Counterfeiting Congestion Control Algorithms

Margarida Ferreira[†,⊥], Akshay Narayan[⋆], Inês Lynce[⊥], Ruben Martins[†], Justine Sherry[†]

[†] *Carnegie Mellon University,* [⊥] *INESC-ID/IST Universidade de Lisboa,* [⋆] *MIT CSAIL*

Abstract

Congestion Control Algorithms (CCAs) impact numerous desirable Internet properties such as performance, stability, and fairness. Hence, the networking community invests substantial effort into studying whether new algorithms are safe for wide-scale deployment. However, operators today are continuously innovating and some deployed CCAs are unpublished – either because the CCA is in beta or because it is considered proprietary. How can the networking community evaluate these new CCAs when their inner workings are unknown?

In this paper, we propose 'counterfeit congestion control algorithms' – reverse-engineered implementations derived using program synthesis based on observations of the original implementation. Using the counterfeit (synthesized) CCA implementation, researchers can then evaluate the CCA using controlled empirical testbeds or mathematical analysis, even without access to the original implementation. Our initial prototype, 'Mister 880,' can synthesize several basic CCAs including a simplified Reno using only a few traces.

ACM Reference Format:

Margarida Ferreira, Akshay Narayan, Inês Lynce, Ruben Martins, Justine Sherry. 2021. Counterfeiting Congestion Control Algorithms. In *The Twentieth ACM Workshop on Hot Topics in Networks (HotNets '21), November 10–12, 2021, Virtual Event, United Kingdom.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3484266.3487381

1 Introduction

Today's Internet hosts an expanding corpus of Congestion Control Algorithms (CCAs) including Reno at Netflix [51], Copa at Facebook [4], and BBR at Google [10]. CCAs determine important properties such as whether or not competing applications share network bandwidth fairly [13, 26, 55]; how stable bandwidth allocations are (or whether performance oscillates) [1]; how heavily occupied network buffers are and hence what latency applications can expect [36]; and whether or not network links are utilized efficiently [25]. These properties and others are a topic of active study for researchers [1, 3, 6, 32, 37, 52, 60].

HotNets '21, November 10–12, 2021, Virtual Event, United Kingdom

ACM ISBN 978-1-4503-9087-3/21/11.
https://doi.org/10.1145/3484266.3487381

However, understanding properties like fairness, utilization, and stability on the Internet is increasingly challenging because service providers do not always publish the details of novel CCAs they deploy. Sometimes CCAs are in a state of 'beta' deployment and unready for public scrutiny (as is currently the case with Google's BBRv2 [9]), and other CCAs are considered proprietary (as are the details of Akamai's FastTCP [57][1]). As user-space implementations of CCAs become more prevalent [38, 43], we expect that unpublished, experimental CCAs will gain more popularity.

Lack of scrutiny into unpublished algorithms is not merely an academic disappointment; a buggy CCA can have damaging performance implications not only for a service provider deploying the new, buggy CCA X, but also for competing services using legacy CCAs Y and Z. If X exhibits unfairness to flows using CCA Y, then services using Y who share a bottleneck link with service using X will suffer; if X exhibits highly oscillatory behavior, other services using CCAs Y or Z and sharing the same bottleneck link can expect to see performance instability as well. Hence, research into CCA properties [3, 35, 56, 60] safeguards Internet performance for *all* services, even those services which deploy legacy, well-studied CCAs.

This leads us to our central question in this paper: *How can the Internet community evaluate deployed CCAs for fairness, utilization, stability, and other properties when the CCA details have not been made public?*

One pragmatic closed-source approach is to use active measurements and to initiate controlled downloads to public services and empirically observe the relevant properties (fairness, stability, etc); some researchers have already taken this tack [5, 45]. As we will discuss in §2, this is a good start, but unfortunately closed source approaches have fundamental limitations compared with analytical/mathematical open source studies of a known algorithm. For example, they cannot provide upper and lower bounds on performance nor can they provide insights into why a CCA behaves in a certain way.

Given that the original implementation remains out of reach, we propose to use active measurements of the real CCA to reverse-engineer the underlying algorithm. We refer to this reverse-engineered CCA as a 'counterfeit congestion control algorithm' or cCCA. Using the cCCA, researchers can then perform mathematical modeling, explore modifications to the

[1] Although the early basis of the FastTCP algorithm has been published [57], its current incarnation remains private.

algorithm, or empirically test the cCCA in diverse, controlled network testbeds. To generate cCCAs, we propose using *program synthesis* [30], a technique from the programming languages community. Program synthesis allows a user to provide a set of expected input/output pairs to a *synthesizer*; the synthesizer then generates a program which, given a particular input, will produce the corresponding output.

Despite the considerable literature on program synthesis [2, 7, 22, 24, 29, 33, 50], reverse engineering congestion control algorithms remains out of reach for existing technology. We discuss numerous challenges in §4, but in this vision paper we focus on the most fundamental obstacle: existing program synthesizers typically target stateless programs because exploring the space of possible stateful programs is computationally much more demanding than exploring only stateless programs. Typical synthesis-by-example takes a set of input-output pairs and finds a program such that the output is the result of applying that program to its respective input. When synthesizing a stateful program, we are given an initial state and a final state, and we must find a series of updates that will update the state to achieve the final state. This is a much harder problem, as it involves asking the synthesizer to consider a large number of *unknown variables* representing the state of the system at each timestep of the input.

Our prototype synthesis tool, Mister880,[2] uses domain-specific knowledge about congestion control to guide the search for an implementation to produce a cCCA more quickly. For a simplified version of Reno, Mister880 can reverse-engineer the correct algorithm in only 13 minutes on a commodity laptop. Despite many remaining open questions (§4), our experience with Mister880 leaves us optimistic that we will be able to generate cCCAs which capture the behavior of truly unknown algorithms in the future.

2 Motivation and Approach

Congestion control properties such as fairness, utilization, and stability all impact the performance of Internet applications; historically, there have been few barriers to their analysis due to an 'open source' ecosystem around CCA algorithms. Many of the fundamental results about Internet performance have relied on open source analysis, *e.g.*:

- TCP-Friendly Rate Control guarantees that a streaming service will be fair to co-existing Reno flows; the authors derive their guarantees via mathematical analysis of the Reno algorithm [26].
- Controlled testbed experiments [21, 47, 54] and mathematical modeling [56] using Google's open sourced BBR code showed that BBR can be unfair to competing flows using any loss-based congestion control algorithm; insights from the research community continue to inform Google's ongoing design of BBR'v2' [9].
- PCC Vivace [21] was designed to have fairer interactions with deployed CCAs after analysis of the original PCC [20] objective function.
- Finally, researchers can prove properties using mathematical models of CCAs [3, 60, 62]: e.g., whether it fully utilizes available bandwidth.

It is worth noting that all of the above discoveries relied on the research community at large having access to the source code, allowing teams *other* than a CCA's own developers to provide insights about the algorithm. Indeed, the research community can even serve as a 'watchdog' to ensure that deployed algorithms are indeed performing as their creators claim. Thus, our goal in this paper is to enable open source analysis like the studies we list above for closed source CCAs.

2.1 Analyzing Closed-Source CCAs

Prior methods of analyzing closed-source CCAs remain more limited than open-source analysis and experimentation.

Classification. Researchers have proposed tools based on both machine learning and heuristics to determine from empirical observations which CCA a flow is using [28, 41, 44, 46], or a less granular classification of the flow's type of CCA [27, 53, 61]. Unfortunately, these classifiers merely *identify* CCAs – they can label a particular server as using BBR, or identify that two servers are using the *same* CCA, but they cannot tell researchers anything about the properties of a previously unseen CCA. Classification is nevertheless useful in helping us identify servers which are running unknown CCAs, as these CCAs are the target of our study.

Empirical Studies. Researchers have instrumented network links to observe utilization and fairness "in the wild" [19] or initiated controlled downloads to Internet servers and measured, e.g., the observed throughput of these connections from the edge [5, 41, 45]. Empirical approaches require no knowledge of what CCA is running at the server nor how it works, but by introducing background loss, inflating latencies, or initiating additional connections through the same bottleneck link, researchers can study the unknown CCA under a range of conditions. However, empirical studies cannot *prove* properties about CCAs, nor can they identify performance bounds and edge cases (e.g, under what loss rate, buffer capacity, and RTT is this CCA's throughput expected to be lowest?). Further, empirical approaches are limited by their vantage points: if the vantage point is 100ms away from the server, it cannot test CCA's behavior in lower latency settings.

3 A Proof-of-Concept: Mister880

In the absence of the algorithm itself to study, we propose to *reverse engineer* deployed closed source CCAs using empirical observations of the true CCA. Similarly to empirical studies,

[2] As an initial prototype, Mister880 is quite limited in its capabilities – for example, it can synthesize Reno, but not Tahoe or Cubic. Hence, we name it for a similarly limited counterfeiter who only ever counterfeited $1 bills [40].

we can observe unknown CCAs, but instead of measuring throughput or fairness directly we can instead measure the inputs a CCA uses to make decisions and its resulting outputs: the number of inflight packets ("visible window"), rate of packets injected into the network, acknowledgments returned to the server, and packet RTT. We call this a *network trace*. Using this information, we can apply *program synthesis* [30] to generate a program which, given the observed inputs, will generate the observed outputs. We refer to this generated algorithm as a counterfeit CCA (cCCA).

Researchers can then study the cCCA like any other open-source algorithm (*e.g.* with mathematical models or controlled testbed experiments). As we will discuss further in §4, although the cCCA is not guaranteed to be identical to the true algorithm, we believe that generating an algorithm that is similar will still catalyze new lines of study and research.

To explore the feasibility of finding cCCAs, we design and test a simplified CCA synthesizer called Mister880. Mister880 is an early prototype: it operates over traces generated in simulation where we can perfectly observe packet arrivals/transmissions in a deterministic setting. Our goals for Mister880 are very limited: to synthesize a version of TCP Reno in our simulation environment. We leave more complex algorithms and real network scenarios (including packet loss and non-determinism) to future work (§4).

3.1 Program Synthesis Basics

Programming-by-example (PBE) [2, 17] is a sub-field of program synthesis where the user provides input-output examples as the behavioral specification of the program to be synthesized. A program synthesizer takes as input (1) a set of input-output examples and (2) a Domain-Specific Language (DSL) describing the space of possible programs (in our case, a language describing the arithmetic operations that make up a CCA). Synthesizers use DSLs to guide the analysis towards programs which are likely to be useful for the given task. The synthesizer then returns a program written in the provided DSL that will produce the correct output for the corresponding input in the examples. PBE is a common approach to solve many practical problems such as data structure transformations [24], spreadsheet data manipulation [29], data preparation tasks [23], as well as applications to computer networks [11, 48, 59].

There are multiple approaches to explore the space of feasible programs, but the predominant approaches use constraint solving [33, 50], machine learning [7, 34, 39], or a combination of both [12, 22]. In our experience, constraint-based approaches tend to outperform machine learning approaches when (a) the input/output specification can be encoded using logic, and (b) the space of satisfying programs is highly constrained, *i.e.* much smaller than the space of possible programs. Since both of the above hold for CCA synthesis, we take a constraint-based approach.

3.2 Challenges and Key Ideas

Constraint solvers rely on a translation ("*encoding*") of a problem into logical formulae. The algorithms that solvers run over these formulae can have exponential [15] or even undecidable [8] complexity, so it is crucial to limit the encoding's size to avoid passing an intractable query to the solver. Unfortunately, in the domain of CCAs, the encoding grows with the size of the trace. There are, of course, more inputs and outputs to represent ('known variables'), but most costly is the need to encode the unknown state at every timestep, creating many 'unknown variables' for the synthesizer to reason about.[3]

Thus, our key challenge in designing Mister880 is to encode a broad range of candidate cCCAs while using a minimal encoding to increase the likelihood that the synthesizer will output a cCCA before we timeout the search (we typically set a limit of four hours). Mister880 thus applies two key ideas:

(1) Event-Driven Structure. CCA implementations have a regular structure: all deployed frameworks have an event-driven structure with handlers for events such as timeouts and incoming acknowledgments, and further use a standard set of congestion signals [16, 43]. Even a hypothetical non-event-driven CCA implementation which made state changes at any time would only have its output (*i.e.*, a congestion window or pacing rate) manifest when the network stack sent the next packet, and by that time an event-driven implementation could make the same update. Therefore, instead of synthesizing a CCA as one big program, we decompose a CCA as a set of event handlers and synthesize them independently, leading to smaller and more tractable queries to the constraint solver.

(2) Arithmetic Pruning. As we will discuss in §3.3, our DSL can generate arbitrary arithmetic functions which update the cCCAs congestion window or CWND. However, many such functions can be pruned easily. With Mister880, we encode a few *CCA prerequisites*, or properties we know *must hold* for a cCCA to be a viable match for the true CCA.

One such prerequisite is unit agreement. Since the congestion window has units *bytes*, we only allow event handlers whose output is in *bytes*. For example, $CWND * AKD$ is $bytes^2$ and thus invalid. We also know that CCAs both *increase* and *decrease* the CWND, and hence an ACK handler which only *decreases* the window size is an invalid candidate algorithm. We enforce these two prerequisites with Mister880, and in future work expect to include more as we tackle more complex cCCAs.

[3] While prior work (*e.g.* in synthesizing SDN policies [59]) also considered state, synthesizing a CCA requires reasoning about its internal state iteratively through the trace, unlike previously considered domains.

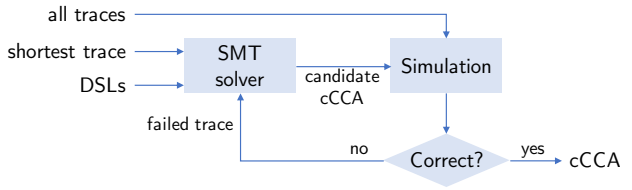

Figure 1: Mister880 iteratively generates candidate cCCAs with the SMT solver and checks them against the corpus of traces.

3.3 Design Sketch

From §3.1, we know that we need to pass a DSL and a trace encoding to a constraint solver such as an SMT solver. We discuss how to build these two components.

Domain-Specific Language. Recall that to improve tractability, we design our cCCAs as a set of independent *event handlers*. Mister880 supports two event handlers: (i) *win-ack* handler, used when the trace shows an ACK, and (ii) *win-timeout* handler, used when the trace shows a timeout. The event handlers update the congestion window (CWND) given the sender's state (the previous window size *CWND* and the initial window w_0) and a set of *input values* (the number of acknowledged bytes at the current timestep *AKD* and the maximum segment size *MSS*). We plan to extend this in the future to (a) include more handlers, *e.g.* for triple dup-acks or more general timers, and (b) to provide a richer set of congestion signals as input values (*e.g.* average ACK arrival rate [10]; RTT gradient [42]).

The event handlers operate over these inputs using DSLs of arithmetic integer operators. The operands can be either the functions' input values (*CWND*, *AKD*, *MSS*, w_0) or arbitrary integer constants (*const*). Equations 1a and 1b show Mister880's DSLs for *win-ack* and *win-timeout*, respectively:

$$\begin{aligned} Int \rightarrow\ & CWND \mid MSS \mid AKD \\ & \mid const \mid Int + Int \\ & \mid Int \times Int \mid Int / Int \end{aligned} \tag{1a}$$

$$\begin{aligned} Int \rightarrow\ & CWND \mid w_0 \mid const \\ & \mid Int / Int \mid max(Int, Int) \end{aligned} \tag{1b}$$

Equation 2 shows a simple CCA (hereafter referred to as Simple Exponential A or SE-A for short) that can be built from these DSLs:

$$win\text{-}ack(CWND, AKD, MSS) = CWND + AKD, \tag{2a}$$

$$win\text{-}timeout(CWND, w_0) = w_0. \tag{2b}$$

Constraint-Based Search. We represent the search space of all event handlers as an abstract syntax tree — deeper trees represent longer expressions. Even our simple DSL has a large search space, and naively enumerating its abstract syntax tree is infeasible. Following Occam's razor ('the simplest solution is often the best one'), Mister880 considers simpler event handler expressions before more complex ones; if there is a *win-ack* handler with a depth-3 expression tree that satisfies the trace, we will not consider depth-4 (or greater) trees.

Partitioning the search into smaller searches for individual handlers rather than one big program improves performance. For example, just encoding Reno's *win-ack* handler (Equation 1a) requires exploring the tree to depth 4, which encompasses 20,000 possible functions. If we further consider all possible *win-ack* handlers in combination with all *win-timeout* handlers, there are several hundred million possible cCCAs.

To limit the number of combinations to consider, we can check the *win-ack* function independently of the *win-timeout* function. In the initial portion of the input trace, we know no loss-timeout has occurred yet; until this first timeout we can thus consider only the *win-ack* function. If at some point before the first timeout the *win-ack* function produces a visible window not compatible with the trace, we know that it will never fit the whole trace (regardless of *win-timeout*) and thus we can discard that *win-ack* function without ever considering *win-timeout*. So, in practice, we can split the synthesis into two parts, which reduces the search space combinatorially: first, we find a *win-ack* that is consistent with the start of the trace. This uses a smaller encoding, since the initial portion of the trace is smaller and there is only one event handler. After we find a *win-ack* function that passes this check, we move on to a *win-timeout* using the rest of the trace.

To additionally simplify our search, we prune the set of possible handlers by enforcing *arithmetic prerequisites* as discussed in §3.2. We tell the solver not to consider functions which, *e.g.*, would always result in a decreasing CWND for a *win-ack*. As we will show in §3.4, arithmetic pruning allows us to quickly discard non-viable solutions and subtrees – synthesizing Reno does not complete with a four hour timeout without this aspect of our design.

Putting it together. We collect dozens of traces at varying RTTs and loss rates for each true CCA. However, encoding all traces to input into the SMT solver results in a formula that is too complex to solve efficiently. Instead, we split the synthesis into the two stages depicted in Figure 1: *SMT solving* and *simulation*. The SMT solver takes as initial input *only one* encoded trace (the shortest one) and the DSL above. The solver will then return a candidate cCCA whose execution satisfies just one trace.

This 'candidate' cCCA may satisfy all of the remaining traces – or it may satisfy just the shortest trace, but not all of the others. Consider Equation 3, which shows another simple CCA, SE-B:

$$win\text{-}ack(CWND, AKD, MSS) = CWND + AKD, \tag{3a}$$

$$win\text{-}timeout(CWND, w_0) = CWND/2. \tag{3b}$$

Suppose Mister880 is synthesizing SE-B from 2 traces: trace *a*, with duration 200ms, and trace *b*, with duration 400ms. However, by just observing the first trace – of 200ms

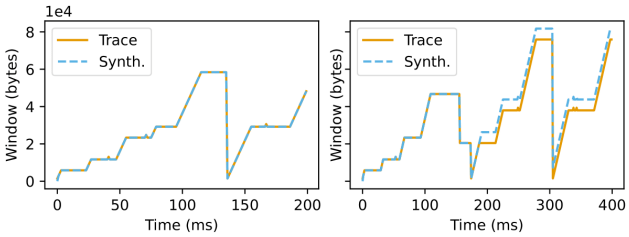

Figure 2: The dashed line shows the visible window produced by the candidate cCCA (*win-ack* : *CWND* + *AKD*; *win-timeout* = w_0), compared to the trace's CCA (*win-ack* : *CWND* + *AKD*; *win-timeout* : *CWND*/2) shown by the solid line, for two traces with durations 200ms on the left and 400ms on the right.

– the SMT solver might produce SE-A instead of SE-B as a candidate cCCA. Figure 2 shows how using only one trace under-specifies the CCA: SE-A produces the same visible window as SE-B, and hence the SMT solver cannot tell that SE-A is the wrong solution.

Rather than feeding all traces into the SMT solver – which would explode the search space – we instead test each candidate cCCA in simulation, which is only a linear-time test. For each trace, we run the candidate cCCA on the inputs for the trace and verify that the candidate cCCA produces the expected outputs. In Figure 2, we would see that SE-A would provide incorrect outputs with the second trace of 400ms (on the right). If the candidate cCCA produces the wrong output, we end simulation and add *just the discordant trace* to the encoded SMT input. We then ask the SMT solver for a new candidate cCCA and repeat the process until the SMT solver provides a cCCA which satisfies all of the remaining traces in simulation. Of the four algorithms we tested in §3.4, two required multiple traces to find a satisfying cCCA.

3.4 Testing our Synthesis Approach

We implemented Mister880 on Python 3.9, using Z3 (version 4.8.10) [18] to encode and solve all SMT formulas. We ran Mister880 on a laptop with a dual-core Intel Core i5 (2.9 GHz) with 8GB of RAM, running MacOS Big Sur 11.4. We tested our synthesizer for 4 CCAs supported by our DSLs: the previously introduced SE-A and SE-B (Equations 2 and 3, respectively), SE-C, described in Equation 4:

$$\textit{win-ack}(\textit{CWND},\textit{AKD},\textit{MSS}) = \textit{CWND} + 2\textit{AKD}, \quad (4a)$$

$$\textit{win-timeout}(\textit{CWND}, w_0) = \textit{max}(1, \textit{CWND}/8). \quad (4b)$$

and a simplified version of Reno shown in Equation 5:

$$\textit{win-ack}(\textit{CWND},\textit{AKD},\textit{MSS}) = \textit{CWND} + \textit{AKD} * \textit{MSS}/\textit{CWND} \quad (5a)$$

$$\textit{win-timeout}(\textit{CWND}, w_0) = w_0. \quad (5b)$$

We generated 16 simulator traces for each true CCA with durations ranging from 200 to 1000ms, RTTs between 10 and 100ms, and loss rates at 1 and 2%. Table 1 shows that it took

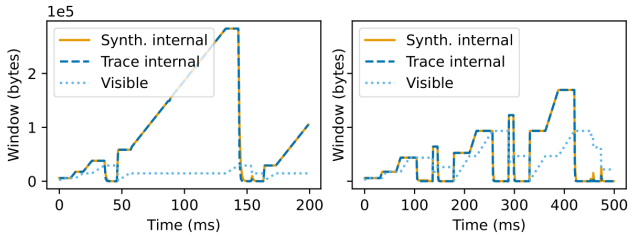

Figure 3: The solid line shows the internal window sizes produced by the cCCA (*win-ack* : *CWND* + 2*AKD*; *win-timeout* : *CWND*/3) compared to the trace's, dashed (*win-ack* : *CWND* + 2*AKD*; *win-timeout* : *max*(1, *CWND*/8)) for 2 traces, with 200ms duration on the left and 500ms on the right. The dotted line shows the visible window, which is identical for both CCAs.

CCA	Synthesis time (s)
SE-A	0.94
SE-B	64.28
SE-C	83.13
Simplified Reno	782.94

Table 1: Synthesis times for each tested CCA. SE-C is shaded because the synthesized cCCA's *win-timeout* handler differs from the ground truth.

from less than one second for SE-A to several minutes for Reno to synthesize cCCAs the CCAs we considered.

SE-A. SE-A is the fastest because its event handler implementations are among the first few functions the solver considers: *CWND* + *AKD* is the third *win-ack* function, and w_0 is the second *win-timeout*. In this case, the SMT solver produces the correct solution with the shortest trace, so the synthesis cycle in Figure 1 executes only once.

SE-B. The synthesis of SE-B takes slightly over a minute. As Figure 2 illustrated, the shortest trace (trace *a*) under-specifies SE-B, so Mister880 needs to encode a second trace into the SMT formula. The synthesizer produces the initial candidate cCCA in 1 second, but takes another 1 minute to get a cCCA (the correct one) for the encoding with 2 traces.

SE-C. The synthesis of SE-C takes over a minute as well. Because *CWND* + 2*AKD* has more DSL components than the previous *win-ack* handlers, a few more functions need to be considered before arriving at the correct one. To find a CCA compatible with all traces for SE-C, Mister880 must encode 3 traces, with durations of 200, 400 and 500ms.

Surprisingly, the resulting synthesized *win-ack* is the correct one, but *win-timeout* is incorrect: *CWND*/3, instead of *win-timeout* (*CWND*, w_0) = *max*(1, *CWND*/ 8). Digging deeper, Figure 3 reveals the slight difference in the *internal* window size when executing the traces. They are the same for all but a few timesteps right after a timeout, where the trace CCA's window decreases faster. However, this difference in

the internal window size does not affect the *visible* window size; the correct bytes are still sent in the correct timesteps.

Simplified Reno. Simplified Reno's *win-ack* handler has more operators than the previous CCAs. Because Mister880 considers event handlers in increasing order of number of DSL components, it takes longer to arrive at the correct *win-ack* handler. Even though a single trace is sufficient to arrive at the correct event handlers, it takes 13 minutes to synthesize.

Due to the larger depth of Simplified Reno's *win-ack* handler, the advantage of arithmetic pruning techniques becomes apparent. If we leave out the SMT constraints enforcing the non-increasing property for *win-ack* handlers, the synthesis time doubles. If we remove the unit agreement constraints (that ensure the handlers' output is in *bytes*) Mister880 is no longer able to find a cCCA for Simplified Reno – the synthesis times out after 4 hours.

4 Conclusion and Future Work

As deploying new CCAs becomes easier, it will be more common to encounter unknown CCAs in the Internet. To make analyzing these CCAs possible, we propose *counterfeiting*, or reverse-engineering, them using program synthesis. Our initial prototype, Mister880, shows that reverse engineering is indeed feasible for an initial limited set of simple CCAs. While this leaves us optimistic that reverse engineering richer, truly unknown CCAs is possible, there remain numerous technical challenges to resolve first. We thus conclude with some open questions to the community.

Noisy Network Traces. While our simulator provides ground truth traces, in a real network any tap or vantage point will incur measurement noise. For example, the network could drop a packet the true CCA sees before it reaches our vantage point (or, conversely, it could drop an ACK our vantage point observes before it reaches the CCA), or ACK compression could obscure the inter-packet timings the CCA used.

Mister880's looks for an exact match between the true CCA's inputs/outputs and the cCCA's, which is impossible to find with noisy traces. Inspired by ongoing work in noisy data and program synthesis [31], we propose that instead of asking for an exact match, we can ask the SMT solver to maximize an objective function measuring how closely a cCCA matches a given trace. For instance, we can consider the number of time steps where cCCA produces the same output as observed in the trace. This turns generating a cCCA from a decision problem into an optimization problem.

Solving an SMT optimization problem is more computationally challenging than solving a decision problem: a single optimization problem can involve multiple solver calls. We believe our system design can address this additional scalability challenge because of our decomposition of event handlers: we can separately enumerate event handlers that satisfy a given similarity threshold with the trace before considering the following event handler. Similarly, the simulation validation step could return a score indicating how close the cCCA is to the trace rather than a boolean success value.

More complex CCAs. Our proof-of-concept DSL only encodes the minimum number of operators needed to express Simplified Reno; more complex CCAs require more operators. For instance, slow-start requires conditionals, Cubic requires exponentiation, and BBR requires pacing rate enforcement. Extending our DSL to support these features will be straightforward; we simply need to support additional event handlers, state variables, and congestion signals. Importantly, while the DSL will need *some* additional operators, the set of operators is finite since CCAs have regular structure [14, 43].

However, some CCA proposals may break some of the assumptions in Mister880. For example, rate-based algorithms are not technically CWND based, and hence break out of Mister880's window-based model. As we saw in §3.3, different CCAs can still produce the same external behavior. Synthesis may thus help uncover properties about known algorithms; *e.g.* recent results have showed that BBRv1 (which was initially promoted as rate-based) typically operates in a windowed fashion [56], so a window-based cCCA would capture its behavior.

More challenging are machine learning based CCAs [21, 49]. It is unlikely that our DSL can ever incorporate enough expressivity to synthesize CCAs which use, e.g., deep reinforcement learning (although it is feasible to express *some* learning-based approaches, such as the decision trees used in Remy [58]). Here, we do not hope to generate an exact-match cCCA to correspond to the true CCA, but we do curiously ask: *is it possible to find cCCAs that use simpler functions and yet have similar properties to complex CCAs?*

Furthermore, it would be a surprising (but interesting) result if we are able to generate new, simple algorithms with Mister880's successor which exhibit similar behavior to more complex, ML-based approaches. Here, we expect that our solution for noise will help us: rather than attempt to search for a cCCA that exactly replicates the true ML-based CCA's behavior, we will search for the most similar cCCA we can generate with our DSL. Thus, we end with this thought: perhaps the most valuable lessons from reverse-engineering CCAs will lie not in the algorithms we counterfeit identically, but in those we counterfeit imperfectly, but more simply.

Acknowledgments

This work was supported by NSF Awards No. CCF-1762363 and CNS-1850384, a VMware Systems Research Award, a CMU Portugal Dual Degree PhD scholarship, project ANI 045917 funded by FEDER and FCT, and project UIDB/50021/2020 funded by FCT. We also thank Carlos Xavier for sleeping well the night of the paper deadline.

References

[1] A. Akella, S. Seshan, R. Karp, S. Shenker, and C. Papadimitriou. 2002. Selfish Behavior and Stability of the Internet: A Game-Theoretic Analysis of TCP. In *SIGCOMM*. 1

[2] Rajeev Alur, Rishabh Singh, Dana Fisman, and Armando Solar-Lezama. 2018. Search-Based Program Synthesis. *CACM* 61, 12 (2018). 1, 3.1

[3] Venkat Arun, Mina Arashloo, Ahmed Saeed, Mohammad Alizadeh, and Hari Balakrishnan. 2021. Formally Verifying Congestion Control Performance. In *SIGCOMM*. 1, 2

[4] Venkat Arun and Hari Balakrishnan. 2018. Copa: Congestion Control Combining Objective Optimization with Window Adjustments. In *NSDI*. 1

[5] Rukshani Athapathu, Ranysha Ware, Aditya Abraham Philip, Srinivasan Seshan, and Justine Sherry. 2020. Prudentia: Measuring Congestion Control Harm on the Internet. In *SIGCOMM N2Women Workshop*. 1, 2.1

[6] Hamsa Balakrishnan, Nandita Dukkipati, Nick McKeown, and Claire J Tomlin. 2007. Stability Analysis of Explicit Congestion Control Protocols. *IEEE Communications Letters* 11, 10 (2007). 1

[7] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2017. DeepCoder: Learning to Write Programs. In *ICLR (Poster)*. 1, 3.1

[8] Nikolaj Bjørner, Nikolai Tillmann, and Andrei Voronkov. 2009. Path Feasibility Analysis for String-Manipulating Programs. In *TACAS*. 3.2

[9] Neal Cardwell. 2020. BBR Update. https://datatracker.ietf.org/meeting/109/materials/slides-109-iccrg-update-on-bbrv2-00. (2020). 1, 2

[10] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. 2016. BBR: Congestion-Based Congestion Control. *ACM Queue* 14, 5 (Oct. 2016). 1, 3.3

[11] Haoxian Chen, Anduo Wang, and Boon Thau Loo. 2018. Towards Example-Guided Network Synthesis. In *APNet*. https://doi.org/10.1145/3232565.3234462 3.1

[12] Qiaochu Chen, Xinyu Wang, Xi Ye, Greg Durrett, and Isil Dillig. 2020. Multi-Modal Synthesis of Regular Expressions. In *PLDI*. 3.1

[13] D-M. Chiu and R. Jain. 1989. Analysis of the Increase and Decrease Algorithms for Congestion Avoidance in Computer Networks. *Computer Networks and ISDN Systems* 17 (1989). 1

[14] Christian Benvenuti. 2009. *Understanding Linux Network Internals*. O'Reilly Media. 4

[15] Stephen A. Cook. 1971. The Complexity of Theorem-Proving Procedures. In *STOC*. 3.2

[16] Jonathan Corbet. 2005. Pluggable Congestion Avoidance Modules. https://lwn.net/Articles/128681/. (2005). 3.2

[17] Allen Cypher, Daniel C. Halbert, David Kurlander, Henry Lieberman, David Maulsby, Brad A. Myers, and Alan Turransky (Eds.). 1993. *Watch What I Do: Programming by Demonstration*. MIT Press. 3.1

[18] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS*. 3.4

[19] A. Dhamdhere, D. Clark, A. Gamero-Garrido, M. Luckie, R. Mok, G. Akiwate, K. Gogia, V. Bajpai, A. Snoeren, and k. claffy. 2018. Inferring Persistent Interdomain Congestion. In *SIGCOMM*. 2.1

[20] Mo Dong, Qingxi Li, Doron Zarchy, Philip Brighten Godfrey, and Michael Schapira. 2015. PCC: Re-architecting Congestion Control for Consistent High Performance. In *NSDI*. 2

[21] Mo Dong, Tong Meng, D Zarchy, E Arslan, Y Gilad, B Godfrey, and M Schapira. 2018. PCC Vivace: Online-Learning Congestion Control. In *NSDI*. 2, 4

[22] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018. Program Synthesis Using Conflict-Driven Learning. In *PLDI*. 1, 3.1

[23] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-Based Synthesis of Table Consolidation and Transformation Tasks From Examples. In *PLDI*. 3.1

[24] John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing Data Structure Transformations From Input-Output Examples. In *PLDI*. 1, 3.1

[25] Sally Floyd. 2003. HighSpeed TCP for Large Congestion Windows. https://www.ietf.org/rfc/rfc3649.txt. (2003). 1

[26] S. Floyd, M. Handley, J. Padhye, and J. Widmer. 2000. Equation-Based Congestion Control for Unicast Applications. In *SIGCOMM*. 1, 2

[27] Mojgan Ghasemi, Theophilus Benson, and Jennifer Rexford. 2017. Dapper: Data Plane Performance Diagnosis of TCP. In *SOSR*. https://doi.org/10.1145/3050220.3050228 2.1

[28] Sishuai Gong, Usama Naseer, and Theophilus A Benson. 2020. Inspector Gadget: A Framework for Inferring TCP Congestion Control Algorithms and Protocol Configurations. In *IFIP TMA*. 2.1

[29] Sumit Gulwani, William R. Harris, and Rishabh Singh. 2012. Spreadsheet Data Manipulation Using Examples. *CACM* 55, 8 (2012). 1, 3.1

[30] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. 2017. Program Synthesis. *Foundations and Trends in Programming Languages* 4 (2017). https://doi.org/10.1561/2500000010 1, 3

[31] Shivam Handa and Martin C. Rinard. 2020. Inductive Program Synthesis over Noisy Data. In *ESEC/FSE*. 4

[32] Mario Hock, Roland Bless, and Martina Zitterbart. 2017. Experimental Evaluation of BBR Congestion Control. In *ICNP*. 1

[33] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-Guided Component-Based Program Synthesis. In *ICSE*. 1, 3.1

[34] Ashwin Kalyan, Abhishek Mohta, Oleksandr Polozov, Dhruv Batra, Prateek Jain, and Sumit Gulwani. 2018. Neural-Guided Deductive Search for Real-Time Program Synthesis from Examples. In *ICLR (Poster)*. 3.1

[35] Muhammad Khan, Yasir Zaki, Shiva Iyer, Talal Ahamd, Thomas Poetsch, Jay Chen, Anirudh Sivaraman, and Lakshmi Subramanian. 2021. The Case for Model-Driven Interpretability of Delay-Based Congestion Control Protocols. *SIGCOMM CCR* 51, 1 (2021). 1

[36] Leonard Kleinrock. 2018. Internet Congestion Control Using the Power Metric: Keep the Pipe Just Full, But No Fuller. *Ad Hoc Networks* (2018). 1

[37] T. Lan, D. Kao, M. Chiang, and A. Sabharwal. 2010. An Axiomatic Theory of Fairness. In *INFOCOM*. 1

[38] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan 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. 2017. The QUIC Transport Protocol: Design and Internet-Scale Deployment. In *SIGCOMM*. 1

[39] Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. 2018. Accelerating Search-Based Program Synthesis Using Learned Probabilistic Models. In *PLDI*. 3.1

[40] St. Clair McKelway. 1949. Old Eight-Eighty. *The New Yorker* (Aug. 1949). 2

[41] Ayush Mishra, Xiangpeng Sun, Atishya Jain, Sameer Pande, Raj Joshi, and Ben Leong. 2019. The Great Internet TCP Congestion Control Census. *Proc. ACM Meas. Anal. Comput. Syst.* 3, 3 (2019). 2.1

[42] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. 2015. TIMELY: RTT-based Congestion Control for the Datacenter. In *SIGCOMM*. 3.3

[43] Akshay Narayan, Frank Cangialosi, Deepti Raghavan, Prateesh Goyal, Srinivas Narayana, Radhika Mittal, Mohammad Alizadeh, and Hari Balakrishnan. 2018. Restructuring Endpoint Congestion Control. In *SIGCOMM*. 1, 3.2, 4

[44] Jitendra Pahdye and Sally Floyd. [n. d.]. On Inferring TCP Behavior. In *SIGCOMM*. 2.1

[45] Jan Rüth, Ike Kunze, and Oliver Hohlfeld. 2019. An Empirical View on Content Provider Fairness *(IFIP TMA)*. 1, 2.1

[46] Constantin Sander, Jan Rüth, Oliver Hohlfeld, and Klaus Wehrle. 2019. DeePCCI: Deep Learning-Based Passive Congestion Control

Identification. In *NetAI*. 2.1

[47] Dominik Scholz, Benedikt Jaeger, Lukas Schwaighofer, Daniel Raumer, Fabien Geyer, and Georg Carle. 2018. Towards a Deeper Understanding of TCP BBR Congestion Control. In *IFIP Networking*. 2

[48] Lei Shi, Yahui Li, Boon Thau Loo, and Rajeev Alur. 2021. Network Traffic Classification by Program Synthesis. In *TACAS*. 3.1

[49] Viswanath Sivakumar, Tim Rocktäschel, Alexander H. Miller, Heinrich Küttler, Nantas Nardelli, Mike Rabbat, Joelle Pineau, and Sebastian Riedel. 2019. MVFST-RL: An Asynchronous RL Framework for Congestion Control with Delayed Actions. http://arxiv.org/abs/1910.04054. (2019). 4

[50] Armando Solar-Lezama, Rodric M. Rabbah, Rastislav Bodík, and Kemal Ebcioglu. 2005. Programming by Sketching for Bit-Streaming Programs. In *PLDI*. 1, 3.1

[51] Bruce Spang, Brady Walsh, Te-Yuan Huang, Tom Rusnock, Joe Lawrence, and Nick McKeown. 2019. Buffer Sizing and Video QoE Measurements at Netflix. In *Workshop on Buffer Sizing*. 1

[52] R. Srikant. 2004. *The Mathematics of Internet Congestion Control*. Birkhauser. 1

[53] Belma Turkovic and Fernando Kuipers. 2020. P4air: Increasing Fairness among Competing Congestion Control Algorithms. In *ICNP*. 2.1

[54] Belma Turkovic, Fernando A. Kuipers, and Steve Uhlig. 2019. Fifty Shades of Congestion Control: A Performance and Interactions Evaluation. https://arxiv.org/abs/1903.03852. (2019). 2

[55] 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 *HotNets*. 1

[56] Ranysha Ware, Matthew K. Mukerjee, Srinivasan Seshan, and Justine Sherry. 2019. Modeling BBR's Interactions with Loss-Based Congestion Control. In *IMC '19*. https://doi.org/10.1145/3355369.3355604 1, 2, 4

[57] D.X. Wei, C. Jin, S.H. Low, and S. Hegde. 2006. FAST TCP: Motivation, Architecture, Algorithms, Performance. *IEEE/ACM Trans. on Networking* 14, 6 (2006), 1246–1259. 1, 1

[58] Keith Winstein and Hari Balakrishnan. 2013. TCP ex Machina: Computer-Generated Congestion Control. In *SIGCOMM*. 4

[59] Yifei Yuan, Dong Lin, Rajeev Alur, and Boon Thau Loo. 2015. Scenario-Based Programming for SDN Policies. In *CoNEXT*. 3.1, 3

[60] Doron Zarchy, Radhika Mittal, Michael Schapira, and Scott Shenker. 2019. Axiomatizing Congestion Control. *SIGMETRICS* (2019). 1, 2

[61] Yin Zhang, Lee Breslau, Vern Paxson, and Scott Shenker. 2002. On the Characteristics and Origins of Internet Flow Rates. In *SIGCOMM*. 2.1

[62] Yibo Zhu, Monia Ghobadi, Vishal Misra, and Jitendra Padhye. 2016. ECN or Delay: Lessons Learnt from Analysis of DCQCN and TIMELY. In *CoNEXT*. 2