This article has been accepted for publication in IEEE Security & Privacy. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/MSEC.2026.3741807

Article Type: Theme Article. Special Issue on Cyber Hard Problems

Who Verifies the Machine-generated Congestion Control? The Trust Gap in Automated CCA Design.

Mingrui Zhang, *University of Nebraska-Lincoln, Lincoln, NE, 68588, USA*

Hamid Bagheri, *University of Nebraska-Lincoln, Lincoln, NE, 68588, USA*

Lisong Xu, *University of Nebraska-Lincoln, Lincoln, NE, 68588, USA*

Abstract—Machine-generated congestion control algorithms outpace our ability to certify them. We characterize this trust gap as a hard problem, a structural mismatch between generative freedom and certification methods that require structure. We take the position that generation and certification must be designed together, and outline three research directions.

Congestion control algorithms (CCAs) run on virtually every Internet host and coordinate to share finite network resources. For decades, CCAs were designed primarily by human experts. Increasingly, machines are entering the design loop. For example, reinforcement learning (RL) can directly control sending behavior [1] or produce switching policies [2], program synthesis can assemble CCAs from restricted grammars [3], and large language models (LLMs) make it increasingly plausible to generate implementation-level CCA code from high-level intent.

Unlike most machine-generated code, a fairness or correctness problem in a generated CCA has a broader impact beyond the local host, and the harm can be externalized onto the shared network. Specifically, the deployer of a machine-generated CCA may obtain higher throughput benefit; however, other flows sharing the same bottleneck may experience increased queueing delay, packet loss, or reduced throughput. In this scenario, the deployer receives the benefit while other users bear part of the failure cost, which creates a **misaligned incentive problem**. In a multi-tenant cloud, for example, a generated CCA that is too aggressive in obtaining resources in one tenant can affect other tenants' performance that share the same host NIC, switch, or bottleneck. Such behavior need not be malicious. It can arise simply because the generated algorithm was never certified against the safety properties that matter in shared networks.

XXXX-XXX © 2021 IEEE
Digital Object Identifier 10.1109/XXX.0000.0000000

Historically, such fairness or correctness problems have also occurred in human-designed CCAs, and production algorithms typically require years of deployment, measurement, and expert analysis to identify and correct such issues. For example, BBRv1's successive revisions continue to improve coexistence fairness with loss-based CUBIC congestion control; a verification work uncovered implementation bugs in CUBIC years after its first deployment [4]. Now, automated generation techniques can produce new candidate CCAs much faster than before, and the speed of generation makes this verification problem harder to address. Publicly available RL-based CCAs already exhibit unfairness and excessive packet loss across many tested configurations, since fairness was not a reward goal in tested designs [5]. Yet, the literature has not demonstrated a comparable cycle of revision and recertification for automatically generated algorithms. For LLM-generated CCAs, fairness may also depend on whether the fairness requirements are explicitly expressed in the design intent that is passed to the generator, and on whether the generator implements those fairness requirements. These observations make it important to systematically compare human-designed and machine-generated CCAs in fairness and other metrics.

The result is a widening trust gap, that is, our ability to generate CCAs now exceeds our ability to certify them. Many existing CCA verification tools need human effort before automated reasoning can begin, such as a hand-built abstraction, a known control structure, or a carefully selected property set [4], [6]. As a consequence, the field has already begun responding

© 2026 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Security & Privacy. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/MSEC.2026.3741807

to this gap, even if not always explicitly. One visible response is a switch from Aurora-style [1] direct control toward Mutant-style [2] policy switching among known CCAs. This trades away generative freedom for trust because choosing among vetted algorithms is easier to reason about than deploying a newly generated control law.

Different verification methods certify at different levels, and are not interchangeable. A mathematical proof or model checker can certify an abstract control idea under a formal model [6]. Symbolic execution can certify properties of a particular code implementation over the explored program paths [4]. Empirical evaluation can certify the behavior observed in the simulations, emulations, or deployments that were actually run [5]. For example, a CCA that performs well in tested scenarios may still fail under an untested network condition, and it may satisfy an abstract model while its implementation behaves differently.

Machine generation makes this mismatch a **technically complex hard problem**, as they seek opposite goals. Specifically, existing verification methods usually assume some **structure** in advance, such as a model, grammar, control form, or implementation pattern that the verifier is built to reason about, while generators gain power by relaxing those assumptions. Recent uses of LLMs sharpen this tension: they can translate user intent into design goals, generate implementation-level code, and iterate rapidly over candidate CCAs. The result is two extremes: the easiest to generate and hardest to certify. This pattern can arise in related cyber network systems, where generated control logic acts locally but affects a common shared resource.

We therefore advocate that the next step in machine-designed congestion control should not be more powerful generation techniques alone. Certification must be designed alongside generation from the beginning.

THE FRONTIER

Existing CCA design methods fall along a frontier between generativity and certifiability. Methods with stronger certifiability usually require predefined structures, such as a fixed control law, a network model, a bounded grammar, or a known implementation pattern; while methods with higher generativity tend to remove that structure, which makes the resulting artifacts harder to certify before deployment. Figure 1 summarizes this tradeoff.

The horizontal axis represents the generative power of the design method: how much novelty it can introduce into the CCA it produces. At the low end is parameter tuning of a fixed algorithm, such as choosing parameters for CUBIC's window function [7]. Above tuning, in increasing generativity, are DRL policies for switching among vetted algorithms [2], DRL policies trained over fixed action spaces [1], program synthesis over a domain-specific grammar [3], and free code generation, where the method emits arbitrary control code in a general-purpose language.

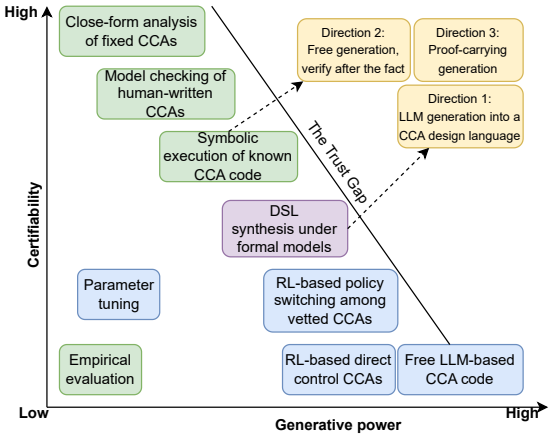


FIGURE 1. The Generativity and Certifiability ability of frontier approaches in machine-generated Congestion Control. Green boxes are existing verification methods, blue boxes are existing generation methods, and the purple boxes are mixed certification and generation power. Yellow boxes are opening directions that this article points out.

The vertical axis represents the certifiability of the output: how strong a verification we can make about the algorithm before it runs on a shared network. At the low end is empirical evaluation [5], where we know how the CCA behaved in the simulations, emulations, or testbeds we tried. Above that are model checking against an abstracted network model [6], symbolic execution against properties of the implementation [4], and closed-form proof of the CCA's behavior in a tractable analytical model, such as CUBIC.

The certifiable corner

At the top-left of Figure 1 are approaches that provide the strongest certification and the least design novelty. Classical mathematical models can characterize throughput as a function of loss rate, round-trip time, and algorithm parameters. These results are valuable as providing mathematical guarantees beyond measurements, but applied only to those CCA with a simple, fixed structure.

Model checking, such as CCAC [6], checks prop-

© 2026 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Security & Privacy. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/MSEC.2026.3741807

erties of human-designed CCAs by building the CCA abstraction models and abstracted network models, then solving counterexamples to claimed behavior. The search is systematic over the model, but the CCA abstraction model, the network abstraction, and the checked properties are all human-supplied. Therefore, CCAC certified behaviors are not guaranteed to be equal to the actual implementations.

Symbolic execution of CCA code

Symbolic execution is closer to deployed reality. Instead of reasoning only about an analytical model or an abstracted CCA, it analyzes implementation code by treating program inputs as symbolic values and exploring the execution paths those inputs induce. For congestion control, these inputs can represent the initial CCA state and network-event information. This matters because failures can live in the implementation rather than in the high-level design.

ASM [4] is a recent example of this point on the frontier. On top of symbolic execution to Linux CCA implementation, it aggregates equivalent ACK events, and reaches tens of thousands of ACK events emulation. By applying those techniques, ASM found implementation bugs in Linux CUBIC that could lead to unfair behavior. This demonstrates that formal verification of real CCA implementations is feasible, which is beyond abstract models. However, their harness, properties, and abstractions are designed around congestion control algorithms' properties that admit ACK aggregation, and therefore is limited in certifiability.

A complementary example for unknown generated code is MetaEase [8], which directly analyzes heuristic source code and searches for worst-case inputs from the implementation itself. It demonstrates that source-code-level analysis can preserve substantial implementation freedom while still exposing problematic behaviors in generated code.

The high-generativity corner

At the bottom-right of Figure 1 are methods with the greatest design freedom and the weakest certification. Direct RL-based CCAs, such as Aurora [1], replace a hand-written control law with a learned policy from reward signals and empirical network interactions. Therefore, the policy's behavior is primarily understood through the training and evaluation environments. Recently, a systematic evaluation found those RL-based CCAs expose unfairness behavior in nearly all tested network configurations and excessive packet loss under normal operation [5].

One response has been to retreat down the generativity axis. That is, rather than learning a sending policy directly, recent systems learn to select among human-designed CCAs [2]. This preserves adaptation ability while restricting the action space to algorithms that are already known, implemented, and certified.

LLM-based CCA generation pushes the frontier further right. Unlike RL-based systems that learn a policy within a fixed action space, an LLM can generate CCA code from user intent. NECC [9], for example, uses an LLM to translate user requirements into deployable CCA code. This lowers the expertise barrier and accelerates design iteration. However, it is the least certifiability point on the frontier, as its verification mainly comes from empirical evaluation [9].

Domain-specific language synthesis

A compromise between certifiable and generative is to constrain the design space with a grammar or domain-specific language (DSL), then search for feasible solutions within that space. As a result, the certifier already understands its structure from the design space definition. If the grammar is designed together with the proof method, then every synthesized CCA can inherit a form of analyzability.

CCmatic [3] is such DSL-based CCA design work. CCmatic system uses a belief framework and program synthesis to search for CCAs that satisfy specified performance properties under formal network models. The resulting CCAs are short, modular, human-interpretable, and come with provable performance guarantees under the modeled conditions. CCmatic searches within a predefined class of belief-based CCAs and verifies properties against specified network models. However, its formal results depend on the expressiveness of that design space and the fidelity of those models. For example, CCmatic explicitly focuses on the single-flow case, and the approach has a length limit for code synthesis. Therefore, DSL synthesis is more generative than model checking or symbolic execution of an existing CCA, but less generative than free LLM code generation, placing it at the middle point in Figure 1.

THE STRUCTURAL GAP

Why the trust gap remains unsolved

Each point on the frontier only addresses part of the trust gap. Empirical testing shows behavior in the scenarios that were run, but it does not certify behavior outside testing scenarios. The limitation is more important for CCAs because the relevant failures often depend on network link conditions, peer algorithms,

© 2026 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Security & Privacy. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/MSEC.2026.3741807

and transient interactions that are hard to enumerate.

Formal verification gives stronger certifiability once the CCA has been translated into a form the verifier can reason about. Model checking requires a network model, an abstraction of the CCA, and a property set. Symbolic execution depends on a harness and properties designed for recognizable control structures. These methods are powerful when the artifact already provides useful verification infrastructure, while an LLM-generated arbitrary control code may require verification methods to handle arbitrary new states, new signals, or new control modes, which makes those verification artifacts not reliable.

RL-based CCA design also fails to close the trust gap. It can generate adaptive behavior, but its verification is mostly empirical. A learned policy may perform well under its training and evaluation environments, but it can still produce unfairness, excessive loss, or instability when it coexists with other algorithms.

The gap remains because existing approaches address generation and certification separately. Empirical evaluation covers tested executions but not all possible executions. Formal verification can provide stronger guarantees but depends on models, abstractions, harnesses, and properties defined for the CCA under analysis. RL and LLM generation can produce new policies but do not provide the corresponding evaluation test case, model, harness, or property specification that a verifier needs.

CCAs are not ordinary generated software

The trust gap in machine-designed CCAs may look like a special case of a broader problem. Large models now generate code for many production systems, and those systems also need verification. One might argue that congestion control can wait for the general problem of code generation and verification to be solved. That view overlooks the CCAs' unique properties. Specifically, three properties distinguish them from ordinary generated software: feedback control, open coexistence, and externalized harm.

First, a CCA is a feedback controller in a closed loop with the network. Most software verification asks whether a program satisfies a specification for a given class of inputs. A CCA does not have a fixed input and output relation of that kind. Its behavior at time t depends on the network state, but the network state at time t also depends on the CCA itself, on other flows using other CCAs, and on path conditions that change over time. The properties for CCAs include stability, fairness, convergence, bounded queue growth, and bounded loss, which are not properties of one control decision. They are properties of the coupled evolution of sender, receiver, queue, bottleneck, and competing flows. Certification for CCAs must cover network-level properties, not only local functional correctness.

Second, a CCA must coexist with peer algorithms. A generated function is usually asked to be correct on its own inputs. A generated CCA is asked to remain safe while sharing a bottleneck with CUBIC , BBR , and other generated CCAs whose control laws it may never have seen. Correctness is a property of the algorithm in isolation and of how it competes for a finite shared resource. A systematic evaluation found unfairness in nearly all tested coexistence settings and observed excessive packet loss during normal operation [5]. Their training objectives and evaluation pipelines did not address coexistence properties before deployment, while other coexist CCAs faced unfairness results.

Third, the deployment surface externalizes the harm. A buggy generated application function usually harms the application, host, or user that runs it. A generated CCA can affect other flows sharing the same bottleneck. This changes the resilience question because it creates misaligned incentives. The incentive problem becomes clearer in a multi-tenant cloud. A tenant, application provider, or CDN operator may use an automated tool to generate a CCA tuned for its own traffic. While the benefit is local, such as higher throughput, lower completion time, faster adaptation to a workload, and less need for congestion-control expertise, the cost of such a design is paid by tenants that share the resource. If the generated CCA is too aggressive, other tenants sharing the same host NIC, switch, or bottleneck path may see inflated queues, packet loss, and reduced throughput.

This separates the benefits and failure costs of generation to different parties. That separation is what makes the trust gap a cyber-resilience problem rather than only a software-quality problem. Without a certification, the rational deployment choice can be unsafe for the system as a whole.

Certifiers and generators require different structures

The structural gap comes from a simple conflict. A certifier needs structure. It needs a model, a grammar, a known control form, a property set, or a recognizable implementation pattern. Every verification method works by using such a structure to reason from assumptions to behavior. A generator gains power by relaxing that structure. The more freedom it has to produce a new CCA, the less likely the output is to fit the assumptions that existing certifiers require.

© 2026 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Security & Privacy. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/MSEC.2026.3741807

This is why the frontier slopes. The top right corner is empty because no existing method achieves both requirements. Two requirements point in different directions. Certification asks the design to commit to a form that can be analyzed. Generation asks the design space to remain open enough to discover behavior that the analyst did not specify in advance.

The history of congestion control shows this tension. Loss-based CCAs such as Reno and CUBIC have relatively fixed update structures. Their behavior is complex enough that their control laws give verification methods something to grip. Closed-form analysis can reason about their throughput under simplified models. Symbolic execution can reason about implementation paths because the update rules, state variables, and loss responses are recognizable.

Newer CCAs move away from that structure. BBR estimates bottleneck bandwidth and round-trip propagation time, then steers the sender toward an operating point derived from those estimates. HPCC uses in-network telemetry to compute rate adjustments from signals such as queue occupancy [10]. These algorithms may improve performance in the environments they target, and they fit less directly with the assumptions used by classical throughput analysis or loss-based symbolic verification. The control state is richer, the signals are different, and the operating point is computed rather than emerging from a fixed loss response law.

Machine-generated CCAs continue this movement. With RL-based direct control CCA, the control policy may be represented by learned weights; with LLM-based code generation, the control structure is whatever the model emits. The generator may introduce new states, combine signals in unexpected ways, or switch between control modes. The same freedom that makes the approach attractive also makes later certification harder.

The current certification methods certify by reasoning from an assumed structure. If future methods can certify distributions of generated artifacts, prove safety against broad classes of peer algorithms, or generate machine-checked proofs together with code, then the frontier may move. Those methods remain to be developed for CCAs. For now, the field has powerful generators on one side and structure-dependent certifiers on the other. The trust gap is the space between them.

Summary

The gap diagnosis frames the next step: where should automated CCA design sit on the frontier?

LLMs make this question unavoidable. Compared with learning-based CCA design, an LLM is free of a fixed policy class, action space, or reward function. It can generate new state variables, signals, control logic, and implementation code. It can also translate user intent into objectives, constraints, design sketches, and candidate implementations. Conventional learning methods require the control problem and reward to be specified before learning begins. LLM-based generation moves part of that specification work into the generator itself.

The open directions, therefore, assume that LLMs will participate in future automated CCA design loops. This leaves learning-based CCAs with their own verification problem. They raise a different verification problem where the artifact is usually a learned policy behind a fixed interface. LLM-generated code is different because the artifact may be implementation code, with explicit state, branches, signals, and control modes. That code is more accessible to program analysis than a neural policy, and its structure is less predictable than the hand-written CCAs existing tools assume.

THE OPENING DIRECTIONS

We outline three directions in this section. The first asks the LLM to generate CCAs within a restricted CCA design language. The second lets the LLM generate freely and tries to certify the resulting code after generation. The third asks for proof-carrying generation, where the generated artifact includes both code and a machine-checked argument about its behavior. These directions place certification at different stages of generation: in the design space, after code generation, or in the generated output itself. We also propose following RFC 9743 to guide the construction of network models for CCA verification, including competing CCAs, network conditions, and misbehaving nodes or external attackers. Then, explicitly state the competing CCAs and network conditions as part of the certification.

LLM generation into a CCA design language

The first direction trades generativity for certifiability before implementation. Specifically, rather than letting the LLM emit C code directly, the LLM emits into a higher-level design representation, such as a grammar, a domain-specific language, or a structured policy specification. That representation is then compiled to executable code by a procedure that preserves its structural properties.

The goal of this pattern is to separate design from implementation. The LLM performs the creative step

© 2026 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Security & Privacy. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/MSEC.2026.3741807

within a representation that the verifier is designed to analyze. The verification step is applied at that same level as predefined structures, where reasoning is more tractable. Only after the design passes the required checks is it translated into deployable code.

This pattern has analogs in adjacent areas. In software engineering and algorithm design, systems that ask an LLM to produce a plan or specification before code have become an active line of work. For example, Evolution of Heuristics [11] proposes a workflow where the LLM proposes algorithmic ideas, the system implements them, and candidates are selected against an objective. This separation between the idea generated and the implementation achieves better results than previous methods.

In networking, the synthesis side of this direction already has a strong precedent. [3], [12] show that CCAs expressed in a predefined policy space can be synthesized with provable performance guarantees under modeled network conditions. What is absent is the generation half. That work uses search, not LLM generation. To our knowledge, no system yet lets an LLM generate a CCA into a built design language and then preserves the verified properties of that language through compilation.

The hard part is the joint design of three languages: (1) the CCA design, (2) the network model, and (3) the properties being certified. The CCA language must be expressive enough to represent algorithms that the field would actually consider useful. Such algorithms may combine multiple signals, maintain state across acknowledgments, and switch behavior over the lifetime of a connection. At the same time, the language must be restricted enough that the verifier can reason about every generated artifact. The network-modeling side is equally important. A generated CCA may look certifiable under a single-flow model but fail when it coexists with CUBIC or another generated CCA. A CCA design language should build on the algorithms it can express and the network behaviors its verifier can describe. The property language is the third part of the stack. The relevant claims are not ordinary functional-correctness claims. They concern stability, convergence, fairness against peer algorithms, bounded queue growth, and tolerable loss over executions of a coupled networked system. A DSL should have a clean CCA syntax, state these properties, and state network assumptions to close the trust gap.

This cannot be obtained for free by general software engineering work on structured code generation. A CCA design language is not only an interface for the LLM. It is part of a verification stack that must expose the assumptions under which a generated CCA is safe to deploy.

A useful result in this direction would need three pieces. First, a candidate CCA design language paired with an explicit network-modeling language and property language. Second, evidence that an LLM can generate nontrivial CCA designs in that language. Third, a demonstration that guarantees from the synthesis and verification layer still apply to the LLM-generated designs under the stated network assumptions. Especially, the verification claim must address coexistence and fairness on shared infrastructure.

Free LLM generation, verify after the fact

This direction lets the LLM generate CCA code freely in a general-purpose language. The generated CCA may introduce new signals, state variables, and control regimes. The verification method then has to certify the resulting implementation.

To verify implementation code, this direction naturally extends recent work on symbolic execution for CCA code. ASM [4] showed that scalable symbolic verification of CCA implementations is feasible on production Linux CCA implementations, reaching executions with tens of thousands of acknowledgments. It also found bugs in real CCA implementations, which shows that code-level verification can follow this approach.

The open question is whether the same approach can verify a CCA whose structure was not chosen in advance. ASM was designed around whether a CCA property admits ACK aggregation. The current properties, harnesses, and abstractions fit congestion control and relevant event sequences that can be represented by the aggregation algorithm, while an LLM-generated CCA may not have that shape. That LLM-generated CCA may use delay as its primary signal, combine delay with loss or telemetry, introduce new states, or switch among control modes during a connection.

The state-of-the-art design around ACK aggregation needs improvement to achieve the free generation verification goal. Verifying known CCAs is easier than verifying arbitrary congestion control code the LLM emits. The hard points are behavior properties, network environment interaction representation, and network state representations. The ideal verifier should have a comprehensive networking simulation design to represent and handle these cases, including which properties matter, how to express them for the generated structure, and which environmental assumptions make the result meaningful.

In general software applications, symbolic execution usually models and simulates *effects* of the envi-

© 2026 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Security & Privacy. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/MSEC.2026.3741807

ronment or system calls, and certifies specific correctness properties or bug classes. In CCA verification, we also need to model and simulate the networks and multi-host coexistence interactions that affect the CCA, rather than only its effects. Therefore, the engine can check networking properties, such as fairness against arbitrary peers, bounded contribution to bottleneck queues, or stability under path drift.

The appeal of this direction is usability. The user can ask the LLM for code directly, and the certification step is applied afterward. The weakness is that the verification method bears nearly all of the burden, as it needs to simulate relevant possible events, interactions, and results. Also, the generator would also benefit from verifier design guidance on how to generate a certifiable CCA, such as which generated structures are certifiable and which structures fall outside the verifier's model.

Proof-carrying generation

Instead of generating code first and verifying it later, the LLM generates code in a proof-aware substrate such as Lean 4, Coq, Dafny, or a successor system. That is, the LLM-generated artifact would include both the CCA and a machine-checked proof of the properties the CCA is supposed to satisfy. In summary, there are three roles for this direction: code generation, proof construction, and proof checking, where the generator takes the first two, and the proof system will check the proofs.

In adjacent fields, systems such as SynVer combine LLM-based program synthesis with machine-checked proofs for C programs in Rocq/Coq [13]. Proof-carrying code completion systems have explored having an LLM produce Dafny code together with proofs of selected safety properties [14].

For CCAs, generating proof-aware code faces two difficult problems. The first problem is constructing the proof itself. Given a property and proof, a proof assistant can check a completed proof process. However, the LLMs still have to produce the reasoning that connects the generated CCA to its claimed behavior. This is difficult for congestion control because properties such as convergence, bounded queue growth, bounded loss, and fairness depend on the interaction between the CCA and the network over time. The proof therefore has to reason about both the generated control logic and the network states that it can encounter.

The second difficult problem is to decide what to prove, as there is no standard machine-checkable definition of CCA correctness. For example, the fairness property depends on the peer algorithms the CCA coexists with, and queueing, loss, or convergence properties depend on the network model assumptions. Without such specifications, a proof assistant has no agreed CCA properties against which to certify generated code.

Progress in this direction may begin with formal CCA specifications, such as Canopy [15] certifies congestion control properties. A useful first step would be a small proof assistant library for simple loss-based CCAs and simple network models. It would define basic notions such as sending rate, queue evolution, loss, utilization, convergence, and coexistence. Finally, the certification and generation ability of this direction depends on the CCA behaviors that the proof system can represent and prove.

Summary

Proposed Direction 1: "LLM generation into a CCA design language." Where certification depends: This direction pushes the certification responsibility to the design language, and the generative ability depends on how much that language can provide. **Our view:** We view this as the most conservative direction, as it improves certifiability by limiting the LLM's design space. **Concrete first step:** To design a small DSL set with clearer and easier causality chains, such as a design based on [3], then evaluate whether an LLM can generate valid and nontrivial CCAs within this language, and finally extend it to more comprehensive and complex congestion control behaviors.

Proposed Direction 2: "Free LLM generation, verify after the fact." Where certification depends: This direction pushes the certification responsibility to the symbolic engine, and the certification ability depends on how many network states it can represent, while the generative ability is entirely on the LLM's side. **Our view:** We view this direction as preserving the most generative freedom, but requiring the verifier to keep up with control structures and network signals that were not known in advance. **Concrete first step:** To extend existing aggregated ACK-based long-term network states [4] to cover more signals in the symbolic engine.

Proposed Direction 3: "Proof-carrying generation." Where certification depends: This direction pushes the certification responsibility to the proof engine's representation, and the certification ability depends on how many network states and properties we can state. **Our view:** We view this as the most integrated but also the longest-horizon direction, because both the CCA properties and their proofs must be represented in the proof system. **Concrete first**

© 2026 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Security & Privacy. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/MSEC.2026.3741807

step: Design a minimal set of machine-checkable CCA correctness properties and link them to the proof system, then extend it to cover more properties.

CONCLUSION

Machine-designed congestion control is becoming an active direction. Reinforcement learning, program synthesis, and large language models already show that machines can produce new CCAs and are much faster than traditional expert design. Unfortunately, existing certification methods cannot match the fast generation speed to verify new CCAs behavior before deployment on shared network infrastructure, creating a trust gap and cyber-resilience concern. The trust gap persists because generators gain power by relaxing design constraints, while certification methods rely on predefined structures that they can model and reason about.

Congestion control is a feedback controller deployed in a shared environment, where its behavior affects queues, bottlenecks, and competing flows outside its owner's control. We believe the way forward is not to reject automated CCA design workflow, but to design generation and certification together, make explicit design tradeoffs, and avoid hiding them behind benchmark performance. We propose three future directions: CCA design languages, post-generation verification, and proof-aware generation. We believe they will become important infrastructure to support automated congestion control.

ACKNOWLEDGMENT

This work was supported in part by NSF 2124116, and UNL Research and Innovation Grant 2026.

REFERENCES

1. N. Jay, N. Rotman, B. Godfrey, M. Schapira, and A. Tamar, "A deep reinforcement learning perspective on internet congestion control," in *Proceedings of the 36th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, K. Chaudhuri and R. Salakhutdinov, Eds., vol. 97. PMLR, 09–15 Jun 2019, pp. 3050–3059.
2. L. Pappone, A. Sacco, and F. Esposito, "Mutant: Learning congestion control from existing protocols via online reinforcement learning," in *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, Philadelphia, PA, Apr. 2025, pp. 1507–1522.
3. A. Agarwal, V. Arun, D. Ray, R. Martins, and S. Seshan, "Towards provably performant congestion control," in *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, Santa Clara, CA, Apr. 2024, pp. 951–978.
4. M. Vu, H. Bagheri, L. Xu, W. Sun, and M. Zhang, "Scalable verification of multi-ack properties in loss-based congestion control implementations," in *Proceedings of the IEEE 32nd International Conference on Network Protocols (ICNP)*, 2024, pp. 1–12.
5. L. Giacomoni and G. Parisis, "Reinforcement learning-based congestion control: A systematic evaluation of fairness, efficiency and responsiveness," in *Proceedings of the IEEE Conference on Computer Communications (INFOCOM)*, 2024, pp. 1451–1460.
6. V. Arun, M. T. Arashloo, A. Saeed, M. Alizadeh, and H. Balakrishnan, "Toward formally verifying congestion control behavior," in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, ser. SIGCOMM '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1–16.
7. W. Yang, Y. Liu, C. Tian, J. Jiang, and L. Guo, "Gemini: Divide-and-conquer for practical learning-based internet congestion control," in *Proceedings of the IEEE Conference on Computer Communications (INFOCOM)*, 2023, pp. 1–10.
8. P. Karimi, S. K. R. Kakarla, R. Beckett, S. Segarra, P. Namyar, M. Alizadeh, and B. Arzani, "Heuristic analysis from source code via Symbolic-Guided optimization," in *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*, Renton, WA, May 2026, pp. 2151–2174.
9. M. Zhang, H. Bagheri, and L. Xu, "Toward non-expert customized congestion control," in *Proceedings of the 2025 IEEE International Conference on Communications*, 2025, pp. 3606–3612.
10. Y. Li, R. Miao, H. H. Liu, Y. Zhuang, F. Feng, L. Tang, Z. Cao, M. Zhang, F. Kelly, M. Alizadeh, and M. Yu, "Hpcc: high precision congestion control," in *Proceedings of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '19, 2019, pp. 44–58.
11. F. Liu, X. Tong, M. Yuan, X. Lin, F. Luo, Z. Wang, Z. Lu, and Q. Zhang, "Evolution of heuristics: towards efficient automatic algorithm design using large language model," in *Proceedings of the 41st International Conference on Machine Learning*, ser. ICML'24. JMLR.org, 2024.
12. A. Agarwal, V. Arun, and S. Seshan, "FRCC: Towards provably fair and robust congestion control," in *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*. Renton, WA:

© 2026 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Security & Privacy. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/MSEC.2026.3741807

USENIX Association, May 2026, pp. 2755–2778.

13. P. Mukherjee, M. Lu, and B. Delaware, "Towards automated verification of llm-synthesized c programs," 2025. [Online]. Available: https://arxiv.org/abs/2410.14835
14. P. Kamran, P. Devanbu, and C. Stanford, "Vision paper: Proof-carrying code completions," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering Workshops*, ser. ASEW '24. New York, NY, USA: Association for Computing Machinery, 2024, pp. 35–42.
15. C. Yang, D. Saxena, R. Dwivedula, K. Mahajan, S. Chaudhuri, and A. Akella, "Canopy: Property-driven learning for congestion control," in *Proceedings of the 21st European Conference on Computer Systems*, ser. EUROSYS '26, 2026, pp. 1658–1673.

Mingrui Zhang is a Ph.D. student in the School of Computing at the University of Nebraska-Lincoln, Lincoln, NE 68588 USA. His current research interests include congestion control, formal verification, and ML/AI for networked systems. Zhang received his M.S. degree in computer science from the University of Nebraska-Lincoln. Contact him at mzhang23@huskers.unl.edu.

Hamid Bagheri is an associate professor in the School of Computing at the University of Nebraska-Lincoln, Lincoln, NE 68588 USA. His current research interests include software security, applied formal methods, and AI for software engineering. Bagheri received his Ph.D. degree in computer science from the University of Virginia. Contact him at bagheri@unl.edu.

Lisong Xu is a professor in the School of Computing at the University of Nebraska-Lincoln, Lincoln, NE 68588 USA. His current research interests include network protocols, congestion control, and formal verification. Xu received his Ph.D. degree in computer science from North Carolina State University. Contact him at xu@unl.edu.

© 2026 IEEE. All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.