Automated Attack Discovery in TCP Congestion Control Using a Model-guided Approach

Samuel Jero
Purdue University
sjero@purdue.edu

Endadul Hoque[†]
Florida International University
ehoque@fiu.edu

David Choffnes, Alan Mislove and Cristina Nita-Rotaru
Northeastern University
{choffnes,amislove,crisn}@ccs.neu.edu

***Abstract*—One of the most important goals of TCP is to ensure fairness and prevent congestion collapse by implementing congestion control. Various attacks against TCP congestion control have been reported over the years, most of which have been discovered through manual analysis. In this paper, we propose an automated method that combines the generality of implementation-agnostic fuzzing with the precision of runtime analysis to find attacks against implementations of TCP congestion control. It uses a model-guided approach to generate *abstract attack strategies*, by leveraging a state machine model of TCP congestion control to find vulnerable state machine paths that an attacker could exploit to increase or decrease the throughput of a connection to his advantage. These abstract strategies are then mapped to *concrete attack strategies*, which consist of sequences of actions such as injection or modification of acknowledgements and a logical time for injection. We design and implement a virtualized platform, TCPWN, that consists of a a proxy-based attack injector and a TCP congestion control state tracker that uses only network traffic to create and inject these concrete attack strategies. We evaluated 5 TCP implementations from 4 Linux distributions and Windows 8.1. Overall, we found 11 classes of attacks, of which 8 are new.**

I. Introduction

TCP is the protocol that underlies most of the Internet traffic including encrypted traffic via TLS and HTTPS. In addition to reliable and in-order data delivery, TCP has two critical goals – efficient delivery based on network conditions and fairness with respect to other TCP flows in the network. These two goals are achieved by using congestion control mechanisms that cause a sender to adapt its sending rate to the current network conditions (*e.g.*, network congestion) or to the receiver's processing resources (*e.g.*, slow receiver). Without congestion control, the network can enter a condition where the majority of sent data is eventually dropped, known as *congestion collapse*; such a collapse occurred on the Internet in 1986, causing throughput to drop by a factor of a thousand [20].

TCP congestion control relies on acknowledgement packets (see Appendix A for details) from the receiver to explicitly provide the sender with *correct* information about the number of data bytes received (and implicitly about the *real* network conditions). However, TCP does not have any cryptographic mechanisms to ensure authentication and integrity of the sent packets, including acknowledgments. Application-layer secure protocols such as TLS provide no protection for TCP headers or TCP control messages, and network-layer secure protocols such as IPsec [23] require separate infrastructure and protect only up to the tunnel termination point. Thus, an attacker that can intercept acknowledgment packets can modify them without being detected by the intended recipient, who will blindly trust the information. TCP has a protection mechanism against packet injection in the form of a sequence number included on each packet. However, numerous attacks demonstrate that this protection mechanism can be bypassed by blind attackers performing TCP sequence guessing [33], [32], [10], [18] or by *off-path* or *on-path* attackers that can observe the target stream. Thus, an attacker can also inject well-crafted acknowledgment packets into a TCP stream without detection. By creating such crafted acknowledgments that propagate malicious information about the data received, an attacker can manipulate TCP congestion control into sending data at rates that benefit the attacker. For example, by creating an acknowledgement that acknowledges data packets prior to receiving them and injecting it into a target stream, an adversarial TCP receiver can persuade the sender to increase its sending rate beyond the rate prescribed by correct congestion control, possibly forcing the network into congestion collapse [37].

Several manipulation attacks against TCP congestion control have been discovered; some of these attacks use external data flows to create the impression of congestion [26], [25] and others use acknowledgement packets to directly mislead the congestion control mechanisms [22], [37], [12], [2]. These attacks are more subtle and difficult to detect than traditional crash or control-hijacking attacks. Acknowledgement-based attacks, in particular, do not raise suspicions as long as the packets are consistent with the receiver's state (unlike data that might not assemble properly at the application level). We focus on attacks against congestion control created through maliciously crafted acknowledgement packets (by fabrication of new ones or modification of existing ones) and refer to them as *manipulation* attacks.

Manipulation attacks on congestion control can have severe implications such as *financial loss*. Consider an attacker who wishes to degrade video quality and streaming experience for a subset of Netflix users. While Netflix recently began to encrypt all of its video traffic with TLS [40], TLS relies on TCP to

[†] This work was done while Endadul Hoque was at Northeastern University

Network and Distributed Systems Security (NDSS) Symposium 2018
18-21 February 2018, San Diego, CA, USA
ISBN 1-1891562-49-5
http://dx.doi.org/10.14722/ndss.2018.23115
www.ndss-symposium.org

transfer data across the network. As a result, an attacker can simply launch an attack misleading TCP into believing that the network is congested. This will cause TCP to repeatedly slow down its sending rate, causing rebuffering events and reduced video quality for any Netflix user subjected to this attack. Due to poor streaming experience, the users may consider turning to other video providers.

Previous work on attacks against TCP congestion control relied mainly on manual analysis. The only work we are aware of that used automation for finding attacks in TCP congestion control implementations is the work in [24] which relies on the user to provide a vulnerable line of code and then performs static analysis. The vulnerable line of code from the user is critical to ensure scalability of the approach. In addition, the method is restricted to a specific implementation, language, and operating system.

In this paper, we aim to automatically discover manipulation attacks on congestion control without requiring the user to provide any vulnerable line of code and without being dependent on specific implementation, language, or operating system characteristics. *Protocol fuzzing* [27], [1], [16] is a well-known approach where packet contents are either randomly generated and injected into the network or randomly mutated in-transit. However, without explicit guidance, given a vast input space, fuzzing fails to concentrate on relevant portions of the source code (*i.e.*, for inducing protocol-compliant behaviors).

Previous work on testing TCP connection establishment [21] used the protocol's connection state machine to guide the fuzzing process and prune unnecessary executions. However, unlike attacks against connection establishment which usually consist of one action, attacks against congestion control require a potentially long sequence of actions spanning several states and transitions, where each action might trigger a new state, which in turn might require a different attack action. Automatically discovering these combinations at runtime is not practical for scalability reasons. For example, using the approach in [21] for congestion control would require a search space of about 1.2×10^{24} cases, assuming only 5 types with 4 parameter choices for creating the malicious acknowledgements and 4 possible states for injecting them. Even limiting this to test at most one manipulation at a time in each state would generate 194,480 cases, which is still impractical for testing in a real network.

To address this scalability challenge while still guaranteeing that we test relevant portions of the code, we use *model-based testing* (MBT) [43], an approach that generates effective test cases based on a model of the program. The approach uses a *model*, an abstract representation of the desired behavior of the program that is typically derived from specifications, to derive *functional tests*. These functional tests contain the same level of abstraction as the model, and are converted to concrete test cases to be tested against the implementation. MBT does not require the source code and guides the testing to concentrate only on relevant portions of the source code.

Our approach. We propose to automatically find manipulation attacks by guiding a protocol fuzzer with *concrete* attack actions derived from *abstract attack strategies*, which are obtained using a model-guided technique inspired by model-based testing. Our model is a finite state machine (FSM) that captures the main functionality of several types of congestion control algorithms used by deployed TCP implementations and is constructed from RFC specifications. We use this abstract model to generate *abstract attack strategies* by exploring the different paths in the FSM that modify state variables controlling throughput, and thus can be leveraged to mount an attack. We then map these abstract strategies to *concrete attack strategies* that correspond to real attacker capabilities; a concrete strategy consists of acknowledgment-packet-level actions with precise information about how the packets should be crafted and the congestion control states in which these actions should be performed. Our approach provides maximum coverage of the model of congestion control while generating an optimum number of abstract strategies. The number of concrete attack strategies is bounded by the number of malicious actions that describe an attacker's capabilities. We consider off-path attackers and on-path attackers; both can sniff traffic and obtain TCP sequence numbers and data that has been acknowledged or sent. However, there is one fundamental difference, an off-path attacker can only inject malicious acknowledgements, but cannot prevent the correct ones from reaching the receiver; an on-path attacker can modify acknowledgements such that the victim sees only acknowledgments from the attacker.

We created and implemented a platform, TCPWN, to create and inject concrete attack scenarios. The platform combines virtualization (to run different implementations in their native environment), proxy-based attack injection, and runtime congestion control state machine tracking (to inject the attacks at the right time during execution). Our state machine tracking at runtime does not require instrumenting the code. Specifically, we use a general congestion control state machine (*e.g.*, TCP New Reno) and infer the current state of the sender by monitoring network packets exchanged during fuzzing. While this option is less accurate than extracting the state machine from an implementation's code, it is less complex and more general. TCPWN is publicly available at https://github.com/samueljero/TCPwn.

Our model-based attack generation finds 21 abstract strategies that are mapped into 564 (for on-path attackers) and 753 (for off-path attackers) concrete strategies. Each strategy can be tested independently and takes between 15 and 60 seconds. We evaluated 5 TCP implementations from 4 Linux distributions and Windows 8.1, all using congestion control mechanisms that can be modeled as the finite state machine we used to generate abstract strategies. Overall, we found 11 classes of attacks, of which 8 were previously unknown.

The rest of the paper is organized as follows. First, we describe the TCP congestion control state machine model we assume in this work, in Section II. We then describe our attacker model in Section III. We provide details on the design of our system in Section IV and describe our implementation in Section V. We summarize our results in Section VI and present related work in Section VII. Finally, we conclude the paper in Section VIII.

II. TCP CONGESTION CONTROL MODEL

We present the finite state machine (FSM) for TCP congestion control considered in this work. This FSM is based on the classic TCP New Reno [19], [4]. We then discuss op-

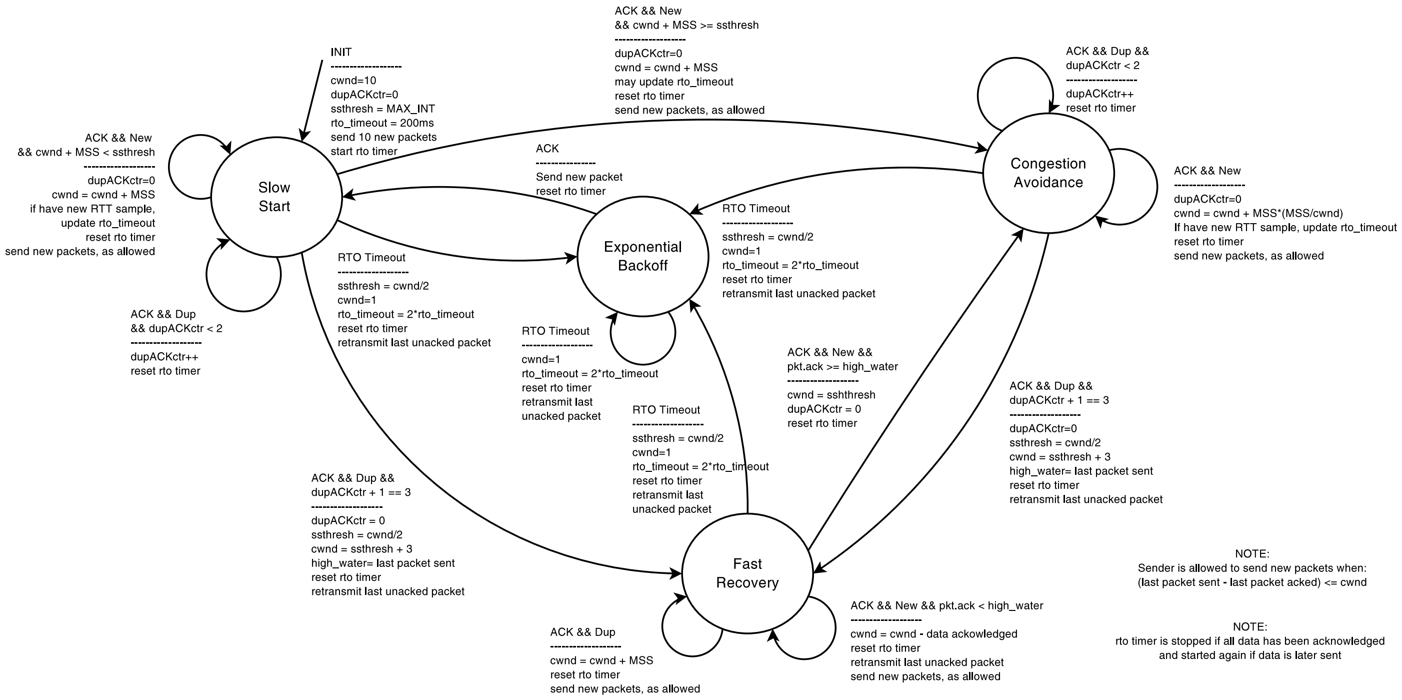


Fig. 1. TCP New Reno State Machine

tional improvements and variants like SACK [8], DSACK [7], TLP [17], PRR [28], FRTO [35], and others [34], [13].

A. Finite State Machine Model

At a high level, the congestion control of TCP New Reno consists of four phases: (1) slow start, (2) congestion avoidance, (3) fast recovery, and (4) exponential backoff. During the slow start phase the sender is probing the network to quickly find the available bandwidth without overloading the network; once such bandwidth is found, the sender enters a congestion avoidance phase in which the sender can send without causing congestion; in case of congestion and data loss, fast recovery or exponential backoff reduce the rate at which data is sent. The fast recovery phase is intended for less significant events where the beginning of congestion is detected through lost packets and acknowledgments, while the exponential backoff phase deals with more significant events where congestion is detected by the expiration of a large timeout. We present the finite state machine model assumed for congestion control in Figure 1. Below we describe the associated events, variables, and states.

(1) Events. TCP congestion control relies on two events for its operation, the reception of an acknowledgement (ACK) and the occurrence of a timeout (RTO Timeout):

ACK. This event denotes that an acknowledgement packet was received by the sender. We describe in detail in Appendix A these TCP acknowledgments. TCP acknowledgements are byte-based and *cumulative*, i.e. the receiver acknowledges the highest byte of data at which all prior data has been received. A duplicate acknowledgment, and particularly three duplicate acknowledgments, are used to signal timely information about the network conditions.

RTO Timeout. This event denotes that a timeout occurred when data was outstanding and no acknowledgements were received for several Round-Trip-Times (RTTs). This indicates more severe conditions in the network since the last acknowledgement. This timer is started when new data packets are sent, reset on every acknowledgement, and stopped if all data has been acknowledged.

(2) Variables. The variables capturing the main functionality of congestion control can be grouped into three categorizes: variables related to the amount of data to be sent (`cwnd` and `ssthresh`), variables keeping track of acknowledged data (`dupACKctr` and `high_water`), and variables controlling timeouts (`rto_timeout`).

Congestion window – `cwnd`. This variable represents the number of bytes of data that TCP is allowed to have in the network at any given time. It is modified by TCP congestion control to increase or decrease the sending rate in response to network conditions.

Slow start threshold – `ssthresh`. This variable indicates the value of the congestion window `cwnd` at which TCP switches from slow start to congestion avoidance. TCP uses this information later in the connection by growing the window exponentially up to `ssthresh` after a timeout or idle period.

Duplicate ACK – `dupACKctr`. This variable tracks the number of duplicate acknowledgements received in slow start and congestion avoidance. Receiving three duplicate acknowledgements triggers a transition to fast recovery.

Highest sequence sent – `high_water`. This variable records the highest sequence number sent prior to entering fast recovery. Only once this sequence number has been acknowledged (or a timeout occurred) will fast recovery be exited.

RTO Timeout – `rto_timeout`. This variable indicates the current length of the RTO Timeout. It is usually set to $max(200ms, 2 * RTT + 4 * RTT\_Variance)$. If the RTO timer expires, this value is doubled, resulting in an exponential backoff.

(3) States. We can now describe the state machine from Figure 1. The states capture the four high-level phases described before.

Slow Start. In this state TCP rapidly increases its sending rate, as indicated by the congestion window `cwnd`, in order to quickly utilize the available bandwidth of the path while not overloading the network with a huge initial burst of packets. For each acknowledgement acknowledging new data, `cwnd` is incremented by MSS (Maximum Segment Size), which results in a doubling of the sending rate every RTT. TCP exits slow start on the RTO Timeout, after three duplicate acknowledgements—which indicate a lost packet—, or when the congestion window `cwnd` becomes bigger than the slow start threshold `ssthresh`. This last condition indicates that TCP is approaching a prior estimate of the fair-share connection bandwidth. TCP connections start in the slow start state with `ssthresh` set to MAX_INT, such that slow start is only exited on timeout or packet loss, and `cwnd` set to 10, allowing a burst of ten packets to be sent initially.[1]

Congestion Avoidance. In this state TCP is sending close to its estimate of the available bandwidth while also slowly probing for additional bandwidth. Every RTT `cwnd` is increased by one MSS sized packet. In practice, this is done by increasing `cwnd` by a small amount($(MSS * \texttt{cwnd})/MSS$) for every new ACK received. TCP exits congestion avoidance either on an RTO Timeout or after receiving three duplicate acknowledgments, indicating a lost packet.

Fast Recovery. In this state, TCP is recovering from a lost packet indicated by three duplicate acknowledgements. TCP assumes that packet loss signals network congestion, so it cuts its sending rate in half by halving `cwnd`, and retransmits the last unacknowledged packet. `ssthresh` is set to this new value of `cwnd`, providing an approximate bandwidth estimate in case of a timeout. TCP remains in fast recovery until all data outstanding at the time it entered fast recovery has been acknowledged or an RTO timeout occurs. This is achieved by saving the last packet sent in `high_water` upon entry and exiting once this packet has been acknowledged.

Exponential Backoff. In this state, TCP is retransmitting a lost packet each time the RTO timer expires. With each timer expiration, `rto_timeout` is doubled, resulting in an exponential backoff between retransmissions. This state is entered from any other state when the RTO timer expires, indicating that data is outstanding in the network but no acknowledgements have been received in `rto_timeout` seconds (at least 2 RTTs). This situation indicates the loss of a large number of packets and, likely, significant changes in network conditions. As a result, `ssthresh` is set to half of `cwnd`, `cwnd` is set to 1 MSS, and the last unacknowledged packet is retransmitted. TCP remains in this state, retransmitting this packet each time the RTO timer expires, until an acknowledgement is received, at which point it transitions to slow start.

B. Variations and Optimizations

The classic TCP New Reno congestion control algorithm we described above has seen a number of variations and optimizations over the years. These include SACK [8], DSACK [7], TLP [17], PRR [28], CUBIC [34], and RACK [13]. These variations and optimizations consist of fairly minor changes to the basic New Reno algorithm. SACK [8], for example, provides the sender with additional information about received packets and uses this information to determine when to enter fast recovery. The logic of the decision does not change: fast recovery is entered when three packets above a loss have been received. SACK simply uses a more accurate method to detect this condition. Similarly, PRR [28] modifies New Reno by adopting paced packet sending during the self-loop in fast recovery. TLP [17] introduces a new, faster timeout state before exponential backoff. CUBIC TCP [34] changes precisely how `cwnd` is increased in congestion avoidance and decreased during fast recovery.

While these changes affect the performance of TCP in certain network conditions, they follow the same phases of TCP congestion control. In this work, our attack generation models all TCP congestion control as classic TCP New Reno due to difficulty inferring more detailed congestion control state from network traffic alone (see Section IV-D). Nevertheless, we successfully tested modern Windows and Linux TCP stacks with many of these variations and optimizations, and we successfully identified attacks against them.

III. Attack Model

In this section we discuss the attacker capabilities and congestion control attacks that we consider in this work.

A. Attacker and Attack Goals

A typical attacker might be a botnet trying to enhance the power of a DDoS attack by using increased throughput attacks to render TCP flows insensitive to congestion. This gives the attacker the power of a UDP flood with the ubiquity of TCP traffic; perfect for the coremelt attack [41]. Alternately, a nation-state actor could launch decreasing throughput attacks to discourage or prevent use of certain undesirable services.

Decreasing Throughput. In this case, the attacker manipulates the congestion control algorithm of a target connection such that it falsely detects congestion, resulting in a rate reduction. This rate reduction can have significant impact at the application level, especially for inelastic data streams like streaming video.

Example. Consider the *Blind Throughput Reduction Attack* [12]. In this attack, the attacker sends spoofed invalid acknowledgements to the target connection's receiver, which cause the receiver to send duplicate acknowledgements to the sender. These duplicate acknowledgements, when received in the congestion avoidance or slow start states, mislead the sender about the existence of lost packets and the level of congestion in the network, causing the sender to transition to the fast recovery state and slow down (see Fig. 3). The sender will continue to slow down as long as the attacker emits its spoofed acknowledgements.

[1] This initial window was originally 2-4 packets [4], but has been increased to 10 packets in more recent standards [15] and implementations.

Increasing Throughput. In this case, the attacker manipulates the congestion control algorithm such that it perceives significant available bandwidth along with low latency and loss. As a result, the sender rapidly increases its sending rate beyond what is fair to competing connections. Any actual congestion in the network will not be observed, which may be used to damage or deny service to target links or to other connections sharing the same links.

Example. Consider the *Optimstic Ack Attack* [37]. In this attack, the receiver repeatedly sends acknowledgements for data that has not actually been received yet in order to dramatically increase its sending rate and render the sender insensitive to actual congestion in the network. Acknowledging data not yet received in the congestion avoidance, slow start, or fast recovery states misleads the sender about the data that has been received and the RTT of the connection. As a result, the sender does not react to actual congestion in the network and is unfair to any competing connections.

Target Flows. Any TCP flow that sends more than an initial window (10 packets, about 15KB) of data is vulnerable to these attacks. In this work, we focus on bulk data transfers because they result in the widest array of attacks, are easiest to automate, and easiest to explain; however, these attacks are not restricted to such flows. Short transfers, like web pages, are also vulnerable to attacks on congestion control, and flows with a limited bitrate, like streaming video, are vulnerable to decreasing throughput attacks. Interactive flows are vulnerable if their sending rate is limited by congestion control and not by the availability of data from the application.

B. Attack, Strategy, Action

Congestion control constrains the sender's data-transfer rate, primarily through acknowledgements. Thus, we consider attacks conducted through acknowledgement packets.

Congestion control manipulation attacks. These are attacks conducted by manipulation of TCP acknowledgements in order to mislead congestion control about current network conditions and cause it to set an incorrect sending rate. They can result in either increasing or decreasing the throughput, and sometimes in connection stall. In order to achieve the high-level goals of manipulating congestion control, an attacker applies an *attack strategy*.

Attack strategy. Given a TCP stream, where a sender sends data to a receiver, we define a concrete attack strategy as a sequence of acknowledgment-based malicious actions and the corresponding sender states (as described in Fig. 1) when each action is performed.

Malicious actions. A malicious action itself requires an attacker to (1) craft acknowledgements by leveraging protocol semantics to mislead congestion control, (2) infer the state at the sender, and (3) inject the malicious acknowledgment on the path and in the target stream. For example, a malicious action can be to craft an acknowledgment that acknowledges data not yet received and inject it when the sender is assumed to be in congestion avoidance.

Crafting malicious acknowledgements. TCP does not use any cryptographic mechanisms to ensure authentication and integrity of packets; thus, an attacker can fabricate packets or modify intercepted ones with malicious payload. In order to intercept, the attacker will need to be on the path and be able to sniff the target stream. Moreover, these crafted acknowledgement are *semantic-aware*, that is, the attacker is aware of the meaning of the bytes acknowledged. For example, in the example above, an attacker will need to know the highest byte of data that was acknowledged in order to acknowledge data that has not been received yet.

Inferring the state machine at the sender. We assume that the attacker can observe the network traffic, but it does not have access to the source code and thus cannot instrument it.

Injecting malicious acknowledgments. This requires an attacker to spoof packets and have knowledge of the TCP sequence number, the only protection TCP has against injection. We do not consider blind attackers in this work, since, while they can inject spoofed packets into the network, they have no knowledge of sequence numbers or data being acknowledged and thus are restricted to guessing this information. We distinguish between off-path and on-path attackers. An off-path attacker can observe packets in the target connection or link and inject spoofed packets. For example he can sniff traffic on the client's local network — e.g., coffee house Wi-Fi. An on-path attacker can intercept, modify, and control delivery of legitimate packets in some target connection or link, as well as inject new spoofed packets. For example, such an attacker can be a switch on the path between client and server.

IV. TCPwn Design

In this section we describe the design of TCPwn, our automated platform for finding attacks on congestion control. We first provide a high-level overview, then discuss our model-guided attack strategy generation and congestion control protocol state tracking.

A. Overview

We motivate our approach with the *Optimistic Ack* [37] attack. Consider its interactions with the congestion control state machine as shown in Fig. 3. In order to be successful, the attacker must inject packets with an acknowledgement number above the real cumulative acknowledgment number and below the highest sequence number that the sender has sent, and it has to do this in either the congestion avoidance, slow start, or fast recovery states. Each time the sender receives one of these new acknowledgements in those states, it causes a self-loop transition (in blue in Fig. 3), increasing the congestion window `cwnd`, which directly controls the sending rate.

Finding all these transitions (i.e. that impact the sending rate at runtime) is challenging because of the large search space. We address this challenge by using a model-based attack strategy generation that finds *all possible attack strategies* in a model of the congestion control (i.e. shown in Figure 1). We refer to these as *abstract strategies*. To test them in real implementations, we translate them to *concrete attack strategies*, obtained by mapping the abstract strategies to attack actions corresponding to attacker capabilities and consisting of specific content for a malicious packet and the state in which it will be injected. An attack injector takes these concrete packet-based attack strategies and injects them in our testing

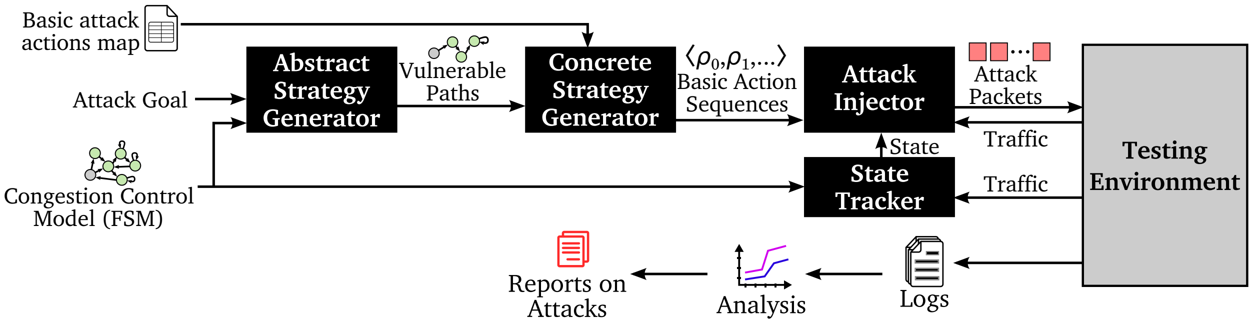


Fig. 2. Design of TCPWN

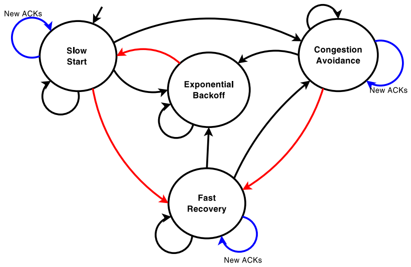


Fig. 3. Interactions between New Reno congestion control and the Optimistic Ack attack. Transitions in blue increase throughput while those in red decrease throughput.

environment during an actual execution of the target implementation. Our attack injector requires information about the current congestion control state of the sender. A state tracker determines this current protocol state so that actions can be performed as specified by the strategy. After the execution of each attack, our system collects logs that capture performance metric(s). By comparing the resulting performance with the expected baseline performance, TCPWN identifies whether the strategy indeed leads to a successful attack. Fig. 2 shows the conceptual design of our system, TCPWN.

Testing strategies with real implementations provides strong soundness properties since any strategy that TCPwn identifies as an attack caused noticeable performance changes in a real TCP connection of the implementation under test. This prevents most classes of false positives, except tests with performance outside of the considered normal range ($>$ 2 standard deviations from average). Our completeness is limited by the accuracy of the congestion control model and state tracking. Here, we choose to trade off some completeness for the ability to test many implementations and use a generalized congestion control model and inferred state tracking.

Example for TCPWN attack generation. We demonstrate the attack strategy generation approach using the same example as above, where the attacker's goal is to increase the sending rate; this can also be expressed as an increase in the sender's cwnd variable. Our abstract strategy generator identifies each of those paths in the FSM (Fig. 1) containing at least one transition that increments the cwnd variable. One of the identified paths (say, $\mathcal{P}$) looks as follows:

$\mathcal{P}$: SlowStart $\rightarrow$ FastRecovery $\rightarrow$ CongestionAvoidance $\circlearrowleft$

where the self-loop in CongestionAvoidance increments cwnd (see Fig. 1). An *abstract strategy* $\mathbb{S}$ is a projection on the condition of each transition along $\mathcal{P}$and is represented as the following sequence of (*state*, *condition*) pair:

(**In**: SlowStart, **Condition**: `ACK && Dup && dupACKctr`$\geq$3)
(**In**: FastRecovery, **Condition**: `ACK && New && pkt.ack` $\geq$ `high_water`)
(**In**: CongestionAvoidance, **Condition**: `ACK && New`)$^{+}$

This strategy $\mathbb{S}$ dictates that when the sender is in SlowStart and is sending data to the receiver, the attacker can send 3 duplicate ACKs to the sender so that it moves to FastRecovery. Next the attacker can send the sender 1 new ACK (that acknowledges all the outstanding data). As a result, the sender moves to CongestionAvoidance, and the attacker can keep on sending new ACKs that optimistically acknowledge all outstanding data even if the receiver has not received it yet. $+$ (the superscript) signifies that the attacker can apply this segment of $\mathbb{S}$ repeatedly.

TCPWN maps $\mathbb{S}$ to several concrete strategies that can be directly tested inside the testing environment running the given implementation. TCPWN relies on a map which associates the abstract network conditions to concrete *basic actions*. For $\mathbb{S}$, TCPWN generates 72 Concrete strategies, based on actions mimicking both off-path and on-path attackers. One such concrete strategy is:

(**In**: SlowStart, **Action**: 3 $\times$ Inject `Dup-Ack`)
(**In**: FastRecovery, **Action**: Inject `Pre-Ack`)
(**In**: CongestionAvoidance, **Action**: Inject `Pre-Ack`)$^{+}$

This concrete strategy dictates that when the sender is in SlowStart, the attacker can use the `Dup-Ack` basic action to inject 3 duplicate ACKs. Similarly, for acknowledging all the outstanding data in the next step, the attacker can use the `Pre-Ack` basic action. Once the sender is in CongestionAvoidance, the attacker can repeatedly apply `Pre-Ack`. We will describe all supported basic actions in §IV-C.

B. Abstract Strategy Generation

We now describe in detail the core of our approach. We observe that a successful attack will (1) trigger a transition that causes an increase or decrease in the congestion window `cwnd` and (2) traverses a cycle in the congestion control state machine.

Changes to `cwnd`. The congestion window, `cwnd`, adjusts the sending rate of TCP to avoid congestion collapse and provide fairness. [2] Further, congestion control modifies `cwnd` frequently during the course of its normal operation. These modifications are done on many transitions of the congestion control state machine and either increase or decrease `cwnd` depending on the transition. As a result, an attacker can increase or decrease `cwnd`, and therefore TCP's sending rate, merely by inducing TCP to follow specific normal transitions in the congestion control state machine.

State Machine Cycles. Successful congestion control attacks traverse a cycle in the congestion control state machine. This is due to the highly dynamic and cyclical nature of congestion control where a sender often traverses the same set of states many times over the course of a connection and multiple state transitions in a single second are common. As a result, the impact on `cwnd` from a single transition is quickly diminished by other transitions. For an attack to be effective and achieve measurable, lasting impact, an attacker has to frequently induce TCP to follow some desirable transition. Such a series of desirable transitions will form either a cycle or a unique path in the state machine. Given the relatively small size (under 10 states) of the congestion control state machine and the frequency of state transitions, anything but the shortest connections would require a cycle to achieve a sufficiently long series of desirable transitions.

Note that these characteristics are necessary but not sufficient for an attack on congestion control. For instance a cycle may contain two manipulations to `cwnd` that balance each other out, or a cycle may not be triggerable by the attacker.

Our abstract strategy generator takes as input an FSM model of congestion control and a description of the desirable transitions. In our case, a desirable transition is one that modifies `cwnd`. It outputs a list of all paths with cycles that contain a desirable transition and can therefore be used by an attacker to achieve his goal. This list includes the transitions in each path as well as the conditions that cause each transition. We use a modified depth-first traversal to enumerate all paths in the FSM. We formally define the abstract strategy generation problem and our algorithm below.

State Machine Model. We define a model $\mathcal{M}$ describing the state machine of the congestion control algorithm as a tuple $(\mathcal{S}, \mathcal{N}, \mathcal{V}, \mathcal{C}, \mathcal{A}, \sigma, \mathcal{T})$. $\mathcal{S}$ is a finite set of states $\{s_0, \ldots, s_n\}$, and the initial state is $\sigma \in \mathcal{S}$. $\mathcal{N}$ represents a finite set of network events (*e.g.*, `ACK` signifies the reception of a TCP acknowledgment). $\mathcal{V}$ is a finite set of variables including both some fields of a received packet and some program variables. For instance, `New` means the received `ACK` acknowledges some new data and `cwnd` indicates the program variable that represents congestion window size. $\mathcal{C}$ represents a finite set of conditional statements such that each element $c \in \mathcal{C}$ is a quantifier-free first order logic (QF-FOL) formula [30] over $\mathcal{V}$ (*e.g.*, `dupAckCtr` < 2). $\mathcal{A}$ represents a finite set of assignment statements (*i.e.*, protocol actions) over a subset of $\mathcal{V}$ (*e.g.*, "`cwnd = 1`" means the congestion window is set to 1). In addition, $\mathcal{N}$, $\mathcal{V}$, $\mathcal{C}$, and $\mathcal{A}$ are pairwise disjoint. $\mathcal{T}$ represents the transition relations such that $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{N} \times \mathcal{C} \times 2^{\mathcal{A}} \times \mathcal{S}$.

Let $\psi : \mathcal{T} \mapsto \mathcal{S}$ and $\xi : \mathcal{T} \mapsto \mathcal{S}$ be two maps indicating the *source* and *target* of a transition. For example, if a transition $t \in \mathcal{T}$ begins at s_b and ends at s_e, then $\psi(t) = s_b$ and $\xi(t) = s_e$. Let $\lambda : \mathcal{T} \mapsto \mathcal{N} \times \mathcal{C}$ and $\aleph : \mathcal{T} \mapsto \mathbb{P}(\mathcal{A})$ be two maps to indicate the triggering conditions and the set of actions of a transition, respectively. Now we define a path as follows.

Definition 1. *Path: A* **path** *P in $\mathcal{M}$ is a sequence of pairs of states and transitions $\langle (s_{i_0}, t_{j_0}), (s_{i_1}, t_{j_1}), \ldots, (s_{i_k}, t_{j_k}) \rangle$, where $k \geq 0$; each $s_{i_x} \in \mathcal{S}$ for $0 \leq x \leq k$ and $s_{i_0} = \sigma$ (the initial state); $\forall y\, [t_{j_y} \in \mathcal{T} \wedge \psi(t_{j_y}) = s_{i_y} \wedge \xi(t_{j_y}) = s_{i_{(y+1)}}]$ where $0 \leq y \leq k-1$; $t_{j_k} \in \{\mathcal{T}, \perp\}$ and $[\psi(t_{j_k}) = s_{i_k} \wedge \xi(t_{j_k}) \in \{\mathcal{S}, \perp\}]$. In addition, $\forall r, s[r \neq s \rightarrow s_{i_r} \neq s_{i_s} \wedge t_{j_r} \neq t_{j_s}]$, where $r, s \in \{0, 1, \ldots, k\}$.*

In other words, a path P starts at σ and moves to the state s_{i_1} by taking the transition t_{j_0}. By following the sequence, P finally reaches at s_{i_k}. The last segment of P (*i.e.*, (s_{i_k}, t_{j_k})) is special as it determines the existence of a cycle. If P contains a **cycle**, then $[t_{j_k} \neq \perp \wedge t_{j_k} \in \mathcal{T}]$ and $\exists z[\xi(t_{i_k}) = s_{i_z}]$, where $z \in \{0, 1, \ldots, k\}$. When P has no cycle, $t_{j_k} = \perp$ and $\xi(t_{j_k}) = \perp$.

Definition 2. *Vulnerable path: Given a vulnerable action $\alpha \in \mathcal{A}$, a path P in $\mathcal{M}$ is a* **vulnerable path** *if P has a segment (s_{i_x}, t_{j_x}) such that $\alpha \in \aleph(t_{j_x})$, where $x \in \{0, \ldots, k\}$ and $k \geq 0$.*

Definition 3. *Abstract strategy: Given a vulnerable path P in $\mathcal{M}$ such that $P = \langle (s_{i_0}, t_{j_0}), \ldots, (s_{i_k}, t_{j_k}) \rangle$ for some $k \geq 0$, the corresponding* ***abstract strategy*** *$\mathbb{S}$ is defined as $\langle (s_{i_0}, \lambda(t_{j_0})), (s_{i_1}, \lambda(t_{j_1})), \ldots, (s_{i_k}, \lambda(t_{j_k})) \rangle$, where $\lambda(t_{j_x}) \in (\mathcal{N} \times \mathcal{C})$ if $t_{j_x} \in \mathcal{T}$ or $\lambda(t_{j_x}) = \perp$ if $t_{j_x} = \perp$ for each $0 \leq x \leq k$.*

Abstract Strategy Generator. Given $\mathcal{M}$, a directed *multigraph* with cycles, and the attacker's goal $\alpha \in \mathcal{A}$, the Abstract Strategy Generator aims to find all the vulnerable paths in $\mathcal{M}$ with respect to α. We devise the algorithm shown in Algorithm 1, which begins the search from the function `VulnerablePathFinder`. Intuitively, the algorithm traverses the entire graph in a depth-first fashion, starting at the initial state $\sigma \in \mathcal{S}$. For each transition $t \in \mathcal{T}$ such that $\psi(t) = \sigma$, the algorithm initializes a new path P, appends (σ, t) to P, and recursively continues its exploration of the subgraph rooted at $\xi(t)$. For P, the recursion stops when it encounters a cycle (line 13) or a terminating state (line 15). If any of these stop conditions is met, the algorithm checks if P is a vulnerable path with respect to α; if so, it adds P to the set of the vulnerable paths (line 20). Unlike traditional depth-first traversal, the algorithm restores the subgraph rooted at $\xi(t)$ by marking it unvisited (line 28) in order to find all possible vulnerable paths w.r.t. α. Upon termination, the algorithm returns the set of vulnerable paths w.r.t. α (line 10) identified during the exploration. This set of vulnerable paths contain our abstract strategies. We generate our abstract strategies $\{\mathbb{S}\}$ by taking projections on conditions of the transitions along each path.

[2] This is true for all congestion control algorithms except Google's new BBR [11] congestion control. This includes Reno [4], New Reno [19], CUBIC [34], Compound TCP [39], and Vegas [9], among others. However, BBR does maintain a variable containing the explicitly computed allowed sending rate, which has similar properties for our purposes. As BBR's public release was concurrent with this work, we do not consider it further here.

Algorithm 1: Abstract Strategy Generator

```
Input: Multigraph $\mathcal{M} = (\mathcal{S}, \mathcal{N}, \mathcal{V}, \mathcal{C}, \mathcal{A}, \sigma, \mathcal{T})$, $\psi$, $\xi$, $\lambda$, $\aleph$
       and a vulnerable action $\alpha \in \mathcal{A}$
Output: All vulnerable paths with respect to $\alpha$

$VulnerablePaths := \emptyset$
                /* to store all the vulnerable paths */
Function VulnerablePathFinder($\mathcal{M}$, $\alpha$)
    $root := \sigma$             /* initial state */
    Mark $root$ as visited
    foreach transition $t$ such that $\psi(t) = root$ do
        Create a new path $P$
        $P := P \| (root, t)$        /* concatenating */
        $v := \xi(t)$
        RecursiveSearch($v, P, \alpha$)
    return $VulnerablePaths$

Function RecursiveSearch($v, P, \alpha$)
                             /* search continue from $v$ */
    $base\_case :=$ false
    if $v$ is already visited then        // reached a cycle
        $base\_case :=$ true
    else if exists no $t$ such that $\psi(t) = v$ then
                            /* $v$ is a terminating state */
        $base\_case :=$ true
        $P := P \| (v, \perp)$               /* concatenating */
    if $base\_case$ is true then
        if $P$ is a vulnerable path w.r.t. $\alpha$ then
            $VulnerablePaths := VulnerablePaths \cup P$
    else
        Mark $v$ as visited
        foreach transition $t$ such that $\psi(t) = v$ do
            $v' := \xi(t)$
            $P' := P$          /* creating a copy */
            $P := P \| (v', t)$      /* concatenating */
            RecursiveSearch($v', P', \alpha$)
        Mark $v$ as unvisited
    return                                  /* void */
```

C. Concrete Strategy Generation

An abstract strategy just specifies a path in the FSM that can lead to an attack. However, there are usually several ways in which this path can be concretely achieved at runtime. Concrete strategy generation takes our abstract strategies and converts them into sets of basic message-based actions that can be applied by our attack injector in particular states of the FSM.

Our concrete strategy generator considers each abstract strategy individually and iterates through each transition in that strategy. Each of these network conditions is *mapped* to a basic action that the attacker can directly utilize to trigger that network condition in that state. This results in a set of $(state, action)$ pairs which we call a *concrete strategy*. A transition condition may be triggered by multiple basic actions, in which case this mapping results in a set of basic actions that could be applied in that state to cause the next transition. Our generator creates one concrete strategy for each combination of actions from these sets. Note that we require a domain expert to provide the mapping of network conditions to basic actions since it relies on domain knowledge. This mapping only needs to be updated when the state machine model changes or new actions are added; generating concrete actions for a given implementation is completely automated.

We developed our set of basic actions based on an extensive study of TCP and known congestion control attacks. We consider two categories: injection of acknowledgements, which captures the capabilities of an *off-path attacker*, and modification of acknowledgements, which captures the capabilities of an *on-path attacker*.

Injection of acknowledgements (off-path attacker). This type of action injects new spoofed acknowledgement packets for either the client or server of a target connection. We support a number of different ways of injecting acknowledgements:

(1) Duplicate Acknowledgements (param: dup_no, delay, offset) — Injecting many acknowledgements with the same acknowledgement number as an apparent set of duplicate acknowledgements. This enables an off-path attacker to slow down a connection. This action assumes that target connection's sequence and acknowledgement numbers are known or can be guessed. Parameters control the number of duplicates injected (2, 10, 1000), the spacing between these duplicates (1ms), and offset from the current acknowledgement number (0, 3000, 90000).

(2) Offset Acknowledgements (param: num, delay, data, offset) – Injecting a series of acknowledgements with an acknowledgement number offset from the legitimate acknowledgement number. Acknowledges either less or more data than is acknowledged by the receiver. This action assumes that target connection's sequence and acknowledgement numbers are known or can be guessed. Parameters control the number of acknowledgements injected (10000, 50000), the spacing between these acknowledgements (1ms, 2ms), any bytes of data included (0, 10), and any offset from the current acknowledgement number (0, 100, 3000, 9000, 90000).

(3) Incrementing Acknowledgements (param: num, delay, data) — Injecting a series of acknowledgements where the acknowledgement number increases by a variable amount each time. Congestion control expects these acknowledgements to indicate the successful receipt of new data and will act accordingly. This action assumes that target connection's sequence and acknowledgement numbers are known or can be guessed. Parameters control the number of acknowledgements injected (50000), the spacing between these acknowledgements (1ms), and the amount the acknowledgement is incremented with each packet (9000, 90000).

Modification of acknowledgements (on-path attacker). This type of action changes the manner in which acknowledgements for the sequence space are sent. We support a number of manipulations to the sequence of acknowledgements for a data stream:

(1) Division (param: data) — Acknowledge the sequence space in chunks much smaller than a single packet. This splits a single acknowledgement packet into many acknowledgement packets that acknowledge separate ranges. A parameter controls the number of bytes to acknowledge in a single chunk (100).

(2) Duplication (param: dup_no) — Duplicate acknowledgements of chunks of the sequence space repeatedly. A parameter controls the number of duplicate acknowledgements to create (1 ,4, 100). This breaks the assumption that each acknowledgement received corresponds to a packet that left the network.

(3) Pre-acknowledging (param: none) — Acknowledging portions of the sequence space that have not been received yet. This hides any losses, preventing slow downs, and effectively shrinks the connection's RTT, allowing faster than normal throughput increases.

(4) Limiting (param: none) — Prevents the acknowledgement number from increasing. This generates duplicate acknowledgements but also prevents any new data from being acknowledged. This is likely to stall the connection and lead to an RTO.

D. State Tracker

In order to test a strategy against an implementation, TCPwn needs to know that the state of the sender with respect to congestion control. This is not an easy problem as there are several implemented congestion control algorithms such as Reno [4], New Reno [19], CUBIC [34], Compound TCP [39], and Vegas [9]. Implementations may also choose to include an Application Limited state, adjustable `dupACKctr` thresholds, and optional enhancements like SACK [8], DSACK [7], TLP [17], F-RTO [35], and PRR [28]. Additionally, we desire to do this *without* modifying the sender or making assumptions about what kind of debugging information it makes available. Finally, key variables that determine the state of the sender (like `cwnd`, `ssthresh`, and `rto_timeout`) are not exposed by the sender and are not readily computable from network traffic.

To overcome these challenges, we choose to *approximate* the congestion control state machine by focusing on its core states and *assume* a bulk transfer application that always has data available to send. This is practical because nearly all TCP congestion control algorithms contain the same basic core set of states from TCP New Reno (see Fig. 1) with the differences being in terms of small changes in the actions done on each transition or the insertion of extra states. For example, CUBIC TCP simply modifies the additive increase and multiplicative decrease constants on the transitions to fast recovery and congestion avoidance. Similarly, TLP adds a single state before exponential backoff. It is entered using a slightly smaller timeout and sends a single new packet to try and avoid an expensive RTO.

We developed a novel algorithm to track the sender's congestion control state using only network traffic. We find that this algorithm works well even when used with implementations containing complex state machines and enhancements and approximating these using only TCP New Reno. Our algorithm detects the fast recovery state even when the `cwnd` reduction is CUBIC's 0.8 factor and not the 0.5 used by New Reno. It still identifies retransmitted packets and enters fast recovery even if SACK is in use and fast recovery was triggered via SACK blocks. TLP is a case where our approximation fails, but even here we misclassify a tail-loss-probe as an RTO. This is only a minor issue because both states are entered via by timeouts and trigger the transmission of a single packet. The pseudocode of our algorithm is presented in Appendix B.

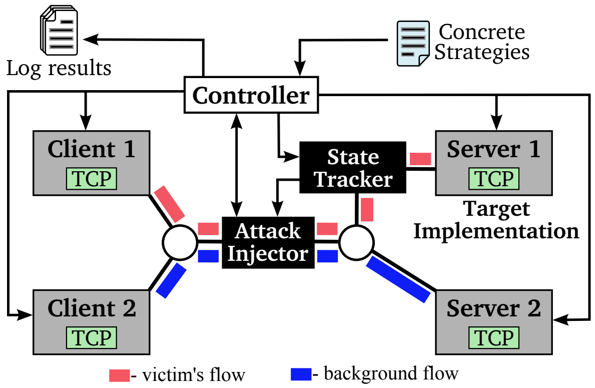


Fig. 4. Testing Environment of TCPwn

V. Implementation

Given the different variants of TCP congestion control algorithms, features, and optimizations [8], [7], [17], [28], [35], [34], [13] any implementation has to make choices about what configuration and combination of features will be provided. This leads to minor differences in congestion control behavior between implementations which can enable or prevent particular attacks or even attack classes.

A. Testing Environment

We developed a testing environment (Fig. 4) which leverages virtualization for both client and server, enabling us to run a wide range of implementations, independent of operating system, programming language, libraries, or availability of source code.

We connect four virtual hosts into a dumbbell topology with two clients on one side, two servers on the other, and a single bottleneck link between. When each client connects to one of the servers, this topology provides an environment where two flows have to compete for bandwidth on the bottleneck link. This competition is precisely what an attacker must influence in order to either increase or decrease the throughput of his target flow. We connect the virtual machines together with Linux tap devices and bridges. We artificially cap the bandwidth on the bottleneck link and introduce a 10ms delay, using Linux traffic control. This gives us a virtual network based on the widely used Linux networking stack that supports throughput in excess of 800Mbits/sec.

One of the servers runs the target TCP implementation under test. The other hosts run a standard TCP stack and serve simply to complete the test harness and generate necessary traffic. To generate traffic, our tests use file transfers over HTTP. This simplifies setting up the target implementation, as HTTP servers are available for a wide variety of operating systems and implementations.

The Attack Injector is implemented as a proxy placed in the middle of the bottleneck link. It intercepts all packets in the target connection and applies any on-path basic actions. It can also inject new packets into the network to emulate an off-path attacker. The proxy also measures connection length and amount of data transferred for attack detection and is implemented in C++.

The State Tracker component is also implemented as a proxy and is placed in our testing environment as near to the target sender as possible. This proxy observes the packets sent and received by the sender over small timeslices to automatically infer the current state of the sender's TCP congestion control state machine. This proxy is also implemented in C++.

This whole environment is controlled and coordinated by a Controller script that takes a concrete strategy from our strategy generator, orchestrates the virtual machines, applications, Attack Injector, and State Tracker components to test that strategy, collects the results, and returns them for analysis.

B. Attack Detection

The goal of an attacker targeting congestion control is to impact throughput. We distinguish between four cases for a target connection that are the observable outcome of an attack:

- *Benign*: no attack occurs.
- *Faster*: the sender sends at a faster rate than it should; throughput is larger than the benign case; this corresponds to a sender bypassing congestion control to send faster.
- *Slower*: the sender is made to send at a slower rate than what the network conditions will allow, the throughput is smaller than a benign connection.
- *Stall*: the connection has stalled and will never complete; corresponds to the case where the attacker made the connection to stall.

Measuring the time it takes to transfer a file at the application layer is not sufficient because it does not allow us to distinguish between two cases: sending faster or connection stalled. Both appear, in some cases, as stalled because the TCP receiver has blocked reassembling data, while all data has already been sent. Thus, the first metric we use is the time it takes to transfer and acknowledge all data packets at the TCP level, referred to simply as $Time$ below.

The time needed to transfer the data at the TCP level is not sufficient to accurately classify attacks because it does not capture the case when the connection stalls out part way through due to an attack and the file has actually not been transferred in its entirety. To detect this case, we use a second metric, the amount of data transferred in the connection at the TCP-level, referred to as $SentData$ below.

We perform 20 tests transferring a file of size $FileSize$ without any attacks being injected to create baseline average and standard deviation values of $TimeBenign$ and $stddev$. Then, using the $Time$ and $SentData$ metrics defined above, our detection works as follows:

```
if Time is > (TimeBenign + 2*stddev):
    Attack: Slower
else if Time is < (TimeBenign + 2*stddev):
    if SentData >= (0.8*FileSize):
        Attack: Faster
    else:
        Attack: Stall
else:
    Benign
```

TABLE I. SUMMARY OF TCPWN RESULTS

Implementation	Attacker	Tested	Marked	FP	Attacks
Ubuntu 16.10 (Linux 4.8)	On-path	564	38	3	35
Ubuntu 14.04 (Linux 3.13)	On-path	564	37	1	36
Ubuntu 11.10 (Linux 3.0)	On-path	564	16	6	10
Debian 2 (Linux 2.0)	On-path	564	3	0	3
Windows 8.1	On-path	564	9	1	8
Ubuntu 16.10 (Linux 4.8)	Off-path	753	466	8	458
Ubuntu 14.04 (Linux 3.13)	Off-path	753	448	9	439
Ubuntu 11.10 (Linux 3.0)	Off-path	753	564	10	554
Debian 2 (Linux 2.0)	Off-path	753	425	0	425
Windows 8.1	Off-path	753	471	3	468
Total		6585	2477	41	2436

VI. RESULTS

We tested five different implementations of TCP in five operating systems: Ubuntu 16.10, Ubuntu 14.04, Ubuntu 11.10, Debian 2, and Windows 8.1. The tests were run on a hyperthreaded 20 core Intel® Xeon® 2.4GHz system with 125GB of RAM. We configured the bottleneck link to be 100Mbits/sec, with a 20ms RTT, and generated traffic for both the target and competing TCP connections with a 100MB HTTP file download for all implementations except Debian 2. Due to limitations with the virtualized NIC, Debian 2 was limited to 10Mbits/sec, so we also limited the bottleneck link to that same rate with a 20ms RTT while traffic generation used a 10MB file. We used the Apache webserver for Linux and IIS on Windows.

Testing each implementation took about 13 hours for the on-path testing and 21 hours for the off-path testing, using only 6 cores. Testing each strategy is independent and takes between 15 and 60 seconds. With 48 cores running eight testing environments (each needs 6 cores), the on-path testing could have been completed in 1.6 hours and the off-path testing in 2.6 hours.

Over all the tested systems, we tested 6,585 strategies and found 2,436 attacks, which we classified into 11 classes. 8 of these classes were previously unknown in the literature. We summarize the attacks in Tables I and II. For lack of space, below we discuss only the new attacks and we present the rest in Appendix C.

While this analysis was performed manually, we observe that it is amenable to automation. In our results, three classes of attacks—Optimistic Ack, Desync, and Ack Lost Data—make up the majority of marked strategies. An automated classification of these three categories leaves only 281 (11%) strategies to manually examine.

A. On-path Attacks

We only consider attacks resulting in increased throughput for some target connection to be of interest to this attacker. Our model-guided strategy generation produced 564 strategies based on the basic actions described in section IV-C. As shown in Table I, our system marked between 3 and 38 of these strategies (depending on implementation). A few of these marked attacks were false positives, due to the imprecision of testing with a real network and real implementations. In

TABLE II. Classes of Attacks Discovered by TCPwn

Num	Attack	Attacker	Description	Impact	Impl	New
1	Optimistic Ack	On-path	Acking data that has not been received	Increased Throughput	ALL	No [37]
2	On-path Repeated Slow Start	On-path	Repeated cycle of Slow Start, RTO, Slow Start due to fixed ack number during Fast Recovery	Increased Throughput	U(buntu)16.10, U11.10	Yes
3	Amplified Bursts	On-path	Send acks in bursts, amplifying the bursty nature of TCP	Increased Throughput	U11.10	Yes
4	Desync Attack	Off-path	Inject data to desynchronize sequence numbers and stall connection	Connection Stall	ALL	No [22]
5	Ack Storm Attack	Off-path	Inject data into both sides of connection, creating ack loop	Connection Stall	D(ebian)2, W(indows)8.1	No [2]
6	Ack Lost Data	Off-path	Acknowledge lost data during Fast Recovery or Slow Start	Connection Stall	ALL	Yes
7	Slow Injected Acks	Off-path	Inject acks for little data slowly during Congestion Avoidance	Decreased Throughput	U11.10	Yes
8	Sawtooth Ack	Off-path	Send incrementing acks in Congestion Avoidance/Fast Recovery, but reset on entry	Decreased Throughput	U16.10,U14.04, U11.10, W8.1	Yes
9	Dup Ack Injection	Off-path	Inject >= 3 duplicate acks repeatedly	Decreased Throughput	D2, W8.1	Yes
10	Ack Amplification	Off-path	Inject acks for lots of new data very rapidly during Congestion Avoidance or Slow Start	Increased Throughput	U16.10,U14.04, U11.10, W8.1	Yes
11	Off-path Repeated Slow Start	Off-path	Repeated cycle of Slow Start, RTO, Slow Start due to increased duplicate ack threshold	Increased Throughput	U11.10	Yes

particular, while our target connection typically incurs its first loss within 0.5 seconds of starting, due to competing with the background connection, in these false positive tests the first loss in the target connection does not occur until after at least 1.5 seconds. Since TCP continues to increase its sending rate until it gets a loss, this results in an unusually high sending rate. This longer time to loss is not attributable to any basic action applied, but is simply a result of variations in packet arrival and departure times, packet processing delays, operating system scheduling, and other random variations. The remaining marked strategies are real attacks against a TCP implementation. We identified between 3 and 36 of these, depending on the implementation. Through manual analysis, we grouped these into 3 classes (Table II), two of which are previously unknown in the literature.

On-path Repeated Slow Start (new). These attacks operate by repeatedly inducing an RTO followed by Slow Start. Thanks to Linux's choice to use a short RTO timer, the rapid increase in sending rate during Slow Start balances out the idle period needed to cause an RTO and in many tests actually provides a higher average sending rate. This is partly due to the significant impact this attack has on competing connections because of the repeated, rapid sending periods that end in a loss for both connections. These repeated losses cause the competing connection to slow down repeatedly. We found this attack class against both Ubuntu 11.10 and Ubuntu 16.10. For both versions, this behavior is best induced by preventing an increase of the cumulative acknowledgement in Fast Recovery, preventing recovery of losses and causing an RTO.

Amplified Bursts (new). This class of attack operates by collecting acknowledgement packets and then sending them together in a burst. This additional burstiness often causes more frequent losses in the competing connection which causes it to slow down and our target flow to increase its throughput. We found this attack class against Ubuntu 11.10 with a strategy that collected acknowledgement packets to send them in bursts during Congestion Avoidance and optimistically acknowledged data during Slow Start, increasing the size of `cwnd`. It is interesting to note that without our model-guided strategy generation we would have been extremely unlikely to find this attack. This is because delaying acknowledgements and sending them in bursts is only a good idea during Congestion Avoidance. During Slow Start, `cwnd` is small enough that there may not be enough acknowledgements in flight to make a single burst, leading to a connection stall. Similarly, in Fast Recovery, the sender needs to get acknowledgements as soon as possible so that it can recover from the loss and keep sending data. Delaying acknowledgements and collecting enough for a single burst tends to cause the connection to stall.

This attack bears significant resemblance to the Induced-Shrew Attack [25]. However, that attack seeks to manipulate a TCP connection to cause catastrophic throughput reduction on other competing connections while maintaining a minimal sending rate itself. Instead, the Amplified Burst attack focuses on increasing the throughput of our target connection.

B. *Off-path Attacks*

An off-path attacker can observe network traffic but cannot directly modify such traffic. As a result, they are limited to injecting new (possibly spoofed) packets into the network. In addition to increasing throughput, possibly as part of a denial of service attack, an off-path attacker might be interested in decreasing the throughput or stalling some target connection.

Our model-guided strategy generation produced 753 strategies based on injecting spoofed packets. As shown in Table I, our system marked between 425 and 564 of these strategies (depending on implementation) as attacks. A few of these marked attacks turned out to be false positives. These are mostly cases where, due to imprecision from testing real implementations, the target connection does not see its first loss for an abnormally long time, leading to a higher sending rate than normal. We present a summary in Table II.

Ack Lost Data (new). This class of attacks contains a wide range of operations that cause lost data to be perceived as acknowledged at some point in the connection. This occurs when an attacker injects a spoofed acknowledgement packet acknowledging data above the current cumulative acknowledgement when the network is about to enter Fast Recovery. In this case, at least some of the lost data will be deemed acknowledged by the victim, causing that data to never be retransmitted. At this point, anything the sender retransmits or sends will not cause the receiver to increase the cumulative acknowledgement and the connection permanently stalls. We found a wide variety of strategies in this attack class against all implementations we tested.

Slow Injected Acks (new). These attacks operate by injecting spoofed acknowledgements that increase their acknowledgment number at a slow and constant rate. As these acknowledgement packets are injected, each one causes TCP to send a few packets–equivalent to the amount of data acknowledged–, due to TCP's self-clocking design. This essentially causes TCP to bypass congestion control and $cwnd$ entirely and send at the rate at which the spoofed acknowledgements are acknowledging data: $ack\_amount * injection\_frequency$. This rate can be made much slower than TCP would otherwise achieve. Additionally, due to the spoofed acknowledgements, any real acknowledgements for data will be considered old and ignored. We found this class of attacks against Ubuntu 11.10.

Sawtooth Ack (new). These attacks also operate using spoofed acknowledgements that increase their acknowledgement number at a steady pace. However, these packets may acknowledge more data and occasionally reset their acknowledgment number to the true cumulative acknowledgement point. This starting over, typically at a state transition from Congestion Avoidance to Fast Recovery or back, results in a long string of spoofed acknowledgements with increasing acknowledgement numbers that eventually reaches the previous high acknowledgement, at which point the sender begins sending new data. This causes a very prominent sawtooth pattern in a time sequence graph of the connection. Due to the increasing number of acknowledgements that must be sent to reach the highest acknowledgement each time, the sending rate of a connection under this type of attack continuously decreases. We found this class of attacks against Ubuntu 16.10, Ubuntu 14.04, Ubuntu 11.10, and Windows 8.1 using a variety of strategies. In our tests, this attack usually resulted in approximately a 12x reduction in throughput. The attacker is required to expend approximately 40Kbps for this attack.

Dup Ack Injection (new). This class of attack operates by repeatedly injecting three or more spoofed duplicate acknowledgements into the target connection in hopes of spuriously triggering Fast Recovery and slowing the connection down. We have found this class of attack to be very effective against Windows 8.1 and Debian 2. Newer Linux versions are not vulnerable to this attack due the use of DSACK [7] to detect spurious retransmissions and a mechanism to dynamically adjust the duplicate acknowledgement threshold needed to trigger Fast Recovery [42]. In our tests, this attack often resulted in approximately a 12x reduction in throughput when using Windows 8.1 or Debian 2. The connection repeatedly enters Fast Recovery and needlessly retransmits significant data. The attacker needs only 40Kbps to launch this attack.

Ack Amplification (new). This class of attack operates similarly to Slow Injected Acks. Instead of sending spoofed acknowledgements with increasing sequence numbers slowly, the attacker sends them very quickly. Each one causes the sender to send a large burst of packets, effectively bypassing congestion control and `cwnd` completely. This effect is even more pronounced in Slow Start, where the sender can send two bytes for every one acknowledged. Additionally, since any losses are masked by the spoofed acknowledgements, TCP will never slow down. This results in a very powerful class of attack where an attacker can cause the target connection to consume all available bandwidth up to the network and/or sending system capacity by simply sending acknowledgements at around 40Kbps. In our tests, the competing connection was left starved for bandwidth, with throughput near zero, and often doing repeated RTOs for the duration of the attack. The low bandwidth required makes this ideal for a denial of service attack. We found a wide variety of strategies in this attack class against Ubuntu versions 16.10, 14.04, 11.10, and Windows 8.1.

Off-path Repeated Slow Start (new). This class of attacks is very similar to the On-path Repeated Slow Start attacks discussed previously. We found this attack in Ubuntu 11.10.

VII. Related Work

Attacks on congestion control. Previous work manually identified several attacks against TCP congestion control. First work in this area is [22] which identified the Desynchronization Attack which causes the sender and receiver to become desynchronized with respect to the location of the cumulative acknowledgement, resulting in a connection stall. Three attacks were shown in [37]: Optimistic Ack, Ack Division, and Dup Ack Spoofing. These attacks allow a malicious receiver or on-path attacker to increase the throughput of a target connection by modifying how it acknowledges data, either acknowledging more data than it should, acknowledging it in many little pieces, or repeatedly acknowledging the same data. Ack Division and Dup Ack Spoofing has since been widely mitigated by applying Appropriate Byte Counting [3] and similar implementation-level mitigations.

The work in [26] and [25] introduced two attacks which degrade TCP throughput along some target link while expending minimal bandwidth in an attempt to avoid detection.

A security analysis of TCP commissioned by the British Government [12] identified two additional attacks available to a blind attacker. These are the Blind Flooding Attack and Blind Throughput Reduction Attack. Both operate by sending spoofed acknowledgements which will cause the receiver to send a duplicate acknowledgment if the packet is out of the acceptable sequence window.

Finally, [2] identified the Ack Storm attack where the injection of data into a target connection prevents further data transfer and generates an infinite series of acknowledgements, as both parties respond to what they consider to be an invalid acknowledgement with an acknowledgement.

In contrast to these, TCPwn performs an automated analysis of TCP congestion control, based on a state machine description, to identify potential attacks and then automatically tests real implementations of TCP for possible attacks.

Automated vulnerability discovery in protocols. Prior work has looked at automatically finding vulnerabilities in network protocols by using fuzzing. While random fuzz testing [29] is often effective in finding interesting corner case errors, the probability of "hitting the jackpot" is low because it typically mutates the well-formed inputs and tests the program on the resulting inputs. To overcome this inherent problem of fuzzing, a set of works like SNOOZE [6], KiF [1], and SNAKE [21] leverage the protocol state machines to cover deeper and more relevant portions of the search space. They require the end users to provide the protocol specification (*e.g.*, message format, state machines) and various fault injection scenarios to discover vulnerabilities in stateful protocols. These tools primarily search for crashes or other fatal errors. In contrast, TCPWN aims to automatically discover attacks on the runtime performance of TCP congestion control by leveraging a model-guided search technique.

Several other research efforts [14], [24], [38], [36], [5] leverage program analysis, for example, symbolic execution, to find vulnerabilities in protocol implementations. MAX [24] focuses on two-party protocols to find performance attacks mounted by a compromised participant that can manipulate the victim's execution control flow. However, MAX relies on user specified information about a known vulnerability of the code to limit the search space during symbolic execution. In contrast, TCPWN relies on vulnerable actions common to the protocol state machine, not specific to a particular implementation. MACE [14] combines symbolic execution with concrete execution to infer the protocol state machine and use it as a search space map to allow deep exploration for bugs. While these state machines can represent the behavior of protocols with various message types (*e.g.*, RFB, SMB), they cannot capture the different aspects of TCP congestion control because the number of messages (*e.g.*, ACKs) plays a significant role in lieu of types. Therefore, TCPWN utilizes the congestion control state machine derived from the specifications to generate an effective, but reduced, set of test scenarios.

VIII. Conclusion

Today, the testing of congestion control and the discovery of attacks against it is mostly a manual process performed by protocol experts. We developed TCPWN, a system to automatically test real implementations of TCP by searching for attacks against their congestion control. TCPWN uses a model-guided attack generation strategy to generate abstract attack strategies which are then converted to concrete attack scenarios made up of message-based actions or packet injections. Finally, these concrete attack scenarios are applied in our testing environment, which leverages virtualization to run real implementations of TCP independent of operating system, programming language, or libraries. We evaluated 5 TCP implementations including both open- and closed- source systems, using TCPWN. We found 2,436 attack strategies which could be grouped into 11 classes, of which 8 are new.

Acknowledgment

This material is based in part upon work supported by the National Science Foundation under Grant Numbers CNS-1600266, CNS-1617728, and CNS-1409191. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] H. Abdelnur, R. State, and O. Festor, "KiF: A stateful SIP fuzzer," in *International Conference on Principles, Systems and Applications of IP Telecommunications*, 2007, pp. 47–56.

[2] R. Abramov and A. Herzberg, "TCP ack storm DoS attacks," in *IFIP International Information Security Conference*, 2011, pp. 29–40.

[3] M. Allman, "TCP Congestion Control with Appropriate Byte Counting (ABC)," RFC 3465 (Experimental), 2003.

[4] M. Allman, V. Paxson, and E. Blanton, "TCP congestion control," RFC 5681 (Draft Standard), 2009.

[5] R. Banabic, G. Candea, and R. Guerraoui, "Finding trojan message vulnerabilities in distributed systems," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014, pp. 113–126.

[6] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer, and G. Vigna, "SNOOZE: Toward a stateful network protocol fuzzer," in *International Conference on Information Security*, 2006, pp. 343–358.

[7] E. Blanton and M. Allman, "Using TCP Duplicate Selective Acknowledgement (DSACKs) and Stream Control Transmission Protocol (SCTP) Duplicate Transmission Sequence Numbers (TSNs) to Detect Spurious Retransmissions," RFC 3708 (Experimental), 2004.

[8] E. Blanton, M. Allman, L. Wang, I. Jarvinen, M. Kojo, and Y. Nishida, "A Conservative Loss Recovery Algorithm Based on Selective Acknowledgment (SACK) for TCP," RFC 6675 (Proposed Standard), 2012.

[9] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson, "TCP vegas: New techniques for congestion detection and avoidance," in *Conference on Communications Architectures, Protocols and Applications*, 1994.

[10] Y. Cao, Z. Qian, Z. Wang, T. Dao, S. V. Krishnamurthy, and L. M. Marvel, "Off-path TCP exploits: Global rate limit considered dangerous," in *USENIX Security Symposium*, 2016, pp. 209–225.

[11] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, "BBR: Congestion-based congestion control," *ACM Queue*, vol. 14, September-October, pp. 20 – 53, 2016.

[12] Centre for the Protection of National Infrastructure, "Security assessment of the transmission control protocol," Centre for the Protection of National Infrastructure, Tech. Rep. CPNI Technical Note 3/2009, 2009.

[13] Y. Cheng, N. Cardwell, and N. Dukkipati, "RACK: a time-based fast loss detection algorithm for TCP," draft-ietf-tcpm-rack-01.txt, 2016.

[14] 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 *USENIX Conference on Security*, 2011.

[15] J. Chu, N. Dukkipati, Y. Cheng, and M. Mathis, "Increasing TCP's Initial Window," RFC 6928 (Experimental), 2013.

[16] J. de Ruiter and E. Poll, "Protocol state fuzzing of TLS implementations," in *USENIX Security Symposium*, 2015.

[17] N. Dukkipati, N. Cardwell, and Y. Cheng, "Tail Loss Probe (TLP): An Algorithm for Fast Recovery of Tail Losses," draft-dukkipati-tcpm-tcp-loss-probe-01.txt, 2013.

[18] Y. Gilad and A. Herzberg, "Off-path attacking the web," in *WOOT*, 2012, pp. 41–52.

[19] T. Henderson, S. Floyd, A. Gurtov, and Y. Nishida, "The NewReno modification to TCP's fast recovery algorithm," RFC 6582 (Proposed Standard), 2012.

[20] V. Jacobson, "Congestion avoidance and control," *ACM SIGCOMM Computer Communication Review*, vol. 18, no. 4, pp. 314–329, 1988.

[21] S. Jero, H. Lee, and C. Nita-Rotaru, "Leveraging State Information for Automated Attack Discovery in Transport Protocol Implementations," in *IEEE/IFIP International Conference on Dependable Systems and Networks*, 2015.

[22] L. Joncheray, "A simple active attack against TCP," in *USENIX Security Symposium*, 1995.

[23] S. Kent and K. Seo, "Security Architecture for the Internet Protocol," RFC 4301 (Proposed Standard), 2005.

[24] N. Kothari, R. Mahajan, T. Millstein, R. Govindan, and M. Musuvathi, "Finding Protocol Manipulation Attacks," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, 2011.

[25] V. A. Kumar, P. S. Jayalekshmy, G. K. Patra, and R. P. Thangavelu, "On remote exploitation of TCP sender for low-rate flooding denial-of-service attack," *IEEE Communications Letters*, vol. 13, no. 1, pp. 46–48, 2009.

[26] A. Kuzmanovic and E. Knightly, "Low-rate TCP-targeted denial of service attacks and counter strategies," *IEEE/ACM Transactions on Networking*, vol. 14, no. 4, pp. 683–696, 2006.

[27] H. Lee, J. Seibert, E. Hoque, C. Killian, and C. Nita-Rotaru, "Turret: A platform for automated attack finding in unmodified distributed system implementations," in *International Conference on Distributed Computing Systems*, 2014, pp. 660–669.

[28] M. Mathis, N. Dukkipati, and Y. Cheng, "Proportional Rate Reduction for TCP," RFC 6937 (Experimental), 2013.

[29] B. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Communications of the ACM*, vol. 33, no. 12, 1990.

[30] G. Nelson and D. Oppen, "Fast decision procedures based on congruence closure," *J. ACM*, vol. 27, no. 2, pp. 356–364, 1980.

[31] J. Postel, "Transmission control protocol," RFC 793 (Standard), 1981.

[32] Z. Qian and Z. M. Mao, "Off-path TCP sequence number inference attack - how firewall middleboxes reduce security," in *IEEE Symposium on Security and Privacy*, 2012, pp. 347–361.

[33] Z. Qian, Z. M. Mao, and Y. Xie, "Collaborative TCP sequence number inference attack: how to crack sequence number under a second," in *ACM Conference on Computer and Communications Security*, 2012.

[34] I. Rhee, L. Xu, S. Ha, A. Zimmermann, L. Eggert, and R. Scheffenegger, "CUBIC for Fast Long-Distance Networks," draft-ietf-tcpm-cubic-02.txt, 2016.

[35] P. Sarolahti, M. Kojo, K. Yamamoto, and M. Hata, "Forward RTO-Recovery (F-RTO): An Algorithm for Detecting Spurious Retransmission Timeouts with TCP," RFC 5682 (Proposed Standard), 2009.

[36] 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 *IEEE International Conference on Information Processing in Sensor Networks*, 2010, pp. 186–196.

[37] S. Savage, N. Cardwell, D. Wetherall, and T. Anderson, "TCP congestion control with a misbehaving receiver," *ACM SIGCOMM Computer Communication Review*, vol. 29, no. 5, 1999.

[38] J. Song, C. Cadar, and P. Pietzuch, "SymbexNet: Testing Network Protocol Implementations with Symbolic Execution and Rule-Based Specifications," *IEEE Transactions on Software Engineering*, vol. 40, no. 7, pp. 695–709, 2014.

[39] M. Sridharan, K. Tan, D. Bansal, and D. Thaler, "Compound TCP: A New TCP Congestion Control for High-Speed and Long Distance Networks," draft-sridharan-tcpm-ctcp-02.txt, 2009.

[40] R. Stewart, S. Long, D. Gallatin, A. Gutarin, and E. Livengood, "The netflix tech blog: Protecting netflix viewing privacy at scale," 2016. [Online]. Available: http://techblog.netflix.com/2016/08/protecting-netflix-viewing-privacy-at.html

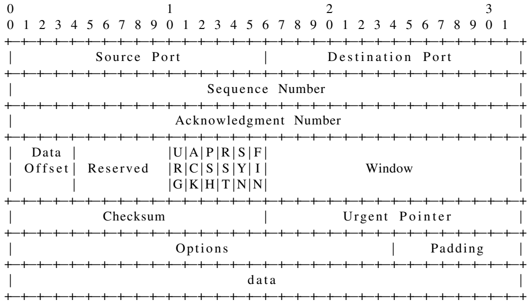
[41] A. Studer and A. Perrig, "The coremelt attack," in *European Symposium on Research in Computer Security*, 2009, pp. 37–52.

[42] The Linux Kernel Community, "/proc/sys/net/ipv4/* variables," 2017. [Online]. Available: https://www.kernel.org/doc/Documentation/networking/ip-sysctl.txt

[43] M. Utting and B. Legeard, *Practical model-based testing: a tools approach*. Morgan Kaufmann, 2010.

Appendix A
TCP Acknowledgements

All TCP packets contain a single, common header. (shown in Fig. 5). This header contains source and destination ports, a sequence number, an acknowledgment number, a set of control bits, a checksum, and options. One of the control bits is the ACK bit, which indicates whether the acknowledgment number included in the header is meaningful. After the initial TCP handshake, all packets set this bit and include the current TCP acknowledgement number. Hence, all TCP packets include acknowledgement information and may acknowledge new data or indicate duplicate acknowledgements.



```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Sequence Number                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Acknowledgment Number                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Data |           |U|A|P|R|S|F|                               |
| Offset| Reserved  |R|C|S|S|Y|I|            Window             |
|       |           |G|K|H|T|N|N|                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Checksum            |         Urgent Pointer        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             data                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Fig. 5. TCP Header fields. Each tick represents a bit position. [31]

In most TCP connections, only one side of the connection is sending data at any given time. In order to provide feedback to the sender, TCP requires that receivers that are quiescent, that is, not currently sending data themselves, must periodically send an empty TCP packet to supply the sender with a current acknowledgement. These empty TCP packets are simply TCP packets with no data and are usually called pure acknowlegements, or simply *acknowledgements*. We focus on these acknowledgement packets in this work because connections where both hosts are sending simultaneously are rare. However, our attacks are equally applicable in such scenarios and would usually be launched by injecting additional pure acknowledgement packets into the connection.

Appendix B
State Tracking Algorithm

Our attack injector uses the state information supplied by the state tracking algorithm to apply the basic actions corresponding to the current congestion control state during testing. Our state tracking algorithm observes the network traffic from a TCP sender in order to track its current congestion control state. We approximate the exact congestion control algorithm in use by the sender and consider it to be New Reno for the purposes of state tracking.

The core idea of our algorithm is to take a small (sub-RTT) time slice and observe the packets received and sent by an implementation. If about twice as many bytes of data have been sent as acknowledged, the state is inferred to be Slow Start and the sending rate is increasing exponentially. If about an equal number of bytes have been sent and acknowledged, the state is inferred to be Congestion Avoidance since the sender is maintaining a steady sending rate. If fewer bytes have been sent than acknowledged or there are retransmitted packets, the state is inferred to be Fast Recovery, and if no packets are received and only a few packets are sent, then an RTO event was observed and the sender is in state Exponential Backoff.

Our algorithm uses two timers, the first fires every `sub_rtt` seconds and the second fires `max_burst_gap` seconds after each packet unless reset. This first timer handles the case where TCP is operating at high speed and has packets in flight constantly while the second handles the case where TCP has not yet reached peak efficiency and is sending packets in bursts and then waiting for their acknowledgements before sending more. We experimentally set `sub_rtt` to 10ms and `max_burst_gap` to 5ms based on a network RTT of 20ms.

Whenever either of these timers expires, the algorithm determines whether TCP is sending data smoothly or in bursts. If TCP is sending data in bursts and it has been less than `max_burst_gap` seconds since the last packet, this timer expiration, is ignored. Otherwise, the state inference is updated. If the most recent packet was a SYN, FIN, or Reset, then the connection state is INIT or END. Otherwise, we compute the ratio of sent to acknowledged data and the space between the two most recent packets, and use this information to determine what state the sender is in based on the intuition presented above. We then reset our data sent and data acknowledged counters. For the slow start and congestion avoidance state, we average the ratios from the last two sampling periods as we found experimentally that this helped to produce more accurate results. Finally, if the ratio is less than 0.8, a situation that should never occur, we ignore this sample.

Appendix C
Additional Results

Below we present attacks that we have automatically found with TCPwn and which were known.

A. *On-path Attacks*

Optimistic Ack This class of attack operates by optimistically acknowledging data that the receiver has not received and acknowledged yet. This reduces the effective RTT of the connection, allowing TCP to increase its sending rate faster, and hides lost packets, preventing TCP from slowing down in response to congestion. By hiding lost packets, the receiver will not receive the complete data transfer, but this may be acceptable if the data stream can tolerate losses or if the attacker does not care about the data, *i.e.*, is simply conducting a denial of service attack.

This attack class was first identified in [37]. Unfortunately, the mitigations proposed require non-backwards-compatible modifications to TCP, such as inserting a random nonce into each packet. As a result, this attack class is still present in modern TCP implementations, and we found many instances of it in all 5 of the implementations we tested. In our tests, this attack usually caused the target connection to consume all available bandwidth, starving competing connections to near zero throughput for the duration of the attack.

B. *Off-path Attacks*

Desync Attack This class of attacks operates by spoofing packets containing a few bytes of data to both sender and receiver in the target connection. If a host is not currently receiving data, this injected data will incorrectly cause its cumulative acknowledgement number to increase. All future packets by this host will then have an acknowledgement number higher than anything the other host sent and will be ignored, causing an unrecoverable connection stall.

These attacks were first identified by [22]. The only known mitigation is encryption to prevent access to the sequence numbers of the packets. We identified many instances of this attack class against all tested implementations and in all congestion control states.

Ack Storm Attack Ack Storm attacks are similar to Desync Attacks but spoof packets with data into both sides of idle connections. By doing so, the cumulative acknowledgement numbers of both sender and receiver are increased. Unfortunately, since neither side actually sent any data, both will consider any future acknowledgements invalid and respond with a duplicate acknowledgement as required by the TCP specification [31]. This leads to an infinite storm of acknowledgements between both sides of the connection, as each responds to the invalid acknowledgements from the other.

This is a known attack, first identified by [2]. One mitigation to this attack is to ignore invalid acknowledgements if they show up too frequently. Unfortunately, neither Debian 2 nor Windows 8.1 provide this mitigation, enabling us to discover this attack with several different strategies.

Algorithm 2: State Tracking

```
Function Init()
    Start timer $intervalTimer$ to expire every sub_rtt ms (10ms)
    $priorPkt$ = $curPkt$ = now()
    $urgEvent$ = false
    $state$ = UNKNOWN
Function OnPacket($p$)
    update $dataBytes$, $dataPkts$, $ackBytes$, $ackPkts$, $seqHigh$, $highAck$, $curPktType$ and $rexmits$ based on $p$
    if $curPkt <$ now()$-$ max_burst_gap then
        $lastIdle$ = now()
    $priorPkt$ = $curPkt$
    $curPkt$ = now()
    Reset timer $packetTimer$ to expire in max_burst_gap ms (5ms)
    if $rexmits > 0$ then
        $urgEvent$ = true
        Reset timer $packetTimer$ to expire now
Function OnTimer()
    if $urgEvent$ or $curPkt >$max_burst_gap or $lastIdle > 4*$sub_rtt then
        $urgEvent$ = false
        if $curPktType$ is SYN then
            $state$ = INIT
            return
        if $curPktType$ is FIN or RST then
            $state$ = END
            return
        $curRatio$ = $dataBytes$ / $ackBytes$
        $pktSpace = curPkt - priorPkt$
        if $dataPkts > 0$ and ($pktSpace > 200ms$) then
            $state$ = EXP_BACKOFF
        else if $state$ == FAST_RECOV and $ackHigh < ackHold$ then
            $state$ = FAST_RECOV
        else if $rexmits > 0$ or ($ackBytes$ == 0 and $ackPkts > 3$) then
            $ackHold = seqHigh$
            $state$ = FAST_RECOV
        else if $(curRatio + priorRatio)/2 > 1.8$ then
            $state$ = SLOW_START
        else if $(curRatio + priorRatio)/2 > 0.8$ then
            $state$ = CONG_AVOID
        else if $state$ == EXP_BACKOFF and $curRatio < 0.1$ then
            $ackPkts$ = 0
        else
            $priorRatio = 0.8 * curRatio + 0.2 * priorRatio$
            return
        $priorRatio$ = $curRatio$
        $ackPkts$ = $ackBytes$ = 0
        $dataPkts$ = $dataBytes$ = 0
        $rexmits$ = 0
```