CC-Fuzz: Genetic Algorithm-based Fuzzing for Stress Testing Congestion Control Algorithms.

Devdeep Ray
Carnegie Mellon University

Srinivasan Seshan
Carnegie Mellon University

ABSTRACT

Recent congestion control research has focused on purpose-built algorithms designed for the special needs of specific applications. Often, limited testing before deploying a CCA results in unforeseen and hard-to-debug performance issues due to the complex ways a CCA interacts with other existing CCAs and diverse network environments. We present CC-Fuzz, an automated framework that uses genetic search algorithms to generate adversarial network traces and traffic patterns for stress-testing CCAs. Initial results include CC-Fuzz automatically finding a bug in BBR that causes it to stall permanently, and automatically discovering the well-known low-rate TCP attack, among other things.

CCS CONCEPTS

• **Networks** → **Network performance analysis**; **Protocol testing and verification**; **Transport protocols**;

KEYWORDS

Congestion Control, Fuzz Testing, Genetic Algorithm

ACM Reference Format:
Devdeep Ray and Srinivasan Seshan. 2022. CC-Fuzz: Genetic Algorithm-based Fuzzing for Stress Testing Congestion Control Algorithms..
In *The 21st ACM Workshop on Hot Topics in Networks (HotNets '22), November 14–15, 2022, Austin, TX, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3563766.3564088

1 INTRODUCTION

Recent networking research has shown an increased interest in designing custom congestion control algorithms (CCAs) for meeting application-specific performance goals (e.g. SCReAM [11], GoogCC [7], Sprout [22] for low latency video streaming) or for specific network environments (e.g. Swift [12], DCTCP [1], CCAs for hybrid optical networks [14] for data-center network environments). In addition, CCAs like Copa [3], Nimbus [9], and TCP-BBR [6] use complex, stateful network modeling techniques in order to achieve their performance targets. It is important to evaluate the robustness of a CCA and it's implementation across a wide range of scenarios before it is deployed in the wild. For CCAs developed by the academic community, the opportunities for large scale real-world testing are limited. Many newly proposed CCAs are evaluated using small scale deployments [23], and local, scenario-based emulation and simulation [15]. Testing of these new, complex CCAs performed at an academic scale can easily miss situations where the algorithm fails to achieve it's goals (like high utilization, fairness, or low delay [9, 19]), or corner cases where implementation bugs are triggered.

In this paper, we describe the design of our testing framework called "CC-Fuzz[1]", and demonstrate how genetic algorithms (GAs) for searching the space of link and cross traffic patterns can be used to identify issues with CCAs and their implementations, and inspire confidence in a CCA before it is deployed. GAs are search heuristics inspired by the Darwinian theory of biological evaluation - on each iteration, each entity in a gene pool is assigned a fitness score that depicts the chances of survival of an entity across generations (akin to natural selection). In our case, the entities are network traces, the fitness score of a trace is determined by the performance of the CCA under that trace, and evolution involves modifying/eliminating traces based on the fitness scores such that a trace that causes poor performance is more likely to survive. In order to generate realistic network traces, CC-Fuzz (1) uses heuristics during trace generation, and (2) leverages the generality of GAs, using carefully designed fitness scores for implicitly modeling trace properties that are hard to implement using heuristics. In Section 5, we propose an alternate way to impose realism on network traces as part of future work.

CC-Fuzz currently uses NS3 [8] for simulating CCAs and assigning fitness scores for traces. In the future, we plan to use emulation of CCA implementations, since CCA implementations in NS3 can sometimes differ from the real implementations. Results summary (§ 4):

HotNets '22, November 14–15, 2022, Austin, TX, USA

ACM ISBN 978-1-4503-9899-2/22/11.
https://doi.org/10.1145/3563766.3564088

[1]CC-Fuzz is a pun on Sisyphus. Wikipedia notes, "tasks that are both laborious and futile are therefore described as Sisyphean" [21]

(1) **BBR** - CC-Fuzz found traces that cause BBR to permanently stall due to the way ACKs and spurious retransmissions interact with each other during a retransmission timeout, and found traffic patterns that trigger BBR to cause high queuing delays.
(2) **CUBIC** - CC-Fuzz found a bug in NS3's CUBIC implementation regarding CWND updates.
(3) **Reno** - CC-Fuzz found traffic patterns that are similar to the TCP low-rate attack [13].

In the remainder of the paper, we discuss the design of CC-Fuzz (§ 3) and present directions for future work (§ 5).

2 MOTIVATION AND RELATED WORK

CCAs are often evaluated using metrics such as throughput, delay and fairness across a limited range of simple scenarios, like varying bandwidth at macroscopic time-scales, and making the CCA compete with other CCAs. Past work has shown that commonly evaluated scenarios often fail to catch surprising failure modes - In [19], the authors use mathematical modeling to show that multiple BBR flows are unfair towards loss-based CCAs. In CCAC [2], the authors argue that basic evaluation techniques are not sufficient for capturing every scenario that causes undesirable behavior, and propose a formal technique that generates network behavior in response to queries about CCA performance. Formal approaches are limited - they analyze "theoretical models" of CCAs, which overlooks potential bugs in real implementations, and become intractable for high-fidelity models and modeling longer time-scale behaviors.

Fuzzing [17] is a widely used technique for discovering vulnerabilities in code. TCPwn [10] uses model-based fuzzing in order to identify manipulation attacks (e.g. dup ACK injection, ACK storm, sequence desynchronization) on CCAs. TCP-Fuzz [24] tests TCP stack implementations for bugs.

The goal of CC-Fuzz is to find realistic situations where CCA performance suffers due to packet delivery timing and losses *automatically*, and not bugs that are triggered by injecting spoofed packets - the tools mentioned above can be used for protocol-level bug finding. ACT [16] and Packetdrill [5] are perhaps the closest in spirit to CC-Fuzz. ACT searches for numerical values of the state variables in the CCA implementation that represent a bad performance state - in contrast, CC-Fuzz directly searches for realistic network traces that lead to poor performance. Packetdrill [5] uses scripted tests to detect bugs in the networking stack and for regression testing of CCAs. Packetdrill requires clearly laid out networking scenarios that must be developed by hand - CC-Fuzz automatically generates such scenarios based on high-level performance goals.

Algorithm Genetic Algorithm Loop

```
procedure CC-Fuzz
    TRACES ← Initial pool of traces
    kElite ← Number of traces that live on unmodified.
    kCrossover ← Count of new traces generated by
                 combining traces with high scores.
    repeat
        for trace ∈ TRACES do
            SCORE(i) ← Score when CCA run with
                       TRACES(i)
        ELITE ← Top kElite traces
        CROSSOVER ← kCrossover traces that are
                    generated by combining traces
        MUTATED ← len(TRACES) - kElite - kCrossover
                  traces generated by modifiying traces
        TRACES ← ELITE + CROSSOVER + MUTATED
    until convergence
```

Figure 1

3 DESIGN

CC-Fuzz uses a GA for generating network traces that cause a CCA to perform poorly. CC-Fuzz's high level loop is described in Figure 1. CC-Fuzz's core components include the following:

(1) **Trace Generator**: Generates initial traces, performs cross-overs between trace pairs, and mutates traces.
(2) **Scoring Function**: Assigns a score based on the property being evaluated (e.g. throughput, delay, loss, or a combination) based on simulated/emulated results.
(3) **Selection Algorithm**: Selects traces for cross-overs and mutations for the next generation.

These components are discussed in further detail below.

3.1 Network Model

CC-Fuzz uses a simple network topology with two sources (one source uses the CCA being tested, and the other source generates cross traffic) that are connected to a gateway with high speed links. The gateway is connected to a sink via a bottleneck link with a fixed propagation delay. The gateway consists of a fixed-size drop-tail FIFO queue.

CC-Fuzz's current design separates link-based fuzzing (searching the space of bottleneck service curves with a fixed queue size), and cross-traffic based fuzzing (searching the space of cross-traffic patterns on a fixed-rate link with a fixed queue size). These two approaches can trigger different behaviors - link-fuzzing models a variable link with unbounded delay jitter (e.g. wireless with link-layer retransmissions), whereas in traffic-fuzzing, the maximum delay

Algorithm Packet Distribution Algorithm

```
procedure DistPackets(num, start, end)
    if num == 0 then return [ ]
    if num == 1 then return [ (start+end)/2 ]
    rate ← num/(end−start)
    loop
        tsplit ← U(start, end)
        numleft ← U(0, num)
                          ▷ U is uniform random sampling.
        if end − start < kAgg then break
        lrate ← numleft/(tsplit−start), rrate ← (num - numleft)/(end−tsplit)
        if lrate > 2 × rate or rrate > 2 × rate then
            continue
        if lrate < 0.5 × rate or rrate < 0.5 × rate then
            continue
    return DistPackets(numleft, start, tsplit)
              + DistPackets(num - numleft, tsplit, end)
```

Figure 2

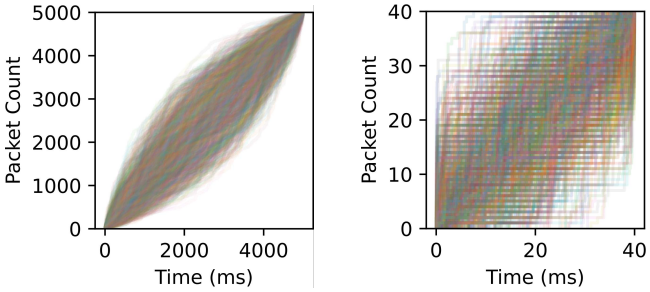
is bounded (e.g. fixed rate wired link with variable cross traffic). In addition, while a realistic link may exhibit aggregation, delay jitter, and some degree of long-term temporal rate variation, cross-traffic can be highly adversarial. Note that these two modes do not cover every scenario, e.g. random packet losses. We defer evaluation of CC-Fuzz on more comprehensive network models to future work (§ 5).

3.2 Link Fuzzing

A link trace (bottleneck service curve) is represented as packet transmission opportunities (similar to MahiMahi [15]). This representation lends itself well to modeling unbounded packet delays (large gaps between two packet transmissions). The duration and total number of packets is fixed for the entire run (i.e. fixed average bandwidth).

Initial Trace Generation. CC-Fuzz uses heuristics to limit the range of long-term bandwidth variation, while allowing jitter and aggregation. This is done using DistPackets (Figre 2), which recursively divides packets by splitting the trace duration and number of packets into two in each step, and ensures that the average rate for each partition is within a multiplicative range of the average rate. Deeper in the recursion, when the duration is below *kAgg*, the rate check is disabled to model packet aggregation and jitter. Figure 3 shows sample service curves generated by this algorithm at two time-scales.

Evolution Mechanism. When creating a new generation from a pool of link traces, CC-Fuzz must ensure that the same



(a) 5 second interval. **(b) 50 millisecond interval.**

Figure 3: Service curves generated using DistPackets, with an average rate of 12 Mbps and *kAgg* = 50 ms.

properties as that of the initial generation hold - otherwise, the constraints imposed by DistPackets can be violated significantly after a few generations. Mutations generate new traces for the next generation by slightly perturbing some traces with desirable properties in the current generation. For mutations, CC-Fuzz selects a random split point in the trace, and redistributes packets (DistPackets) either before or after the split point (chosen using a coin toss). This inductively preserves the properties imposed by DistPackets. Crossover picks two or more traces that have desirable properties, and combines them to generate a new trace. CC-Fuzz currently does not use crossovers for link fuzzing, since we could not come up with a combining function that preserves rate variation and average properties.

3.3 Traffic Fuzzing

CC-Fuzz uses the same algorithm (DistPackets) for generating traffic traces, with some modifications.

(1) **Trace generation heuristics:** We eliminate the local rate constraints, allowing arbitrary traffic bursts.
(2) **Crossover operation:** Without the local rate constraints, the crossover operation as follows: randomly choose a split point by packet count, randomly select the left half of one trace and the right half of the other trace around the split point, and combine the two sets of timestamps.

In the case of traffic fuzzing, it is desirable to generate "minimal" traffic vectors that induce poor behavior in CCAs. E.g., a large traffic burst where many cross-traffic pakcets are lost will behave the same even if the lost packets aren't transmitted. In addition, cross traffic at the bottleneck queue when the CCA is quiescent (e.g. when TCP is waiting for ACKs after filling the CWND) has no impact. CC-Fuzz enforces a maximum average rate instead of a fixed average rate for traffic fuzzing. When mutating a portion of the trace, the number of packets in that portion are changed randomly.

During a crossover operation, the number of traffic packets also changes depending on the number of packets in the trace on the right side. This is combined with a "total traffic" penalty in the scoring function (§ 3.4), which steers the GA towards minimizing the traffic vectors.

3.4 Scoring Function

The scoring function is a key aspect of a genetic algorithm - it determines which traces were successful in triggering specific performance behavior, and allows implicit modeling of desirable properties in a link or traffic trace. As part of calculating the score for a trace, CC-Fuzz runs the CCA using the link or traffic trace (simulated using NS3. Emulation using tools like MahiMahi can also be used - comparison in Section 3.6), and analyzes the queuing behavior. The score assigned to each trace in a generation has two components: performance score and trace score.

Performance Score. The performance score can be designed for specific types of poor behavior like high loss rate, high delay or low utilization. For quantifying low utilization, CC-Fuzz calculates windowed throughput for the run, and takes the average of the lowest 20% of the windows. This prevents algorithmic bias towards traces that trigger poor behavior early on, which improves trace diversity.

Trace Score. CC-Fuzz can also assign a separate score to the trace itself to implicitly impose additional properties of the traces which are hard to model using heuristics. For example, CC-Fuzz scores traffic traces using the (negation of) total traffic packets and the total traffic packets dropped in order to make the GA prefer minimal traces where few traffic packets are lost.

3.5 Selection Algorithm

Once the traces in a generation have been assigned scores, we rank the traces from highest score to lowest score. We first pick *kElite* of the highest scored traces that make it to the next generation unchanged. We assign a relative probability of $\frac{1}{rank}$ to each trace and then choose *kCrossover* pairs of traces according to these probabilities, and combine them for generating crossover traces. The same probabilities (based on rank) are used for picking traces that undergo mutation for generating the rest of the traces in order to maintain a constant population size.

3.6 Emulation vs. Simulation

CC-Fuzz currently simulates CCAs using NS3 to calculate the CCA performance score for each trace. An alternative is emulation (e.g. using MahiMahi). In either case, CC-Fuzz will test a combination of the CCA implementation and the run-time framework, finding failures in either system and their interactions.

The benefit of emulation is the ability to test a real implementation of a CCA. Unfortunately, emulating multiple traces in parallel in a reproducible manner is challenging. We need to ensure that the performance is not affected due to CPU and memory bottlenecks, and that the start time of a flow is synchronized with the network trace. Otherwise, the CCA behavior can be very different across generations for a given trace, which can delay or even prevent convergence of the genetic algorithm.

Simulation, on the other hand, will generate identical results across repeated runs, resulting in faster convergence. In addition, for link rates in 10s of Mbps, simulation is likely to be faster than real-time emulation, and the results of the simulation are not affected by machine load - this makes it easy to massively parallelize the algorithm on a single machine. The key drawback of simulation is that it does not test the actual implementation, but a re-implementation in the simulation framework (e.g. NS3). Tools like DCE [18] can mitigate this drawback by simulating real network stacks.

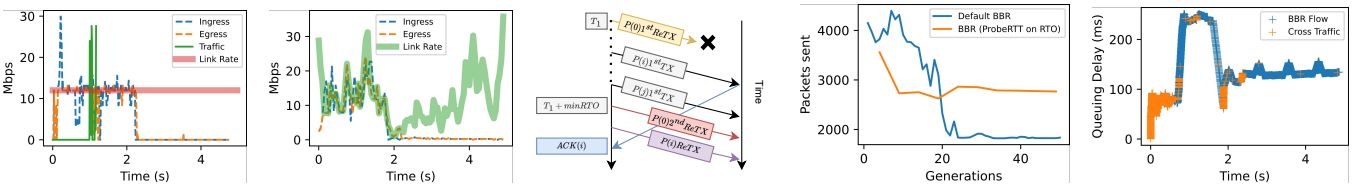
In addition, randomization in a CCA's implementation can also prevent convergence. In such cases, we need to modify the CCA implementation so that the randomization is repeatable (fix the random seed). This is much easier in a simulated setup as opposed to modifying kernel CCA code in an emulated environment. In the future, we plan to explore the use of emulation for CC-Fuzz.

4 FINDINGS

In this section, we will discuss some interesting findings that CC-Fuzz was able to automatically discover. For all of our tests, we set the bottleneck bandwidth to 12 Mbps (average bandwidth in the case of link fuzzing) and set the propagation delay of the bottleneck link to 20 ms. TCP-SACK and delayed ACKs are enabled (Linux defaults), and min-RTO is set to 1 second (as per RFC 6298/2.4, Linux uses 200 ms). We use a population size of 500, and use an island-isolation [20] strategy with 20 islands for solution diversity, where 10% of the traces migrate every 10 generations. Across island generations, the best trace is preserved (*kElite = 1*), 30% of the traces are crossovers, and the rest are mutations. We simulate each CCA for around 5 seconds. CC-Fuzz took approximately 5 seconds per generation, where we parallelized the simulation across 32 cores on an Intel Xeon machine.

4.1 BBR - Stuck Throughput

We tested NS3's version of TCP-BBR with CC-Fuzz, and after a few generations, it produced traces that triggered low throughput for BBR where it get's stuck permanently. One such trace is shown in Figure 4a. We verified that this issue was specific to BBR by running Reno and Cubic on such traces, and they worked as expected. For understanding



(a) CC-Fuzz traffic trace that causes BBR to get stuck.

(b) CC-Fuzz link trace that causes BBR to get stuck.

(c) Timeline showing how BBR's bug is triggered.

(d) CC-Fuzz performance with and without BBR patch.

(e) CC-Fuzz triggering high delays in BBR with cross traffic.

Figure 4: Analyzing BBR with CC-Fuzz.

the root cause, we dug into NS3 code and generated various internal logs from BBR's code and from the NS3 TCP socket code.

BBR uses an 8-RTT gain cycle for estimating bandwidth, where it sends at 1.25X the current bandwidth estimate for the first RTT, 0.75X on the second RTT and at 1X for the rest of the gain cycle. Each RTT is considered as a probing round. The measured rate in each probing round is processed through a windowed max-filter that keeps the estimates from the last 10 rounds of probing.

We found the root cause to be BBR's mechanism for timing it's bandwidth probing cycles in terms of RTT. For each packet, the TCP send buffer tracks the number of bytes delivered when that packet was sent in the SKB. At the beginning of a probing round, BBR records the number of bytes delivered so far. The probe ends when the prior delivered of the packet most recently ACK (i.e. bytes delivered when the ACKed packet was sent) exceeds the bytes delivered at the beginning of the probing round.

Suppose a packet $P(0)$ is transmitted at time T_0, and is lost. Fast retransmit will cause the first retransmission to occur at some time $T_1 > T_0 + RTT$, and an RTO timer will be set for $T_1 + minRTO$. At $T_1 + minRTO$, $P(0)$ is retransmitted for the second time. Suppose $P(i)...P(j)$ were the last few packets sent before the second retransmission for $P(0)$, and the SACKs for these have not arrived yet. After transmitting $P(0)$ for the second time, $P(i)$ will be transmitted again (a spurious retransmission). Here, the prior delivered for $P(i)$ is updated in the SKB for $P(i)$ to the current bytes delivered. If the SACK for the original transmission of $P(i)$ arrives right after the second transmission of $P(i)$, BBR will prematurely end the current probe cycle, since the value of prior delivered for $P(i)$ increased when the spurious retransmission was sent, and now likely exceeds threshold at which the current probing round was supposed to end. This sequence of events is depicted in Figure 4c. Thus, BBR's rate sample is now incorrect, as it is using the time and bytes delivered between the ACK for the original packet, and the packet's spurious retransmission, to calculate the rate. This can result in a low value for the bandwidth sample. This can repeat for the other packets $P(i+1)...P(j)$ that were in-flight when $P(0)$ was transmitted the second time. If this continues for 10 or more packets, the true bandwidth estimates in the bandwidth max-filter expire, and BBR's bandwidth estimate becomes low. With a very low bandwidth estimate, delayed ACKs can cause a positive feedback loop, causing BBR to send slower and slower, stalling BBR indefinitely. It is possible that this is the same issue being referenced in [4].

CC-Fuzz was able to trigger this behavior with both, link fuzzing and traffic fuzzing. Figure 4b shows a link trace generated by CC-Fuzz that triggers the same bug. The traffic trace generated by CC-Fuzz is very easy to understand - CC-Fuzz's implicit constraints on traffic traces generate a clean, minimal trace. On the other hand, despite our trace annealing mechanism significantly smoothing out the bandwidth variations, the link trace is harder to reason about. In the future, we plan to implement better heuristics and implicit constraints in order to generate easier to understand link traces that trigger poor behavior.

In order to try and mitigate this behavior, we made BBR trigger a minRTT probe when an RTO occurs - this slows down BBR momentarily which allows BBR to receive the in-flight ACKs, and thus avoid the spurious retransmissions that cause poor RTT-clocking for BBR's bandwidth probes. Figure 4d plots the average of the top 20 traces with the lowest throughput in each generation. Our proposed fix reduces throughput a little bit, but avoids the permanent stalling behavior observed in BBR without the fix.

4.2 TCP-CUBIC Incorrect CWND Update

When testing TCP-CUBIC, we discovered a bug in NS3's implementation of CUBIC's window update during slow start. When a packet is lost, and it's retransmission triggered by fast retransmit is also lost, the CCA goes into slow start after RTO. The sender performs a second retransmission for the packet, and when the ACK for this is received, there is a large jump in the cumulative ACK. CUBIC's slow start window-increase function is called with the large number

of segments ACKed. At this point, the CWND must only be increased upto the slow-start threshold. In NS3, this check is not performed, and the congestion window is increased by a large value - causing CUBIC to send almost 1-RTO (1 second in the case of NS3) worth of pending data, causing catastrophic losses. This leads to CUBIC going into slow start again. As of commit `60e1e403`, this bug is still present in NS3. This computation is performed correctly in the Linux kernel source code.

4.3 Other Findings

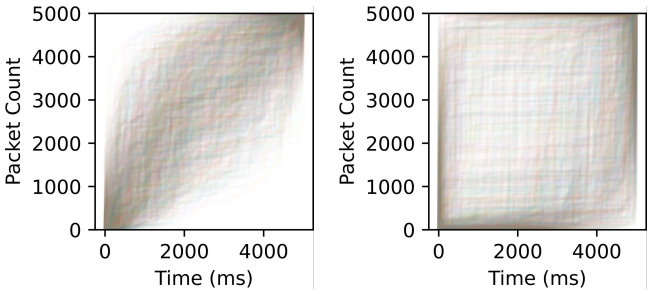
For TCP-Reno, CC-Fuzz was able to find a traffic trace similar to the well-known low-rate TCP attack [13] for a single flow, where traffic bursts cause the same packet sequence to get lost after each retransmission, which triggers exponential RTO back-off. This prevented Reno from ever ramping up after the initial slow start phase. CC-Fuzz can also test CCAs for goals other than low throughput, by just changing the performance component of the scoring function. For example, we ran traffic fuzzing on BBR with the goal of inducing high delays by setting the score function to the 10th percentile delay. This caused CC-Fuzz to generate a traffic vector that (1) fills up the queue just before BBR starts, so that BBR cannot see the true link RTT, and (2) injects traffic right after BBR's slow start phase to accelerate queue-growth caused by BBR. This is shown in Figure 4e.

5 FUTURE WORK

Realism Scoring. The current version of CC-Fuzz uses a heuristic-based approach for generating realistic traces. In the future, we plan to explore an alternate technique for generating realistic traces using aggregate performance across multiple CCAs as a score function to quantify the realism of a trace, assigning high scores to traces under which at least a few algorithms perform well, and vice versa. Figure 5 shows the traces accepted and rejected by this mechanism. Note how traces that have low bandwidth initially and higher bandwidth later are rejected - such traces will naturally cause low throughput in most CCAs.

This approach can be thought of as "differential" testing, where the GA tries to find traces where one CCA performs poorly, whereas other CCAs work fine. In order to reduce the amount of computation required, the realism score can be computed every few iterations instead of every iteration, or can be computed for a single randomly chosen CCA instead of all CCAs in each generation.

Diversity and Semantic Scoring. Currently, CC-Fuzz tends to converge at a point where most traces trigger the easiest to induce performance bug. In order to find other bugs, an iterative process of fixing the bug and retesting, or



(a) Valid traces. **(b) Invalid traces.**

Figure 5: Distribution of service curves according to realism scores assigned by testing on multiple CCAs. The traces were generated with DistPackets, but without the local rate constraints.

defining a score function that negatively weights the manifestation of that bug can be used. In the future, we plan to explore techniques that automatically result in a diverse set of bugs being found automatically by using machine learning to classify different behavior and dropping traces that trigger similar bugs across generations. We also plan to implement a framework to translate logical specifications of performance goals into score functions, so that the user does not have to come up with complex score functions themselves in order to make CC-Fuzz work.

Random Losses and Combined Fuzzing. Random packet losses are common on wireless links. CC-Fuzz's two modes, link, and traffic fuzzing, do not cover scenarios where random losses occur without a corresponding queue build up. It is not difficult to also do this in CC-Fuzz, we just limited our scope for this submission. In the future, we plan to explore loss fuzzing in order to increase the testing coverage, and also explore combining link, traffic, and loss fuzzing into a single process. Combined fuzzing will result in much more complex network traces that include link variations, cross traffic and loss - these are harder to understand, and thus, it is harder to pin-point the bug. We plan to address the challenge of generating easier to understand network traces in order to aid debugging.

6 CONCLUSION

In this paper, we have presented the design of an automated congestion control testing tool, CC-Fuzz. Our results are highly promising with an initial prototype of CC-Fuzz are finding both known and unknown issues with existing well-tested CCAs. We believe that with further development, CC-Fuzz could fill an important gap in the development of new CCAs by providing a simple way to identify settings in which a particular CCA performs poorly.

REFERENCES

[1] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference*. 63–74.

[2] Venkat Arun, Mina Tahmasbi Arashloo, Ahmed Saeed, Mohammad Alizadeh, and Hari Balakrishnan. 2021. Toward Formally Verifying Congestion Control Behavior. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference (SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 1–16. https://doi.org/10.1145/3452296.3472912

[3] Venkat Arun and Hari Balakrishnan. 2018. Copa: Practical {Delay-Based} Congestion Control for the Internet. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. 329–342.

[4] BBR-Development. 2022. Question on strange BBR behavior. https://groups.google.com/g/bbr-dev/c/XUOKHJiAW80. (2022). [Online; accessed 23-June-2022].

[5] Neal Cardwell, Yuchung Cheng, Lawrence Brakmo, Matt Mathis, Barath Raghavan, Nandita Dukkipati, Hsiao-keng Jerry Chu, Andreas Terzis, and Tom Herbert. 2013. packetdrill: Scriptable network stack testing, from sockets to packets. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. 213–218.

[6] Neal Cardwell, Yuchung Cheng, C Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. 2017. BBR: congestion-based congestion control. *Commun. ACM* 60, 2 (2017), 58–66.

[7] Gaetano Carlucci, Luca De Cicco, Stefan Holmer, and Saverio Mascolo. 2016. Analysis and design of the google congestion control for web real-time communication (WebRTC). In *Proceedings of the 7th International Conference on Multimedia Systems*. 1–12.

[8] Gustavo Carneiro. 2010. NS-3: Network simulator 3. In *UTM Lab Meeting April*, Vol. 20. 4–5.

[9] Prateesh Goyal, Akshay Narayan, Frank Cangialosi, Srinivas Narayana, Mohammad Alizadeh, and Hari Balakrishnan. 2018. Elasticity detection: A building block for internet congestion control. *arXiv preprint arXiv:1802.08730* (2018).

[10] Samuel Jero, Md Endadul Hoque, David R Choffnes, Alan Mislove, and Cristina Nita-Rotaru. 2018. Automated Attack Discovery in TCP Congestion Control Using a Model-guided Approach.. In *NDSS*.

[11] Ingemar Johansson and Zaheduzzaman Sarker. 2017. *Self-clocked rate adaptation for multimedia*. Technical Report.

[12] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan MG Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, et al. 2020. Swift: Delay is simple and effective for congestion control in the datacenter. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*. 514–528.

[13] Aleksandar Kuzmanovic and Edward W Knightly. 2003. Low-rate TCP-targeted denial of service attacks: the shrew vs. the mice and elephants. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*. 75–86.

[14] Matthew K Mukerjee, Christopher Canel, Weiyang Wang, Daehyeok Kim, Srinivasan Seshan, and Alex C Snoeren. 2020. Adapting {TCP} for Reconfigurable Datacenter Networks. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 651–666.

[15] Ravi Netravali, Anirudh Sivaraman, Keith Winstein, Somak Das, Ameesh Goyal, and Hari Balakrishnan. 2014. Mahimahi: A lightweight toolkit for reproducible web measurement. *ACM SIGCOMM Computer Communication Review* 44, 4 (2014), 129–130.

[16] Wei Sun, Lisong Xu, Sebastian Elbaum, and Di Zhao. 2019. {Model-Agnostic} and Efficient Exploration of Numerical State Space of {Real-World}{TCP} Congestion Control Implementations. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 719–734.

[17] Michael Sutton, Adam Greene, and Pedram Amini. 2007. *Fuzzing: brute force vulnerability discovery*. Pearson Education.

[18] Hajime Tazaki, Frédéric Urbani, and Thierry Turletti. 2013. DCE Cradle: Simulate network protocols with real stacks. In *Workshop on NS3 (WNS3)*.

[19] Ranysha Ware, Matthew K Mukerjee, Srinivasan Seshan, and Justine Sherry. 2019. Modeling bbr's interactions with loss-based congestion control. In *Proceedings of the internet measurement conference*. 137–143.

[20] Darrell Whitley, Soraya Rana, and Robert B Heckendorn. 1999. The island model genetic algorithm: On separability, population size and convergence. *Journal of computing and information technology* 7, 1 (1999), 33–47.

[21] Wikipedia contributors. 2022. Sisyphus — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Sisyphus&oldid=1091831787. (2022). [Online; accessed 23-June-2022].

[22] Keith Winstein, Anirudh Sivaraman, and Hari Balakrishnan. 2013. Stochastic forecasts achieve high throughput and low delay over cellular networks. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. 459–471.

[23] Francis Y Yan, Jestin Ma, Greg D Hill, Deepti Raghavan, Riad S Wahby, Philip Levis, and Keith Winstein. 2018. Pantheon: the training ground for Internet congestion-control research. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 731–743.

[24] Yong-Hao Zou, Jia-Ju Bai, Jielong Zhou, Jianfeng Tan, Chenggang Qin, and Shi-Min Hu. 2021. {TCP-Fuzz}: Detecting Memory and Semantic Bugs in {TCP} Stacks with Fuzzing. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 489–502.