Model-Agnostic and Efficient Exploration of Numerical State Space of Real-World TCP Congestion Control Implementations

Wei Sun and Lisong Xu, *University of Nebraska-Lincoln;*
Sebastian Elbaum, *University of Virginia;* Di Zhao, *University of Nebraska-Lincoln*

https://www.usenix.org/conference/nsdi19/presentation/sun

This paper is included in the Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19).
February 26–28, 2019 • Boston, MA, USA
ISBN 978-1-931971-49-2

Model-Agnostic and Efficient Exploration of Numerical State Space of Real-World TCP Congestion Control Implementations

Wei Sun[1], Lisong Xu[1], Sebastian Elbaum[2], Di Zhao[1]

[1]*Department of Computer Science and Engineering, University of Nebraska-Lincoln*
Lincoln, NE, {wsun, xu, dzhao}@cse.unl.edu
[2]*Department of Computer Science, University of Virginia*
Charlottesville, Virginia, selbaum@virginia.edu

Abstract

The significant impact of TCP congestion control on the Internet highlights the importance of testing the correctness and performance of congestion control algorithm implementations (CCAIs) in various network environments. Many CCAI testing questions can be answered by exploring the numerical state space of CCAIs, which is defined by a group of numerical (and nonnumerical) state variables of the CCAIs. However, the current practices for automated numerical state space exploration are either limited by the approximate abstract CCAI models or inefficient due to the large space of network environment parameters and the complicated relation between the CCAI states and network environment parameters. In this paper, we propose an automated numerical state space exploration method, called ACT, which leverages the model-agnostic feature of random testing and greatly improves its efficiency by guiding random testing under the feedback iteratively obtained in a test. Our experiments on five representative Linux TCP CCAIs show that ACT can more efficiently explore a large numerical state space than manual testing, undirected random testing, and symbolic execution based testing, while without requiring an abstract CCAI model. ACT successfully detects multiple design and implementation bugs of these Linux TCP CCAIs, including some new bugs not reported before.

1 Introduction

TCP congestion control algorithms are crucial to Internet performance and stability. We have seen many of them emerged in the last decades [1, 6, 20, 43, 50], and we have witnessed how billions of computers, servers, routers, smartphones, and other Internet devices are affected, when new TCP Congestion Control Algorithm Implementations (CCAIs) are deployed, such as Linux CUBIC [20] and Windows Compound-TCP [43]. That is why a significant effort is placed in testing the correctness and performance of CCAIs in various network environments [16].

1.1 Numerical state space exploration

In this paper, we focus on how to explore the *numerical state space* $\mathbf{S}$[1] of a CCAI in various network environments. $\mathbf{S}$ is defined by a group of numerical state variables of the CCAI, such as congestion window size (*cwnd*), slow start threshold (*ssthresh*), and smoothed round-trip time (RTT, *rtt*). $\mathbf{S}$ may also have some additional nonnumerical state variables, such as the Linux TCP variable *ca_state* whose value indicates the current status of CCAI (e.g., 0:normal, 3:recovery, 4:timeout) but does not have numerical meanings. Space $\mathbf{S}$ contains all possible combinations of the values of the state variables, and each point in $\mathbf{S}$ is called a *state* or state vector. Exploring $\mathbf{S}$ aims to answer questions like the following.

Motivating Example 1: Does Linux CUBIC increase its *cwnd* appropriately in various network environments? The aggressiveness of CUBIC is determined by its state variable *target* [20], which is the expected congestion window size after one RTT. It is typically expected [15] that a CCAI increases its *cwnd* less aggressively in the congestion avoidance stage (i.e., when $cwnd > ssthresh$) than in the slow start stage (i.e., when $cwnd \leq ssthresh$) where it doubles its *cwnd* every RTT. This requirement can be tested by answering a numerical state space exploration question: does Linux CUBIC ever visit any states satisfying the condition $cwnd > ssthresh$ (i.e., congestion avoidance stage) and $target > 2 \times cwnd$ (i.e., more aggressively)?

Motivating Example 2: Does a Linux CCAI appropriately decrease its *cwnd* during fast recovery in various network environments? It is typically expected [2] that a CCAI decreases its *cwnd* in fast recovery when a congestion is detected (e.g., three duplicate ACKs). For example, CUBIC decreases its *cwnd* to $0.7 * prior\_cwnd$ and AIMD[2] [2] to $0.5 * prior\_cwnd$ right after a fast recovery, where state variable *prior_cwnd* is the congestion window size right before the fast recovery. This requirement can be roughly tested

[1]$\mathbf{S}$ is not to be confused with the TCP connection management state space [10] such as LISTEN, SYN-SENT, and CLOSED.

[2]Additive Increase Multiplicative Decrease of Reno and NewReno

by answering a numerical state space exploration question: does a Linux CCAI ever visit any states satisfying the condition $previous\_ca\_state == 3$ and $ca\_state == 0$ (i.e., just finished fast recovery) and $cwnd \geq prior\_cwnd$ (i.e., no window decrease at all)?

Similar to these two motivating examples, many CCAI requirements can be tested if we can explore the **S** of a CCAI in various network environments. Specifically, in this paper, we consider the *numerical state space exploration problem*: how to automatically sample a network environment parameter space **P** in order to efficiently visit as many as different regions of **S** within a given amount of testing time? Space **P** contains the parameter values of all possible network environments that a tester needs to check, and each point in **P** is called a *network environment* or network environment parameter vector. A region of **S** contains a group of nearby states in **S**, and is defined and discussed in Section 2.1.

1.2 Challenges

The numerical state space exploration problem, however, is challenging to solve. *The first challenge* is that space **P** is usually too large to check exhaustively. For example, suppose that a tester is testing a CCAI using a simple network topology with a single link, where the packet loss rate parameter is in the range of [0%, 10%] with a granularity of 10^{-6}, the link bandwidth parameter is in the range of [0.1, 10000] Mbps with a granularity of 0.1 Mbps, and the packet delay parameter is in the range of [0, 1000] ms with a granularity of 1 ms. The **P** of this simple example already contains about 10^{13} possible network environments (i.e., combinations).

The second challenge is that the mapping from the **P** to **S** of a CCAI is usually very complicated so that it is difficult to directly find a network environment in **P** that can lead the CCAI to visit certain regions in **S**. 1) A real-world CCAI, such as Linux CUBIC, involves multiple intertwined components contributed by tens of developers spanning tens of years. Many state variables, such as *cwnd*, are affected by multiple components, such as slow start, congestion avoidance, fast recovery, timeout, and undo components. 2) This is exacerbated by the fact that many states in a large **S** can be visited only after a large number of packets. For example, thousands of packets are needed in order to increase *cwnd* and *ssthresh* to over thousands of packets. That is, the exploration path from the start state to a final state may contain thousands of intermediate states. 3) There are currently no complete abstract models (e.g., state machines, or high-order logic) of real-world TCP implementations capturing all state variables and all components of the CCAIs, because they are very challenging to develop and verify. For example, a relatively complete TCP model [3] took several man-years of effort and deals with only the traditional AIMD.

Because of the unknown mapping from a large **P** to a large **S**, it is hard to efficiently explore **S** by either randomly or systematically sampling **P** and it is challenging to answer general numerical state space exploration questions, like the motivating examples.

1.3 Our contributions

We propose *an Automated Congestion control Testing method*, called ACT, to model-agnostically and efficiently explore a general numerical state space **S** of real-world CCAIs for a given **P**. ACT belongs to the class of feedback-guided random testing methods [28] or guided fuzzing methods [51] used in the software testing and verification community. While the general idea of feedback-guided random testing or guided fuzzing is not new, to the best of our knowledge, our work is the *first one* to use it in automatically exploring a large **S** of a CCAI. Specifically, ACT randomly selects network environments in a large **P** to explore a large **S**, and the random selection of new network environments is guided by the feedback iteratively obtained from the region coverage information of previously selected network environments. We propose two novel types of feedback to explore the low-probability regions of **S**: 1) parameter estimation to explore the low-probability regions due to the unknown nonlinear mapping from **P** to **S**, 2) parameter concatenation to explore the low-probability regions due to the correlation among the state variables of **S**. Intuitively ACT randomly samples in **P** but favoring those network environments that are more likely to explore different regions of **S**. By doing so, ACT is scalable to a large **P** (i.e., first challenge) and does not require an abstract CCAI model (i.e., second challenge).

Our contributions are threefold. First, we propose an automated and model-agnostic method, ACT, which can efficiently explore a large **S** for a large **P** without requiring an abstract CCAI model, and then output the states satisfying the specified conditions along with the concrete data necessary to deterministically reproduce the detected states.

Second, we present an ACT implementation using the widely used network simulator NS3 with Direct Code Execution (DCE) [44] to execute the original Linux networking stack. It can be easily used for testing, debugging, and studying the correctness and performance of real-world CCAIs in various reproducible and controllable network environments.

Third, we conduct a family of experiments on five representative Linux TCP CCAIs showing that ACT can more efficiently explore different regions of **S** than manual testing, undirected random testing, and symbolic execution based testing. ACT successfully detects multiple design and implementation bugs of these CCAIs, including *several new bugs not reported before*. For example, ACT finds that Linux CUBIC (current default) sometimes misjudges the network congestion and then mistakenly aggressively increases its throughput (i.e., motivating example 1). ACT also detects that Linux AIMD (previous default) sometimes mistakenly doubles its throughput right after a fast recovery (i.e., motivating example 2) or suddenly increases its throughput to an extremely large number.

2 Design of ACT

2.1 Regions of numerical state space S

Intuitively, a region of **S** contains a group of nearby states all satisfying or not satisfying a specified condition, and a region is visited if at least one state of the region is visited. We attempt to explore different regions of **S** instead of different individual states because of two unique properties of numerical state variables. 1) Because numerical state variables usually have a large number of possible values, the **S** of a CCAI is usually prohibitively large (e.g., in the order of 10^{11} states in our experiments). As a result, it is impossible to visit each individual state in **S** in any reasonably amount of testing time. 2) For a numerical state space exploration problem (e.g., the two motivating examples), there are usually one or multiple regions of nearby states (instead of only a single state) all satisfying the same condition. As long as we find at least one state in these regions (i.e., one counterexample), we can answer the exploration problem.

The shape and the size of a region might depend on the CCAIs, **S**, **P**, and the specified conditions. Without making any special assumption and for the sake of simplicity, we divide **S** into equal-sized non-overlapping regions of size k. Specifically, the range of each numerical state variable is divided into equal-sized intervals with size k, and the range of each nonnumerical state variable (if any) is divided into intervals with size 1. *A region contains all the states with each state variable in the same interval.* For example, let's consider a 2-dimensional $\mathbf{S} = \{(cwnd, ssthresh) \mid cwnd \in [1,1024],\ ssthresh \in [1,1024]\}$. If $k = 512$, **S** is divided into 4 equal-sized non-overlapping regions $\mathbf{S} = R_1(512) \cup R_2(512) \cup R_3(512) \cup R_4(512)$, where $R_i(k)$ denotes the i-th region when the region size is k. For instance, $R_1(512) = \{(cwnd, ssthresh) \mid cwnd \in [1,512],\ ssthresh \in [1,512]\}$, and $R_4(512) = \{(cwnd, ssthresh) \mid cwnd \in [513,1024],\ ssthresh \in [513,1024]\}$. In the extreme case of $k = 1$, **S** is divided into $1024 \times 1024 = 1{,}048{,}576$ regions $R_1(1), R_2(1), ..., R_{1048576}(1)$, each containing only one state. In another extreme case of $k = 1024$, the whole **S** is a single region $R_1(1024) = \mathbf{S}$.

Without making any special assumptions about the CCAIs, **S**, **P**, and the specified conditions, we do not consider a specific region size k. Instead, we attempt to explore as many as different regions for all possible k values within a given amount of testing time.

Note that, it is reasonable to group nearby states of **S** into regions, but it is not reasonable to group nearby network environments of **P**. This is because even a tiny difference between two network environments may lead to significantly different CCAI behaviors. For example, two packet loss rates of 10^{-5} and 10^{-6} with a tiny difference with respect to a parameter range $[0\%, 10\%]$ lead to about six times of different throughputs for CUBIC [20].

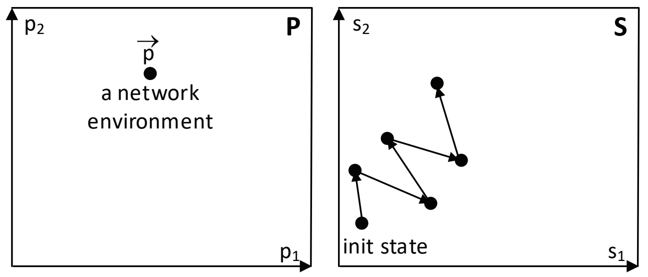

Figure 1: A network environment $\vec{p} \in \mathbf{P}$ leads a CCAI to visit a sequence of states in **S**.

2.2 Numerical state space exploration

Each network environment $\vec{p} \in \mathbf{P}$ leads a CCAI to visit a sequence of states in **S** starting from the initial state, as illustrated in Fig. 1 using a two-dimensional $\mathbf{P} = \{(p_1, p_2)\}$ and two-dimensional $\mathbf{S} = \{(s_1, s_2)\}$. In a network simulation, the sequence of visited states depends not only on $\vec{p}$ but also a random seed e, which are collectively referred to as a *simulation configuration* $G = (e, \vec{p})$. The simulation results (e.g., visited states) are deterministic for a given G.

The numerical state space exploration problem is given a number N, how to select N simulation configuration G's in order to maximize $coverage(\mathbf{S}, k)$ for any $k \geq 1$, where $coverage(\mathbf{S}, k)$ is the percentage of visited regions of **S** when the region size is k.

$$\max_{N \text{ selected } G's} coverage(\mathbf{S}, k) \qquad \forall\, k \geq 1 \tag{1}$$

Note that we attempt to maximize $coverage(\mathbf{S}, k)$, instead of exploring only a specific region of **S** for a specific condition that is nevertheless very challenging too. This is because state space exploration is time consuming, and it is more convenient to explore **S** once and then use the explored **S** to answer multiple different questions for the same **S**.

We say that a testing method is *more efficient* than another one, if given the same N, the $coverage(\mathbf{S}, k)$ of the former is higher than or equal to that of the latter for any $k \geq 1$. In this and next sections, we propose ACT to solve the numerical state space exploration problem, and in Section 4 we empirically evaluate the efficiency of ACT by comparing with other related exploration methods.

The design of ACT is based on the following theorem, where $|\mathbf{S}|$ denotes the total number of states in **S**. The proof is shown in the appendix.

Theorem 1 *Among all state exploration methods that visit state $i \in [1, |\mathbf{S}|]$ with probability q_i, the exploration method with $q_i = q_j$ for $\forall i, j \in [1, |\mathbf{S}|]$ maximizes $coverage(\mathbf{S}, k)$ for any $k \geq 1$.*

2.3 Feedback-guided random testing

Theorem 1 shows that the optimal exploration method should uniformly visit the states. While it is hard or impossible to design such an optimal exploration method, we attempt to design a method that visits as large as a fraction of **S** as possible within a limited testing time budget, instead of thoroughly visiting certain regions.

Our proposed ACT is based on undirected random testing [41] that randomly samples **P** accordingly to a distribution, because it is scalable to a large **P** and does not require an abstract model of a CCAI. Without making any special assumptions about the CCAIs, **S**, **P**, and the specified conditions, ACT uses the simple uniform distribution for the undirected random testing. However, because of the unknown and complicated mapping from **P** to **S**, undirected random testing tends to repeatedly visit the high-probability regions of **S** and thus is inefficient in covering different regions of **S**. In other words, *uniformly sampling* **P** *does not lead to uniform coverage of* **S**.

ACT leverages the model-agnostic feature of undirected random testing, and greatly improves the region coverage of **S** by guiding random testing under the feedback iteratively obtained in the test. Thus, ACT belongs to the class of feedback-guided random testing [28] or guided fuzzing methods [51]. We have identified two major reasons that undirected random testing has low probabilities to visit certain regions of **S**, and correspondingly propose two types of feedback to visit these low-probability regions of **S**: 1) parameter estimation to visit the low-probability regions due to the unknown nonlinear mapping from **P** to **S**, 2) parameter concatenation to visit the low-probability regions due to the correlation among different state variables of **S**.

2.4 Parameter estimation

One reason that undirected random testing has low probabilities to visit some regions of **S** is the unknown nonlinear mapping from **P** to **S**. For example, let's consider packet loss rate parameter *loss* in the range of [0%, 10%] and state variable *cwnd* in the range of [1, 1024] packets for AIMD. The average *cwnd* of AIMD is greater than 379 packets if *loss* is lower than 10^{-5} [15]. If *loss* is uniformly distributed in [0, 10%], the probability that $cwnd > 379$ is approximately lower than 0.01%, and thus the regions with $cwnd > 379$ have very low probabilities to be visited. With an unknown nonlinear mapping from **P** to **S**, it is impossible for undirected random testing with any specific distribution (not just uniform) to uniformly visit different states of **S**.

Parameter estimation attempts to visit the low-probability regions due to the unknown nonlinear mapping from **P** to **S**. Specifically, for an unvisited state $\overrightarrow{s^*} \in \mathbf{S}$, it attempts to find a network environment $\overrightarrow{p^*}$ such that the tested CCAI is likely to visit region $R(\overrightarrow{s^*}, k)$, which is the region of state $\overrightarrow{s^*}$ when the region size is k. ACT starts with the smallest region size $k = 1$ to find $\overrightarrow{p^*}$, if not successful, it gradually doubles k until it finds $\overrightarrow{p^*}$.

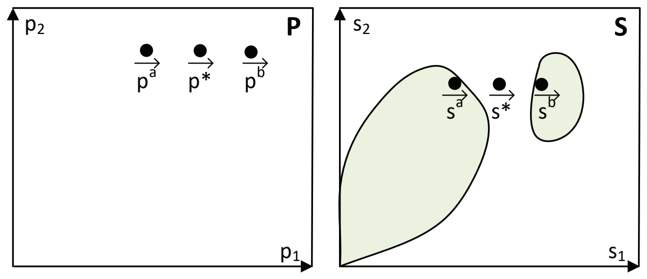

Figure 2: Interpolation finds $\overrightarrow{p^*}$ using $\overrightarrow{p^a}$ and $\overrightarrow{p^b}$ to cover the unvisited gap (e.g., the region of state $\overrightarrow{s^*}$) between two visited regions (e.g., the regions of $\overrightarrow{s^a}$ and $\overrightarrow{s^b}$).

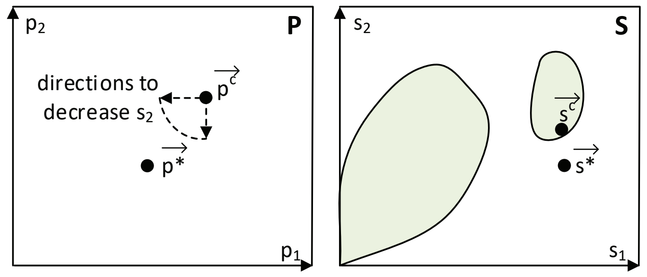

Figure 3: Extrapolation finds $\overrightarrow{p^*}$ using $\overrightarrow{p^c}$ to visit an unvisited corner or side of **S** (e.g., the region of $\overrightarrow{s^*}$ below the region of $\overrightarrow{s^c}$). The directions to decrease s_2 are for illustration purpose.

Parameter estimation is illustrated in Figs. 2 and 3 where shaded areas indicate the regions already visited by undirected random testing. The pseudo-code of parameter estimation is given in Method 1. Basically, for an unvisited state $\overrightarrow{s^*}$, we find a new network environment $\overrightarrow{p^*}$ using either the interpolation or extrapolation of the past selected network environments. Interpolation is used to cover the unvisited gap between two visited regions in **S**, such as state $\overrightarrow{s^*}$ in Fig. 2, and extrapolation is used to cover an unvisited corner or side of **S**, such as state $\overrightarrow{s^*}$ in Fig. 3.

To implement parameter estimation, each state $\vec{s} \in \mathbf{S}$ is associated with a pool of simulation configurations. Each simulation configuration $G = (e, \vec{p})$ contains the random seed e and the network environment $\vec{p}$ of a simulation that visited state $\vec{s}$. An unvisited state has an empty pool, and a visited state may have multiple simulation configurations if it has been visited multiple times by different simulations.

As an example of interpolation, for state $\overrightarrow{s^*}$ in Fig. 2, ACT randomly finds a pair of states $\overrightarrow{s^a}$ and $\overrightarrow{s^b}$ so that region $R(\overrightarrow{s^*}, k)$ lies in between $R(\overrightarrow{s^a}, k)$ and $R(\overrightarrow{s^b}, k)$ for the smallest possible k. In order to visit $R(\overrightarrow{s^*}, k)$, ACT estimates $\overrightarrow{p^*}$

Method 1 Parameter estimation to find a G^* for $\overrightarrow{s^*}$

```
function ESTIMATION($\overrightarrow{s^*}$)
    $e^* \leftarrow$ randomly selected random seed
    for region size $k \leftarrow 1$; ; $k \leftarrow 2 \times k$ do
        // First try interpolation
        find a pair of states $\overrightarrow{s^a}$ and $\overrightarrow{s^b}$ such that $R(\overrightarrow{s^*},k)$ lies in between $R(\overrightarrow{s^a},k)$ and $R(\overrightarrow{s^b},k)$.
        if find at least one pair of $\overrightarrow{s^a}$ and $\overrightarrow{s^b}$ then
            randomly and uniformly select one pair
            $G^a \leftarrow$ randomly and uniformly select one from the pool of simulation configurations associated with $\overrightarrow{s^a}$
            $\overrightarrow{p^a} \leftarrow$ the network environment in $G^a$ for $\overrightarrow{s^a}$
            $\overrightarrow{p^b} \leftarrow$ similarly a network environment for $\overrightarrow{s^b}$
            for $i$ from 1 to $dim(\mathbf{P})$ do
                $p_i^* \leftarrow$ random($p_i^a$, $p_i^b$)
            return $G^* \leftarrow (e^*, \overrightarrow{p^*})$
        // If interpolation fails, then do extrapolation
        find state $\overrightarrow{s^c}$ such that $R(\overrightarrow{s^c},k)$ and $R(\overrightarrow{s^*},k)$ differ in only one state variable.
        if find at least one state $\overrightarrow{s^c}$ then
            randomly and uniformly select an $\overrightarrow{s^c}$
            $\overrightarrow{p^c} \leftarrow$ a network environment for $\overrightarrow{s^c}$
            $j \leftarrow$ the state variable index that $R(\overrightarrow{s^c},k)$ and $R(\overrightarrow{s^*},k)$ differ
            for $i$ from 1 to $dim(\mathbf{P})$ do
                if $\frac{d\bar{s}_j}{dp_i}\Big|_{\overrightarrow{p^c}}$ and $s_j^* - s_j^c$ have same sign then
                    $p_i^* \leftarrow$ random($p_i^c$, max)
                else if different signs then
                    $p_i^* \leftarrow$ random(min, $p_i^c$)
                else                                        ▷ zero gradient
                    $p_i^* \leftarrow$ random(min, max)
            return $G^* \leftarrow (e^*, \overrightarrow{p^*})$
```

using the interpolation of the parameter vectors $\overrightarrow{p^a}$ and $\overrightarrow{p^b}$ of the pair of states. The interpolation is implemented by lines 4 to 13 of the pseudo-code. Because ACT does not make any assumption about the mapping from **P** to **S**, it randomly and uniformly selects a network environment $\overrightarrow{p^*}$ within the range of $\overrightarrow{p^a}$ and $\overrightarrow{p^b}$ instead of possibly a linear or some other interpolations.

As an example of extrapolation, for state $\overrightarrow{s^*}$ in Fig. 3, ACT randomly finds one state $\overrightarrow{s^c}$ lying beside $\overrightarrow{s^*}$ so that their regions $R(\overrightarrow{s^c},k)$ and $R(\overrightarrow{s^*},k)$ differ only in one state variable, say state s_j with $j \in [1, dim(S)]$, where $dim(\mathbf{S})$ denotes the dimension of **S**. That is, state $\overrightarrow{s^c}$ and $\overrightarrow{s^*}$ have a major difference only in s_j, and have similar other state variables. In order to visit $R(\overrightarrow{s^*},k)$, ACT estimates $\overrightarrow{p^*}$ using the extrapolation of network environment $\overrightarrow{p^c}$ (i.e., lines 14 to 27 of the pseudo-code). Specifically, the extrapolation estimates $\overrightarrow{p^*}$ by

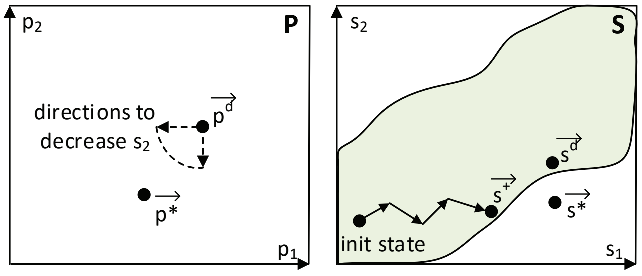

Figure 4: If s_1 and s_2 are positively correlated, $\overrightarrow{p^*}$ estimated by extrapolation leads to not only a smaller s_2 but also a smaller s_1, and thus visits the region of $\overrightarrow{s^+}$ instead of $\overrightarrow{s^*}$.

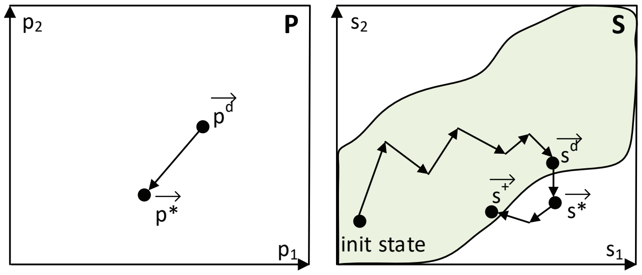

Figure 5: Parameter concatenation visits the region of $\overrightarrow{s^*}$ by first following the path from the initial state to state $\overrightarrow{s^d}$ using $\overrightarrow{p^d}$, and then the path from state $\overrightarrow{s^d}$ to state $\overrightarrow{s^+}$ using $\overrightarrow{p^*}$.

increasing or decreasing each parameter of $\overrightarrow{p^c}$ based on the impact of that parameter on state variable s_j. The impact of a parameter p_i ($i \in [1, dim(P)]$) on s_j is measured using the gradient of $\bar{s}_j$ with respect to p_i, where $\bar{s}_j$ is the average of all visited s_j values in a simulation and is defined as $\bar{s}_j = \frac{1}{T}\int_0^T s_j(t)dt$ with T as the simulation time. The gradient at $\overrightarrow{p^c}$ is estimated using the simulation results of undirected random testing. For example, states $\overrightarrow{s^*}$ and $\overrightarrow{s^c}$ in Fig. 3 differ mainly in state variable s_2, and specifically state $\overrightarrow{s^*}$ has a smaller s_2 than state $\overrightarrow{s^c}$. Then extrapolation estimates $\overrightarrow{p^*}$ by randomly adjusting $\overrightarrow{p^c}$ in the directions to decrease s_2.

2.5 Parameter concatenation

We notice that some regions of **S** have low probabilities to be visited by both undirected random testing and parameter estimation because of the correlation among the state variables of **S**. For example, state variables *cwnd* and *ssthresh* are positively correlated due to the window reduction at each congestion event (i.e., three duplicate acknowledgements), where *ssthresh* is set to a certain fraction of *cwnd* (e.g., CUBIC: $ssthresh = 0.7 * cwnd$, AIMD: $ssthresh = 0.5 * cwnd$). Because of this positive correlation, regions with very high *cwnd* values but very low *ssthresh* values and regions with very low *cwnd* values but very high *ssthresh* values have low probabilities to be visited by both undirected random testing and parameter estimation.

Let's use states $\overrightarrow{s^*}$ and $\overrightarrow{s^d}$ in Fig. 4 to illustrate why extrapolation does not work if there is a strong positive correlation between s_1 and s_2. Because $\overrightarrow{s^*}$ has a smaller s_2 than $\overrightarrow{s^d}$, extrapolation estimates $\overrightarrow{p^*}$ by randomly adjusting network environment $\overrightarrow{p^d}$ in the directions to decrease s_2. However, because of the positive correlation between s_1 and s_2, $\overrightarrow{p^*}$ leads to not only a smaller s_2 but also a smaller s_1. As illustrated in Fig. 4, $\overrightarrow{p^*}$ leads the tested CCAI to visit the region of state $\overrightarrow{s^+}$ by following the path from the initial state to $\overrightarrow{s^+}$, instead of visiting the expected region of $\overrightarrow{s^*}$.

Parameter concatenation attempts to visit the low-probability regions due to the state variable correlation. It is illustrated in Fig. 5 where the shaded area indicates all the region visited by the undirected random testing and parameter estimation. The pseudo-code is given in Method 2. Basically, parameter concatenation runs a network simulation with a list of network environments at different time periods in order to visit the unvisited region of state $\overrightarrow{s^*}$.

To implement parameter concatenation, we extend the simulation configurations used in parameter estimation. A simulation configuration associated with state $\overrightarrow{s^d}$ is changed to $G^d = (e, \overrightarrow{p^{d_1}}, t_1, \overrightarrow{p^{d_2}}, t_2, ..., \overrightarrow{p^{d_n}}, t_n)$, which means state $\overrightarrow{s^d}$ was visited by a simulation with random seed e, network environment $\overrightarrow{p^{d_1}}$ from the beginning to time t_1, $\overrightarrow{p^{d_2}}$ to time t_2, ..., and finally $\overrightarrow{p^{d_n}}$ visiting state $\overrightarrow{s^d}$ at time t_n. The visiting time t_n is added to the configuration by ACT during the simulation.

Parameter concatenation runs a network simulation using both the previous network environments $\overrightarrow{p^{d_1}}$, $\overrightarrow{p^{d_2}}$,..., $\overrightarrow{p^{d_n}}$ of $\overrightarrow{s^d}$ and the new network environment $\overrightarrow{p^*}$ estimated by extrapolation. At time t_n when the simulation just visits state $\overrightarrow{s^d}$, parameter concatenation changes the current network environment from $\overrightarrow{p^{d_n}}$ to $\overrightarrow{p^*}$. As illustrated in Fig. 5, such a list of network environments lead the tested CCAI to first visit state $\overrightarrow{s^d}$ by following the path from the initial state to $\overrightarrow{s^d}$, and then visit state $\overrightarrow{s^+}$ by following the path from $\overrightarrow{s^d}$ to $\overrightarrow{s^+}$.

The path from $\overrightarrow{s^d}$ to $\overrightarrow{s^+}$ in Fig. 5 may possibly visit new regions, such as the region of $\overrightarrow{s^*}$, which are not visited by the path from the initial state to $\overrightarrow{s^+}$ in Fig. 4 for two reasons. First, although both paths finally reach the same state $\overrightarrow{s^+}$ that is determined by network environment $\overrightarrow{p^*}$, they have different starting states and thus go through different paths.

Second, we observe that two state variables may be correlated strongly only over a long time scale but not in a short time scale. For example, over a long time scale, such as spanning multiple window reductions, *cwnd* and *ssthresh* are strongly correlated. But in a short time scale, such as within a congestion avoidance stage between two window reductions, they are weakly correlated in that only *cwnd* changes and *ssthresh* remains unchanged. For example, Fig. 6 shows the positive correlation between *cwnd* and *ssthresh* becomes stronger as the time scale n increases. Specifically, the Pearson's correlation coefficient is measured in a sliding window of 10 pairs of *cwnd* and *ssthresh* sampled every n-th ACKs in a simulation and averaged over 30,000 simulations. Because of the strong correlation between s_1 and s_2 in a long time scale, both the path from $\overrightarrow{s^d}$ to $\overrightarrow{s^+}$ in Fig. 5 and the path from the initial state to $\overrightarrow{s^+}$ in Fig. 4 reach the same state $\overrightarrow{s^+}$, which has both a smaller s_1 and a smaller s_2 than $\overrightarrow{s^d}$. But because of the weak correlation in a short time scale, the path from $\overrightarrow{s^d}$ to $\overrightarrow{s^+}$ in Fig. 5 may possibly visit the region of $\overrightarrow{s^*}$ where only s_2 is changed.

Method 2 Parameter concatenation to find a G^* for $\overrightarrow{s^*}$

```
1:  function CONCATENATION($\overrightarrow{s^*}$)
2:      for region size $k \leftarrow 1$; ; $k \leftarrow 2 \times k$ do
3:          find state $\overrightarrow{s^c}$ such that $R(\overrightarrow{s^d}, k)$ and $R(\overrightarrow{s^*}, k)$ differ in only one state variable
4:          if find at least one state $\overrightarrow{s^d}$ then
5:              randomly and uniformly select an $\overrightarrow{s^d}$
6:              $G^d \leftarrow$ randomly and uniformly select one from the pool of simulation configurations associated with $\overrightarrow{s^d}$
7:              $\overrightarrow{p^{d_n}} \leftarrow$ the last parameter vector in $G^d$
8:              $j \leftarrow$ the state variable index that $R(\overrightarrow{s^d}, k)$ and $R(\overrightarrow{s^*}, k)$ differ
9:              for $i$ from 1 to $dim(\mathbf{P})$ do
10:                 if $\frac{d\bar{s}_j}{dp_i}\Big|_{\overrightarrow{p^{cn}}}$ and $s^*_j - s^d_j$ have same sign then
11:                     $p^*_i \leftarrow$ random($p^{d_n}_i$, max)
12:                 else if different signs then
13:                     $p^*_i \leftarrow$ random(min, $p^{d_n}_i$)
14:                 else                                    ▷ zero gradient
15:                     $p^*_i \leftarrow$ random(min, max)
16:             $G^* \leftarrow$ append $\overrightarrow{p^*}$ to end of $G^d$
17:             return $G^*$
```

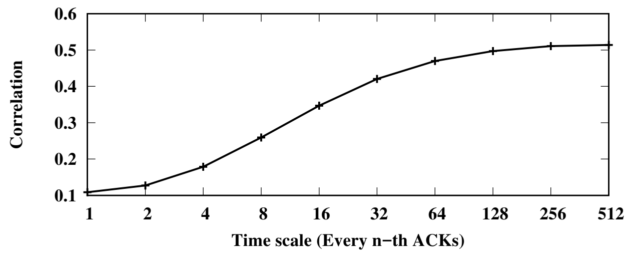

Figure 6: The longer the time scale, the stronger the positive correlation between *cwnd* and *ssthresh*.

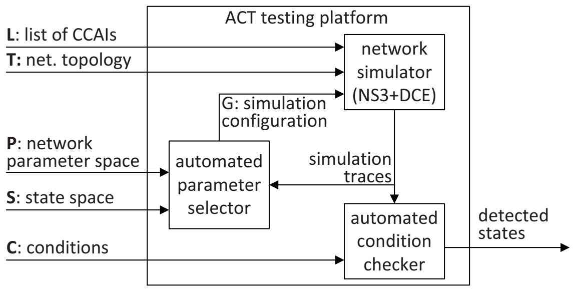
ACT testing platform
L: list of CCAIs
T: net. topology
P: network parameter space
S: state space
C: conditions
network simulator (NS3+DCE)
G: simulation configuration
automated parameter selector
simulation traces
automated condition checker
detected states

Figure 7: The ACT testing platform consists of three components: the existing network simulator NS3+DCE, our automated parameter selector, and automated rule checker.

3 Implementation of ACT

3.1 Testing platform

The ACT testing platform as illustrated in Fig. 7 takes as the input a list **L** of CCAIs, a state space **S** of the CCAIs, a network topology **T**, a network environment parameter space **P** for the topology, and a set **C** of state conditions to check. It automatically outputs the states satisfying the conditions along with specific network environments and other data necessary to deterministically reproduce the detected states.

The platform consists of three components. 1) A network simulator simulates CCAI flows of **L** in a network described by topology **T** and simulation configuration G that includes a random seed e and one or multiple network environments in **P**. We choose the widely used NS3 enabled with DCE [44], which can execute the original Linux networking stack in reproducible and controllable network environments. The output of each simulation is a trace of the timestamped CCAI state variables. 2) The automated parameter selector automatically selects network environments in **P** and generates the next simulation configuration G based on the feedback of the region coverage of the previously selected network environments. 3) The automated condition checker automatically checks whether any visited states satisfy the conditions in **C**.

3.2 Test input

A test input is a 5-tuple (**L**, **T**, **P**, **S**, **C**). CCAI list $\mathbf{L} = (l_1, l_2, ..., l_m)$ with $m \geq 1$ indicates the CCAIs of a total of m tested CCAI flows, where l_i with $i \in [1, m]$ is the CCAI of the i-th tested CCAI flow. That is, ACT can be used to test not only a single CCAI flow, but also the interaction among multiple different/same CCAI flows. Network topology **T** describes the topology (e.g., the total number of nodes and the routing information) of the tested network environments in which the m tested CCAI flows run. Leveraging the powerful NS3, our testing platform supports various types of network topologies, such as a single link, the dumbbell topology, and the parking lot topology. Network environment parameter space **P** describes the parameter ranges of the network topology **T**. Each point $\overrightarrow{p} \in \mathbf{P}$ is a network environment parameter vector $\overrightarrow{p} = (p_1, p_2, ...)$ (also called network environment), where p_i with $i \in [1, dim(\mathbf{P})]$ is a network environment parameter. CCAI state space **S** describes the possible states of the tested CCAI flows. Each point $\overrightarrow{s} \in \mathbf{S}$ is a state vector $\overrightarrow{s} = (s_1, s_2, ...)$ (also called state), where s_i with $i \in [1, dim(\mathbf{S})]$ is a numerical or nonnumerical state variable of a CCAI in **L**. **C** contains a set of the conditions of the state variables of the CCAIs, and is implemented as a script that reads and analyzes the simulation traces generated by NS3.

Different CCAI tests may need different test inputs. For example, a throughput test checks only a single CCAI flow whereas a fairness test checks multiple CCAI flows, and thus their test inputs have different **L**'s. Also the same CCAI state conditions may be used for different test inputs, for example, with different network topologies and/or parameter spaces. This paper focuses on the testing methods, and does not consider the design of comprehensive test inputs for CCAIs.

3.3 Test output

After a test, the testing platform reports all detected states satisfying the conditions. For each detected state, it outputs the corresponding simulation configuration G, which can be used to deterministically reproduce the detected state using NS3. In addition, it outputs the percentage of the regions covered in the test.

3.4 ACT method

ACT has the following four steps.

Step 1, *undirected random testing* repeatedly simulates CCAIs of **L** in a network specified by **T** and $G = (e, \overrightarrow{p})$ with randomly selected seed e and uniformly selected $\overrightarrow{p} \in \mathbf{P}$, until the coverage saturates. The goal of this step is not only to have an initial coverage of the state space, but also to profile the mapping from **P** to **S** to estimate the gradients used in parameter estimation and concatenation. Without making any assumptions for **L**, **T**, **P**, and **S**, ACT uses the simple uniform distribution for the undirected random testing.

Step 2, *parameter estimation* iteratively simulates CCAIs in a network specified by **T** and G^*=Estimation($\overrightarrow{s^*}$) for a uniformly selected unvisited state $\overrightarrow{s^*} \in \mathbf{S}$, until the coverage saturates. This step is used to improve the coverage of the low-probability regions due to the unknown nonlinear mapping from **P** to **S**.

Step 3, *parameter concatenation* iteratively simulates CCAIs in a network specified by **T** and G^*=Concatenation($\overrightarrow{s^*}$) for a uniformly selected unvisited state $\overrightarrow{s^*} \in \mathbf{S}$, until the coverage saturates. This step is used to improve the coverage of the low-probability regions due to the state variable correlation.

Step 4, *condition checking* reports all visited states in **S** satisfying the conditions in **C**.

ACT checks the coverage saturation using two parameters: saturation size κ and threshold δ. The coverage has reached saturation if the growth rate of $coverage(\mathbf{S},\kappa)$ is lower than δ. Note that, these two parameters are used only to determine the total testing time, and ACT still attempts to maximize $coverage(\mathbf{S},k)$ for all possible k values within that testing time. Smaller κ and δ increase $coverage(\mathbf{S},k)$ for all k values but require longer testing times.

4 Experiments

4.1 General setup

We consider five representative CCAIs of Linux kernel 3.10: the traditional AIMD [2], the current Linux default CUBIC [20], HTCP [33] as a time-based CCAI, HSTCP [15] as a high-speed CCAI, and VENO [17] as a delay-based CCAI. We choose Linux kernel 3.10 for two reasons. First, this is the Linux kernel extensively tested with DCE-enhanced NS3 [44], and thus we can minimize the impact of the potential DCE-enhanced NS3 bugs on our experiments. Second, all the tested CCAIs were initially developed before 2005, and their implementations were already relatively stable in Linux kernel 3.10 that was released in 2013. For all the experiments, we use the default TCP parameters of Linux kernel 3.10, except that the maximum buffer size is increased to not limit the TCP throughput.

Each CCAI has a default test input, which is mainly used for comparing the region coverage of different testing methods, so it does not have any conditions in $\mathbf{C}$. The default test inputs for different CCAIs are the same, except different $\mathbf{L}$. For example, the default test input for CUBIC has $\mathbf{L} = (\text{CUBIC})$, and the default test input for AIMD has $\mathbf{L} = (\text{AIMD})$. In every default test input, the network topology $\mathbf{T}$ has a single (virtual) link, which is simple and yet very powerful in simulating various network environments with random packet dynamics in terms of packet bandwidth, delay, loss, and reordering. The network environment parameter space $\mathbf{P}$ contains all possible network environments $\vec{p} = (p_1, p_2, p_3, p_4, p_5, p_6)$, with random packet loss rate $p_1 \in [0\%, 10\%]$ with granularity 10^{-6}, link bandwidth $p_2 \in [0.1, 10000]$ Mbps with granularity 0.1 Mbps, link delay $p_3 \in [1, 1000]$ ms with granularity 1 ms, random queuing delay following a Gamma distribution [26] with shape parameter $p_4 \in [0, 20]$ and scale parameter $p_5 \in [0, 80]$ both with granularity 0.01, and application rate $p_6 \in [0.001, 10000]$ Mbps with granularity 0.1 Mbps. The ranges of the parameters are selected to cover most of possible Internet conditions.

In every default test input, the state space $\mathbf{S}$ contains all possible states $\vec{s} = (s_1, s_2, s_3, s_4, s_5)$, where s_1 is the congestion window size variable $cwnd \in [1, 1024]$ packets with granularity 1 packet, s_2 is the slow start threshold variable $ssthresh \in [1, 1024]$ packets with granularity 1 packet, s_3 is the smoothed RTT variable $rtt \in [0, 2048]$ ms with default Linux granularity 4 ms, s_4 is the smoothed RTT deviation variable $rttvar \in [0, 1024]$ ms with granularity 4 ms, and s_5 is the congestion avoidance state variable $ca\_state \in$ [0:normal, 1:disorder, 2:cwr, 3:recovery, 4:timeout]. These variables are the basic CCAI state variables, and are maintained in the *tcp_sock* structure in the Linux kernel. The ranges of these state variables are selected to cover most of possible TCP states in the Internet, except that *cwnd* and *ssthresh* could be even larger for ultra-high-speed networks. In addition to these basic state variables, more state variables can be added into $\mathbf{S}$ depending on the tested conditions, such as congestion window size *prior_cwnd* right before fast recovery, and CCAI-specific variables like *target* for CUBIC.

In each experiment, each tested CCAI flow transfers a long file of size 15 MBytes, which is selected to be long enough to generate tens of thousands of packets so that all CCAIs can possibly increase their *cwnd* and *ssthresh* to over 1024 packets (i.e., their ranges in $\mathbf{S}$).

4.2 Evaluation: region coverage

We compare the region coverage of ACT with manual testing (MAN) and with other model-agnostic methods: undirected random testing (RAN) and symbolic execution based testing (SYM). We are unable compare ACT with model-guided methods, because there is no abstract model that can capture all state variables used in our experiments.

Methods: ACT: For each default test input, ACT runs DCE-enhanced NS3 simulations with the following saturation parameter values: κ=128, and δ=1.5% per 5000 simulation runs. That is, the coverage has reached saturation if the growth rate of $coverage(\mathbf{S}, 128)$ is slower than 1.5% per 5000 simulation runs. These parameter values are selected so that ACT can finish every test in about three days.

MAN: For each default test input, MAN repeatedly runs simulations with our manually selected network environments, which are similar to those selected for the response function test in a representative CCAI test [24]. Specifically, we consider packet loss rates p_1=0, 10^{-6}, 10^{-5}, ..., and 10^{-1}, bandwidths p_2=1, 10, 100, and 250 Mbps, link delays p_3=8, 20, 40, 80, and 160 ms, queuing delay shape values p_4=1 and 2.5 and scale values p_5=0, 1, and 10, and application rate p_6=10000 Mbps. There are a total of 840 network environments (i.e., combinations), and MAN repeatedly runs simulations with these network environments with different random seeds for the same total number of times as ACT.

RAN: For each default test input, RAN repeatedly runs simulations with uniformly and randomly selected network environments for the same total number of times as ACT.

SYM: Symbolic execution based testing [40, 46] executes the network simulator using symbolic execution platforms, where the packet dynamics (e.g., delay) are represented using symbolic variables with ranges defined according to $\mathbf{P}$. Because DCE-enhanced NS3 is a huge system where each simulated network node runs a virtualized Linux networking stack, we symbolically execute the simulations using a powerful symbolic execute platform S^{2}E [8], which is capable of symbolically testing a virtual machine. We find that SYM systematically checks all possible TCP behaviors including congestion control behaviors and non-congestion-control behaviors, such as all possible retransmissions and consecutive timeouts of the data packets, and all possible ways to establish and terminate a connection. As a result it can only test a TCP flow for a small file size of a few KBytes within three days instead of the expected 15-MByte file size, and thus it can only increase *cwnd* by a few packets that is far below our expected 1024 packets.

Result: We show the results of only CCAI $\mathbf{L}$ = (CUBIC) in Fig. 8, and the results of other CCAIs are similar. Fig. 8 shows the $coverage(\mathbf{S},k)$ results of ACT, RAN, and MAN, which are measured by the percentages of visited regions with size k. The region coverage of SYM is too low (lower than all others) and not shown in the figure. As k increases, the size of a region increases and the total number of regions in $\mathbf{S}$ decreases, and thus the region coverages of all methods increase. As an extreme case, when $k = 1024$, the whole state space $\mathbf{S}$ is treated as a single region, and thus all three methods achieve 100% coverage. It is interesting that MAN is more efficient (i.e., higher or the same coverage) than RAN for big regions but not for small ones. This is because the network environments used in MAN are representative network environments in $\mathbf{P}$ selected by TCP experts [24], and thus MAN covers a broader range of states than RAN. As a result, MAN is more efficient than RAN for big regions (i.e., $k > 4$). However, MAN has only a limited number of network environments (i.e., 840), and thus covers a smaller number of distinct states than RAN. As a result, MAN is less efficient than RAN for small regions (i.e., $k \leq 4$). We can see that *ACT is more efficient than MAN, RAN, and SYM for all possible region sizes*. Note that ACT achieves high coverage without requiring an abstract CCAI model.

Figs. 9 and 10 show the growth of $coverage(\mathbf{S},2)$ and $coverage(\mathbf{S},128)$, respectively. When $k = 2$, there are a total of about 10^{10} regions and all three methods achieve very small coverage percentages in three days. When $k = 128$, there are a total of 2048 regions and then all three methods achieve higher coverage percentages. We can see that ACT covers slightly more small regions (i.e., $k = 2$) than RAN, but significantly more big regions (i.e., $k = 128$) than RAN. This is because ACT uniformly selects unvisited states in $\mathbf{S}$ and thus is more likely to visit different big regions, whereas RAN uniformly selects parameter vectors in $\mathbf{P}$ and thus is more likely to redundantly visit the same big regions. Fig. 10 shows that ACT step 2 (i.e., estimation) without requiring an abstract CCAI model already achieves a higher coverage than both MAN and RAN, and ACT step 3 (i.e., concatenation) further greatly improves the coverage.

Note that when k is small (e.g., ≤ 16), all three methods including ACT achieve low coverage (e.g., $\leq 10\%$). This is because we only run each test for three days, and there are

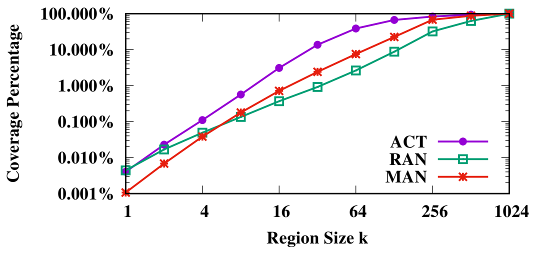

Figure 8: $coverage(\mathbf{S},k)$ with different k values.

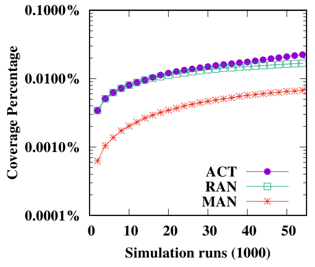

Figure 9: $coverage(\mathbf{S},2)$.

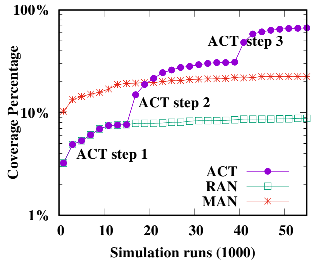

Figure 10: $coverage(\mathbf{S},128)$.

very large number of regions when k is small. For example, when k is 16, there are already 8,388,608 regions. In three days, RAN explores about 31,000 different regions, and ACT explores about 260,000 different regions. The coverages of all methods can be improved by running each test for a longer time by reducing parameters κ and δ. But our experiments already clearly demonstrate that ACT is significantly more efficient than MAN, RAN, and SYM giving the same amount of testing time.

4.3 Use case 1: Checking generic behaviors

We demonstrate the capability of ACT in detecting design and implementation bugs using three types of state conditions in the following three subsections, respectively: 1) a condition that checks generic CCAI behaviors, 2) a condition that checks the window increase behavior of a CCAI, 3) a condition that checks the window decrease behavior.

This group of experiments demonstrates that even a simple condition that checks generic CCAI behaviors might be useful for detecting bugs. The test inputs are the same as the default test inputs, except that $\mathbf{C}$ contains a simple condition: $cwnd > 10^7$ packets. Intuitively, this test checks whether the *cwnd* of a CCAI could be mistakenly larger than some upper bound, such as 10^7 packets that approximately corresponds to the throughput of a TCP flow with a rate of 100 Gbps and an RTT of 1000 ms. Note that although 10^7 is outside of the specified range $[1,1024]$ for *cwnd*, it is still possible for ACT to detect such states, because ACT keeps track of all the visited states, not just the states in the specified ranges. ACT with this simple condition detects *an implementation bug*. Due to a bug triggered by two consecutive undos, all tested CCAIs with *tcp_sack* disabled, except CUBIC, mistakenly

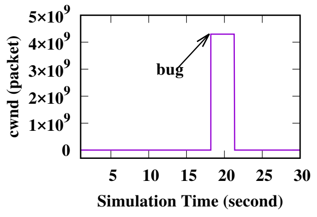

Figure 11: AIMD implementation bug: Suddenly extremely large *cwnd* after consecutive undos.

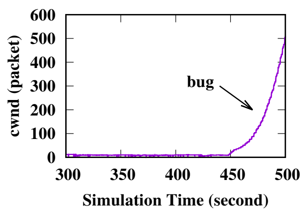

Figure 12: CUBIC design bug: Too aggressive after application rate-limited periods.

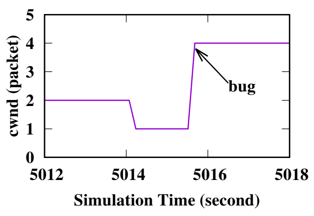

Figure 13: AIMD implementation bug: mistakenly increases *cwnd* after fast recovery.

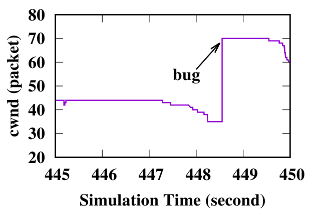

Figure 14: VENO implementation bug: mistakenly increases *cwnd* after fast recovery.

set *cwnd* to an extremely large number (i.e., 4,294,967,294 packets), as demonstrated in Fig 11. We thought that it was a new and severe bug and reported it to Linux kernel developers [38], and then were told that it was just fixed a few months ago.

4.4 Use case 2: Checking increase behavior

This group of experiments checks the first motivating example in Section 1. The test inputs are the same as the default test inputs, except that **C** contains a condition: $cwnd > ssthresh$ and $target > 2 \times cwnd$, and **S** contains additional *target*. Intuitively, this test checks whether CUBIC could be mistakenly more aggressive in congestion avoidance than in slow start. ACT detects multiple states satisfying this condition. There are three types of cases. 1) *New design bug* detected by ACT steps 1 and 2: CUBIC is designed to be a time-based congestion control algorithm, and its window increment in one RTT is a function of the duration of the RTT. As a result, in cases of extremely long propagation or queueing delays, CUBIC may set *target* to be higher than twice of the current *cwnd*, which is reasonable for long propagation delays but is questionable for long queueing delays that are possible signs of network congestion. This is an extreme case that we did not consider when we were designing CUBIC [20]. 2) *Design bug* detected by ACT step 3: Linux CUBIC mistakenly increases its *target* too aggressively after a long idle period. This bug was first reported in 2015 [25], and has been fixed in the latest Linux kernel. 3) *New design bug* shown in Fig. 12 detected by ACT step 3: Linux CUBIC mistakenly increases its *target* too aggressively after a long application rate-limited period. Both this and the previous bugs are special cases that we did not consider when we were designing and implementing CUBIC [20].

4.5 Use case 3: Checking decrease behavior

This group of experiments checks the second motivating example in Section 1. The test inputs are the same as the default test inputs, except that **C** contains a condition: $prior\_ca\_state == 3$, $ca\_state == 0$, and $cwnd \geq prior\_cwnd$, and **S** contains additional state variables used in the condition. Intuitively, this test checks whether a CCAI appropriately decreases its *cwnd* in fast recovery. ACT detects multiple states satisfying this condition, all by steps 1 and 2. There are two types of cases. 1) *New implementation bug* of AIMD and HTCP shown in Fig. 13. Due to a calculation boundary bug (happens only when $cwnd < 4$), AIMD and HTCP mistakenly increase *cwnd* to 4 after an undoed fast recovery. This is a new bug and was recently fixed after we reported it to Linux kernel developers [39]. *This is an important bug, because in a highly congested network where we desperately need CCAIs, this bug makes the network even more congested.* 2) *Implementation bug* of VENO and HSTCP shown in Fig. 14. VENO and HSTCP mistakenly double their *cwnd* after an undoed fast recovery, because they mistakenly use the default undo function that was designed for AIMD. This bug has been reported before and was fixed in 2016 [47].

5 Discussions

What domain knowledge is required to use ACT? An ACT user needs to know the state variables of a tested CCAI (e.g., by reading the related RFC or papers) in order to define the state space **S**. In addition, currently the user needs to manually instrument the source code of CCAIs and NS3 to keep track of the values of the state variables. The contribution of our work is that ACT is model-agnostic so that the user does not need to know how multiple intertwined components of CCAIs change the state variables and does not need to know the complicated mapping from **P** to **S**.

An ACT user needs to know the correct behavior of a tested CCAI (e.g., by reading the related RFC or papers) in order to define the set of conditions **C**. In addition, an ACT user needs to manually analyze the outputted simulation traces with buggy behavior (i.e., satisfying the conditions) and then manually check the source code of CCAIs to identify the reasons for the bugs. The contribution of our work is that ACT efficiently searches an extremely large number of possible network environments **P**, and automatically finds the specific network environments where the tested CCAIs show the buggy behavior, so that the user only needs to manually analyze the specific simulation traces with buggy behavior.

What kind of CCAIs ACT can or can't test? Although we haven't evaluated ACT for all current CCAIs, we conjecture that ACT works for general current and future CCAIs for the following two reasons. First, ACT does not make any specific assumptions about the network environment parameters of **P** and the state variables of **S**, except that **S** contains mainly numerical state variables and state variables should not be strongly correlated in a short time scale. Second, ACT checks the general behaviors of a tested CCAI by analyzing the impact of **P** on **S**, instead of checking the detailed implementations of the CCAI by analyzing its source code. While different CCAIs may have quite different implementations (e.g., loss based or delay based, expert designed or computer generated, kernel space or user space), they have same or similar general behaviors (e.g., increase or decrease *cwnd* based on network congestion). Having said that, an important future work is to evaluate the effectiveness of ACT for new CCAIs, such as BBR [6], Remy [48], and PCC Vivace [12].

What kind of bugs ACT can or can't detect? ACT can be used to detect the bugs that can be described by state variables of **S**, like the two motivating examples. ACT does not work well for the bugs related to the specific packet behaviors, such as whether an acknowledgement packet with the correct acknowledgment number is sent right after receiving a data packet, because it is hard or impossible to describe such a behavior as a condition of state variables. In addition, ACT does not work well for bugs that happen only with certain TCP configuration parameters, because ACT does not search the large space of TCP configuration parameters.

Are there false positives and false negatives? ACT does not have false positives, because ACT can output the specific network environments and the actual simulation traces for each reported bug. However, ACT does have false negatives as it is possible that a tested CCAI satisfies a condition but ACT could not find it. This is because ACT attempts to maximize the region coverage of **S** within a testing time budget, instead of covering all regions which requires an unrealistically long time for small k values. Intuitively, this implies that ACT can be used for bug detection but not for correctness guarantee, which is consistent with a fundamental testing principle "Program testing can be used to show the presence of bugs, but never to show their absence" [11] in the software testing community. For real-world networking systems, correctness can be verified only for special cases, such as for the abstract models of the code [34, 37], for code built on verified libraries [52], and for partial pieces of code [42].

6 Related work

6.1 TCP numerical state space exploration

Three types of methods can be potentially used to address TCP numerical state space exploration problems. 1) These problems are usually studied by *manual testing* [19, 24], where a tester manually selects some representative network environments in **P** to test whether a CCAI visits certain regions in **S**. Not only is manual testing unscalable to a prohibitively large **P** (e.g., only an order of 10^2 network environments are selected in [19, 24]), but also the effectiveness of manual testing highly depends on how much the tester knows about a CCAI.

2) *Automated and model-guided methods* such as [22] have the potential to automatically and efficiently explore a limited **S** of a CCAI under the guidance of an abstract model of the CCAI. But the choice of **S** is limited by the state variables captured in the abstract model. For example, the model used in [22] does not capture CUBIC state variable *target*, and thus cannot be used to explore the **S** of CUBIC in the first motivating example. More importantly, the regions of **S** that can be explored are limited by the CCAI components captured in the abstract model. For example, the model used in [22] does not capture the undo component of Linux CCAIs. As a result, it is unable to guide the exploration of the regions that can be reached by the undo component, and then hard to detect the bugs caused by the interference between the undo and fast recovery components in the second motivating example. However, there is currently no complete abstract model for real-world CCAIs, as described in the second challenge.

3) *Automated and model-agnostic methods*, such as undirected random testing [41] and symbolic execution based testing [40, 46], can automatically explore a general **S** of a CCAI without requiring an abstract CCAI model. However, they are inefficient to explore different regions of a large **S**, because they blindly visit **S** and as a result tend to repeatedly or densely explore some regions of **S**. Symbolic execution based testing [40, 46] groups all the network environments leading to exactly the same CCAI execution path into equivalence classes in order to improve the scalability over exhaustive testing that exhaustively tests each $\vec{p} \in \mathbf{P}$. However, it is still inefficient in exploring different regions of a large **S** for the following reasons. First, it still blindly explores **S**, because different equivalence classes of network environments may still repeatedly or densely explore the same regions of **S**. Second, the number of equivalence classes of network environments is still prohibitively large, and is roughly an exponential function of the number of packets (i.e., path explosion problem [4]). As a result, it can be used to test CCAI with only a small number of packets [40, 46] or test partial code of CCAIs [42].

ACT attempts to combine the advantages of the model-guided and model-agnostic methods, that is, the efficiency of model-guided methods and the generality of model-agnostic methods. First, ACT is based on undirected random testing instead of symbolic execution based testing, so that it is scalable to a large **P** and a large number of packets. Second, ACT guides the selection of network environments under the feedback iteratively obtained in a test, so as to select new network environments that are more likely to explore different regions. As a result, ACT does not blindly explore **S**, and can more efficiently explore different regions of **S** than undirected random testing and symbolic execution based testing.

6.2 Enhancements to random testing

The efficiency of random testing can be improved by incorporating the right guidance, such as feedback-based random testing [28] and guided fuzzing (e.g., AFL [51]), or by combining with symbolic execution in various ways (e.g., DART [18], Driller [36], MACE [9]). The major difference between these techniques and our proposed ACT is that these techniques explore the general program execution state space by maximizing the code coverage or edge coverage, whereas ACT explores the specific numerical state space of CCAIs where maximizing code coverage may not always be helpful. First, maximizing code coverage might waste the testing resources on covering code with no or little impact on congestion control, such as the TCP code related to connection management or packet formats. Second, many congestion control states can be explored only by repeatedly visiting the already visited code blocks for many times. For example, in order for AIMD to reach from a state with $cwnd = ssthresh = 500$ packets to another state with $cwnd = 1000$ packets and $ssthresh = 500$ packets, AIMD needs to repeatedly visit the same additive increase code for $500+501+502+\ldots+999 = 374,750$ times.

The efficiency of random testing can also be improved using genetic algorithms [29], where new test inputs can be generated by recombining two existing test inputs (called crossover), or by randomly changing one existing test input (called mutation). The parameter estimation of ACT is inspired by genetic algorithms. Specifically, the interpolation is inspired by the crossover, as it generates a new network environment by combining two existing network environments. The extrapolation is inspired by mutation, as it generates a new network environment by changing one existing network environment. The major difference between ACT and genetic algorithms is the parameter concatenation of ACT that concatenates a sequence of network environments instead of combining them into a single network environment, as interpolation and extrapolation (similarly crossover and mutation) do not work well for **S** with correlated state variables.

6.3 General state space exploration

In addition to random testing, many automated techniques have been proposed to explore various state spaces (e.g., program execution space, TCP connection management space) of network programs.

Implementation-level model checking techniques [27, 31] recursively explore the next states from the start state by enumerating all possible events at each state. They are effective for systematically exploring a small state space, but are not scalable to a large one [28]. The path explosion problem [4] limits symbolic execution based techniques [32,35,40,42] to testing only a small number of packets [40], a component of a network protocol [42], or an abstract network model [37]. Static analysis techniques [7,13,45] analyze the network programs at compilation time to infer their run-time behaviors. These techniques [13] are effective at quickly checking shallow behaviors of large programs, but not at accurately checking the deep program behaviors, such as finding the exact network environments that lead a CCAI to certain states after thousands of packets. Model learning techniques [14, 21] attempt to automatically build an abstract model and then explore the state space of the model. But they work only for a small state space.

The major difference between all above techniques and our proposed ACT is that these techniques attempt to explore different individual states and are more suitable for small state spaces, such as nonnumerical state spaces (e.g., TCP connection management state space [27]) or small numerical state spaces of simple protocols (e.g., TFTP [40]), whereas ACT is specifically designed to efficiently explore different regions of an extremely large numerical state space of CCAIs where certain regions can be reached only after thousands of intermediate states (i.e., thousands of packets).

6.4 Other related TCP testing work

Pantheon [49] provides a training ground for evaluating the performance of CCAIs in real-world settings and can automatically calibrate the parameters of a network emulator to match a real network path so that a tested CCAI achieves similar average throughput and delay, whereas ACT attempts to maximize the coverage of the whole state space and then detect bugs. PacketDrill [5] is an automated TCP testing tool that checks whether TCP meets a requirement in a specific network environment $\overrightarrow{p}$, whereas ACT checks whether a CCAI meets a requirement in a large space **P** of network environments. Automated trace analysis [3, 23, 30] checks the correctness of TCP packet traces against some formal models or rules mainly about the TCP connection establishment and termination, whereas ACT checks the correctness of TCP congestion control.

7 Conclusion

This paper proposes a CCAI testing tool ACT, and presents several design and implementation bugs of Linux TCP. Most of them are due to the mismatch among different TCP components, because they were designed by different researchers but their interfaces are evolving and not clearly defined. In the future, we plan to extend ACT to test other congestion control algorithms, such as those based on UDP and those in information-centric networking.

ACKNOWLEDGMENT

We thank our shepherd, Anirudh Sivaraman, and the reviewers for their constructive comments. The work presented in this paper was supported in part by NSF CNS-1526253 and NSF SHF-1718040.

Appendix

Proof: Let $I(k)$ denote the total number of regions in $\mathbf{S}$ when the region size is k. Let $Q_i(k)$ denote the probability to visit region $R_i(k)$ with $i \in [1, I(k)]$, which is the probability that at least one state in region $R_i(k)$ is visited. In the special case when $k = 1$, we have $I(k) = |\mathbf{S}|$ and $Q_i(k) = q_i$.

Suppose that a method runs the network simulator for N times and each time visits M states in $\mathbf{S}$. The probability that region $R_i(k)$ is visited at least once is $1-(1-Q_i(k))^{N\times M}$. The expected number of visited regions is $coverage(S,k) = \sum_{i=1}^{I(k)} \left(1-(1-Q_i(k))^{N\times M}\right)$. Thus, the numerical state space exploration problem can be rewritten as follows.

$$\text{Maximize} \qquad \sum_{i=1}^{I(k)} \left(1-(1-Q_i(k))^{N\times M}\right) \qquad (2)$$

$$\text{Subject to} \qquad \sum_{i=1}^{I(k)} Q_i(k) = 1 \qquad (3)$$

Using the Karush-Kuhn-Tucker conditions, we can prove that the maximum coverage is achieved when $Q_i(k) = Q_j(k)$ for $\forall i, j \in [1, I(k)]$. If and only if $q_i = q_j$ for $\forall i, j \in [1, |\mathbf{S}|]$, we have $Q_i(k) = Q_j(k)$ for $\forall i, j \in [1, I(k)]$ and for any $k \geq 1$. That is, given the same amount of testing time (i.e., the same N), the uniform distribution is the only one that maximizes $coverage(S,k)$ for any $k \geq 1$. ▪

References

[1] M. Alizadeh, A. Greenberg, D. Maltz, and J. Padhye et al. Data center TCP (DCTCP). In *Proceedings of ACM SIGCOMM*, New Delhi, India, August 2010.

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

[3] 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.

[4] C. Cadar and K. Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, February 2013.

[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.

[6] N. Cardwell, Y. Cheng, C. Gunn, S. Yeganeh, and V. Jacobson. BBR: Congestion-based congestion control. *Coomunications of the ACM*, 60(2):pp. 58–66, February 2017.

[7] 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 CCS*, Denver, CO, October 2015.

[8] V. Chipounov, V. Kuznetsov, and G. Candea. The S2E platform: design, implementation, and applications. *ACM Transactions on Computer Systems*, 30(1), February 2012.

[9] C. Cho, D. Babic, P. Poosankam, K. Chen, E. Wu, and D. Song. MACE: model-inference-assisted concolic exploration for protocol and vulnerability discovery. In *Proceedings of USENIX Conference on Security (SEC)*, San Francisco, CA, August 2011.

[10] DARPA Internet Program. Transmission control protocol – protocol specification. *RFC 793*, September 1981.

[11] E. Dijkstra. *Notes on Structured Programming in Book Structured Programming*. Academic Press, 1972.

[12] M. Dong, T. Meng, D. Zarchy, E. Arslan, Y. Gilad, P. Godfrey, and M. Schapira. PCC Vivace: Online-learning congestion control. In *Proceedings of USENIX NSDI*, Renton, WA, April 2018.

[13] 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, January 2004.

[14] 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*, Canada, July 2016.

[15] S. Floyd. HighSpeed TCP for large congestion windows. *RFC 3649*, December 2003.

[16] S. Floyd and M. Allman. Specifying new congestion control algorithms. *RFC 5033*, August 2007.

[17] C. Fu and S. Liew. TCP Veno: TCP enhancement for transmission over wireless access networks. *IEEE Journal on Selected Areas in Communication*, 21(2):216–228, February 2003.

[18] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proceedings of ACM Programming Language Design and Implementation*, Chicagi, IL, June 2005.

[19] S. Ha, L. Le, I. Rhee, and L. Xu. Impact of background traffic on performance of high-speed TCP variant protocols. *Computer Networks*, 51(7):1748–1762, May 2007.

[20] S. Ha, I. Rhee, and L. Xu. CUBIC: A new TCP-friendly high-speed TCP variant. *ACM SIGOPS Operating System Review*, 42(5):64–74, July 2008.

[21] Y. Hsu, G. Shu, and D. Lee. A model-based approach to security flaw detection of network protocol implementations. In *Proceedings of IEEE ICNP*, Orlando, FL, October 2008.

[22] 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 NDSS*, San Diego, CA, February 2018.

[23] D. Lee, D. Chen, R. Hao, R. Miller, J. Wu, and X. Yin. Network protocol system monitoring - a formal approach with passive testing. *IEEE/ACM Transactions on Networking*, 14(2):424–437, 2006.

[24] Y. Li, D. Leith, and R. Shorten. Experimental evaluation of high-speed congestion control protocols. *IEEE/ACM Transactions on Networking*, 15(5):1109–1122, October 2007.

[25] 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.

[26] A. Mukherjee. On the dynamics and significance of low frequency components of Internet load. *Internetworking: Research and Experience*, 5:163–205, December 1994.

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

[28] C. Pacheco, S. Lahiri, M. Ernst, and T. Ball. Feedback-directed random test generation. In *Proceedings of International Conference on Software Engineering (ICSE)*, Minneapolis, MN, May 2007.

[29] R. Pargas, M. Harrold, and R. Peck. Test-data generation using genetic algorithm. *Journal of Software: Testing, Verification and reliability*, 9(4):263–282, December 1999.

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

[31] W. Rathje and B. Richards. A framework for model checking UDP network programs with Java Pathfinder. In *Proceedings of ACM High Integrity Language Technology (HILT) International Conference*, Portland, OR, October 2014.

[32] R. Sasnauskas, O. Landsiedel, M. Alizai, C. Weise, S. Kowalewski, and K. Wehrle. KleeNet: discovering insidious interaction bugs in wireless sensor networks before deployment. In *Proceedings of ACM/IEEE IPSN*, Stockholm, Sweden, April 2010.

[33] R. Shorten and D. Leith. H-TCP: TCP for high-speed and long-distance networks. In *Proceedings of PFLDNet*, Argonne, IL, February 2004.

[34] M. Smith and K. Ramakrishnan. Formal specification and verification of safety and performance of TCP selective acknowledgment. *IEEE/ACM Transactions on Networking*, 10(2):193–207, August 2002.

[35] J. Song, C. Cadar, and P. Pietzuch. SymbexNet: Testing network protocol implementations with symbolic execution and rule-based specifications. *IEEE Transactions on Software Engineering*, 40(7):695–709, July 2014.

[36] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbette, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: augmenting fuzzing through selective symbolic execution. In *Proceedings of NDSS*, San Diego, CA, Feburary 2016.

[37] R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu. SymNet: Scalable symbolic execution for modern networks. In *Proceedings of ACM SIGCOMM*, Brazil, August 2016.

[38] W. Sun. A bug report for Linux TCP congestion control algorithms, May 2017. https://patchwork.ozlabs.org/patch/767239/.

[39] W. Sun. A buggy behavior for Linux TCP Reno and HTCP, July 2017. Report https://www.spinics.net/lists/netdev/msg444955.html, Fix https://patchwork.ozlabs.org/patch/797520/.

[40] W. Sun, L. Xu, and S. Elbaum. Improving the cost-effectiveness of symbolic testing techniques for transport protocol implementations under packet dynamics. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, Santa Barbara, CA, July 2017.

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

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

[43] K. Tan, J. Song, Q. Zhang, and M. Sridharan. A compound TCP approach for high-speed and long distance networks. In *Proceedings of IEEE INFOCOM*, Barcelona, Spain, April 2006.

[44] H. Tazaki, F. Uarbani, E. Mancini, M. Lacage, D. Camara, T. Turletti, and W. Dabbous. Direct code execution: revisiting library OS architecture for reproducible network experiments. In *Proceedings of ACM CoNEXT*, Santa Barbara, CA, December 2013.

[45] 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.

[46] 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 INFOCOM*, Paris, France, April 2019.

[47] F. Westphal. TCP: make undo_cwnd mandatory for congestion modules, November 2016. https://www.mail-archive.com/netdev@vger.kernel.org/msg138481.html.

[48] K. Winstein and H. Balakrishnan. TCP ex machina: computer-generated congestion control. In *Proceedings of ACM SIGCOMM*, Hong Kong, China, August 2013.

[49] 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, July 2018.

[50] P. Yang, J. Shao, W. Luo, L. Xu, J. Deogun, and Y. Lu. TCP congestion avoidance algorithm identification. *IEEE Transactions on Networking*, 22(4):1311–1324, August 2014.

[51] M. Zalewski. American Fuzzy Lop for network fuzzing. https://github.com/jdbirdwell/afl.

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