Scalable Verification of Multi-ACK Properties in Loss-Based Congestion Control Implementations

Minh Vu*, Hamid Bagheri*, Lisong Xu*, Wei Sun†, Mingrui Zhang*
* University of Nebraska-Lincoln, {minh.vu, mzhang23}@huskers.unl.edu, {bagheri, xu}@unl.edu,
† Meta Platform, Inc, wesun@meta.com

***Abstract*—Congestion control algorithms, such as RENO and CUBIC, are vital for the Internet. However, numerous bugs have been discovered and reported in the Congestion Control Algorithm Implementations (CCAIs), even in those that have been extensively tested and used on the Internet for years, such as Linux RENO and Linux CUBIC. Some of these bugs have potentially severe impacts on the performance and stability of the Internet. Unfortunately, current CCAI testing and verification methods are inadequate for proving the absence of bugs, require substantial verification expertise, or are not scalable to a large number of acknowledgment packets (ACKs) that trigger CCAI actions. To address all these shortcomings, we propose an ACK Scalable Method, called ASM. Our experiments with two representative loss-based CCAIs, Linux RENO and CUBIC, demonstrate the promising performance of the proposed ASM even with tens of thousands of ACKs.**

***Index Terms*—TCP congestion control, verification, symbolic execution, mathematical induction**

I. Introduction

Congestion control algorithms are vital for the Internet as they are responsible for efficiently and fairly allocating Internet bandwidth among a large and dynamic number of users. However, there is no one-size-fits-all solution because it is challenging, if not impossible, to design a single congestion control algorithm that works well for all types of Internet applications and across all current and future Internet environments (e.g., wired, wireless, cellular, data centers, and Internet of Things). As a result, various congestion control algorithms, such as RENO [1], CUBIC [2], and BBR [3], have been proposed, implemented, and deployed on the Internet [4]. Furthermore, new congestion control algorithms are still being proposed for improved performance or new types of applications every year.

While congestion control has a tremendous impact on the Internet, numerous bugs have been detected and reported [5]–[8] in *Congestion Control Algorithm Implementations* (CCAIs, i.e., actual code) in popular operating systems, even in those that have been extensively tested and used on the Internet for years, such as Linux CUBIC [9] (current default) and Linux RENO [10] (previous default). This is because CCAI developers may make mistakes in designing and implementing CCAIs, especially since real-world CCAIs, like those in Linux, involve multiple intertwined and evolving components contributed by different groups of developers spanning over many years.

This work was supported in part by NSF CCF-2124116. This paper is based on the work when Minh Vu was a Ph.D. student at UNL.

979-8-3503-5171-2/24/$31.00 ©2024 IEEE

CCAI Properties: A CCAI adapts its TCP throughput to changing network conditions by updating its state variables in response to network events. For example, an important state variable is the congestion window size, `cwnd`, which determines the number of data packets that a CCAI can transmit in a round-trip time (RTT). A network event could be the arrival of an acknowledgment packet (ACK) or a timeout at a TCP sender. Most CCAI properties are the expected state of a CCAI in response to ACKs and can be classified into two types: single-ACK and multi-ACK properties, depending on whether a single or multiple ACKs are involved. For example, a well-known multi-ACK property is the **NoMoreThanOne** property [1], which requires that RENO must not increment its `cwnd` by more than one packet (specifically the maximum segment size) during the congestion avoidance stage in response to all the ACKs in an RTT.

Current practices: There are three potential classes of methods to check the correctness of CCAIs. *Class 1 - Testing methods* aim at detecting CCAI bugs, such as manual testing [11], [12], undirected random testing [13], feedback-guided random testing [7], [14], genetic algorithm-based random testing [15], and symbolic execution-enhanced random testing [16], [17]. They are effective in scalably checking the CCAI behaviors even for a large number of packets; however, they are unable to prove the correctness of CCAIs (i.e., the absence of bugs). *Class 2 - Model verification methods*, such as CCAC [18] and Belief Framework [19], automatically prove the correctness of approximate abstract models of CCAIs using techniques, such as model checking, where the models are written manually using formal specification languages. However, CCAIs may have bugs not captured in the approximate abstract models. *Class 3 - Code verification methods* aim at proving the correctness of the actual code of CCAIs. They can be further classified into two sub-classes. *Class 3.1 - Manual code verification methods*, such as theorem proving, construct and conduct the proofs with the assistance of formal proof management systems (e.g., Coq [20]). *Class 3.2 - Automated code verification methods*, such as SCCT [8], automatically prove the correctness of CCAI for a small number of packets using techniques such as exhaustive symbolic execution [21]–[23] and implementation-based model checking [24]–[26].

Limitations of current practices: While both testing methods (Class 1, e.g., [7], [14]–[16]) and model verification methods (Class 2, e.g., [18], [19]) improve our confidence in CCAI, none of them verifies the correctness of the actual code

of CCAI. This is because testing methods are not guaranteed to detect all possible bugs, and model verification methods only prove the correctness of their approximate abstract models. While manual code verification methods (Class 3.1) verify the actual code of CCAI, they require substantial verification expertise (e.g., theorem proving and formal specification) and thus are hard to use by network protocol designers and developers. For example, recent studies [22], [27] show that "a major stumbling block to the adoption of formal methods" is the lack of verification expertise in the network community. This is why we could not find one example for this class. While automated code verification methods (Class 3.2, e.g., SCCT [8]) verify the actual code of CCAI and require little verification expertise, they are not scalable to multi-ACK properties with a large number of ACKs, which are common for multi-ACK properties. For example, the number of ACKs in the NoMoreThanOne property is proportional to `cwnd`, which could be thousands or higher in high-speed networks.

Our method: We propose an ACK Scalable Method, called ASM, for network protocol designers and developers with little verification expertise (i.e., ASM users) to check the multi-ACK properties of CCAIs. ASM extends the current automated code verification method, SCCT [8], significantly improving its scalability in checking multi-ACK properties using three techniques. 1) *Aggregated Information*: We convert a CCAI to an equivalent Aggregation Congestion Control Algorithm (Agg-Alg), which operates on aggregated information of a sequence of ACKs and has the same congestion control behavior as the CCAI. It is more scalable to verify the Agg-Alg than the CCAI, because the Agg-Alg handles aggregated ACK information whereas the CCAI handles individual ACKs. 2) *Symbolic Aggregated Information*: We use symbolic variables to represent the aggregated information of multiple sequences of ACKs by leveraging the symbolic execution technique (a powerful program analysis technique, Section III-A). Intuitively, symbolic aggregated information enables us to simultaneously check the behaviors of the Agg-Alg in response to multiple sequences, instead of one sequence at a time. 3) *Mathematical Induction*: We scalably prove the equivalence between the CCAI and the Agg-Alg using the proof by mathematical induction on the number of ACKs. As a result, proving the correctness of the Agg-Alg is equivalent to proving the correctness of the CCAI.

Contributions: We make the following contributions in this paper. First, we propose ASM, which is the first CCAI verification method that is scalable to a large number of ACKs, requires little verification expertise, and verifies the actual code of CCAIs. Note that our proposed ASM works only for a CCAI with an equivalent Agg-Alg, for example, loss-based Linux CCAIs, such as Linux RENO, CUBIC, BIC, HIGHSPEED, and SCALABLE. We will discuss the extension of ASM to delay-based CCAIs in Section VII-E. Second, we evaluate the performance of ASM using two representative real-world loss-based CCAIs, Linux RENO and Linux CUBIC. Our experiments show that ASM reduces the verification time by several orders of magnitude compared with the current automated code verification method, SCCT [8]. For example, it takes ASM only 1.5 minutes to check the NoMoreThanOne property of Linux RENO for up to 10,000 ACKs in about 10^8 network environments, whereas SCCT already takes about 200 minutes for up to only 100 ACKs. Due to the superior scalability of ASM, for the first time, we are able to prove some fundamental properties of Linux RENO and CUBIC in common network environments of the Internet.

TABLE I
REQUIRED CCAI API FUNCTIONS IN LATEST LINUX KERNEL

API	Description
`ssthresh()`	calculate a new `ssthresh`
`cong_avoid()`	update `cwnd` in slow start and congestion avoidance
`undo_cwnd()`	revert the unnecessary `cwnd` reduction

II. BACKGROUND AND RELATED WORK

A. Background on CCAI

The TCP code of an operating system has multiple parts for different purposes, such as the connection management part for establishing and terminating a TCP flow, the reliability part for providing reliable transfer in case of packet reordering and loss, and the congestion control part (i.e., CCAI) for determining the efficient TCP throughput while being fair among all users and avoiding congestion collapse. In this paper, we consider the verification of the CCAI part of TCP.

A CCAI generally consists of multiple components, such as slow start, congestion avoidance, fast retransmit, fast recovery, timeout, reordering, and undo, to handle different packet dynamics. An important state variable of a CCAI is the congestion window size, `cwnd`, which determines the number of data packets that the CCAI can transmit in an RTT. Another important state variable is the slow start threshold, `ssthresh`, which determines whether the current stage is slow start (i.e., if `cwnd<ssthresh`) or congestion avoidance.

An operating system usually supports multiple CCAIs, and adopts some architecture to simplify the development of a new CCAI because different CCAIs often differ only in some components (e.g., slow start and congestion avoidance). For example, Linux adopts a pluggable congestion control architecture. To develop a new CCAI in Linux, CCAI developers only need to implement several Application Programming Interface (API) functions defined in structure `tcp_congestion_ops` and can re-use most of the current TCP code. Specifically, Table I lists all three required API functions in the latest Linux kernel (version 6.9) that each CCAI must implement. In addition, there are several optional API functions that a CCAI may or may not implement.

Because a CCAI adjusts its numerical state variables (e.g., `cwnd` and `ssthresh`) to adapt the TCP throughput to changing network conditions, CCAI properties are usually numerical properties about these numerical state variables, such as the NoMoreThanOne property. The CCAI updates its numerical state variables in response to network events, mainly the arrivals of ACKs at a TCP sender, and thus most CCAI

properties are the expected state of the CCAI in response to ACKs. Especially, many CCAI properties specify the expected final state of the CCAI after a sequence of ACKs (e.g., all ACKs in an RTT in the NoMoreThanOne property) instead of the specific state after each individual ACK. By doing so, the CCAI specifications allow that different CCAIs or different versions of the same CCAI all follow the same long-term requirements (e.g., per RTT) but may flexibly have different short-term behaviors (e.g., per ACK) for flexible implementations. For example, Linux RENO has changed its congestion avoidance code multiple times over the past decades but all following the same NoMoreThanOne property.

B. Related work

The works closely related to ASM have been summarized in Section I. Below we summarize more generally related works.

CCAIs have been traditionally tested and evaluated mainly for the performance (e.g., efficiency and fairness) using flow modeling [28]–[30], network simulations [31], network emulations and experiments [5], [11], [12], [32], [33], and trace analysis [34], [35]. However, none of these methods can verify the correctness of CCAIs. Flow modeling typically studies the abstract behaviors of CCAIs that approximate the long-term behaviors of CCAIs and thus is suitable for approximate long-term performance evaluation but not for exact correctness verification. Network simulations, emulations, experiments, and trace analysis can be used to check the exact behaviors of CCAIs but only in a limited number of network environments, and thus can check to some extent but cannot verify the correctness of CCAIs.

Most formal specification, testing, and verification work on TCP implementations [24], [34], [36]–[41] focuses on the TCP connection management, TCP options, or reliability, but with no or little coverage on CCAIs. Some work [7], [14], [16] detects the bugs or security issues of CCAIs but does not verify the correctness of CCAIs. There is also some work on formal specifications of other transport protocols (e.g., QUIC) [42]. However, they focus on other transport components, such as connection management, with little coverage of CCAIs. Formal methods have also been used for synthesizing congestion control algorithms [19].

There also exists a rich body of work on the verification of general network software. Hyperkernel [43] verifies an operating system kernel using a SMT solver. Symbolic execution is used to generate high-coverage test cases for network protocols [44], [45]. Symbolic execution has been combined with model checking to test OpenFlow applications [26]. Symbolic execution has also been combined with theorem proving to verify network function software [22], [23]. Formal methods [46] have been used to reason and verify the network performance. Static analysis-based techniques [47]–[49] have been used for finding bugs in network protocols. These approaches are usually faster than exhaustive symbolic execution when checking large implementations. However, it is hard for them to detect deep bugs that only emerge after a large number of packets.

Code 1. A function to be verified

```
1 int absolute(int a)
2   if (a >= 0)
3       return a;          // true branch
4   else
5       return -a;         // false branch
```

Code 2. Verification code for Code 1

```
1 void check_absolute( )
2   int a = sym_value(-1000000000, 1000000000);
3   int abs_a = absolute(a);
4   assert(abs_a >= 0);     // check property
```

Code 3. Creating a symbolic value

```
1 int sym_value(int low, int high)
2   return a symbolic integer between low and high
3          created by the symbolic execution engine
```

III. Motivating Examples

A. Exhaustive symbolic execution

Symbolic execution [50] has evolved significantly to become a powerful and popular technique to test and verify software programs in recent years [51]. It can find all possible execution paths of a program, each of which is a possible flow of control of the program. If all possible execution paths can be exhaustively explored to verify that a property holds for every execution path, we can formally prove that the property holds for the program, and such a verification technique is referred to as *exhaustive symbolic execution* [21]–[23].

Code 2 illustrates how to use exhaustive symbolic execution to verify function `absolute()` defined in Code 1. The property to check is that the return value of `absolute()` is always non-negative for any integer argument `a` between -10^9 and 10^9. Code 2 first assigns `a` to a symbolic value by calling function `sym_value()` (line 2). Different from normal concrete values, a *symbolic value* can be intuitively treated as a mathematical symbol representing a set of concrete values. For example, `a` is associated with all possible integers between -10^9 and 10^9. Code 2 then calls `absolute()` with argument `a` (line 3), and finally checks whether the property holds (line 4).

When Code 2 is executed by a symbolic execution engine, such as KLEE [52], a total of two execution paths are explored, and their corresponding path constraints are reported. The path constraint of an execution path describes the set of all possible concrete values leading to the execution path. The two execution paths of Code 2 differ only in the `if` statement of Code 1. One path executes the true branch (i.e., Code2:lines1,2, Code3:lines1,2, Code2:line3, Code1:lines1,2,3, Code2:line4), and the corresponding path constraint is `a>=0` (i.e., condition of Code1:line2). The other path executes the false branch and the corresponding path constraint is `a<0`.

The strength of exhaustive symbolic execution is that the path constraint of each execution path essentially defines an equivalence class of concrete values leading to the same path and thus we only need to exhaustively verify that the property holds for each path (i.e., each equivalence class) instead of each concrete value. For example, we only need to verify the

property for the two execution paths of Code 2, although `a` has a total of $2 \times 10^9 - 1$ different concrete values. Specifically, for the path with constraint `a>=0`, we have `abs_a=a` and thus KLEE determines that `assert(abs_a>=0)` is true; for the other path with constraint `a<0`, we have `abs_a=-a` and thus KLEE determines that `assert(abs_a>=0)` is also true. Thus, the property holds for function `absolute()`.

The scalability of exhaustive symbolic execution heavily depends on the total number of execution paths of the code under test. The higher the number, the poorer the scalability. This is because it takes both time and memory to explore each execution path. Therefore, a scalable verification method should have a bounded number of execution paths (defined in Section IV-C). Intuitively, the upper bound of the number of execution paths is independent of the range of symbolic values. For example, Code 2 has at most two execution paths, and that is independent of the range of `a`.

B. SCCT scalability challenge

SCCT [8] based on exhaustive symbolic execution is an automated code verification method (i.e., Class 3.2) proposed recently. It can automatically prove the correctness of CCAIs, but is not scalable to a large number of ACKs as demonstrated below. To simplify our discussion, let's consider a simplified version of Linux RENO, which is referred to as sRENO. We also consider TCP setting where a TCP receiver sends an ACK for each data packet.

CCAI function to check: We check function `sreno_avoid()` of sRENO shown in Code 4, which updates `cwnd` during the congestion avoidance stage in response to each ACK that acknowledges a new data packet. Variable `cwnd_cnt` keeps track of the number of ACKs.

Property to verify: We verify the NoMoreThanOne property of RENO [1] for all possible `cwnd` values in a range, say [1, `max_cwnd`] packets, where `max_cwnd` is a user-chosen constant.

Verification method: Code 5 illustrates how SCCT checks whether the NoMoreThanOne property holds for `sreno_avoid()` in an RTT. It considers a sRENO sender, whose initial value of `cwnd` is `init_cwnd` (line 3) that is a symbolic value between 1 and `max_cwnd` (line 2). The sRENO sender receives a total of `init_cwnd` number of ACKs from the sRENO receiver in an RTT, and handles these ACKs using a `for` loop (lines 5 and 6). Finally, Code 5 compares the new value of `cwnd` with its initial value and checks the property (line 7). SCCT uses KLEE [52] to run Code 5. If the property holds for all the execution paths explored by KLEE, SCCT reports that the property holds for `sreno_avoid()`. Note that variable `init_cwnd` denotes the initial `cwnd` value just before an RTT, and it is not the initial `cwnd` value when a TCP connection is just established. Also, note that Code 5 sets other variables, such as `cwnd_cnt`, to only a concrete value (line 4) to simplify the discussion of this example. In real-world verification, they should also have symbolic values to cover all possible concrete values.

Scalability challenge: SCCT has poor scalability because each concrete value of `init_cwnd` leads to a different execution path of Code 5 and 4. For example, if `init_cwnd` = 1, the corresponding path runs the `for` loop (Code5:lines5,6) for one iteration to handle one ACK. But if `init_cwnd` = 2, the corresponding path runs the `for` loop for two iterations to handle two ACKs and thus is different. Also, note that Code 4 may be executed differently at the `if` statement (Code4:line4) in different `for` loop iterations depending on the values of `cwnd` and `cwnd_cnt`.

Code 4. A function of sRENO for a single ACK

```
1 int cwnd, cwnd_cnt;       // connection variables
2 void sreno_avoid( )
3   cwnd_cnt += 1;                        // one ACK
4   if (cwnd_cnt >= cwnd)
5     {cwnd_cnt -= cwnd;  cwnd++;}        // update cwnd
```

Code 5. SCCT verification code for sRENO

```
1 void scct_check_sreno_avoid( )
2   int init_cwnd = sym_value(1, max_cwnd); // symbolic
3   cwnd = init_cwnd;                     // initial cwnd
4   cwnd_cnt=0;                           // other initials
5   for (int k=1; k<=init_cwnd; k++)      // multiple ACKs
6     sreno_avoid();                      // handle each ACK
7   assert(cwnd <= init_cwnd+1);          // check property
```

We can see that the total number of execution paths of SCCT is unbounded. Specifically, the number of execution paths of SCCT increases as `max_cwnd` increases, because `max_cwnd` is the total number of possible concrete values of `init_cwnd`. Intuitively, SCCT is not scalable to a large number of ACKs, because `max_cwnd` is also the maximum number of ACKs. In this paper, we propose a new verification method that has only a bounded number of execution paths and thus is more scalable than SCCT.

IV. CCAI Verification Problem

A. Which part of CCAIs to verify?

Same as SCCT, we choose to verify the API functions of Linux CCAIs as described in Section II-A. This is because different CCAIs differ only in these API functions. Also, for a new CCAI, these API functions are possibly more likely to have bugs than the other parts that are shared among all CCAIs and thus have already been extensively tested.

B. What properties of CCAIs to verify?

In this paper, we focus on the multi-ACK properties of CCAIs. For example, the NoMoreThanOne property discussed in the motivating example in Section III-B involves `init_cwnd` ACKs, and `init_cwnd` could be tens of thousands in the current Internet. More properties will be discussed in the evaluation experiments in Section VII.

C. Design goals of the proposed verification method

There are three design goals for our proposed method.

- **Goal 1: Code verification.** It proves the correctness of CCAI code against a multi-ACK property.
- **Goal 2: ACK scalable.** It has a bounded number `N` of execution paths with respect to the number `n` of ACKs in an RTT, and thus is scalable to a large number of ACKs.

- **Goal 3: Little verification expertise.** It does not require substantial verification expertise (e.g., theorem proving and formal specification), which network protocol designers and developers, by and large, lack [27].

Definition: `N` is **bounded** with respect to `n`, if there exists a constant C, such that `N` $< C$ for any `n`; otherwise, `N` is **unbounded**. Intuitively, `N` is bounded with respect to `n`, if an upper bound of `N` is independent of `n`.

None of the current CCAI testing and verification methods achieves all three design goals. Especially, SCCT, a state-of-the-art verification method achieves Goals 1 and 3, but not Goal 2, because it has an unbounded number of execution paths as demonstrated in Section III-B.

V. Our Verification Method

We propose an ACK Scalable Method, called ASM, to verify the API functions of a CCAI against a multi-ACK property. ASM is proposed for the network protocol designers and developers (i.e., users), who should have in-depth CCAI expertise but do not need substantial verification expertise.

ASM users take the following three steps to verify the API functions of a CCAI against a multi-ACK property. **Step 1:** Build the Agg-Alg of the CCAI. **Step 2:** Check the equivalence between the Agg-Alg and the original CCAI. **Step 3:** Check the correctness of the Agg-Alg against the property.

Goal 1 is achieved because the Agg-Alg is equivalent to the original CCAI (Step 2), and thus proving the correctness of the Agg-Alg (Step 3) is equivalent to proving the correctness of the original CCAI. Goal 2 is analytically proved in Section VI. Goal 3 is achieved because ASM uses exhaustive symbolic execution so that ASM users do not need to have substantial verification expertise, such as theorem proving. In addition, ASM uses the same programming language as the original CCAI (i.e., C for Linux kernel) to specify the Agg-Alg and use assertions to specify the multi-ACK properties so that ASM users do not need to learn formal specification languages.

A. Step 1: Building the Agg-Alg for a CCAI

Figure 1 illustrates the difference between a CCAI and its Agg-Alg. The CCAI operates on individual ACKs, including normal ACKs (green blocks in the figure) that acknowledge new data packets and are handled in slow start and congestion avoidance stages, and duplicate ACKs (red blocks in the figure) that are handled in other stages (e.g., fast recovery). Because a CCAI mainly stays in the congestion avoidance stages, most ACKs handled by the CCAI are normal ACKs. The Agg-Alg is designed to operate on the aggregated information of these normal ACKs to be ACK scalable. The information of a sequence of consecutive normal ACKs, up to the current congestion window size, is aggregated (blue blocks in the figure). Intuitively, the aggregated information (each blue block) contains the information of at most all the normal ACKs in an RTT. Note that the RTT boundaries are shown in the figure just to help the readers understand the diagram, and we do not need the RTT boundaries when aggregating ACKs.

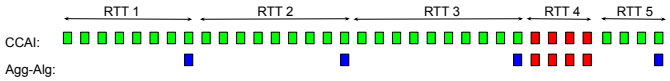

Fig. 1. Operations of CCAI and Agg-Alg in several RTTs. Each green block is a normal ACK that acknowledges new data packets, each red block is a duplicate ACK, and each blue block represents the aggregated information of the corresponding normal ACKs.

Code 6. sRENO Agg-Alg for a sequence of n ACKs

```
1 int cwnd, cwnd_cnt;      // connection variables
2 void sreno_avoid_agg(int n)
3   assert(1 <= n <= cwnd);        //upto the ACKs in a RTT
4   cwnd_cnt += n;                 //n ACKs
5   if (cwnd_cnt >= cwnd)
6     {cwnd_cnt -= cwnd;   cwnd++;}//update cwnd
```

An ASM user builds the Agg-Alg for a CCAI by re-implementing the frequent API functions and keeping the occasional API functions of the CCAI. There are two types of API functions. 1) *Frequent API Functions*: An API function is frequent if it handles normal ACKs, because it is triggered frequently in response to all the normal ACKs. Among the required API functions listed in Table I, `cong_avoid()` is a frequent API function, because it is called repeatedly for each normal ACK during the slow start and congestion avoidance stages. 2) *Occasional API Functions*: An API function is occasional if it handles duplicate ACKs, because it is triggered only occasionally in special cases. Among the required API functions, `ssthresh()` and `undo_cwnd()` are occasional functions. `ssthresh()` is called only when a congestion event (e.g., three duplicate ACKs) is detected and then CCAI needs to reduce `cwnd`. `undo_cwnd()` is called only when a CCAI discovers that a congestion event was mistakenly detected and `cwnd` was unnecessarily reduced.

The aggregated information of a sequence of normal ACKs depends on the CCAI. It may contain one or more of the following variables depending on the CCAI: number of ACKs, arrival time of the last ACK, average packet delay, maximum or minimum packet delay, and other possible aggregated metrics. For example, the aggregated information of Linux RENO contains just the number of ACKs, and the aggregated information of Linux CUBIC contains the number of ACKs and the arrival time of the last ACK.

As an illustrating example, let's consider sRENO described in Section III-B and consider `sreno_avoid()` shown in Code 4 as its implementation of frequent API function `cong_avoid()`. The sRENO Agg-Alg re-implements this API function as `sreno_avoid_agg()` shown in Code 6.

There are several differences between the original CCAI `sreno_avoid()` and the new Agg-Alg `sreno_avoid_agg()`. 1) They are designed to handle different numbers of ACKs. For example, to handle a sequence of `n` normal ACKs, `sreno_avoid()` is called `n` times once for each ACK, whereas `sreno_avoid_agg()` is called only once for the whole sequence. 2) They have different arguments. The original `sreno_avoid()` has arguments (if any) describing

the information of an individual ACK, whereas the new `sreno_avoid_agg()` has an argument, `n`, which describes the aggregated information of a sequence of `n` ACKs. 3) The new `sreno_avoid_agg()` may not have any loop, such as `for`, `while`, and `repeat`, depending on `n`. This programming constraint is necessary to achieve a bounded number of execution paths (i.e., Goal 2).

Functions `sreno_avoid()` and `sreno_avoid_agg()` are *equivalent in that they have the same final values of CCAI state variables*, such as `cwnd`, starting from any initial values of these state variables for the same sequence of `n` normal ACKs in an RTT. The range of `n` depends on `cwnd`. For example, if there are at most `cwnd` data packets in an RTT and there is an ACK for each data packet, there are at most `cwnd` ACKs in an RTT. Then `n` could be any number between 1 and `cwnd`, which is checked at line 3 of `sreno_avoid_agg()`.

B. Step 2: Checking equivalence

For each frequent API function, ASM checks the equivalence between the original CCAI implementation (denoted by `f()`) and the Agg-Alg implementation (denoted by `f_agg()`) developed by an ASM user. For example, if `f()` is `sreno_avoid()`, then `f_agg()` is `sreno_avoid_agg()`. Specifically, the ASM user provides both `f()` and `f_agg()`, and then ASM runs `asm_check_equ()` shown in Code 7 to check whether they have the same final value of `cwnd` starting from any `init_cwnd` after `n` ACKs in an RTT. The equivalence is checked for all possible values of `init_cwnd` in [1, `max_cwnd`] and all possible values of `n` in [1, `init_cwnd`].

The challenge is that `asm_check_equ()` should have a bounded number of execution paths to achieve Goal 2. As a result, we may not call `f()` for `n` times (i.e., a for loop) when finding the final `cwnd` value of `f()` after `n` ACKs.

We propose to address this challenge using proof by mathematical induction as illustrated in Code 7, which has two steps. 1) The base step (lines 3 to 6) checks the equivalence between `f()` and `f_agg()` for only 1 ACK. 2) The induction step (lines 7 to 11) assumes that they are equivalent for `n-1` ACKs, and then calculates the final `cwnd` value of `f()` after `n` ACKs (line 10) by first calling `f_agg()` once to handle the first `n-1` ACKs and then calling `f()` once to handle the last ACK. By doing so, our proposed `asm_check_equ()` does not have any for loop and then has a bounded number of execution paths.

C. Step 3: Checking property

At this step, ASM runs a verification code written by an ASM user to check whether a property holds for the Agg-Alg. Because Step 2 already proves that the Agg-Alg is equivalent to the original CCAI, if a property holds for the Agg-Alg then it holds for the original CCAI (i.e., Goal 1).

A verification code checks the behavior of the API functions of a CCAI in response to a sequence of ACKs according to the property. ASM verification code is very similar to that of SCCT. A general SCCT verification code `scct_check_prop()` is illustrated in Code 8 that checks `f()` for a property involving `n` ACKs, and a general ASM verification code `asm_check_prop()` is illustrated in Code 9 that checks `f_agg()`. They may call other API functions based on the property to check. Their difference is that the SCCT code calls `f()` for `n` times (Code8:line3) and the ASM code calls `f_agg()` only once (Code9:line3). As a result, the SCCT code has an unbounded number of execution paths whereas the ASM code has a bounded number of execution paths.

As an example, a specific SCCT verification code is Code 5 discussed in Section III-B, which checks `sreno_avoid()` against the NoMoreThanOne property involving `init_cwnd` ACKs. A specific ASM code is shown in Code 10, which checks `sreno_avoid_agg()` for the same property. Again, their difference is that the SCCT code calls `sreno_avoid()` for `init_cwnd` times (Code5:lines5,6) and the ASM code calls `sreno_avoid_agg()` only once (Code10:line5).

Note that to reduce the required verification expertise (i.e., Goal 3), the verification code is written in the same programming language as the original CCAI and the property is specified using assertions that ASM users are familiar with.

Code 7. ASM Step 2: Check equivalence

```
1  void asm_check_equ(f(), f_agg())
2    int init_cwnd = sym_value(1, max_cwnd);
3    cwnd = init_cwnd;     // base step
4    check if the following two have the same final cwnd
5    1) call f() for 1 ACK
6    2) call f_agg() for 1 ACK
7    cwnd = init_cwnd;     // induction step
8    n = sym_value(2, init_cwnd);
9    check if the following two have the same final cwnd
10   1) call f_agg() for n-1 ACKs then call f() for 1 ACK
11   2) call f_agg() for n ACKs
```

Code 8. SCCT code to check property

```
1  void scct_check_prop( )
2    initialize variables, such as init_cwnd and RTT
3    call f() n times to handle n ACKs
4    check whether the property holds
```

Code 9. ASM Step 3: Check property

```
1  void asm_check_prop( )
2    initialize variables, such as init_cwnd and RTT
3    call f_agg() once to handle n ACKs
4    check whether the property holds
```

Code 10. ASM verification code for sRENO Agg-Alg

```
1  void asm_check_sreno_avoid_agg( )
2    int init_cwnd = sym_value(1, max_cwnd); // symbolic
3    cwnd = init_cwnd;                  // initial cwnd
4    cwnd_cnt=0;                        // other initials
5    sreno_avoid_agg(init_cwnd);        // handle ACKs
6    assert(cwnd <= init_cwnd+1);       // check property
```

VI. Scalability Analysis

In this section, we analytically study the scalability of ASM and SCCT. We consider a general type of properties that compare the `cwnd` before and after calling frequent API function `cong_avoid()` to handle a total of $n \in [1, \texttt{init\_cwnd}]$ ACKs in an RTT, where $\texttt{init\_cwnd} \in [1, \texttt{max\_cwnd}]$. For example, NoMoreThanOne belongs to this type, and we will evaluate several properties of Linux RENO and CUBIC also belonging to this type in Section VII.

Let `f()` denote the original CCAI implementation of API `cong_avoid()`, for example, `sreno_avoid()` in Code 4. Let $N_{f()}$ denote the maximum number of execution paths of `f()`, which is bounded because it handles only one ACK. For example, the $N_{f()}$ of `sreno_avoid()` is no more than 2 because there is only one `if` statement (line 4).

Let `f_agg()` denote the Agg-Alg implementation of API `cong_avoid()`, for example, `sreno_avoid_agg()` in Code 6. Let $N_{f\_agg()}$ denote the maximum number of execution paths of `f_agg()`, which is bounded by design (Section V-A). For example, the $N_{f\_agg()}$ of `sreno_avoid_agg()` is no more than 8 because there are only three `if` statements (line 5 and also `assert` at line 3 has two `if` statements).

Theorem 1. *The number of execution paths of ASM, denoted by N_{ASM}, is bounded with respect to `max_cwnd`.*

Proof: N_{ASM} is the sum of the numbers of execution paths of ASM at Steps 2 and 3, denoted by N_{step2} and N_{step3}, respectively. That is, $N_{ASM} = N_{step2} + N_{step3}$.

At Step 2, ASM checks the equivalence between `f()` and `f_agg()` using `asm_check_equ()` in Code 7. We have $N_{step2} \leq N_{f()}^2 \times N_{f\_agg()}^3 \times c_{step2}$ where c_{step2} is a constant depending on the number of `if` statements in Code 7, because `f()` is called twice and `f_agg()` is called three times. Also, because both $N_{f()}$ and $N_{f\_agg()}$ are bounded with respect to `max_cwnd`, N_{step2} is bounded. That is, N_{step2} is $\Theta(1)$.

At Step 3, ASM checks the property of `f_agg()` using `asm_check_prop()` in Code 9. We have $N_{step3} \leq N_{f\_agg()} \times c_{step3}$ where c_{step3} is a constant depending on the number of `if` statements in Code 9, because `f_agg()` is called once. Thus, N_{step3} is bounded with respect to `max_cwnd` and is $\Theta(1)$.

Overall, N_{ASM} is bounded and is $\Theta(1)$. □

Theorem 2. *The number of execution paths of SCCT, denoted by N_{SCCT}, is unbounded with respect to `max_cwnd`.*

Proof: SCCT checks the property of `f()` using `scct_check_prop()` in Code 8. N_{SCCT} has a lower bound $\Omega(\text{max\_cwnd})$ and an upper bound $\mathcal{O}({N_{f()}}^{\text{max\_cwnd}})$, because `f()` is called `n` times and there are a total of `max_cwnd` possible values of `n`. That is, in the best case (e.g., when $N_{f()}$ is 1), N_{SCCT} increases linearly as `max_cwnd`, and in the worst case, N_{SCCT} increases exponentially as `max_cwnd`. Therefore, N_{SCCT} is unbounded. □

We measure and show the specific values of both N_{ASM} and N_{SCCT} for Linux RENO and CUBIC in Section VII.

VII. Experiments

A. Overview

We design and conduct experiments to answer the following research questions.

- **RQ1:** Can ASM verify the code of loss-based CCAIs?
- **RQ2:** Is ASM ACK-scalable?

CCAI to verify: To answer RQ1, we consider two real-world loss-based CCAIs: Linux RENO and Linux CUBIC, the previous and current default CCAI of Linux and Android devices, respectively. Both had or have been used by billions of users worldwide. We consider their API functions [9], [10] in the latest Linux kernel (version 6.9).

Verification methods: To answer RQ2, we run the following two verification methods. 1) SCCT [8] belongs to Class 3.2 (automated code verification). It is a state-of-the-art tool to verify real-world CCAIs and has successfully detected multiple CCAI bugs. 2) Our proposed ASM is developed by extending the open-source SCCT. The source code of ASM and all the experiments in this paper is available at https://github.com/verifiabletcp/asm. We run both SCCT and ASM using a popular symbolic execution engine - KLEE [52].

Properties to check: We consider only multi-ACK properties involving a large number of ACKs (e.g., thousands), because SCCT can already check the properties involving a small number of ACKs (e.g., several) [8]. In the experiments, both SCCT and ASM check properties with a symbolic number of ACKs.

Network environments: We check CCAIs mainly in the network environments with `cwnd` up to the order of 10^4 packets and the RTT duration `rtt_duration` up to `max_rtt_duration` = 200 ms, which cover common network environments of the current Internet (e.g., 1 Kbps to 10 Gbps of bandwidth, 1 ms to 200 ms of RTT). Both SCCT and ASM check CCAIs in networks with symbolic bandwidth and symbolic RTTs.

Machine used: We run all our experiments on a DELL Precision 3630 Tower with 128 GByte memory and Intel Core i7-8700K CPU at 3.70 GHz x 12.

B. Case Study 1: Linux RENO

1) Introduction: We demonstrate the applicability and evaluate the performance of ASM using a representative loss-based CCAI, Linux RENO, which was the default CCAI of Linux. It had been used by billions of Linux and Android devices worldwide and is still used by a large number of devices.

2) Properties to check: We check two fundamental properties of RENO [1] about the maximum increment of frequent API function `cong_avoid()`. 1) NoMoreThanDouble: It should not increment its `cwnd` more than double in an RTT in the slow start stage, 2) NoMoreThanOne in the congestion avoidance stage. These two properties involve two input variables: `init_cwnd` (i.e., the initial `cwnd` before an RTT) and `ssthresh`, which determine the stage of RENO. If `init_cwnd` < `ssthresh`, it is the slow start stage; otherwise, it is the congestion avoidance stage. These two maximum increment properties are fundamental to RENO. However, these two properties have not been verified for Linux RENO, to the best of our knowledge, for common network environments of the current Internet, because they involve `init_cwnd` number of ACKs that could be tens of thousands in the current Internet.

We write both ASM and SCCT verification code in language C that is the programming language of Linux RENO, and specify the properties using assertions that Linux network developers are familiar with. The code checks the properties using symbolic values to cover all possible `init_cwnd` in [1, `max_cwnd`] and all possible `ssthresh` in [1, `max_ssthresh`]. In

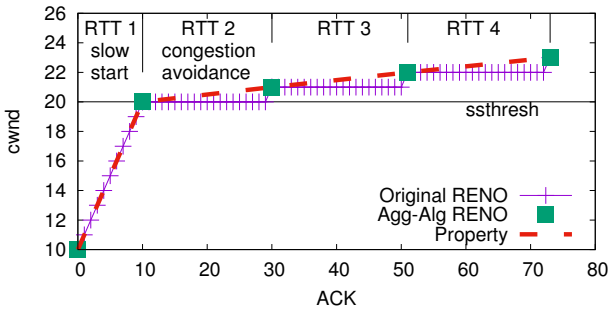

Fig. 2. An example with `ssthresh`=20 and `init_cwnd`=10. The RENO Agg-Alg generates the same final `cwnd` as the original RENO in each RTT, and the maximum increment properties hold (`cwnd` $\leq$ property) in each RTT.

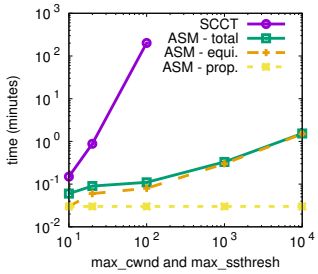

Fig. 3. The verification time of ASM (total = prove equivalent + prove property alone) is several orders of magnitude shorter than SCCT.

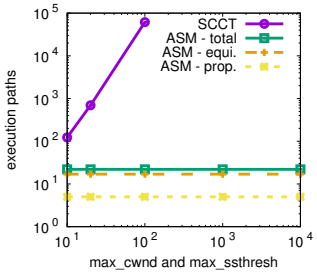

Fig. 4. The number of execution paths of ASM is bounded, whereas that of SCCT is unbounded (increases exponentially).

addition, it checks all possible values of other related variables, such as `cwnd_cnt` in Code 4 and 6.

3) Original Linux RENO implementation: Because the Agg-Alg of CCAI differs from the original CCAI only in the implementations of the frequent API functions, let's consider the only frequent required API function `cong_avoid()`. The original RENO implementation of API `cong_avoid()` is function `tcp_reno_cong_avoid()` in Linux kernel source file `tcp_input.c` [10]. It is called for each received ACK and has one major argument: `acked` that is the number of data packets acknowledged by the ACK.

4) RENO Agg-Alg: We develop the Agg-Alg of Linux RENO by re-implementing API function `cong_avoid()`. The Agg-Alg implementation has the same final `cwnd` as the original implementation `tcp_reno_cong_avoid()` for a sequence of $1 \leq$ `n` $\leq$ `init_cwnd` ACKs. Intuitively, the RENO Agg-Alg captures the behavior of the original RENO for a sequence of ACKs in an RTT. We capture the aggregated information of a sequence of ACKs for RENO using variable `total_acked`, which is the total number of data packets acknowledged by all the ACKs in the sequence. The Agg-Alg implementation directly calculates the final `cwnd` using argument `total_acked`.

To help the readers better understand the relation between the original RENO function `tcp_reno_cong_avoid()` and the RENO Agg-Alg, let's consider an example where `init_cwnd` of the first RTT is 10 packets and `ssthresh` is 20 packets. Figure 2 plots the new `cwnd` values calculated by both functions for the first 4 RTTs involving a total of 73 ACKs. To handle these ACKs, the original RENO function is called 73 times, once for each ACK, and the RENO Agg-Alg is called only 4 times, once for each RTT. The RENO Agg-Alg generates the same final `cwnd` as the original RENO at the end of each RTT. Below we will prove their equivalence for a large range of `init_cwnd` and `ssthresh` and for every ACK instead of just the last ACK in each RTT in Figure 2.

Also to help the readers understand the maximum increment properties of RENO, Figure 2 plots the maximum increment line for each RTT and the properties hold if `cwnd` is always no higher than the corresponding maximum increment line at the end of every RTT. We can see that the properties hold for this example. Below we will check these properties for a large range of `init_cwnd` and `ssthresh`.

We develop the RENO Agg-Alg according to the RENO specification [1] and the original Linux RENO code [10]. The RENO Agg-Alg is written in language C that is the same as the original RENO. We notice that the developed RENO Agg-Alg is almost the same as the original RENO implementation `tcp_reno_cong_avoid()`, except replacing argument `acked` with `total_acked`. That is, we do not need to develop the RENO Agg-Alg from scratch, instead, we only need to make some simple changes to the original RENO code.

5) Check properties and equivalence: We use both SCCT and ASM to verify the property with `max_cwnd` and `max_ssthresh` up to 10^4 packets. For example, if both are 10^4, they check whether the properties hold for all $10^4 \times 10^4 = 10^8$ possible combinations of `init_cwnd` and `ssthresh`, and ASM also checks the equivalence between the original RENO and Agg-Alg for all these 10^8 combinations and all possible 10^4 ACKs. Figures 3 and 4 show their verification times and numbers of execution paths. ASM quickly verifies that the properties hold for Linux RENO for all 10^8 combinations of `init_cwnd` and `ssthresh`, but SCCT is unable to complete the verification within 1000 minutes.

Scalability: Figure 3 shows that *ASM is several orders of magnitude faster than SCCT*. For example, when `max_cwnd` and `max_ssthresh` are 100 packets, it takes SCCT about 200 minutes to verify the properties, whereas it takes ASM about 5 seconds to verify the equivalence, about 1 second to verify the properties alone, and about 6 seconds in total. Even when `max_cwnd` and `max_ssthresh` are 10000 packets (i.e., 10000 ACKs), it takes ASM only 1.5 minutes to verify the equivalence and properties. The reason for the different scalability of ASM and SCCT is that *the number of ASM execution paths is bounded whereas that of ACCT is unbounded* as proved by Theorems 1 and 2 and confirmed in Figure 4. Note that Figure 4 shows that the number of SCCT execution paths increases exponentially. The reason that the verification time of ASM still increases with a bounded number of execution paths is that the constraints of symbolic execution have larger ranges of variables and then take a longer time to solve.

C. Case Study 2: Linux CUBIC

1) Introduction: We demonstrate the applicability and evaluate the performance of ASM using another representative loss-based CCAI, Linux CUBIC, which is currently being used by billions of Linux and Android devices worldwide. Linux CUBIC has replaced Linux RENO as the default congestion control approach of all Linux devices since around 2006. Another reason that we select CUBIC is that its behavior is more complicated than RENO in that it depends on not only the number of ACKs but also their arrival times.

2) Properties to check: We check two maximum increment properties. 1) MaxCwndIncrement: CUBIC should not increment its `cwnd` more than 1.5 times in an RTT in the congestion avoidance stage. 2) MaxTargetIncrement: CUBIC should not increment its `target` more than 1.5 times in an RTT in the congestion avoidance stage. Variable `target` is the expected `cwnd` of CUBIC after one RTT. The MaxTargetIncrement is required in the CUBIC specification [2] and implies the MaxCwndIncrement property. These two properties involve two input variables: `origin` that is the origin point of the cubic function that CUBIC follows to increment `cwnd`, and `rtt_duration` that is the duration of an RTT. We check these properties using symbolic values to cover all possible `origin` in [2, `max_origin`] packets and all possible `rtt_duration` in [1, `max_rtt_duration`] ms.

3) Original Linux CUBIC implementation: The original CUBIC implementation of frequent API `cong_avoid()` is function `cubictcp_cong_avoid()` in Linux kernel source file `tcp_cubic.c` [9]. It is called for each received ACK and has one major argument `acked`.

4) CUBIC Agg-Alg: The general behavior of Linux CUBIC is complicated because it may bypass some computational-intensive code based on some conditions of the ACK arrival times to reduce the CPU load. As a result, the behavior of CUBIC depends on the arrival time of each individual ACK, and thus it is challenging to develop the Agg-Alg of CUBIC for all possible cases. In this paper, we consider a case where the behavior of CUBIC depends on only the arrival time of the last ACK in an RTT instead of the arrival times of all the ACKs in the RTT. In this case, the aggregated information of a sequence of ACKs can be captured by two variables: variable `total_acked`, and variable `last_ack_time` that is the arrival time of the last ACK in the sequence. Specifically, we develop the Agg-Alg of CUBIC by re-implementing API function `cong_avoid()`, and the Agg-Alg implementation has the same final `cwnd` value as the original `cubictcp_cong_avoid()` for a sequence of $1 \leq$ `n` $\leq$ `init_cwnd` ACKs.

5) Check properties and equivalence: We use both SCCT and ASM to check the two maximum increment properties of CUBIC with `max_rtt_duration` = 200 ms and `max_origin` up to 10^4 packets. We check the behavior of CUBIC only in the first RTT after a loss event, because the first RTT has the maximum increment among all the RTTs in the concave region [2] of CUBIC. ASM verifies that the properties hold for Linux CUBIC in all checked network environments, but SCCT is unable to complete the verification within 1000 minutes.

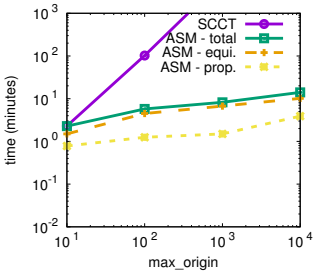

Fig. 5. The verification time of ASM (total = prove equivalent + prove property alone) is several orders of magnitude shorter than SCCT.

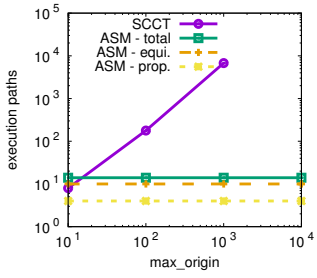

Fig. 6. The number of branches of ASM is bounded, whereas that of SCCT is unbounded (increases exponentially).

Scalability: Figures 5 and 6 show their verification times and numbers of execution paths. We can see that ASM is several orders of magnitude faster than SCCT. For example, when `max_origin` is 100 packets, it takes SCCT about 100 minutes to verify the properties, whereas it takes ASM about 4.5 minutes to verify the equivalence, about 1.3 minutes to verify the properties alone, and about 5.8 minutes in total. Even when `max_origin` is 10000 packets (i.e., 10000 ACKs), it still takes ASM only about 14 minutes to verify the properties. The fundamental reason for the different scalability of ASM and SCCT is that the number of ASM execution paths is bounded whereas that of SCCT is unbounded as proved in Theorems 1 and 2 and confirmed in Figure 6.

Extreme environments: We also run additional experiments to check these two properties in extreme network environments with RTTs up to 2000 ms. With such a wider range of RTTs, both ASM and SCCT take longer to check the properties, because the symbolic execution constraints have larger ranges of variables and then take longer to solve. Both ASM and SCCT report violations of the MaxTargetIncrement property in network environments with very long RTTs. This is because Linux CUBIC does not limit `target` as required in the CUBIC specification [2], and then `target` may increment more than 1.5 times in a very long RTT.

The reported violation of the MaxTargetIncrement property is consistent with the finding in a previous work [7]. Different from the previous work which reports only some specific network environments where the property is violated, ASM reports the ranges of all the network environments where the property is violated. Also different from the previous work, ASM verifies that the MaxCwndIncrement property still holds in all these network environments. Therefore, the final `cwnd` increment of Linux CUBIC still follows the requirement of the CUBIC specification [2], although the intermediate `target` calculation does not follow exactly the CUBIC specification.

D. Case Study 3: Both Linux RENO and CUBIC

1) Introduction: Different from the first two case studies, each involving only one CCAI, this case study involves two CCAIs. Because CCAIs are designed to compete and coexist with other CCAIs, some properties involve two CCAIs.

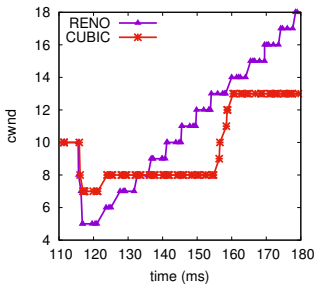

Fig. 7. Linux CUBIC bug: The average `cwnd` of current CUBIC is lower than that of RENO.

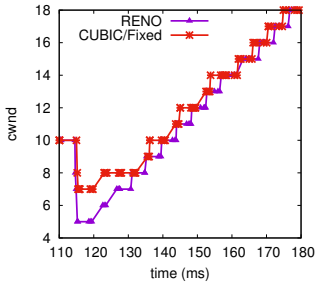

Fig. 8. Fixed Linux CUBIC: CUBIC archives at least the same average throughput as RENO.

2) Property to check: We check the friendliness property of CUBIC [2] that CUBIC should achieve at least the same average throughput as RENO in network environments with small bandwidth-delay products where RENO performs well. According to the network environments discussed in the CUBIC specification [2], we consider networks with `congestion_cwnd` $\in$ [2, 38] packets that is the `cwnd` of a TCP flow right before a congestion event, and with `rtt_duration` $\in$ [1, 100] ms. Both RENO and CUBIC start from a congestion event with the same initial `cwnd` = `congestion_cwnd`. Both call their implementations of API function `ssthresh()` to reduce their `cwnd` in response to the congestion event, and then call their implementations of API function `cong_avoid()` to increment their `cwnd` in response to the ACKs in `R` RTTs. Finally, the code checks whether the average `cwnd` of CUBIC in `R` RTTs is always higher than or equal to that of RENO.

3) Check property: ASM is faster than SCCT. For example, ASM takes 1.3 minutes and SCCT takes 38.2 minutes to check the property with `R` = 2 RTTs in all the above network environments. As expected, both ASM and SCCT take longer to check the property as `R` increases.

Both ASM and SCCT detect violations of the friendliness property. Figure 7 shows the experiment results of one reported violation with `congestion_cwnd` = 10 packets and `rtt_duration` = 4 ms. The results are obtained by running a Mininet experiment where both RENO and CUBIC start with a loss event. CUBIC achieves a lower average `cwnd` than RENO and thus violates the friendliness property. By investigating the Linux CUBIC code [9], we find that this violation is due to three bugs. First, Linux CUBIC may mistakenly not update `cwnd` for several RTTs when the RTT is very short. Second, it does not match the RENO `cwnd` increment once `cwnd` reaches `congestion_cwnd` as described in the latest CUBIC specification [2]. Third, it mistakenly emulates the current `cwnd` of RENO instead of one RTT in the future. We have reported all three bugs [53] to and have been confirmed by the Linux maintainers. We have also submitted patches to fix these bugs and the experiment results of our fixed CUBIC are shown in Figure 8. The fixed CUBIC achieves a similar average `cwnd` as RENO, and has better performance than the original CUBIC.

E. Discussions and Limitations

RQ1: Yes, we have developed the Agg-Algs for two representative loss-based CCAIs, Linux RENO and CUBIC, and checked the correctness of their code using ASM.

RQ2: Yes, ASM is ACK scalable (up to 10000 ACKs in experiments). In the experiments, the numbers of execution paths of ASM are bounded with respect to the number of ACKs in an RTT whereas those of SCCT are unbounded, and as a result, the verification times of ASM are several orders of magnitude shorter than those of SCCT. However, ASM is not necessarily faster than SCCT in detecting bugs.

Can ASM be applied to delay-based CCAIs? ASM works for a delay-based CCAI, if it has an equivalent Agg-Alg. Intuitively, if a delay-based CCAI operates on aggregated ACK information once or several times per RTT, it has an equivalent Agg-Alg. For example, Linux VEGAS [54] adjusts `cwnd` once per RTT using the minimum RTT sample observed in the last RTT, and thus it can be verified using ASM. However, most delay-based Linux CCAIs, such as BBR [3], do not have an equivalent Agg-Alg, because they operate on individual ACK information and may change their `cwnd` value or pacing rate after each ACK. In our future work, we plan to slightly modify these delay-based Linux CCAIs to operate on aggregated ACK information several times per RTT so that they have an Agg-Alg. For example, a modified BBR estimates the bottleneck bandwidth several times per RTT instead of per ACK, and then adjusts its pacing rate and `cwnd` several times per RTT instead of per ACK. The concept of coarse-grained congestion control algorithms using aggregated ACK information has recently garnered significant interest [55], [18], [19]. We anticipate that this concept will become increasingly important and popular, because coarse-grained algorithms can maintain low CPU overhead in ever-increasing high-speed networks while delivering throughput comparable to fine-grained algorithms using per-ACK information [55], and because their correctness can be more scalably verified.

What are other limitations of ASM? ASM is suitable for verifying the multi-ACK properties that describe the behaviors of a CCAI in response to the ACKs in an RTT. For example, all the properties of Linux RENO and CUBIC checked in case studies 1 and 2 involve the ACKs only in a single RTT. However, ASM may not scale well with multi-RTT properties that describe the behaviors of a CCAI in response to the ACKs in multiple RTTs, as the friendliness property in case study 3. One approach to scalably verifying the multi-RTT properties is to develop an Agg-Alg that captures the behavior of a CCAI over multiple RTTs instead of only one RTT as in ASM.

VIII. Conclusion

In this paper, we have proposed ASM to verify the correctness of the actual code of CCAIs for multi-ACK properties. Our experiments on two widely used real-world loss-based CCAIs show that ASM significantly reduces verification time and requires little verification expertise, though it demands substantial CCAI expertise.

REFERENCES

[1] M. Allman, V. Paxson, and E. Blanton, "TCP congestion control," *RFC 5681*, September 2009.

[2] L. Xu, S. Ha, I. Rhee, V. Goel, and L. Eggert, "CUBIC for fast and long-distance networks," *IETF RFC 9438*, August 2023, https://www.rfc-editor.org/rfc/rfc9438.

[3] N. Cardwell, Y. Cheng, C. Gunn, S. Yeganeh, and V. Jacobson, "BBR: Congestion-based congestion control," *Coomunications of the ACM*, vol. 60, no. 2, pp. 58–66, Feb. 2017.

[4] P. Yang, J. Shao, W. Luo, L. Xu, J. Deogun, and Y. Lu, "TCP congestion avoidance algorithm identification," *IEEE Transactions on Networking*, vol. 22, no. 4, pp. 1311–1324, Aug. 2014.

[5] N. Cardwell, Y. Cheng, L. Brakmo, M. Mathis, B. Raghavan, N. Dukkipati, H. Chu, A. Terzis, and T. Herbert, "PacketDrill: Scriptable network stack testing, from sockets to packets," in *Proceedings of USENIX ATC*, San Jose, CA, June 2013, pp. 213–218.

[6] P. McManus, "Thanks Google for open source TCP fix," September 2015, http://bitsup.blogspot.com/2015/09/thanks-google-tcp-team-for-open-source.html.

[7] W. Sun, L. Xu, S. Elbaum, and D. Zhao, "Model-agnostic and efficient exploration of numerical state space of real-world TCP congestion control implementations," in *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, Feb. 2019, pp. 719–734.

[8] W. Sun, L. Xu, and S. Elbaum, "Scalably testing congestion control algorithms of real-world TCP implementations," in *Proceedings of IEEE ICC*, Kansas City, MO, May 2018, pp. 1–6.

[9] Linux CUBIC Source Code in Latest Kernel, https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/net/ipv4/tcp_cubic.c.

[10] Linux RENO Source Code in Latest Kernel, https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/net/ipv4/tcp_cong.c.

[11] S. Ha, L. Le, I. Rhee, and L. Xu, "Impact of background traffic on performance of high-speed TCP variant protocols," *Computer Networks*, vol. 51, no. 7, pp. 1748–1762, May 2007.

[12] Y. Li, D. Leith, and R. Shorten, "Experimental evaluation of high-speed congestion control protocols," *IEEE/ACM Transactions on Networking*, vol. 15, no. 5, pp. 1109–1122, October 2007.

[13] W. Sun, L. Xu, and S. Elbaum, "Limitations of emulating realistic network environments for correctness testing of internet applications," in *Proceedings of IEEE ICC*, Kansas City, MO, May 2018, pp. 1–6.

[14] S. Jero, E. Hoque, D. Choffnes, A. Mislove, and C. Nita-Rotaru, "Automated attack discovery in TCP congestion control using a model-guided approach," in *Proceedings of Network and Distributed Systems Security (NDSS)*, San Diego, CA, Feb. 2018.

[15] D. Ray and S. Seshan, "Cc-fuzz: Genetic algorithm-based fuzzing for stress testing congestion control algorithms," in *Proceedings of ACM Workshop on Hot Topics in Networks (HotNets)*, Austin, TX, USA, November 2022, pp. 1–7.

[16] N. Kothari, R. Mahajan, T. Millstein, R. Govindan, and M. Musuvathi, "Finding protocol manipulation attacks," in *Proceedings of ACM SIGCOMM*, Toronto, Canada, August 2011.

[17] M. Vu, P. Ha, and L. Xu, "Efficient correctness testing of Linux network stack under packet dynamics," in *Proceedings of IEEE International Conference on Communications (ICC)*, Ireland, Jun. 2020.

[18] V. Arun, M. T. Arashloo, A. Saeed, M. Alizadeh, and H. Balakrishnan, "Toward formally verifying congestion control behavior," in *Proceedings of ACM SIGCOMM*, Aug. 2021, pp. 1–16.

[19] A. Agarwal, V. Arun, D. Ray, R. Martins, and S. Seshan, "Towards provably performant congestion control," in *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Santa Clara, CA, Apr. 2024, pp. 951–978.

[20] The Coq proof assistant, https://coq.inria.fr/.

[21] K. Zhang, D. Zhuo, A. Akella, A. Krishnamurthy, and X. Wang, "Automated verification of customizable middlebox properties with Gravel," in *Proceedings of USENIX NSDI*, CA, Feb. 2020, pp. 221–239.

[22] A. Zaostrovnykh, S. Pirelli, R. Iyer, M. Rizzo, L. Pedrosa, K. Argyraki, and G. Candea, "Verifying software network functions with no verification expertise," in *Proceedings of ACM SOSP*, Canada, Oct. 2019.

[23] A. Zaostrovnykh, S. Pirelli, L. Pedrosa, K. Argyraki, and G. Candea, "A formally verified NAT," in *Proceedings of ACM SIGCOMM*, 2017.

[24] M. Musuvathi and D. Engler, "Model checking large network protocol implementations," in *Proceedings of USENIX NSDI*, San Francisco, CA, March 2004.

[25] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou, "MoDist: Transparent model checking of unmodified distributed systems," in *Proceedings of USENIX NSDI*, Boston, MA, April 2009.

[26] M. Canini, D. Venzano, P. Peresini, D. Kostic, and J. Rexford, "A NICE way to test OpenFlow applications," in *Proceedings of USENIX NSDI*, San Jose, CA, April 2012.

[27] R. Jhala, R. Majumdar, R. Alur, A. Datta, D. Jackson, S. Krishnamurthi, J. Regehr, N. Shankar, and C. Tinelli, "NSF workshop on formal methods: Future directions & transition to practice," NSF, Tech. Rep., 2012.

[28] J. Padhye, V. Firoiu, D. Towsley, and J. Kursoe, "Modeling TCP throughput: A simple model and its empirical validation," in *Proceedings of the ACM SIGCOMM*, 1998, pp. 303–314.

[29] F. Kelly, "Mathematical modelling of the Internet," in *Mathematics Unlimited - 2001 and Beyond*, B. Engquist and W. Schmid, Eds. Springer, 2001, pp. 685 – 702.

[30] D. Chiu and R. Jain, "Analysis of the increase/decrease algorithms for congestion avoidance in computer networks," *Journal of Computer Networks and ISDN*, vol. 17, no. 1, pp. 1–14, June 1989.

[31] Network Simulator 3, https://www.nsnam.org/.

[32] F. Yan, J. Ma, G. Hill, D. Raghavan, R. Wahby, P. Levis, and K. Winstein, "Pantheon: the training ground for Internet congestion-control research," in *Proceedings of USENIX ATC*, Boston, MA, Jul. 2018.

[33] J. Padhye and S. Floyd, "On inferring TCP behavior," in *Proceedings of ACM SIGCOMM*, San Diego, CA, August 2001.

[34] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, and K. Wansbrough, "Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP, and sockets," in *Proceedings of ACM SIGCOMM*, Philadelphia, PA, August 2005, pp. 265–276.

[35] V. Paxson, "Automated packet trace analysis of TCP implementations," in *Proceedings of ACM SIGCOMM*, Cannes, France, September 1997.

[36] M. Hippel, C. Vick, S. Tripakis, and C. Nita-Rotaru, "Automated attacker synthesis for distributed protocols," in *Proceeding of International Conference on Computer Safety, Reliability, and Security (SAFECOMP)*, Lisbon, Portugal, Sep. 2020, pp. 133–149.

[37] S. Bishop, M. Fairbairn, H. Mehnert, M. Norrish, T. Ridge, P. Sewell, M. Smith, and K. Wansbrough, "Engineering with logic: Rigorous test-oracle specification and validation for TCP/IP and the sockets API," *Journal of the ACM*, vol. 66, no. 1, Dec. 2018.

[38] P. Fiterau-Brosteam, R. Janssen, and F. Vaandrager, "Combining model learning and model checking to analyze TCP implementations," in *Proceedings of Internation Conference on Computer Aided Verification (CAV)*, Canada, July 2016, pp. 454–471.

[39] M. Smith and K. Ramakrishnan, "Formal specification and verification of safety and performance of TCP selective acknowledgment," *IEEE/ACM Transactions on Networking*, vol. 10, no. 2, pp. 193–207, August 2002.

[40] L. Lcokefeer, D. Williams, and W. Fokkink, "Formal specification and verification of TCP extended with the window scale option," *Science of Computer Programming*, vol. 118, no. 1, pp. 3–23, Mar. 2016.

[41] Z. Shukur, N. Alias, M. Halip, and B. Idrus, "Formal specification and validation of selective acknowledgement protocol using Z/EVES theorem prover," *Journal of Applied Sciences*, vol. 6, no. 8, pp. 1712–1719, 2006.

[42] K. McMillan and L. Zuck, "Formal specification and testing of QUIC," in *Proceedings of ACM SIGCOMM*, Beijing, China, Aug. 2019, pp. 227–240.

[43] L. Nelson, H. Sigurbjarnarson, K. Zhang, D. Johnson, J. Bornholt, E. Torlak, and X. Wang, "Hyperkernel: Push-button verification of an os kernel," in *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP 17)*, Shanghai, China, 2017, p. 252–269.

[44] J. Song, C. Cadar, and P. Pietzuch, "SymbexNet: Testing network protocol implementations with symbolic execution and rule-based specifications," *IEEE Transactions on Software Engineering*, vol. 40, no. 7, pp. 695–709, July 2014.

[45] M. Vu, L. Xu, S. Elbaum, W. Sun, and K. Qiao, "Efficient systematic testing of network protocols with temporal uncertain events," in *Proceedings of IEEE International Conference on Computer Communications (INFOCOM)*, Paris, France, Apr. 2019, pp. 604–612.

[46] M. Arashloo, R. Beckett, and R. Agarwal, "Formal methods for network performance analysis," in *Proceedings of NSDI*, Boston, MA, April 2023, pp. 221–239.

[47] D. Engler and M. Musuvathi, "Static analysis versus software model checking for bug finding," in *Proceedings of International Conference on Verification, Model Checking and Abstract Interpretation*, Venice, Italy, Jan. 2004, pp. 191–210.

[48] O. Udrea, C. Lumezanu, and J. Foster, "Rule-based static analysis of network protocol implementation," in *Proceedings of USENIX Security Symposium*, Vancouver, Canada, July 2006, pp. 130–157.

[49] Q. Chen, Z. Qian, Y. Jia, Y. Shao, and Z. Mao, "Static detection of packet injection vulnerabilities: A case for identifying attacker-controlled implicit information leaks," in *Proceedings of ACM SIGSAC Conference on Computer and Communications Security (CCS)*, Denver, CO, Oct. 2015, pp. 388–400.

[50] J. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, July 1976.

[51] R. Baldoni, E. Coppa, D. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Computing Surveys*, vol. 51, no. 3, July 2018.

[52] C. Cadar, D. Dunbar, and D. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of USENIX OSDI*, San Diego, CA, December 2008.

[53] M. Zhang, "[patch net] tcp_cubic fix to achieve at least the same throughput as reno," https://lore.kernel.org/netdev/20240810223130.379146-1-mrzhang97@gmail.com/t/#u.

[54] Linux Vegas Source Code in Latest Kernel, https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/net/ipv4/tcp_vegas.c.

[55] A. Narayan, F. Cangialosi, D. Raghavan, P. Goyal, S. Narayana, R. Mittal, M. Alizadeh, and H. Balakrishnan, "Restructuring endpoint congestion control," *Proceedings of ACM SIGCOMM*, pp. 30–43, Aug. 2018.