A Formal Analysis of Karn's Algorithm

Max von Hippel[1], Kenneth L. McMillan[2], Cristina Nita-Rotaru[1], and Lenore D. Zuck[3]

[1] Northeastern University, Boston MA, USA
{vonhippel.m,c.nitarotaru}@northeastern.edu
[2] University of Texas at Austin, Austin TX, USA kenmcm@cs.utexas.edu
[3] University of Illinois Chicago, Chicago IL, USA zuck@uic.edu

Abstract. The stability of the Internet relies on timeouts. The timeout value, known as the Retransmission TimeOut (RTO), is constantly updated, based on sampling the Round Trip Time (RTT) of each packet as measured by its sender – that is, the time between when the sender transmits a packet and receives a corresponding acknowledgement. Many of the Internet protocols compute those samples via the same sampling mechanism, known as Karn's Algorithm.

We present a formal description of the algorithm, and study its properties. We prove the computed samples reflect the RTT of some packets, but it is not always possible to determine which. We then study some of the properties of RTO computations as described in the commonly used RFC6298. All properties are mechanically verified.

Keywords: Internet protocols · RTT sampling · Karn's Algorithm.

1 Introduction

This paper examines Karn's algorithm [33], a mechanism that is widely used across the Internet to estimate the round trip time (RTT) of transmissions. Many of the Internet's protocols use Karn's algorithm to measure RTT for the specific purpose of computing the *Retransmission TimeOut* (RTO), which is used to detect congestion and trigger retransmissions of lost data. We examine the RTO computation based on the RTT computations of Karn's algorithm, formally verify properties of the RTT and RTO computations, and discuss their interactions. To the best of our knowledge, this is the first formal and automated study of these algorithms.

Protocols leverage RTT information for many purposes, such as one-way delay estimation [1] or network topology optimization [59, 52]. The most common use of RTT is for RTO computation. Internet protocols are described in *Request for Comments* documents (RFCs), and the RTO computation is defined in RFC6298 [49], which states:

> The Internet, to a considerable degree, relies on the correct implementation of the RTO algorithm [...] in order to preserve network stability and avoid congestion collapse.

On one hand, an RTO that is too low may cause false timeouts by hastily triggering a timeout mechanism that delays the proper functioning of the protocol, and also exposes it to denial-of-service attacks. On the other hand, an RTO that is too high causes overuse

of resources [51] by unnecessarily delaying the invocation of timeout mechanisms when congestion occurs. A poorly chosen RTO can have disastrous consequences, including *congestion collapse*, wherein the demands put on the network far exceed its capacity, leading to excessive message dropping and thus excessive retransmission. Congestion collapse was first observed in October 1986, during which time total Internet traffic dropped by over 1000x [30]. At the time this kind of network failure was an engineering curiosity, but today it would spell global economic disaster, loss of life, infrastructural damage, etc.

Both Karn's algorithm and the RTO computation are widely used across the Internet, as we detail in Subsec. 1.1. Hence, the correctness of these two mechanisms is fundamental for the correctness of the Internet as a whole. It is interesting to note that some theoretical papers analyzing congestion control, the original motivation for computing RTO, explicitly ignore the topic of timeouts, and hence implicitly ignore how RTO is computed (e.g., [45, 7, 64]).

Computing a "good" RTO requires a "good" estimate of the RTT. The RTO computation depends solely on the estimated RTT and some parameters that are fixed. Thus, understanding the mechanism which estimates RTT is fundamental to understanding any quantitative property of the Internet. The RTT of a packet (or message, datagram, frame, segment, etc.) is the time that elapsed between its transmission and some confirmation of its delivery. Both events (transmission and receipt of confirmation of delivery) occur at the same endpoint, namely, the one that transmits the packet, which we call the *sender*. In essence, if the sender transmits a packet at its local time t, and first learns of its delivery at time $t + \delta$, it estimates the RTT for this packet as δ.

TCP uses a *cumulative* acknowledgement mechanism where every packet received generates an acknowledgment (ACK) with the sequence number of the first unreceived packet.[4] Thus, if all packets $p_1, \ldots, p_x$ are received and packet p_{x+1} is not, the receiver will ACK with $x + 1$, which is the sequence number of the first unreceived packet in the sequence, even if packets whose sequence number is greater than $x + 1$ were received.

If the Internet's delivery mechanism were perfect, then packets would be received in order, acknowledged in order, and the sender would always be able to compute the RTT of each packet. In reality, of course, the Internet is not perfect. TCP operates on top of IP, whose only guarantee is that every message received was sent. Thus, messages are neither invented nor corrupted, but at least theoretically, may be duplicated, reordered, or lost. In practice duplication is sufficiently rare that it is ignored, and reordering is sometimes ignored and sometimes restricted. But losses are never ignored, and are the main focus of all congestion control algorithms. When a loss is suspected, a packet is retransmitted (often, the retransmission is at a lower rate than the original transmissions). If it is later acknowledged, one cannot determine whether the ACK is for the initial transmission or for the retransmission. Karn's algorithm [33] addresses this ambiguity by assuming that packets may have been retransmitted and providing an RTT estimate that ignores the retransmitted packets. RFC6298 [49] then computes an estimated RTT as a weighted (decaying) average of the samples output by Karn's algorithm, and computes an RTO based on this estimate and a measure of the RTT variance.

[4] Some implementations of TCP use additional types of acknowledgements, yet, the cumulative ones are common to TCP implementations.

The RTO is then used to gauge whether a packet is lost, and then, usually, to transition a state where transmission rate is reduced. Thus, the RTT sampling in Karn's algorithm is what ultimately informs the transmission rate of protocols. And while RF6298 pertains to TCP, numerous non-TCP protocols also refer to RFC6298 for the RTO computation, as we outline in Subsec. 1.1.

Here, we first formalize (our understanding of) Karn's algorithm [49], and prove some high-level properties about the relationship between acknowledgments and packets. In particular, we show that Karn's algorithm computes the "real" RTT of some packet, but the identity of this packet may be impossible to determine, unless one assumes (as many do) that acknowledgements are delivered in a FIFO ordering, and the identity of this packet can be determined. Next, we examine the RTO computation defined in RFC6298 [49] and its relationship to Karn's algorithm. For example, we show that when the samples fluctuate within a known interval, the estimated RTT eventually converges to the same interval. This confirms and generalizes prior results.

All our results are automatically checked. For the first part, where we study Karn's algorithm, we use Ivy [60]. Ivy is an interactive prover for inductive invariants, and provides convenient, built-in facilities for specifying and proving properties about protocols, which makes it ideal for this part of the work. For the second part, we study the RTO computation (and other computations it relies on), defined in RFC6298. These are purely numerical computations and, in isolation, do not involve reasoning about the interleaving of processes or their communication. Each computation has rational inputs and outputs, and the theorems we prove bound these computations using exponents and rational multiplication. We also prove the asymptotic limits of these bounds in steady-state conditions, which we define. Since Ivy lacks a theory of rational numbers or exponentiation, we turn to the automated theorem prover ACL2s [19, 16] for the remainder of the work. All of our code is open-source and available at github.com/rto-karn. We believe this is the first paper that formalizes properties of the RTT sampling via Karn's algorithm, as well as properties of the quantities RFC6298 computes, including the RTO. Additionally, our work provides a useful example of how multiple formal methods approaches can be used to study different angles of a single system.

1.1 Usage of Karn's Algorithm and RFC6298

Many protocols use Karn's Algorithm to sample RTT, e.g., [49, 9, 27, 2, 42, 20]. Unfortunately, the samples output by Karn's Algorithm could be noisy or outdated. RFC6298 addresses this problem by using a rolling average called the *smoothed RTT*, or srtt. Protocols that use the srtt in conjunction with Karn's Algorithm (at least optionally) include [26, 57, 58, 53, 20, 52, 34, 50, 14, 61]. RFC6298 then proposes an RTO computation based on the srtt and another value called the rttvar, which is intended to capture the variance in the samples. Note, when referring specifically to the RTO output by RFC6298, we use the convention rto. This is a subtle distinction as the RTO can be implemented in other ways as well (see e.g., [35, 10]). These three computations (srtt, rttvar, and rto) are used in TCP and in many other protocols, e.g. [58, 53, 31, 50, 61], although some such protocols omit explicit mention of RFC6298 (see [51]).

Not all protocols use retransmission. For example, in QUIC [29] every packet has a unique identifier, hence retransmitting a packet assigns it a new unique identifier and

the matching ACK indicates whether it is for the old or new transmission. Consequently, Karn's algorithm is only used when a real retransmission occurs, which covers most of the protocols designed when one had to be mindful of the length of the transmitted packets and could not afford unique identifiers. On the other hand, even protocols that do not use Karn's algorithm nevertheless utilize a retransmission timeout that is at least adapted from RFC6298 – and in fact, QUIC is one such protocol.

2 The Formal Setup

We partition messages, or datagrams, into *packets* P and *acknowledgments* A. Each packet $p \in P$ is uniquely identified by its id $p.id \in \mathbb{N}$. Each acknowledgment $a \in A$ is also uniquely identified by its id $a.id$. In the sequel we identify packets and acknowledgments by their ids.

Messages (packets and acknowledgments) typically include additional information such as destination port or sequence number, however, we abstract away such information in our model. Also, some protocols distinguish between packets and *segments*, where a segment is a message containing multiple packets, but we abstract away this distinction as well.

2.1 The Model

The model consists of two endpoints (*sender* and *receiver*) connected over a bi-directional *channel*, as shown in Fig. 1. The sender sends packets through the channel to the receiver, and the receiver sends acknowledgements through the channel to the sender.

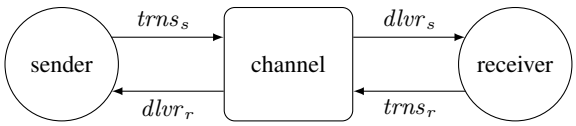

Fig. 1. The sender, channel, and receiver. The sender sends messages by $trns_s$ actions which are received by $dlvr_s$ actions at the receiver's endpoint, and similarly, the receiver sends messages by $trns_r$ actions which are received by $dlvr_r$ actions at the the receiver's endpoint.

Actions. The set of actions, Act, is partitioned into four action types:

1. $trns_s$ that consists of the set of the sender's transmit actions. That is, $trns_s = \cup_{p \in P}\{trns_s(id) : id = p.id\}$.
2. $dlvr_s$ that consists of the set of the receiver's delivery actions. That is, $dlvr_s = \cup_{p \in P}\{dlvr_s(id) : id = p.id\}$.
3. $trns_r$ that consists of the set of the receiver's transmit actions. That is, $trns_r = \cup_{a \in A}\{trns_r(id) : id = a.id\}$.

4. $dlvr_r$ that consists of the set of the sender's delivery actions. That is, $dlvr_r = \cup_{a \in A}\{dlvr_r(id) : id = a.id\}$.

For a finite sequence σ over Act, we denote the length of σ by $|\sigma|$ and refer to an occurrence of an action in σ as an *event*. That is, an event in σ consists of an action and its position in σ.

The sender's input actions are $dlvr_r$, and its output actions are $trns_s$. The receiver's input actions are $dlvr_s$ and its output actions are $trns_r$. The channel's input actions are $trns_s \cup trns_r$ and its output actions are $dlvr_s \cup dlvr_r$.

We assume that the channel is CSP-like synchronously composed with its two endpoints, the sender and the receiver. That is, for every $q \in \{s, r\}$, a $trns_q$ action occurs simultaneously at both q and the channel, and similarly for $dlvr_q$ actions. The sender and the receiver can be asynchronous. Modules are input-enabled in the I/O-automata sense, i.e., modules can always receive inputs (messages). In real implementations, modules' inputs are restricted by buffers, but since the channel is allowed to drop messages (as we see later), restrictions on the input buffer sizes can be modeled using loss. Hence the assumption of input-enabledness does not restrict the model.

2.2 Executions

Let σ be a sequence of actions. We say that σ is an *execution* if every delivery event in σ is preceded by a matching transmission event, that is, both events carry the same message. (This does not rule out duplication, reordering, or loss – more on that below.) Formally, for every $q \in \{s, r\}$ and $x \in \mathbb{N}$, if $e_i = dlvr_q(x) \in \sigma$, then for some $j < i$, $e_j = trns_q(x) \in \sigma$. This requirement rules out corruption and insertion of messages. In addition, for TCP-like executions, we may impose additional requirements on the ordering of $trns$-events of the endpoints. An example execution is illustrated in the rightmost column of Fig. 2.

The Sender. We adopt the convention that it only transmits a packet after it had transmitted all the preceding ones. Formally, for every $x \in \mathbb{N}$, if $e_i = trns_s(x + 1) \in \sigma$, then for some $j < i$, $e_j = trns_s(x) \in \sigma$.

The Receiver. We assume here the model of *cumulative acks*. That is, the receiver executes a $trns_r(id)$ action only if it has been delivered all packets p such that $p.id < id$ and it had not been delivered packet p such that $p.id = id$. Thus, for example, the receiver can execute $trns_r(17)$ only after it had been delivered all packets whose id is < 17 and had not been delivered the packet whose id is 17. In particular, it may have been delivered packets whose id is > 17, just not the packet whose id is 17.

Many TCP models mandate the receiver transmits exactly one acknowledgement in response to each packet delivered (e.g., [7, 6, 22, 13, 17, 21]). The assumption is common in congestion control algorithms where the sender uses the number of copies of the same acknowledgement it is delivered to estimate how many packets were delivered after a packet was dropped, and thus the number of lost packets. There are however some TCP variants, such as Data Center TCP and TCP Westwood, that allow a *delayed* ACK option wherein the receiver transmits an ACK after every n^{th} packet delivery [11, 44],

or Compound TCP that allows *proactive acknowledgments* where the receiver transmits before having receiving all the acknowledged packets, albeit at a pace that is proportional to the pace of packet deliveries [55]. Another mechanism that is sometimes allowed is *NACK* (for Negative ACK) where the receiver sends, in addition to the cumulative acknowledgement, a list of gaps of missing packets [3]. Since TCP datagrams are restricted in size, the NACKs are partial. Newer protocols (such as QUIC) allow for full (unrestricted) NACKs [29].

Our Ivy model assumes the receiver transmits one acknowledgement per packet delivered. That is, we assume that in the projection of σ onto the receiver's actions, $trns_r$ and $dlvr_s$ events are alternating. In fact, the results listed in this paper would still hold even under the slightly weaker assumption that the receiver transmits an acknowledgment whenever it is delivered a packet that it had not previously been delivered, but for which it had previously been delivered all lesser ones. However, the stronger assumption is easier to reason about, and is more commonly used in the literature (for example it is the default assumption for congestion control algorithms where the pace of delivered acknowledgments is used to infer the pace of delivered packets). Consequently, our results apply to traditional congestion control algorithms like TCP Vegas and TCP New Reno where the receiver transmits one acknowledgement per packet delivered, however, our results might not apply to atypical protocols like Data Center TCP, and TCP Westwood, or Compound TCP that use alternative acknowledgment schemes.

The Channel. So far, we only required that the channel never deliver messages to one endpoint that were not previously transmitted by the other. This does not rule out loss, reordering, nor duplication of messages. In the literature, message duplication is assumed to be so uncommon that it can be disregarded. The traditional congestion control protocols ([5, 22, 39, 28, 55]) assume bounded reordering, namely, that once a message is delivered, an older one can be delivered only if transmitted no more than k transmissions ago (usually, $k = 4$). Packet losses are always assumed to occur, but the possibility of losing acknowledgements is often ignored.

It is possible to formalize further constraints on the channel, for example, by restricting the receiver-to-sender path to be loss- and reordering-free. For instance, the work in [4] formalizes a constrained channel by assuming a mapping from delivery to transmission events, and using properties of this mapping to express restrictions. Reordering is ruled out by having this mapping be monotonic, duplication is ruled out by having it be one-to-one, and loss is ruled out by having it be onto.

Most prior works assume no loss or reordering of acknowledgments [8, 25, 64, 7, 6], or did not model loss or reordering at all [40, 24, 18]. Some prior works assume both loss and reordering but do not study the computation of RTO or other aspects of congestion control [4, 54].

Since, as we describe in Sec. 5, some works on RTO assume the channel delivers acknowledgements in perfect order, and since this assumption has implications on the RTT computation (see Ob. 4), we define executions where the receiver's messages are delivered, without losses, in the order they are transmitted as follows. An execution σ is a *FIFO-acknowledgement execution* if $\sigma|_{dlvr_r} \preceq \sigma|_{trns_r}$ is an invariant of sigma, where $\sigma|_a$ is the projection of σ onto the a actions, and $\preceq$ is the prefix relation. That is,

in a FIFO-acknowledgement execution, the sequences of ACK's delivered to the sender is always a prefix of the sequence of ACK's transmitted by the receiver.

The following observation establishes that the sequence of acknowledgements the receiver transmits is monotonically increasing. Its proof follows directly from the fact that the receiver is generating cumulative acks. (Recall that all Observations in this section and the next are established in Ivy.)

Observation 1 *Let σ be an execution, and assume i and j, $i < j$, such that $e_i = trns_r(a_i), e_j = trns_r(a_j)$ are in σ. Then $a_i \leq a_j$.*

2.3 Sender's Computations

So far, we abstracted away from the internals of the sender, receiver, and channel, and focused on the executions their composition allows. As we pointed out at the beginning of this section, real datagrams can contain information far beyond ids, and there are many mechanisms for their generation, depending on the protocol being implemented and the implementation choices made. Such real implementations have *states*. All we care about here, however, is the set of observable behaviors they allow, in terms of packet and acknowledgement ids. We thus choose to ignore implementation details, including states, and focus on executions, namely abstract observable behaviors.

In the next section we study a particular mechanism that is imposed over executions. In particular, we describe a pseudo code for an algorithm for sampling the RTT of packets. This pseudo code, say P, is (synchronously) composed with the sender's algorithm (on which we only make a single assumption, that is, that a packet is transmitted only after all prior ones were transmitted). We can view the algorithm as a *non-interfering monitor*, that is, P "observes" the sender's actions ($trns_s$ and $dlvr_r$) and performs some bookkeeping when each occurs. In fact, after initialization of variables, it consists of two parts, one that describes the update to its variables upon a $trns_s$ action, and one that describes the updates to its variables after a $dlvr_r$ action.

Let V be the set of variables P uses. To be non-interfering, V has to be disjoint from the set of variables that the sender uses to determine when to generate $trns_s$'s and process $dlvr_r$'s. We ignore this latter set of variables since it is of no relevance to our purposes. Let a *sender's state* be a type-consistent assignment of values V. For a sender's state s and a variable $v \in V$, let $s[\![v]\!]$ be the value of v at state s. For simplicity's sake (and consistent with the pseudo code we present in the next section) assume that P is deterministic, that is, given a state s and a sender's action α, there is a unique sender state s' such that s' is the successor of s given α.

Let σ be an execution. Let $\sigma|_s$ be the projection of σ onto the sender's events (the $trns_s$ and $dlvr_r$ events). Since P is deterministic, the sequence $\sigma|_s$ uniquely defines a sequence of sender's states $\kappa_\sigma : s_0, \ldots$ such that s_0 is the initial state, and every s_{i+1} is a successor of s_i under P according to $\sigma|_s$. We refer to κ_σ as the *sender's computation under P and σ*.

3 Karn's Algorithm and its Analysis

As discussed in Sec. 1, having a good estimate of RTT, the round-trip time of a packet, is essential for determining the value of RTO, which is crucial for many of the Internet's

protocols (see Subsec. 1.1 for a listing thereof). The value of RTT varies over the lifetime of a protocol, and is therefore often sampled. Since the sender knows the time it transmits a packet, and is also the recipient of acknowledgements, it is the sender whose role it is to sample the RTT. If the channel over which packets and acknowledgements are communicated were a perfect FIFO channel, then RTT would be easy to compute, since then each packet would generate a single acknowledgement, and the time between the transmission of the packets and the delivery of its acknowledgement would be the RTT. However, channels are not perfect. Senders retransmit packets they believe to be lost, and when those are acknowledged the sender cannot disambiguate which of the transmissions to associate with the acknowledgements. Moreover, transmitted acknowledgments can be lost, or delivered out of order. In [33], an idea, referred to as "Karn's Algorithm," was introduced to address the first issue. There, sampling of RTT is only performed when the sender receives a new acknowledgement, say h, greater than the previously highest received acknowledgement, say ℓ, where all the packets whose id is in the range $[\ell, h)$ were transmitted only once. It then outputs a new sample whose value is the time that elapsed between the transmission of the packet whose id is ℓ and delivery of the acknowledgement h. The reason ℓ (as opposed to h) is used for the base of calculations is the possibility that the id of the packet whose delivery triggers the new acknowledgement is ℓ, and the RTT computation has to be cautious in the sense of over-approximating RTT.

The real RTT of a packet may be tricky to define. The only case where it is clear is when packet i is transmitted once, and an ACK $i+1$ is delivered before any other ACK $\geq i+1$ is delivered. We can then define the RTT of packet i, $\mathsf{rtt}(i)$, to be the time, on the sender's clock, that elapses between the (first and only) $trns_s(i)$ action and the $dlvr_r(i+1)$ action. Since the channel is not FIFO, it's possible that $h > \ell + 1$, and then the sample, that is, the time that elapses between $trns_s[\ell]$ and $dlvr_r(h)$ is the RTT for some packet $j \in [\ell, h)$, denoted by, $\mathsf{rtt}(j)$, but we may not be able to identify j. Moreover, the sample over-approximates the RTT of all packets in the range. Note that rtt is a partial function. We show that when the channel delivers the receiver's messages in FIFO ordering, then the computed sample is exactly $\mathsf{rtt}(\ell)$.

We model the sender's sampling of RTT according to Karn's Algorithm, see pseudocode in Alg. 1. The sampling is a non-interfering monitor of the sender. Its inputs are the sender's actions, the $trns_s(i)$'s and $dlvr_r(j)$'s. Its output is a (possibly empty) sequence of samples denoted by S. To model time, we use an integer counter (τ) that is initialized to 1 (we reserve 0 for "undefined") and is incremented with each step. Upon a $trns_s(i)$ input, the algorithm stores, in *numT*$[i]$, the number of times packet i is transmitted, and in *time*$[i]$ the time of the first time it is transmitted. The second step is for $dlvr_r$ events, where the sender determines whether a new sample can be computed, and if so, computes it. An example execution, concluding with the computation of a sample via Karn's Algorithm, is given in Fig. 2.

In Alg. 1, *numT*$[i]$ stores the number of times a packet whose id is i is transmitted, *time*$[i]$ stores the sender's time where packet whose id is i is first transmitted, $high$ records the highest delivered acknowledgement, and when a new sample is computed (in S) it is outputted. The condition *ok-to-sample*(*numT*, $high$) in Line 13 checks whether sampling should occur. When $high > 0$, that is, when this is not the

Algorithm 1: RTT Sampling

```
   input  : trns_s(i), dlvr_r(j), i, j ∈ ℕ⁺
   output: S ∈ ℕ⁺
 1 numT, time : ℕ⁺ → ℕ init all 0
 2 high : ℕ init 0
 3 τ : ℕ init 1
 4 if trns_s(i) is received then
 5 |   numT[i] := numT[i] + 1
 6 |   if time[i] = 0 then
 7 |   |   time[i] := τ
 8 |   end
 9 |   τ := τ + 1
10 end
11 if dlvr_r(j) is received then
12 |   if j > high then
13 |   |   if ok-to-sample(numT, high) then
14 |   |   |   S := τ − time[high]
15 |   |   |   Ouput S
16 |   |   end
17 |   |   high := j
18 |   end
19 |   τ := τ + 1
20 end
```

first acknowledgement received, then the condition is that all the packets in the range $[high, j)$ were transmitted once. If, however, $high = 0$, since ids are positive, the condition is that all the packets in the range $[1, j)$ were transmitted once. Hence, *ok-to-sample*(*numT*, $high$) is:

$$(\forall k.high < k < j \rightarrow \textit{numT}[k] = 1) \ \wedge \ (high > 0 \rightarrow \textit{numT}[high] = 1)$$

If *ok-to-sample*(*numT*, $high$), Line 14 computes a new sample S as the time that elapsed since packet $high$ was transmitted until acknowledgement j is delivered, and outputs is in the next line. Thus, a new sample is *not computed* when a new ACK, that is greater than $high$, is delivered but some packets whose id is less than the new ACK, yet $\geq high$ were retransmitted. Whether or not a new sample is computed, when such an ACK is delivered, $high$ is updated to its value to reflect the currently highest delivered ACK.

3.1 Properties of Karn's Algorithm

We show, through a sequence of observations, that Alg. 1 computes the true RTT of *some* packet, whose identity cannot also be uniquely determined. While much was written about the algorithm, we failed to find a clear statement of what exactly it computes, but then, unitl now, it is stated informally and the main focus is on its implications to RTO estimates. In [33], it is shown that if a small number of consecutive samples are equal then the computed RTT (which is a weighted average of the sampled RTTs) is

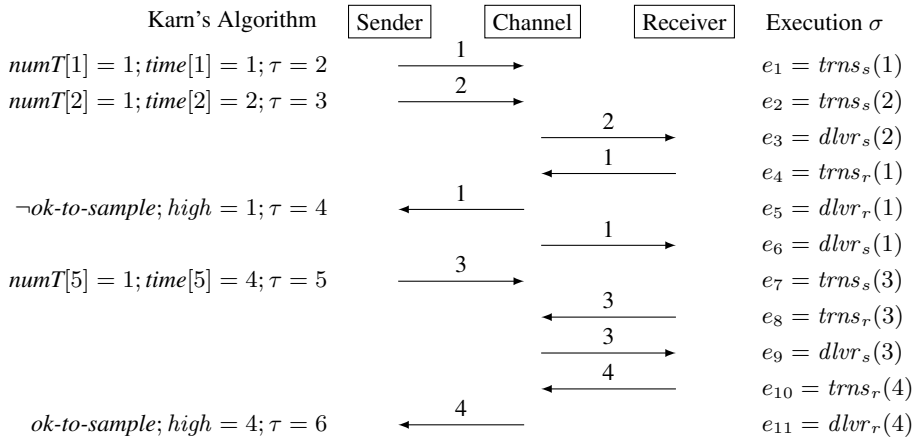

Fig. 2. Message sequence chart illustrating an example execution. Time progresses from top down. Instructions executed by Alg. 1 are shown on the left, and the sender's execution is on the right. $trns_s$ events are indicated with arrows from sender to channel, $dlvr_s$ events with arrows from channel to receiver, etc. After the final $dlvr_r$ event, sender executes Line 14 and outputs the computation $\mathsf{S} = 6 - 2 = 4$.

close to the value of those samples. See the next section for further discussion on this issue. Our focus in this section is what exactly is computed by the algorithm.

The set of variables in Alg. 1 is $V = \{\tau, \mathit{numT}, \mathit{time}, \mathit{high}, \mathsf{S}\}$. Let σ be an execution, and let κ_σ be the sender's computation under Alg. 1 and σ. The following observation establishes two invariants over κ_σ. Both follow from the assumption we made on the sender's execution, namely that the sender does not transmit p without first transmitting $1, \ldots, p-1$. The first establishes that if a packet is transmitted (as viewed by *numT*), all preceding ones were transmitted, and the second that the first time a packet is transmitted must be later than the first time every preceding packet was transmitted.

Observation 2 *The following are invariants over sender's computations:*

$$0 < i < j \ \wedge \ \mathit{numT}[j] > 0 \longrightarrow \mathit{numT}[i] > 0 \qquad \text{(I1)}$$
$$0 < i < j \ \wedge \ \mathit{numT}[j] > 0 \longrightarrow \mathit{time}[i] < \mathit{time}[j] \ \text{(I2)}$$

Assume $\kappa_\sigma : s_0, s_1, \ldots$. We say that a state $s_i \in \kappa_\sigma$ is a *fresh sample* state if the transition leading into it contains an execution of Lines 13–16 of Alg. 1. The following observation establishes that in a fresh sample state, the new sample is an upper bound for the RTT of a particular range of packets (whose ids range from the previous $high$ up to, but excluding, the new $high$), and is the real RTT of one of them.

Observation 3 *Let σ and κ_σ be as above and assume that $s_i \in \kappa_\sigma$ is a fresh sample state. Then the following all hold:*

1. *For every packet with id ℓ, $s_{i-1}[\![high]\!] \leq \ell < s_i[\![high]\!]$ implies that $\mathsf{rtt}(\ell) \leq s_i[\![\mathsf{S}]\!]$. That is, the fresh sample is an upper bound of the RTT for all packets between the old and the new $high$.*

2. *There exists a packet with id ℓ, $s_{i-1}[\![high]\!] \leq \ell < s_i[\![high]\!]$ such that $\mathsf{rtt}(\ell) = s_i[\![\mathsf{S}]\!]$. That is, the fresh sample is the RTT of some packet between the old and new $high$.*

We next show under the (somewhat unrealistic, yet often made) assumption of FIFO-acknowledgement executions, the packet whose RTT is computed in the second clause of Ob. 3 is exactly the packet whose id equals to the prior $high$. In particular, that if s_i is a fresh sample state, then the packet whose RTT is computed is p such that $p.id$ equals to the value of $high$ just before the new fresh state is reached.

Observation 4 *Let σ be a FIFO-acknowledgement execution σ, and assume κ_σ contains a fresh sample state s_ℓ. Then $s_\ell[\![\mathsf{S}]\!] = \mathsf{rtt}(s_{\ell-1}[\![high]\!])$.*

Let σ be a (not necessarily FIFO) execution and let κ_σ be the sender's computation under Alg. 1 and σ that outputs some samples. We deonte by $\mathsf{S}_1, \ldots$ the sequence of samples that is the output of κ_σ. That is, S_k is the k^{th} sample obtained by Alg. 1 given the execution σ.

4 Analyzing RFC6298 Calculation of RTO

We next analyze the computation of RTOs as described in RFC6298. Each new sample triggers a new RTO computation, that depends on sequences of two other variables (srtt and rttvar) and three constants (α, β, and G). In this Section, we consider the scenario in which the samples produced by Karn's algorithm are consecutively bounded. We show that in this context, we can compute corresponding bounds on srtt, as well as an upper bound on rttvar; and that these bounds converge to the bounds on the samples and the distance between those bounds, respectively, as the number of bounded samples grows. These observations allow us to characterize the asymptotic conditions under which the RTO will generally exceed the RTT values, and by how much. In other words, these observations allow us to reason about whether timeouts will occur in the long run.

Let $\{\mathsf{srtt}, \mathsf{rttvar}, \mathsf{rto}, \alpha, \beta, G\} \in \mathbb{Q}^+$ be fresh variables. As mentioned before, $\alpha < 1$, $\beta < 1$, and G are constants. Let σ be an execution and κ_σ be the sender's computation under Alg. 1 and σ. Assume that κ_σ outputs some samples $\mathsf{S}_1, \ldots, \mathsf{S}_N$.

RFC6298 defines the RTO and the computations it depends upon as follows:

$$\mathsf{rto}_i = \mathsf{srtt}_i + \max(G, 4 \cdot \mathsf{rttvar}_i)$$

$$\mathsf{srtt}_i = \begin{cases} \mathsf{S}_i & \text{if } i = 1 \\ (1-\alpha)\mathsf{srtt}_{i-1} + \alpha \mathsf{S}_i & \text{if } i > 1 \end{cases}$$

$$\mathsf{rttvar}_i = \begin{cases} \mathsf{S}_i/2 & \text{if } i = 1 \\ (1-\beta)\mathsf{rttvar}_{i-1} + \beta|\mathsf{srtt}_{i-1} - \mathsf{S}_i| & \text{if } i > 1 \end{cases}$$

where G is the clock granularity (of τ), srtt is referred to in RFC6298 as the *smoothed RTT*, and rttvar as the *RTT variance*. The srtt is a rolling weighted average of the sample values and is meant to give an RTT estimate that is resilient to noisy samples. The rttvar is described as a measure of variance in the sample values, although as we show below, it is not the usual statistical variance. The rto is computed from srtt and

rttvar and is the amount of time the sender will wait without receiving an ACK before it determines that congestion has occurred and takes some action such as decreasing its output and retransmitting unacknowledged messages. We manually compute, and mechanically verify the computations, of these variables using ACL2s. The choice of ACL2s stems from Ivy's lack of support of the theory of the Rationals, which is necessary for this analysis.

Intuitively, the srtt is meant to give an estimate of the (recent) samples, while the rttvar is meant to provide a measure of the degree to which these samples vary. However, the rttvar is not actually a variance in the statistical sense. For example, if $\mathsf{S}_1 = 1$, $\mathsf{S}_2 = 44$, $\mathsf{S}_3 = 13$, $\alpha = 1/8$, and $\beta = 1/4$, then the statistical variance of the samples is $1477/3$ but $\mathsf{rttvar}_3 = 4977361/65536 \neq 1477/3$.

If the rttvar does not compute the statistical variance, then what does it compute? And what does the srtt compute? We answer these questions under the (realistic) restriction that the samples fall within some bounds, which we formalize as follows. Let c and r be positive rationals and let i and n be positive naturals. Suppose that $\mathsf{S}_i, \ldots, \mathsf{S}_{i+n}$ all fall within the bounded interval $[c - r, c + r]$ with center c and radius r. Then we refer to $\mathsf{S}_i, \ldots, \mathsf{S}_{i+n}$ as c/r *steady-state samples*. In the remainder of this section, we study c/r steady-state samples and prove both instantaneous and asymptotic bounds on the rttvar and srtt values they produce. Fig. 3 illustrates two scenarios with c/r steady-state samples – one in which the samples are randomly drawn from a uniform distribution, and another where they are pathologically crafted to cause infinitely many timeouts – and shows for each scenario, the lower and upper bounds on the srtt which we report below in Ob. 5, as well as the upper bound on the rttvar which we report below in Ob. 6. The asymptotic behavior of the reported bounds is also clearly visible.

In [33], Karn and Partridge argue that, given $\alpha = 1/8$ and $\beta = 1/4$, after six consecutive identical samples S, assuming the initial $\mathsf{srtt} \geq \beta\mathsf{S}$, the final srtt approximates S within some tolerable ϵ. We generalize this result in the following observation.

Observation 5 *Suppose α, c, and r are reals, c is positive, r is non-negative, and $\alpha \in (0, 1]$. Further suppose i and n are positive naturals, and $\mathsf{S}_i, \ldots, \mathsf{S}_{i+n}$ are c/r steady-state samples. Define L and H as follows.*

$$L = (1-\alpha)^{n+1}\mathsf{srtt}_{i-1} + (1-(1-\alpha)^{n+1})(c-r)$$
$$H = (1-\alpha)^{n+1}\mathsf{srtt}_{i-1} + (1-(1-\alpha)^{n+1})(c+r)$$

Then $L \leq \mathsf{srtt}_{i+n} \leq H$. Moreover, $\lim_{n\to\infty} L = c - r$, and $\lim_{n\to\infty} H = c + r$.

As an example, suppose that $n = 5$, $\alpha = 1/8$, $\beta = 1/4$, $r = 0$, and $\mathsf{srtt}_{i-1} = 3\beta c$. Then $L = H \approx 0.89c$, hence srtt_{i+4} differs from $\mathsf{S}_i, \ldots, \mathsf{S}_{i+4} = c$ by about 10% or less. Ob. 5 also generalizes in the sense that as n grows to infinity, $[L, H]$ converges to $[c - r, c + r]$, meaning the bounds on the srtt converge to the bounds on the samples – or if $r = 0$, to just the (repeated) sample value $\mathsf{S}_i = c$.

Next, we turn our attention to bounding the rttvar. The following observation establishes that when the difference between each sample and the previous srtt is bounded above by some constant Δ, then each rttvar is bounded above by a function of this Δ. Moreover, as the number of consecutive samples grows for which this bound holds, the

upper bound on the rttvar converges to precisely Δ. Note, in this observation we use the convention $f^{(m)}$ to denote m-repeated compositions of f, for any function f, e.g., $f^{(3)}(x) = f(f(f(x)))$.

Observation 6 *Suppose* $1 < i$, *and* $0 < \Delta \in \mathbb{Q}$ *is such that* $|\mathsf{S}_j - \mathsf{srtt}_{j-1}| \leq \Delta$ *for all* $j \in [i, i+n]$. *Define* $B_\Delta(x) = (1-\beta)x + \beta\Delta$. *Then all the following hold.*

- *Each* rttvar_j *is bounded above by the function* $B_\Delta(\mathsf{rttvar}_{j-1})$.
- *We can rewrite the (recursive) upper bound on* rttvar_{i+n} *as follows:*

$$B_\Delta^{(n+1)}(\mathsf{rttvar}_{i-1}) = (1-\beta)^{n+1}\mathsf{rttvar}_{i-1} + (1-(1-\beta)^{n+1})\Delta$$

- *Moreover, this bound converges to* Δ, *i.e.,* $\lim_{n\to\infty} B_\Delta^{(n+1)}(\mathsf{rttvar}_{i-1}) = \Delta$.

Note that if $\mathsf{S}_i, \ldots, \mathsf{S}_{i+n}$ are c/r steady-state samples then by Ob. 5:

$$|\mathsf{S}_n - \mathsf{srtt}_{n-1}| \leq \Delta = (1-\alpha)^{n+1}\mathsf{srtt}_{i-1} + 2r - (1-\alpha)^{n+1}(c+r)$$

Since $\lim_{n\to\infty} \Delta = 2r$, in c/r steady-state conditions, it follows that the rttvar asymptotically measures the diameter $2r$ of the sample interval $[c-r, c+r]$.

Implications for the rto *Computation.* Assume n are c/r consecutive steady-state samples. As $n \to \infty$, the bounds on srtt_n approach $[c-r, c+r]$, and the upper bound Δ on rttvar_n approaches $2r$. Thus, as n increases, assuming $G < 4\mathsf{rttvar}_n$, $c - r + 4\mathsf{rttvar}_n \leq \mathsf{rto}_n \leq c + 3r$. With these bounds, if rttvar_n is always bounded from below by r, then the rto exceeds the (steady) RTT, hence no timeout will occur. On the other hand, we can construct a pathological case where the samples are c/r steady-state but the rttvar dips below r, allowing the rto to drop below the RTT. One such case is illustrated in the bottom of Fig. 3. In that case, every 100^{th} sample is equal to $c + r = 75$, and the rest are equal to $c - r = 60$. At the spikes (where $\mathsf{S}_i = 75$) the sampled RTT exceeds the rto, and so a timeout would occur. This suffices to show that steady-state conditions alone do not guarantee a "steady-state" in terms of avoiding timeouts. Characterizing the minimal, sufficient conditions for avoiding timeouts during a c/r steady-state is a problem left for future work.

5 Related Work

To the best of our knowledge, ours is the first work to formally verify properties of Karn's algorithm or the RTO defined in RFC6298. However, formal methods have previously been applied to proving protocol correctness [54, 40, 47, 18], and lightweight formal methods have been used for protocol testing [12, 46]. One such lightweight approach, called PACKETDRILL, was used to test a new implementation of the RTO computation from RFC6298 [15]. The PACKETDRILL authors performed fourteen successful tests on the new RTO implementation. After publication, their tool was used externally to find a bug in the tested RTO implementation [62]. In contrast to such lightweight FM, in which an implementation is strategically tested, we took a proof-based approach to the verification of fundamental properties of the protocol design.

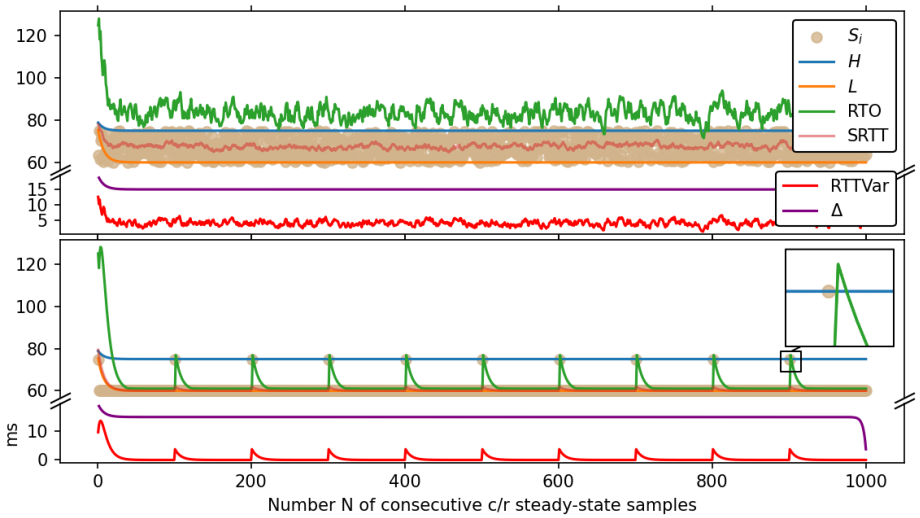

Fig. 3. Top: Uniformly random c/r steady-state scenario. Bottom: Pathological scenario. X-axis is index in the fresh subsequence; Y-axis is time in ms. On top, samples are uniformly distributed over $[60, 75]$, whereas on bottom, every 100^{th} sample equals $c + r = 75$, and the rest equal $c - r = 60$. On top, timeouts rarely occur, but on bottom, a timeout occurs every 100 samples (see zoomed inset). In both, $\alpha = 1/8$, $\beta = 1/4$, $\mathsf{srtt}_{i-1} = 80$, and $\mathsf{rttvar}_{i-1} = 11.25$.

Some prior works applied formal methods to congestion control algorithms [41, 23, 37, 43, 6, 7]. A common theme of these works is that they make strong assumptions about the network model, e.g., assuming the channel never duplicates messages or reorders or loses acknowledgments. In this vein, we study case in which acknowledgments are communicated FIFO in Ob. 4. Congestion control algorithms were also classically studied using manual mathematics (as opposed to formal methods) [45, 64, 56]. One such approach is called *network calculus* [38] and has been used to simulate congestion control algorithms [36]. Network calculus has the advantage that it can be used to study realistic network dynamics, in contrast to our Ivy-based approach, which is catered to logical properties of the system. For example, Kim and Hou [36] are able to determine the minimum and maximum throughput of traditional TCP congestion control, but do not prove any properties about what precisely Karn's algorithm measures, or about bounds on the variables used to compute the RTO.

Another line of inquiry aims to integrate formal methods directly into the RFC drafting process, either by analyzing RFC documents using natural language processing [48, 63], or by manually drafting a formal specification for one or more RFCs, and then using the specification to interrogate real-world implementations [32]. The Internet Engineering Task Force is exploring such techniques with its recently introduced Usable Formal Methods Research Group, of which we are members.

6 Conclusion

In this work we applied formal methods to Karn's algorithm, as well as the rto computation described by RFC6298 and used in many of the Internet's protocols. These two algorithms were previously only studied with manual mathematics or experimentation. We presented open-source formal models of each, with which we formally verified the following important properties.

Obs. 1: Acknowledgements are transmitted in nondecreasing order.
Obs. 2: Two inductive invariants regarding the internal variables of Karn's algorithm.
Obs. 3: Karn's algorithm samples a real RTT, but a pessimistic one.
Obs. 4: In the case where acknowledgments are neither dropped, duplicated, nor reordered, Karn's algorithm samples the highest ACK received by the sender before the sampled one.
Obs. 5: For the rto computation, when the samples are bounded, so is the srtt. As the number of bounded samples increases, the bounds on the srtt converge to the bounds on the samples.
Obs. 6: For the rto computation, when the samples are bounded, so is the rttvar. As the number of bounded samples increases, the upper bound on the rttvar converges to the difference between the lower and upper bounds on the samples.

We concluded by discussing the implications of these bounds for the rto.

In addition to rigorously examining some fundamental building blocks of the Internet, we also provide an example of how multiple provers can be used in harmony to prove more than either could handle alone. First, we used Ivy to model the underlying system and Karn's algorithm. Ivy offers an easy treatment for concurrency, which was vital for the behavior of the underspecfied models we used for the sender, receiver, and channel. The underspecification renders our results their generality. We guided Ivy by providing supplemental invariants as hints, e.g., *if $dlvr_r(a)$ occurs in an execution, then for all $p < a$, $dlvr_s(p)$ occurred previously*. Then, since Ivy lacks a theory of rationals, we turned to ACL2s. We began by proving two lemmas.

- The α-summation "unfolds": $(1-\alpha)\sum_{i=0}^{N}(1-\alpha)^i\alpha + \alpha = \sum_{i=0}^{N+1}(1-\alpha)^i\alpha$.
- The srtt is "linear": if $\mathsf{srtt}_{i-1} \leq \mathsf{srtt}'_{i-1}$ and, for all $i \leq j \leq i+n$, $\mathsf{S}_j \leq \mathsf{S}'_j$, then $\mathsf{srtt}_{i+n} \leq \mathsf{srtt}'_{i+n}$.

Then we steered ACL2s to prove Ob. 5 and Ob. 6 with these lemmas as hints.

Proving the limits of the bounds on srtt and rttvar was much trickier, and required manually writing ϵ/δ proofs directly in the ACL2s proof-builder. Nevertheless, they would have been impossible to do in Ivy. On the other hand, since ACL2s does not come with built-in facilities for reasoning about interleaved network semantics, we opted to leave the RTT computation proofs in Ivy. These choices were easier – and yielded cleaner proofs – compared to doing everything using just one of the two tools.

Acknowledgments. This material is based upon work supported by the National Science Foundation under Grant CCS-2140207, SHF-1918429, CNS-1801546, and GRFP-1938052, as well as by the Department of Defense under Grant W911NF2010310.

References

1. Abdou, A., Matrawy, A., van Oorschot, P.C.: Accurate one-way delay estimation with reduced client trustworthiness. IEEE Communications Letters **19**(5), 735–738 (2015)
2. Aboba, B., Wood, J.: Authentication, authorization and accounting (AAA) transport profile. https://www.rfc-editor.org/rfc/rfc3539 (June 2003), accessed 21 March 2023
3. Adamson, B., Bormann, C., Handley, M., Macker, J.: Negative-acknowledgment (NACK)-oriented reliable multicast (NORM) building blocks. https://www.rfc-editor.org/rfc/rfc3941 (November 2004), accessed 17 March 2023
4. Afek, Y., Attiya, H., Fekete, A., Fischer, M., Lynch, N., Mansour, Y., Wang, D.W., Zuck, L.: Reliable communication over unreliable channels. Journal of the ACM (JACM) **41**(6), 1267–1297 (1994)
5. Allman, M., Paxson, V., Blanton, E.: TCP congestion control. https://www.rfc-editor.org/rfc/rfc5681 (September 2009), accessed 23 February 2023
6. Arun, V., Alizadeh, M., Balakrishnan, H.: Starvation in end-to-end congestion control. In: Proceedings of the ACM SIGCOMM 2022 Conference. pp. 177–192 (2022)
7. Arun, V., Arashloo, M.T., Saeed, A., Alizadeh, M., Balakrishnan, H.: Toward formally verifying congestion control behavior. In: SIGCOMM 2021 (2021)
8. Baccelli, F., Hong, D.: TCP is max-plus linear and what it tells us on its throughput. In: Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication. pp. 219–230 (2000)
9. Balakrishnan, H., Seshan, S.: The congestion manager. https://www.rfc-editor.org/rfc/rfc3124 (June 2001), accessed 21 March 2023
10. Balandina, E., Koucheryavy, Y., Gurtov, A.: Computing the retransmission timeout in coap. In: Internet of Things, Smart Spaces, and Next Generation Networking: 13th International Conference, NEW2AN 2013 and 6th Conference, ruSMART 2013, St. Petersburg, Russia, August 28-30, 2013. Proceedings. pp. 352–362. Springer (2013)
11. Bensley, S., Thaler, D., Balasubramanian, P., Eggert, L., Judd, G.: Data Center TCP (DCTCP): TCP congestion control for data centers. https://www.rfc-editor.org/rfc/rfc8257 (October 2017), accessed 15 March 2023
12. Bishop, S., Fairbairn, M., Norrish, M., Sewell, P., Smith, M., Wansbrough, K.: Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP, and sockets. In: Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications. pp. 265–276 (2005)
13. Brakmo, L.S., Peterson, L.L.: TCP vegas: End to end congestion avoidance on a global internet. IEEE Journal on selected Areas in communications **13**(8), 1465–1480 (1995)
14. Camarillo, G., Drage, K., Kristensen, T., Ott, J., Eckel, C.: The Binary Floor Control Protocol (BFCP). https://www.rfc-editor.org/rfc/rfc8855 (January 2021), accessed 23 February 2023
15. Cardwell, N., Cheng, Y., Brakmo, L., Mathis, M., Raghavan, B., Dukkipati, N., Chu, H.k.J., Terzis, A., Herbert, T.: packetdrill: Scriptable network stack testing, from sockets to packets. In: 2013 USENIX Annual Technical Conference (USENIX ATC 13). pp. 213–218 (2013)
16. Chamarthi, H.R., Dillinger, P., Manolios, P., Vroon, D.: The ACL2 sedan theorem proving system. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 291–295. Springer (2011)
17. Cheng, Y., Cardwell, N., Dukkipati, N., Jha, P.: The RACK-TLP loss detection algorithm for TCP. https://www.rfc-editor.org/rfc/rfc8985 (February 2021), accessed 15 March 2023
18. Cluzel, G., Georgiou, K., Moy, Y., Zeller, C.: Layered formal verification of a TCP stack. In: 2021 IEEE Secure Development Conference (SecDev). pp. 86–93. IEEE (2021)
19. Dillinger, P.C., Manolios, P., Vroon, D., Moore, J.S.: Acl2s:"the ACL2 sedan". Electronic Notes in Theoretical Computer Science **174**(2), 3–18 (2007)

20. Eggert, L., Fairhurst, G., Shepherd, G.: UDP usage guidelines. https://www.rfc-editor.org/rfc/rfc8085 (March 2017), accessed 23 February 2023
21. Gerla, M., Sanadidi, M.Y., Wang, R., Zanella, A., Casetti, C., Mascolo, S.: TCP Westwood: Congestion window control using bandwidth estimation. In: GLOBECOM'01. IEEE Global Telecommunications Conference (Cat. No. 01CH37270). vol. 3, pp. 1698–1702. IEEE (2001)
22. Henderson, T., Floyd, S., Gurtov, A., Nishida, Y.: The NewReno modification to TCP's fast recovery algorithm. https://www.rfc-editor.org/rfc/rfc6582 (April 2012), accessed 15 March 2023
23. Hespanha, J.P., Bohacek, S., Obraczka, K., Lee, J.: Hybrid modeling of TCP congestion control. In: Hybrid Systems: Computation and Control: 4th International Workshop, HSCC 2001 Rome, Italy, March 28–30, 2001 Proceedings. pp. 291–304. Springer (2001)
24. von Hippel, M., Vick, C., Tripakis, S., Nita-Rotaru, C.: Automated attacker synthesis for distributed protocols. In: Computer Safety, Reliability, and Security: 39th International Conference, SAFECOMP 2020, Lisbon, Portugal, September 16–18, 2020, Proceedings 39. pp. 133–149. Springer (2020)
25. Hu, K., Liu, C., Liu, K.: Modeling and verification of custom TCP using SDL. In: 2013 IEEE 4th International Conference on Software Engineering and Service Science. pp. 455–458. IEEE (2013)
26. Hurtig, P., Brunstrom, A., Petlund, A., Welzl, M.: TCP and Stream Control Transmission Protocol (SCTP) RTO restart. https://www.rfc-editor.org/rfc/rfc7765 (February 2016), accessed 23 February 2023
27. Inamura, H., Montenegro, G., Ludwig, R., Gurtov, A., Khafizov, F.: TCP over second (2.5g) and third (3g) generation wireless networks. https://www.rfc-editor.org/rfc/rfc3481 (February 2007), accessed 21 March 2023
28. Iyengar, J., Swett, I.: QUIC loss detection and congestion control. https://www.rfc-editor.org/rfc/rfc9002 (may 2021), accessed 17 March 2023
29. Iyengar, J., Thomson, M.: QUIC: A UDP-Based Multiplexed and Secure Transport. RFC 9000 (May 2021). https://doi.org/10.17487/RFC9000, https://www.rfc-editor.org/info/rfc9000
30. Jacobson, V.: Congestion avoidance and control. ACM SIGCOMM computer communication review **18**(4), 314–329 (1988)
31. Jennings, C., Lowekamp, B., Rescorla, E., Baset, S., Schulzrinne, H.: REsource LOcation And Discovery (RELOAD) Base Protocol. https://www.rfc-editor.org/rfc/rfc6940 (2014), accessed 23 February 2023
32. Kakarla, S.K.R., Beckett, R., Millstein, T., Varghese, G.: SCALE: Automatically finding RFC compliance bugs in DNS nameservers. In: 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22). pp. 307–323. USENIX Association, Renton, WA (Apr 2022), https://www.usenix.org/conference/nsdi22/presentation/kakarla
33. Karn, P., Partridge, C.: Improving round-trip time estimates in reliable transport protocols. ACM SIGCOMM Computer Communication Review **17**(5), 2–7 (1987)
34. Keranen, A., Holmberg, C., Rosenberg, J.: Interactive Connectivity Establishment (ICE): A protocol for Network Address Translator (NAT) traversal. https://www.rfc-editor.org/rfc/rfc8445 (July 2018), accessed 23 February 2023
35. Kesselman, A., Mansour, Y.: Optimizing TCP retransmission timeout. In: Networking-ICN 2005: 4th International Conference on Networking, Reunion Island, France, April 17-21, 2005, Proceedings, Part II 4. pp. 133–140. Springer (2005)
36. Kim, H., Hou, J.C.: Network calculus based simulation for TCP congestion control: Theorems, implementation and evaluation. In: IEEE INFOCOM 2004. vol. 4, pp. 2844–2855. IEEE (2004)

37. Konur, S., Fisher, M.: Formal analysis of a VANET congestion control protocol through probabilistic verification. In: 2011 IEEE 73rd Vehicular Technology Conference (VTC Spring). pp. 1–5. IEEE (2011)
38. Le Boudec, J.Y., Thiran, P.: Network calculus: a theory of deterministic queuing systems for the internet. Springer (2001)
39. Liu, S., Başar, T., Srikant, R.: TCP-Illinois: A loss and delay-based congestion control algorithm for high-speed networks. In: Proceedings of the 1st international conference on Performance evaluation methodolgies and tools. pp. 55–es (2006)
40. Lockefeer, L., Williams, D.M., Fokkink, W.: Formal specification and verification of tcp extended with the window scale option. Science of Computer Programming **118**, 3–23 (2016)
41. Lomuscio, A., Strulo, B., Walker, N.G., Wu, P.: Model checking optimisation based congestion control algorithms. Fundamenta Informaticae **102**(1), 77–96 (2010)
42. Ludwig, R., Gurtov, A.: The Eifel response algorithm for TCP. https://www.rfc-editor.org/rfc/rfc4015 (February 2005), accessed 21 March 2023
43. Malik, M.H., Jamil, M., Khan, M.N., Malik, M.H.: Formal modelling of tcp congestion control mechanisms ecn/red and sap-law in the presence of udp traffic. EURASIP Journal on Wireless Communications and Networking **2016**, 1–12 (2016)
44. Mascolo, S., Casetti, C., Gerla, M., Sanadidi, M.Y., Wang, R.: Tcp westwood: Bandwidth estimation for enhanced transport over wireless links. In: Proceedings of the 7th annual international conference on Mobile computing and networking. pp. 287–297 (2001)
45. Mathis, M., Semke, J., Mahdavi, J., Ott, T.: The macroscopic behavior of the tcp congestion avoidance algorithm. ACM SIGCOMM Computer Communication Review **27**(3), 67–82 (1997)
46. McMillan, K.L., Zuck, L.D.: Formal specification and testing of QUIC. In: Proceedings of the ACM Special Interest Group on Data Communication, pp. 227–240 (2019)
47. Okumura, N., Ogata, K., Shinoda, Y.: Formal analysis of rfc 8120 authentication protocol for http under different assumptions. Journal of Information Security and Applications **53**, 102529 (2020)
48. Pacheco, M.L., von Hippel, M., Weintraub, B., Goldwasser, D., Nita-Rotaru, C.: Automated attack synthesis by extracting finite state machines from protocol specification documents. In: 2022 IEEE Symposium on Security and Privacy (SP). pp. 51–68. IEEE (2022)
49. Paxson, V., Allman, M., Chu, J., Sargent, M.: Computing TCP's retransmission timer. https://www.rfc-editor.org/rfc/rfc6298 (June 2011), accessed 22 February 2023
50. Petit-Huguenin, M., Salgueiro, G., Rosenberg, J., Wing, D., Mahy, R., Matthews, P.: Session Traversal Utilities for NAT (STUN). https://www.rfc-editor.org/rfc/rfc8489 (February 2020), accessed 23 February 2023
51. Pothamsetty, V., Mateti, P.: A case for exploit-robust and attack-aware protocol RFCs. In: Proceedings 20th IEEE International Parallel and Distributed Processing Symposium (2006)
52. Schinazi, D., Pauly, T.: Happy eyeballs version 2: Better connectivity using concurrency. https://www.rfc-editor.org/rfc/rfc8305 (December 2017), accessed 23 February 2023
53. Shalunov, S., Hazel, G., Iyengar, J., Kuehlewind, M.: Low Extra Delay Background Transport (LEDBAT). https://www.rfc-editor.org/rfc/rfc6817 (December 2012), accessed 23 February 2023
54. Smith, M.A.S.: Formal verification of TCP and T/TCP. Ph.D. thesis, Massachusetts Institute of Technology (1997)
55. Sridharan, M., Tan, K., Bansal, D., Thaler, D.: Compound TCP: A new TCP congestion control for high-speed and long distance networks. https://datatracker.ietf.org/doc/html/draft-sridharan-tcpm-ctcp-02 (November 2008), accessed 15 March 2023
56. Srikant, R., Başar, T.: The mathematics of Internet congestion control. Springer (2004)
57. Stewart, R.: tream control transmission protocol. https://www.rfc-editor.org/rfc/rfc4960 (September 2007), accessed 23 February 2023

58. T. Henderson, A.G.: The Host Identity Protocol (HIP) Experiment Report. https://www.rfc-editor.org/rfc/rfc6538 (March 2012), accessed 23 February 2023
59. Tang, C., Chang, R.N., Ward, C.: Gocast: Gossip-enhanced overlay multicast for fast and dependable group communication. In: 2005 International Conference on Dependable Systems and Networks (DSN'05). pp. 140–149. IEEE (2005)
60. Taube, M., Losa, G., McMillan, K.L., Padon, O., Sagiv, M., Shoham, S., Wilcox, J.R., Woos, D.: Modularity for decidability of deductive verification with applications to distributed systems. In: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 662–677 (2018)
61. Thubert, P.: IPv6 over Low-Power Wireless Personal Area Network (6LoWPAN) selective fragment recovery. https://www.rfc-editor.org/rfc/rfc8931 (November 2020), accessed 23 February 2023
62. Yang, P.: tcp: fix F-RTO may not work correctly when receiving DSACK. https://lore.kernel.org/netdev/165116761177.10854.18409623100154256898.git-patchwork-notify@kernel.org/t/, accessed 24 March 2023
63. Yen, J., Lévai, T., Ye, Q., Ren, X., Govindan, R., Raghavan, B.: Semi-automated protocol disambiguation and code generation. In: Proceedings of the 2021 ACM SIGCOMM 2021 Conference. pp. 272–286 (2021)
64. Zarchy, D., Mittal, R., Schapira, M., Shenker, S.: Axiomatizing congestion control. In: Proceedings of the ACM on Measurement and Analysis of Computing Systems (6 2019)