance than the synthesized handler. This matches visual intuition; in this trace, the fine-tuned handler's pulses are aligned with the pulses observed in the trace. However, this is not true for all traces. In the trace shown in Figure 4b, the synthesized handler achieves a lower distance. This

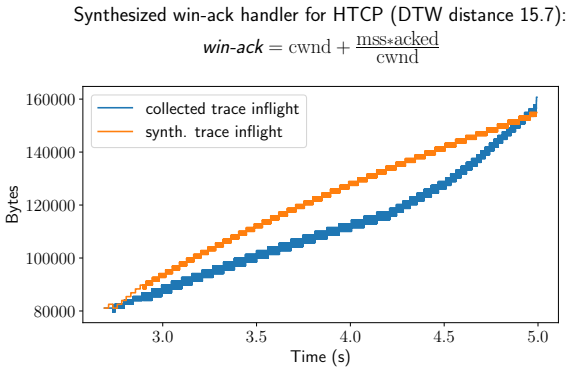


Figure 5: Although this HTCP trace exhibits an inflection point, a simple Reno-variant handler has a low enough distance that Abagnale does not explore more complex handlers.

example demonstrates a limitation of the DTW distance metric; because DTW purposely disregards temporal shifts, it is less likely that Abagnale will produce a synthesized handler which matches the original CCA's pulse behavior. Nevertheless, Abagnale produces a viable expression for BBR which is significantly simpler and more understandable than the original implementation.

5.3 Reno-Variant CCAs

The Reno, Westwood, Scalable, and LP CCAs all behave similarly to Reno, with minor modifications to the cwnd-increase function. Indeed, Abagnale produces similar expressions for traces generated by these CCAs. These CCA expressions matched the kernel implementations' behavior even with combinations of Abagnale's default placeholder constants. Fine-tuning these CCAs only required modifying the constant values. Thus, Abagnale is able to confirm (without needing access to the source code) that these CCAs behave similarly to each other, and is able to estimate each CCA's relative aggressiveness.

Three more CCAs' traces result in handlers with the same Reno-variant structure: Hybla, HTCP, and Illinois. The objective of the Hybla CCA is indeed to increase similarly to Reno, but to scale the increase to compensate for high-delay links [11]. Indeed, the synthesized handler similarly scales the increase proportionally to the link RTT.

Surprisingly, Abagnale also returns a Reno-Variant handler when provided traces from HTCP [44] and Illinois [45]. This is unexpected because we expect both to depend on delay-based signals. Figure 5 digs deeper into this result for HTCP (Illinois is similar). We find that, indeed, the trace segment depicted exhibits an inflection point in the congestion window growth. However, Abagnale is unable to find a handler that more closely represents the behavior observed in the traces, even at higher depths.

5.4 Vegas-Variant CCAs

The Vegas, Veno, NV, Illinois, and YeAH CCAs all use conditional expressions derived from a delay signal to determine their congestion window evolution. Abagnale consistently includes this feature in the synthesized handlers for these traces. Note that we include in the DSL the expression $(\text{RTT} - \text{minRTT}) \times \frac{\text{ack-rate}}{\text{MSS}}$, which is a commonly used estimator of the number of packets in the bottleneck queue. We highlight that even though the classifier is unable to identify NV, Abagnale correctly produces a Vegas-variant handler given traces from NV.

In fact, Abagnale's output given traces from NV is identical to its output for traces from Vegas. We note that the CCAs Vegas and NV (*i.e.*, "New Vegas") use the same fundamental logic [7, 8]; their differences are only in the way they measure the number of packets in the queue. For example, NV uses a moving average of the delay and uses a hidden state variable to reduce the frequency of its window updates to once per RTT. Since Abagnale provides its own definitions of congestion signals and captures behavior rather than implementation details, it correctly returns the same handler for both these sets of traces.

5.5 Remaining CCAs

The remaining CCAs we consider from the Linux kernel are BIC, CDG, Cubic, and HighSpeed.

BIC: The BIC CCA, at a high level, conditionally performs either binary search between the current window and the window at the time of the last loss, and linear probing [66]. The closest expressions Abagnale returns on BIC traces, meanwhile, grow the window according to the time since the last loss. As a result, we suspect that the correct handler for BIC has an AST depth too deep (with multiple levels of nested conditionals) for Abagnale to effectively explore.

CDG: The CDG CCA calculates the probability of reducing the window based on the RTT value and randomly decides to decrease the congestion window based on this drop probability [35]. Since calculating random values is outside the input DSL, it is not possible for Abagnale to synthesize the correct handler. As a result, we do not run Abagnale on CDG traces.

Cubic: When encoding our unit constraints as described above in §4.1, we make a design decision to only encode integer-valued constraints, so that the enumerator formula remains a quantifier-free finite domain formula, which makes queries significantly faster. Unfortunately, as a result, Abagnale cannot unit-check cube-root operations. We thus run Abagnale with unit constraints disabled on Cubic-derived traces. Indeed, the returned expression captures a subset of Cubic's behavior - growing cubically with the time since the last loss - but this expression does not have consistent units.

HighSpeed: The HighSpeed CCA uses logarithmic operations [25]. In the Linux kernel implementation, this is implemented with a large lookup table. As is the case with Cubic, our enumeration constraints cannot reason about exponentiation and logarithm operations. We did not run Abagnale on HighSpeed traces as a result.

5.6 Student CCAs

When we run the CCAnalyzer classifier on the student CCA dataset, the classifier indicates that all 7 CCAs have some similarity to Vegas. Unsurprisingly, when we run Abagnale on these traces many are of Vegas-variant form: Student CCAs 1, 2, 4, and 5 all modify the congestion window by comparing $(\text{RTT} - \text{minRTT}) \times \frac{\text{ack-rate}}{\text{MSS}}$ to a constant threshold value. Note that the synthesized result for the Student 5 CCA is simplifiable, since the first conditional expression is trivially false; as discussed above, Abagnale cannot reason about this simplification due to its reliance on sympy. We discuss

CCA	pos. after iteration 1	pos. after iteration 2
BBR	4/127	3/5
Cubic	7/27	–
HTCP	2/31	4/5
Hybla	4/7	1/5
Illinois	3/63	3/5
LP	1/63	1/6
NV	5/15	2/5
Reno	3/218	1/5
Scalable	1/218	1/5
Vegas	5/15	4/5
Veno	1/7	1/5
Westwood	1/218	1/5
YeAH	1/31	1/5

Table 4: Abagnale's progress through the search space for the CCAs distributed with the Linux kernel.

the student CCAs in more detail, particularly exploring the impact of DSL depth, in §6.3.

6 Evaluation

We previously described the intractably large search space Abagnale must navigate during the synthesis process. We now evaluate how well Abagnale navigates this space. We consider three aspects of Abagnale's exploration:

First, how much of the search space does Abagnale evaluate in order to return the results in §5? We show Abagnale's exploration of the search space for Reno in §6.1.

Second, how far off was Abagnale from returning the fine-tuned CCA a domain expert developed with knowledge of both the synthesized handler as well as the nature of the ground-truth CCA? We discuss this in §6.2.

Third, how important are the DSL inputs to Abagnale in determining whether it will return a good sketch for an unknown CCA? We evaluate this in §6.3.

6.1 Search Efficiency

We evaluate how efficiently Abagnale explores its search space by digging deeper into its exploration of traces produced by Reno. Recall that for this CCA, Abagnale returns the following expression with depth 3: CWND + .7 × *reno-inc*. Note that we encode *reno-inc* as a macro in Abagnale's DSL, so that sub-expression does not increase the depth.

Reno DSL is shown in Listing 1. Between congestion signals, operators, and macros, this DSL contains 11 elements. The space of all depth-3 sketches that can be built in this DSL is then 2 billion. From those, using the enumeration pruning techniques described in §4.1, Abagnale reduces this space to 1617 sketches. This represents the space of type-checked, unit-checked, non-simplifiable Reno-DSL cwnd-ack handler *sketches*. Each of these sketches can get expanded into concrete win-ack handlers, by filling out its holes. In total, the Reno-DSL search space has 101K concrete handlers.

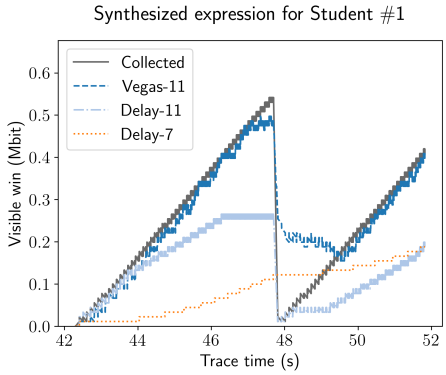
Abagnale first partitions the search space into 218 disjoint buckets. The first iteration of the refinement loop enumerates and scores a sample of 16 handler sketches of each of the 218 buckets. To do this, Abagnale must concretize each sketch with constant values. Each sketch has between 1 and 273 completions, so in this first iteration Abagnale scores a total of 17.5K fully-populated handlers. Scoring these handlers is parallelizable, and completes in 7 minutes on the cluster described above. After this first iteration, Abagnale retains 5 of the 218 buckets. In the second iteration, it samples an additional 112 sketches (totalling 128 across the two iterations) from each bucket. 3 of the buckets contain fewer than 128 sketches in total; we enumerate those buckets exhaustively. Thus, in this iteration Abagnale scores 28.4k fully-populated handlers, in 13 minutes. After this iteration, Reno retains the 2 top buckets. These two buckets both contain fewer than 128 sketches, so they had already been fully enumerated. So, Abagnale returns the handler with the lowest known distance, CWND + .7 × *reno-inc*. Overall, Abagnale finds this handler after exploring only about 1/3 of the viable search space (*i.e.* the search space remaining after all enumeration constraints).

6.2 Search Accuracy

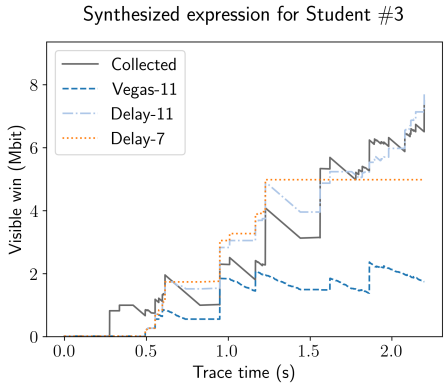
We next evaluate Abagnale's accuracy relative to the fine-tuned handlers described in §5.1. We measure where in the process Abagnale discarded the fine-tuned handler in favor of the one it eventually returned. Note that in some cases, such as with BBR (§5.2), this fine-tuned handler does not have a lower DTW distance to the collected trace than the handler the synthesizer returned.

Table 4 shows this result. Recall from §4.4 that Abagnale's search proceeds iteratively through "buckets" of the search space. The column "position after iteration 1" shows both the rank of the fine-tuned handler's bucket, and the number of possible buckets. For example, for BBR, "4 / 127" indicates that the fine-tuned handler's bucket had the fourth-lowest estimated distance out of 127 total buckets. In the first iteration of the refinement loop, Abagnale retains the top 5 buckets. Thus, in this example, Abagnale correctly discarded 122 of the 127 possible buckets. For Cubic, the first iteration of the refinement loop ranks the fine-tuned handler's bucket 7th. Since only 5 buckets are retained for the second iteration, the fine-tuned handler's bucket gets discarded. If this bucket had not been discarded, exhaustive exploration would have ranked the fine-tuned sketch at 7/4794. Within these sketch completions, the fine-tuned handler would have ranked 1/36. For Cubic, unlike BBR, the fine-tuned handler has lower distance than the expression Abagnale returns. So, if the fine-tuned handler had been sampled from the respective bucket in the first iteration of the loop, Abagnale would have exhaustively searched that bucket and ultimately returned the fine-tuned handler.

The second column shows the same result after the second iteration of the refinement loop. This second iteration has more information about each bucket, because it samples 128 sketches from each of the 5 buckets (in the first iteration Abagnale samples only 16). For BBR, we see that the fine-tuned handler bucket was ranked 3rd in the second iteration, so it was not selected for exhaustive search. Similarly, for Vegas, the fine-tuned handler is in the 4th-ranked bucket after the second iteration of the refinement loop. In both cases, Abagnale exhaustively enumerates, concretizes, and scores the top-scoring buckets (which do not contain the fine-tuned handler). The fine-tuned handler's bucket in the Vegas DSL only contains one sketch; this means that, similarly to BBR, the fine-tuned handler has a higher distance than the handler Abagnale returned.



(a) Student CCA 1 with different DSLs



(b) Student CCA 3 with different DSLs

Figure 6: Three different synthesized CCAs for Student 1 and Student 3 using DSLs identified by CCAnalyzer [63].

6.3 Impact of DSL Input

We use the student CCAs to evaluate the impact of the input DSL on Abagnale's results. In Figure 6a, we show Abagnale's results from three DSLs: a Delay DSL (Listing 1), which includes RTT and rate signals, with constraints of depth 4 and up to 7 or 11 nodes (Delay-7 and Delay-11), and the Vegas DSL, which additionally includes a macro encoding the common sub-expression $(\text{RTT} - \text{minRTT}) \times \frac{\text{ack-rate}}{\text{MSS}}$, with depth 5 and up to 11 nodes (Vegas-11). We observe that with Delay-7, the best-scoring handler cannot capture the behavior of this CCA, while the best-scoring handler from Delay-11 starts to capture the triangular pattern. Finally, using Vegas-11 enables the macro, which frees up sketch nodes for other operations. This handler comes closest to matching the input trace's behavior.

In contrast, we show the result for student CCA #3 in Figure 6b. In this case, the best handler uses the Delay-11 DSL, not Vegas-11. This is because the DSL components that are part of the Vegas DSL but not the Delay DSL do not play a part in student CCA #3. This makes the search space bigger, which in turns means that if we timeout a search at any point, we are less likely to have already explored the lowest-distance sketch. So, even though the lowest-distance sketch was in both the space Abagnale explores with Vegas-11 as its input DSL and the space corresponding to Delay-11, by the time these searches timed out, Abagnale with Delay-11 had already evaluated it and saved it, but Abagnale with Vegas-11 had not.

7 Related Work

Program Synthesis. Traditional approaches for program synthesis with examples (PBE) [21, 24, 29, 30] find a program that satisfies all given examples. Although this is the main focus of PBE research, there is some work [34] on handling cases where examples may have noise. In this scenario, prior work also formulated the synthesis problem as an optimization problem. However, they consider discrete data such as string or tabular data where the noise is limited and discrete, leaving the remaining parts intact and uncorrupted. In our case, we produce a trace of outputs for the same inputs observed in the collected trace and compare them to the outputs visible in the original trace. While we use the DTW distance to measure how good our synthesized CCA is, prior work on strings can use simpler methods like the number of failed examples or the edit distance between strings. Moreover, Abagnale uses the distance metric not only to evaluate a candidate handler's merit, but also to guide the search with our bucket prioritization strategy (§4.4).

Smaller DSLs result in a smaller search space and faster performance but finding a small DSL expressive enough to capture the intended behavior is a challenging task. Chan et al. [13] proposed to start with a generic large DSL and use gradient descent to find a sub-DSL that is effective for a specific problem. They train on several benchmarks and reward sub-DSLs that can quickly solve benchmarks and penalize those that fail to solve. Abagnale also has sub-DSLs for each class of CCAs from the Linux Kernel. Given a network trace, Abagnale runs a CCA classifier to map the trace to a known CCA in the Linux Kernel and uses that sub-DSL.

Synthesis of CCAs. Mister880 [23] first proposed using program synthesis to reverse-engineer CCAs. Mister880 makes several simplifying assumptions that make it unsuitable for analyzing real CCAs. For example, it only considers a single simulated packet trace, and cannot cope with trace noise. Additionally, Mister880's simulation relies on an SMT solver for the simulation procedure, and does not scale to real-world traces, which can be hundreds of times larger than Mister880's simulated traces. It also attempts to fully enumerate the search space, which is impractical for all but the simplest CCAs. However, with Abagnale, we do take inspiration from Mister880's event-driven structure and use of distance to evaluate candidate CCAs.

Meanwhile, CCmatic [1] recently proposed program synthesis techniques to produce novel congestion control algorithms that satisfy desired properties. This is fundamentally a different problem than reverse-engineering; while with Abagnale we seek to provide *fidelity* to an extant CCA, Agarwal et al.'s work need only consider a CCA's performance in some specific setting.

8 Conclusion

In this paper we described a system, Abagnale, that combines existing and novel techniques in program synthesis with domain-specific knowledge of CCAs to take a first step towards reverse-engineering the behavior of arbitrary real-world CCAs. This process is today currently fraught with uncertainty and difficulty; most efforts at CCA analysis simply stop with providing trace collection and performance reports. We argue that automated and mechanized

reverse-engineering, such as with Abagnale, should be an important technique in the toolbox of the modern CCA researcher. The results from our synthesis techniques, even when they do not precisely match the ground-truth implementation, reliably give insights into the signals and structure a target CCA uses.

Acknowledgments

This work was partially supported by the Portuguese Foundation for Science and Technology under the Carnegie Mellon Portugal PhD fellowship SFRH/BD/151467/2021 and the projects UIDB/50021/2020 and 2022.03537.PTDC. Additional support was provided by NSF grant CNS-2212390.

Ethics

This work does not raise any ethical issues.

References

[1] Anup Agarwal, Venkat Arun, Devdeep Ray, Ruben Martins, and Srinivasan Seshan. Automating network heuristic design and analysis. In *HotNets*, pages 8–16. ACM, 2022.

[2] Venkat Arun, Mina Tahmasbi Arashloo, Ahmed Saeed, Mohammad Alizadeh, and Hari Balakrishnan. Toward formally verifying congestion control behavior. In *SIGCOMM*, pages 1–16. ACM, 2021.

[3] Rukshani Athapathu, Ranysha Ware, Aditya Abraham Philip, Srinivasan Seshan, and Justine Sherry. Prudentia: Measuring Congestion Control Harm on the Internet. In *SIGCOMM N2Women Workshop*, 2020.

[4] Andrea Baiocchi, Angelo P Castellani, Francesco Vacirca, et al. Yeah-tcp: yet another highspeed tcp. In *Proc. PFLDnet*, volume 7, pages 37–42, 2007.

[5] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. DeepCoder: Learning to Write Programs. In *ICLR (Poster)*, 2017.

[6] Donald J. Berndt and James Clifford. Using dynamic time warping to find patterns in time series. In *KDD Workshop*, pages 359–370. AAAI Press, 1994.

[7] L. Brakmo. TCP-NV: Congestion Avoidance for Data Centers. *Linux Plumbers Conference*, 2010.

[8] Lawrence S. Brakmo, Sean W. O'Malley, and Larry L. Peterson. TCP vegas: New techniques for congestion detection and avoidance. In *SIGCOMM*, pages 24–35. ACM, 1994.

[9] Ricardo Brancas, Miguel Terra-Neves, Miguel Ventura, Vasco M. Manquinho, and Ruben Martins. Towards reliable SQL synthesis: Fuzzing-based evaluation and disambiguation. In *FASE*, volume 14573 of *Lecture Notes in Computer Science*, pages 232–254. Springer, 2024.

[10] Deklin Caban, Devdeep Ray, and Srinivasan Seshan. Understanding congestion control for cloud game streaming. 2020.

[11] Carlo Caini and Rosario Firrincieli. TCP hybla: a TCP enhancement for heterogeneous networks. *Int. J. Satell. Commun. Netw.*, 22(5):547–566, 2004.

[12] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. BBR: congestion-based congestion control. *ACM Queue*, 14(5):20–53, 2016.

[13] Nicolas Chan, Elizabeth Polgreen, and Sanjit A. Seshia. Gradient descent over metagrammars for syntax-guided synthesis. *CoRR*, abs/2007.06677, 2020.

[14] Haoxian Chen, Anduo Wang, and Boon Thau Loo. Towards example-guided network synthesis. In *APNet*, pages 65–71. ACM, 2018.

[15] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021.

[16] Qiaochu Chen, Xinyu Wang, Xi Ye, Greg Durrett, and Isil Dillig. Multi-Modal Synthesis of Regular Expressions. In *PLDI*, 2020.

[17] Dah-Ming Chiu and Raj Jain. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Comput. Networks*, 17:1–14, 1989.

[18] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. In *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[19] Amogh Dhamdhere, David D. Clark, Alexander Gamero-Garrido, Matthew J. Luckie, Ricky K. P. Mok, Gautam Akiwate, Kabir Gogia, Vaibhav Bajpai, Alex C. Snoeren, and kc claffy. Inferring persistent interdomain congestion. In *SIGCOMM*, pages 1–15. ACM, 2018.

[20] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. Program Synthesis Using Conflict-Driven Learning. In *PLDI*, 2018.

[21] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. Component-Based Synthesis of Table Consolidation and Transformation Tasks From Examples. In *PLDI*, 2017.

[22] Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. Component-based synthesis for complex apis. In *ACM SIGPLAN Symposium on Principles of Programming Languages, POPL*, pages 599–612. ACM, 2017.

[23] Margarida Ferreira, Akshay Narayan, Inês Lynce, Ruben Martins, and Justine Sherry. Counterfeiting congestion control algorithms. In *HotNets*, pages 132–139. ACM, 2021.

[24] John K. Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. In *PLDI*, pages 229–239. ACM, 2015.

[25] Sally Floyd. HighSpeed TCP for Large Congestion Windows. https://www.ietf.org/rfc/rfc3649.txt, 2003.

[26] Sadjad Fouladi, John Emmons, Emre Orbay, Catherine Wu, Riad S. Wahby, and Keith Winstein. Salsify: Low-latency network video through tighter integration between a video codec and a transport protocol. In *NSDI*, pages 267–282. USENIX Association, 2018.

[27] Cheng Peng Fu and Soung C. Liew. TCP veno: TCP enhancement for transmission over wireless access networks. volume 21, pages 216–228, 2003.

[28] Sishuai Gong, Usama Naseer, and Theophilus Benson. Inspector gadget: A framework for inferring TCP congestion control algorithms and protocol configurations. In *TMA*. IFIP, 2020.

[29] Sumit Gulwani. Programming by examples - and its applications in data wrangling. In Javier Esparza, Orna Grumberg, and Salomon Sickert, editors, *Dependable Software Systems Engineering*, volume 45, pages 137–158. IOS Press, 2016.

[30] Sumit Gulwani, William R. Harris, and Rishabh Singh. Spreadsheet data manipulation using examples. *Commun. ACM*, 55(8):97–105, 2012.

[31] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Program synthesis. *Foundations and Trends in Programming Languages*, 4, 2017.

[32] Zheng Guo, David Cao, Davin Tjong, Jean Yang, Cole Schlesinger, and Nadia Polikarpova. Type-directed program synthesis for restful apis. In *ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*, pages 122–136. ACM, 2022.

[33] Sangtae Ha, Injong Rhee, and Lisong Xu. CUBIC: a new tcp-friendly high-speed TCP variant. *ACM SIGOPS Oper. Syst. Rev.*, 42(5):64–74, 2008.

[34] Shivam Handa and Martin C. Rinard. Inductive program synthesis over noisy data. In *ESEC/SIGSOFT FSE*, pages 87–98. ACM, 2020.

[35] David A. Hayes and Grenville J. Armitage. Revisiting TCP congestion control using delay gradients. In *Networking (2)*, volume 6641 of *Lecture Notes in Computer Science*, pages 328–341. Springer, 2011.

[36] Janey C. Hoe. Improving the start-up behavior of a congestion control scheme for TCP. In *SIGCOMM*, pages 270–280. ACM, 1996.

[37] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-Guided Component-Based Program Synthesis. In *ICSE*, 2010.

[38] Ashwin Kalyan, Abhishek Mohta, Oleksandr Polozov, Dhruv Batra, Prateek Jain, and Sumit Gulwani. Neural-Guided Deductive Search for Real-Time Program Synthesis from Examples. In *ICLR (Poster)*, 2018.

[39] Tom Kelly. Scalable TCP: improving performance in highspeed wide area networks. *Comput. Commun. Rev.*, 33(2):83–91, 2003.

[40] Linux Kernel. Tcp bbr implementation. https://elixir.bootlin.com/linux/v4.14/source/net/ipv4/tcp_bbr.c.

[41] Aleksandar Kuzmanovic and Edward W. Knightly. TCP-LP: low-priority service via end-point congestion control. *IEEE/ACM Trans. Netw.*, 14(4):739–752, 2006.

[42] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan R. Iyengar, Jeff Bailey, Jeremy Dorfman, Jim Roskind, Joanna Kulik, Patrik Westin, Raman Tenneti, Robbie Shade, Ryan Hamilton, Victor Vasiliev, Wan-Teh Chang, and Zhongyi Shi. The QUIC transport protocol: Design and internet-scale deployment. In *SIGCOMM*, pages 183–196. ACM, 2017.

[43] Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. Accelerating Search-Based Program Synthesis Using Learned Probabilistic Models. In *PLDI*, 2018.

[44] DJ Leith and R. Shorten. H-TCP Protocol for High-Speed Long Distance Networks. In *PFLDNet*, 2004.

[45] Shao Liu, Tamer Basar, and R. Srikant. Tcp-illinois: A loss- and delay-based congestion control algorithm for high-speed networks. *Perform. Evaluation*, 65(6-7):417–440, 2008.

[46] Xavier Marchal, Philippe Graff, Joël Roman Ky, Thibault Cholez, Stéphane Tuffin, Bertrand Mathieu, and Olivier Festor. An analysis of cloud gaming platforms behaviour under synthetic network constraints and real cellular networks conditions. *J. Netw. Syst. Manag.*, 31(2):39, 2023.

[47] Ruben Martins, Jia Chen, Yanju Chen, Yu Feng, and Isil Dillig. Trinity: An extensible synthesis framework for data science. *Proc. VLDB Endow.*, 12(12):1914–1917, 2019.

[48] S. Mascolo, C. Casetti, M. Gerla, M.Y. Sanadidi, and R. Wang. TCP Westwood: Bandwidth Estimation for Enhanced Transport over Wireless Links. In *MobiCom*, 2001.

[49] Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondrej Certík, Sergey B. Kirpichev, Matthew Rocklin, Amit Kumar, Sergiu Ivanov, Jason Keith Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Stepán Roucka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony M. Scopatz. Sympy: symbolic computing in python. *PeerJ Comput. Sci.*, 3:e103, 2017.

[50] Ayush Mishra, Xiangpeng Sun, Atishya Jain, Sameer Pande, Raj Joshi, and Ben Leong. The great internet TCP congestion control census. In *SIGMETRICS (Abstracts)*, pages 59–60. ACM, 2020.

[51] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *OSDI*, pages 561–577. USENIX Association, 2018.

[52] Akshay Narayan, Frank Cangialosi, Deepti Raghavan, Prateesh Goyal, Srinivas Narayana, Radhika Mittal, Mohammad Alizadeh, and Hari Balakrishnan. Restructuring endpoint congestion control. In *SIGCOMM*, pages 30–43. ACM, 2018.

[53] Vikram Nathan, Vibhaalakshmi Sivaraman, Ravichandra Addanki, Mehrdad Khani Shirkoohi, Prateesh Goyal, and Mohammad Alizadeh. End-to-end transport for video qoe fairness. In *SIGCOMM*, pages 408–423. ACM, 2019.

[54] Jitendra Padhye and Sally Floyd. On inferring TCP behavior. In *SIGCOMM*, pages 287–298. ACM, 2001.

[55] Adithya Abraham Philip, Rukshani Athapathu, Ranysha Ware, Fabian Francis Mkocheko, Alexis Schlomer, Mengrou Shou, Zili Meng, Srinivasan Seshan, and Justine Sherry. Prudentia: Findings of an internet fairness watchdog. In *SIGCOMM*, pages 506–520. ACM, 2024.

[56] Devdeep Ray. *Integrating Video Codec Design and Network Transport for Emerging Internet Video Streaming Applications*. PhD thesis, Carnegie Mellon University, 2022.

[57] Devdeep Ray and Srinivasan Seshan. Cc-fuzz: genetic algorithm-based fuzzing for stress testing congestion control algorithms. In *HotNets*, pages 31–37. ACM, 2022.

[58] Jan Rüth, Ike Kunze, and Oliver Hohlfeld. An empirical view on content provider fairness. In *TMA*, pages 177–184. IEEE, 2019.

[59] Constantin Sander, Jan Rüth, Oliver Hohlfeld, and Klaus Wehrle. Deepcci: Deep learning-based passive congestion control identification. In *NetAI@SIGCOMM*, pages 37–43. ACM, 2019.

[60] Lei Shi, Yahui Li, Boon Thau Loo, and Rajeev Alur. Network traffic classification by program synthesis. In *TACAS*, volume 12651 of *Lecture Notes in Computer Science*, pages 430–448. Springer, 2021.

[61] Armando Solar-Lezama, Rodric M. Rabbah, Rastislav Bodík, and Kemal Ebcioglu. Programming by sketching for bit-streaming programs. In Vivek Sarkar and Mary W. Hall, editors, *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 281–294. ACM, 2005.

[62] Ranysha Ware, Matthew K. Mukerjee, Srinivasan Seshan, and Justine Sherry. Modeling bbr's interactions with loss-based congestion control. In *Internet Measurement Conference*, pages 137–143. ACM, 2019.

[63] Ranysha Ware, Adithya Abraham Philip, Nicholas Hungria, Yash Kothari, Justine Sherry, and Srinivasan Seshan. Ccanalyzer: An efficient and nearly-passive congestion control classifier. In *SIGCOMM*, pages 181–196. ACM, 2024.

[64] D.X. Wei, C. Jin, S.H. Low, and S. Hegde. FAST TCP: Motivation, Architecture, Algorithms, Performance. *IEEE/ACM Trans. on Networking*, 14(6):1246–1259, 2006.

[65] Keith Winstein and Hari Balakrishnan. TCP ex machina: computer-generated congestion control. In *SIGCOMM*, pages 123–134. ACM, 2013.

[66] Lisong Xu, Khaled Harfoush, and Injong Rhee. Binary increase congestion control (BIC) for fast long-distance networks. In *INFOCOM*, pages 2514–2524. IEEE, 2004.

[67] Yifei Yuan, Dong Lin, Rajeev Alur, and Boon Thau Loo. Scenario-based programming for SDN policies. In *CoNEXT*, pages 34:1–34:13. ACM, 2015.