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

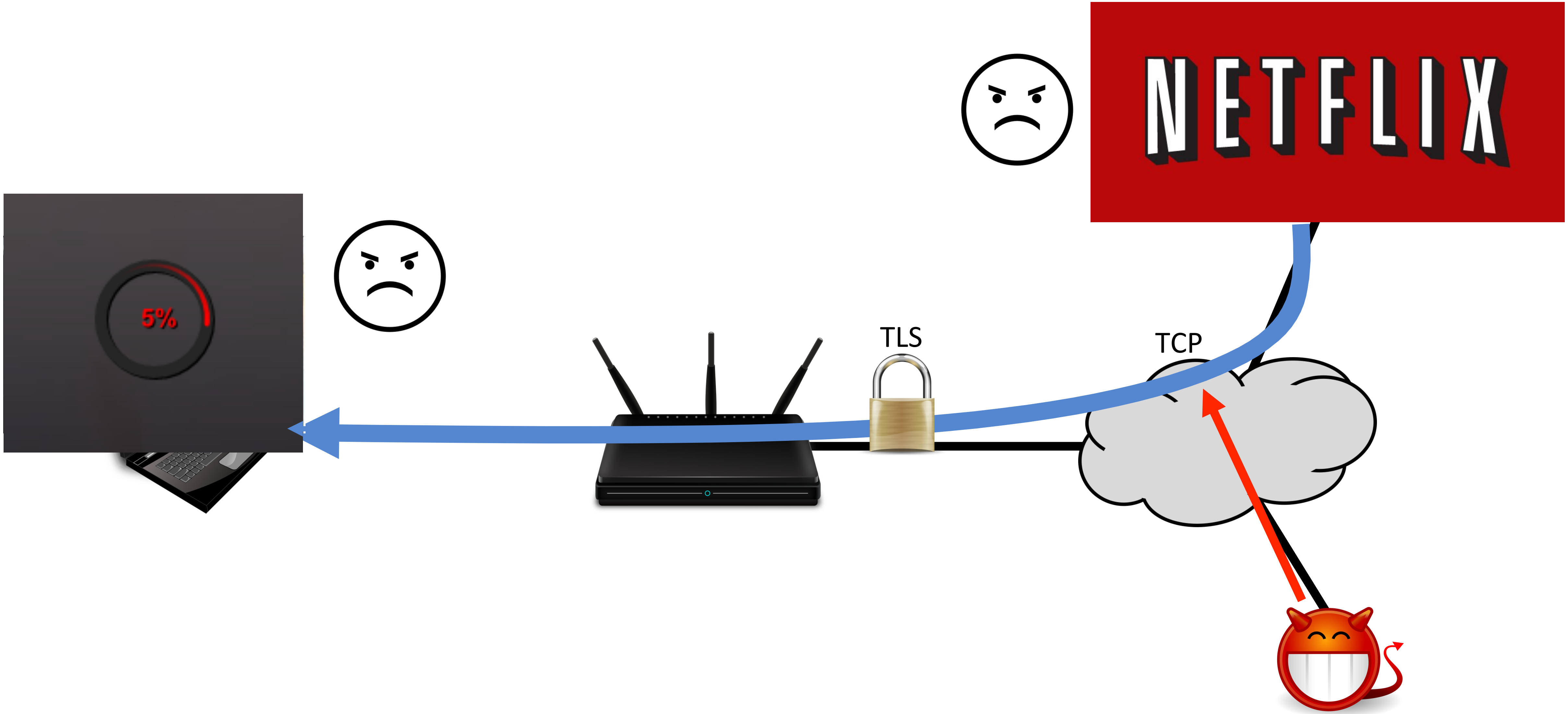
Samuel Jero[1], **Endadul Hoque**[2], **David Choffnes**[3], **Alan Mislove**[3], **and Cristina Nita-Rotaru**[3]

[1]Purdue University, [2]Florida International University, and [3]Northeastern University

NDSS 2018

PURDUE UNIVERSITY FIU FLORIDA INTERNATIONAL UNIVERSITY Northeastern University

A Day In the Life of the Internet

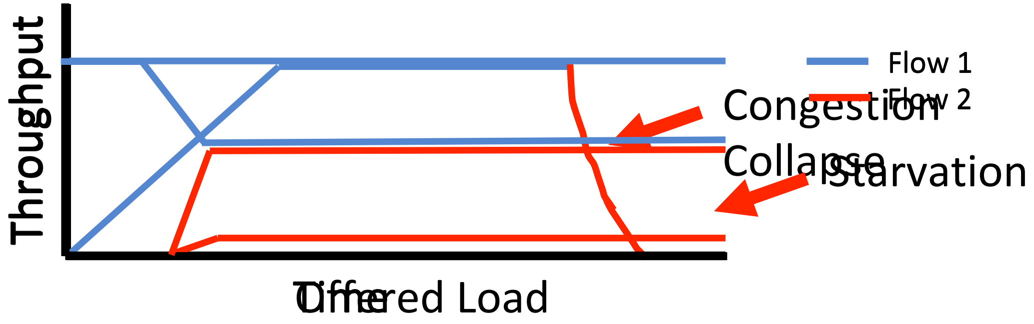


TCP

- Transport protocol used by vast majority of Internet traffic
  - Including traffic encrypted with TLS
  - Including network infrastructure protocols like BGP
- Thousands of implementations
  - Over 5,000 implementation variants detectable by nmap
- Provides:
  - Reliability
  - In-order delivery
  - Flow control
  - *Congestion control*

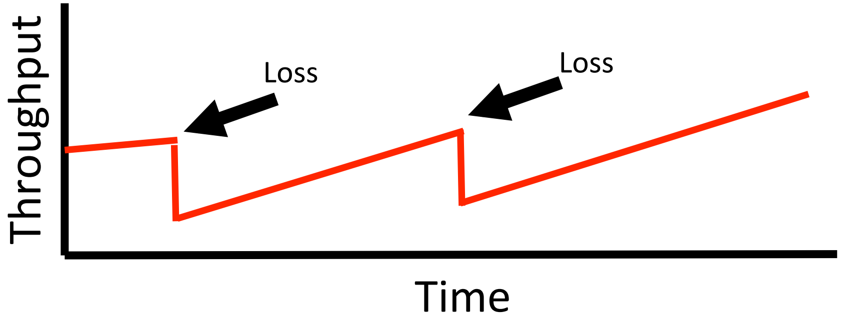
TCP Congestion Control

- Protects against congestion collapse
  - Majority of sent data is dropped later on
  - Caused throughout decrease of 1000x in 1988
- Also ensures fairness between competing flows
  - Prevents one flow from starving others

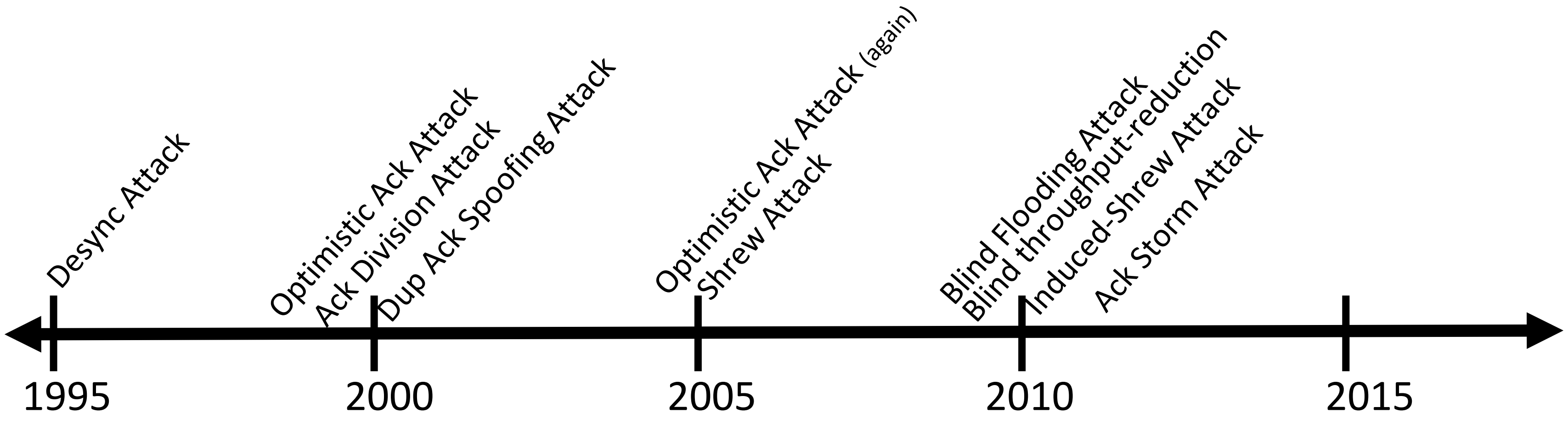


Congestion Control is Crucial for Modern Networks

- General scheme
  - Additive Increase, probing for more bandwidth
  - Loss indicates congestion
  - Multiplicative Decrease, slowing down to clear congestion

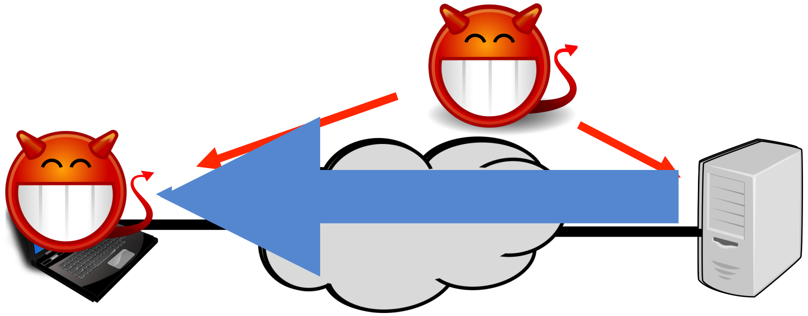


Long History of Powerful Attacks



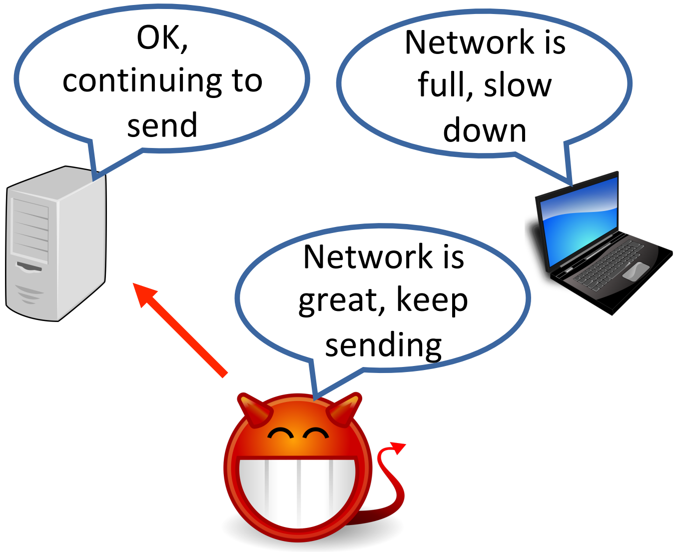
Attacks may result in:

- **Decreased** throughput
- **Increased** throughput that starves competing flows
- **Stalled** data transfer



Why So Many Attacks?

- Attacks leverage designed behavior
  - Congestion control is designed to control throughput
  - Attacks confuse congestion control about network conditions
  - No crashes or unusual control flow
- Many designs and implementations
  - Multiple Variations: Reno, New Reno, SACK, Vegas, BBR
  - Multiple Optimizations: PRR, TLP, DSACK, FRTO, RACK
  - Hundreds of implementations
- Lack of unified specifications
  - Individual components and optimizations are specified separately
  - Understanding unified behavior is difficult
- Very dynamic behavior
  - Congestion control state changes with every acknowledgement
  - Impact of individual packet dilutes quickly with time



RFC 2861 RFC 793 RFC 7323
RFC 5681 RFC 5827 RFC 3390
RFC 2581 RFC 3465 RFC 6937
RFC 2001 RFC 2018 RFC 3708
RFC 6298 RFC 3042 RFC 6582
RFC 4653 RFC 6675 RFC 2883
RFC 4015 RFC 5682 RFC 6528

Current Testing Methods

- Manual Investigation
  - Security researchers manu
- Regression Testing
  - Manually create tests for k
  - Test each implementation
- MAX [SIGCOMM'11]
  - Automatically finds manip
  - Leverages symbolic execut
- SNAKE [DSN'15]
  - Automatically fuzzes transport protocols searching for availability and performance attacks
  - Uses state-machine attack

Labor Intensive, requires human to enumerate all possible attacks, does not scale

Unable to find new vulnerabilities, different implementations may not be vulnerable in the same way

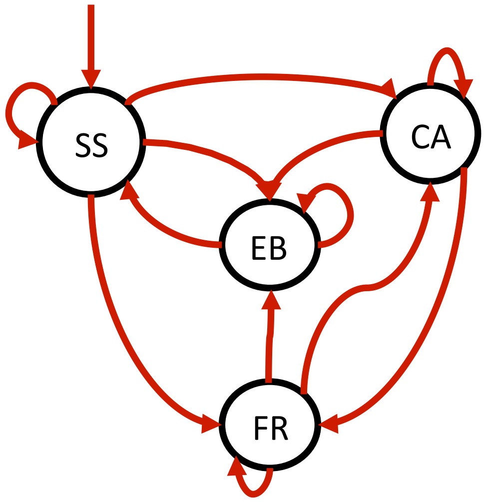
Requires source code in a particular language and manual annotations

Does not scale to highly dynamic systems and complex attacks with many steps

Our Approach: TCPwn

Goal: Automatically test TCP implementations for attacks on Congestion Control

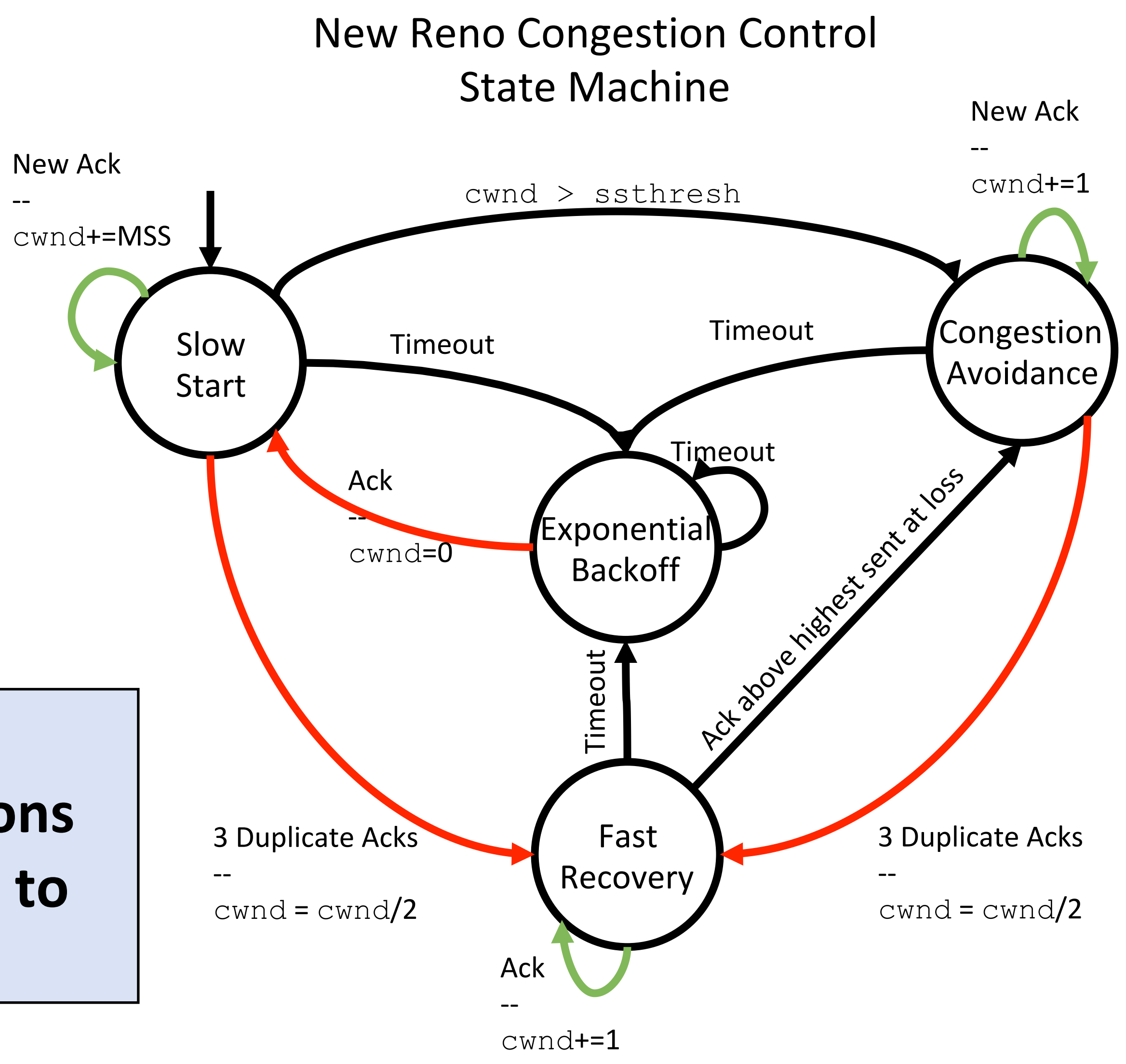
- Test *real, unmodified* implementations
- **Scalability** was the major challenge: attacks are complex and multi-stage, system is highly dynamic
- Model TCP congestion control as a state machine
- Use **model-based testing** to identify all possible attacks in a *scalable* manner
- Create testable attacks using packet manipulation and injection
- Finds attacks causing:
  - Decreased Throughput
  - Increased Throughput
  - A connection stall



Optimistic Ack Attack

Increase sending rate by acknowledging data that has not been received yet

- Acknowledging new data causes green transitions to be taken
- Increases `cwnd` and thus throughput with each loop
- Avoids red transitions which reduce `cwnd` and thus throughput



Key Takeaways:

- **Attacks attempt to cause desirable transitions**
- **Attacks must repeatedly execute transition to have noticeable impact**

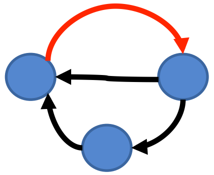
Model-based Attack Generation

Generate all cycles with the following pattern:

- `cwnd` **increases/decreases along cycle**
- **A set of actions exist that force TCP to follow this cycle**

1. Consider state machine model of congestion control
2. Identify cycles containing desirable transitions
   - Abstract strategy generation
3. Force TCP to follow each cycle
   - Concrete strategy generation

State Machine



Abstract Strategies

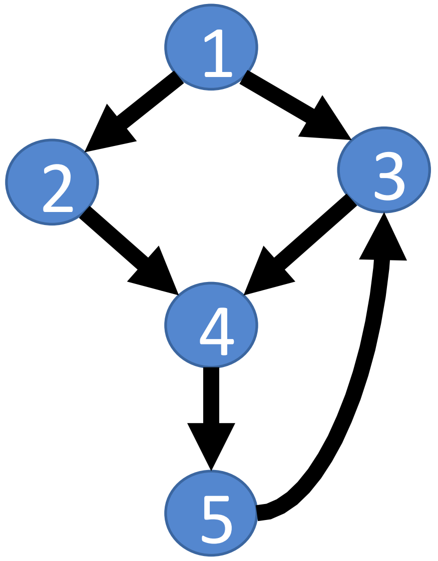
1,2,1…
1,2,3,1…

Concrete Strategies

Delay Msg1, Drop Msg2
Drop Msg3, Dup Msg4

Abstract Strategy Generation

- Enumerate all paths
  - No standard graph algorithm
  - We adapt *depth first search* to this problem
- Check that path contains cycle
- Check that cycle contains desirable transitions
  - Any change to `cwnd`
- Add path and transition conditions to abstract strategies



✓ **Cycle**

✓ **Desirable Transition**

Abstract strategies are merely desirable cycles; they may not be realizable in practice!

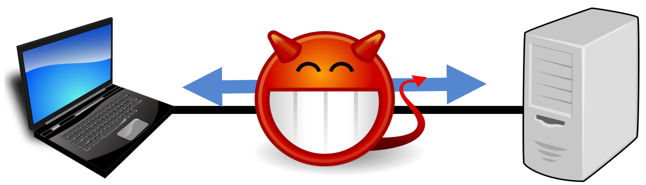
From Abstract to Concrete Strategies

We want to test implementations

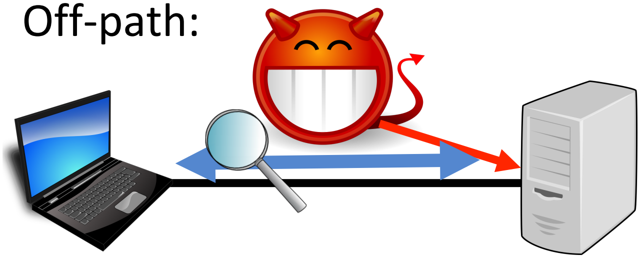
- Limited to packet manipulation and injection to cause abstract strategies
- Consider each abstract strategy separately
- Map each transition to a set of basic malicious actions
  - Actions chosen to cause transition
  - Based on attacker capabilities

Attacker Types:

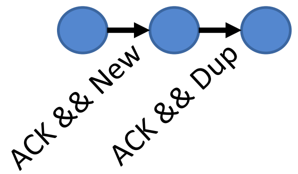
On-path:



Off-path:



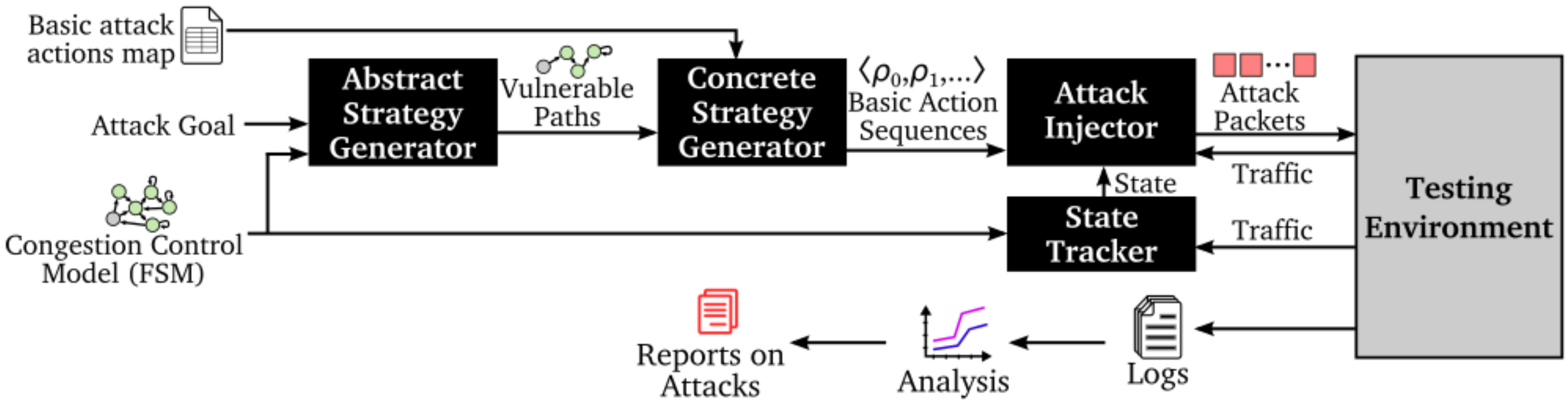
Abstract Strategy



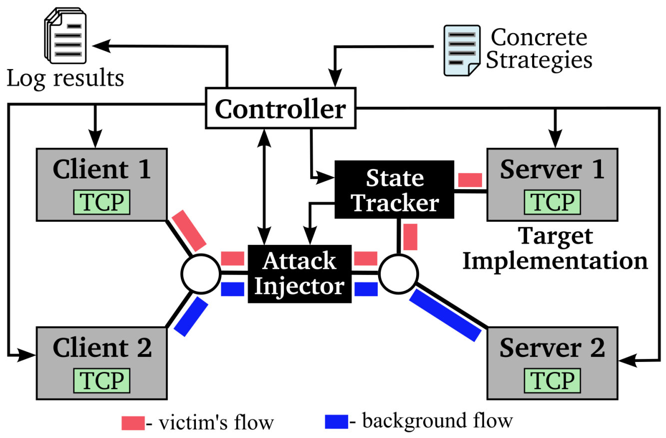
State 1	State 2
Inject Dup Ack Inject Pre Ack Inject Offset Ack	Duplicate Ack Limit Ack Pre Ack

State1:InjectDupAck,State2:DuplicateAck
State1:InjectPreAck,State2:LimitAck
State1:InjectOffsetAck,State2:PreAck
State1:InjectDupAck,State2:DuplicateAcl
…

TCPwn Design



- Test strategies creating using model-based testing and our abstract and concrete strategy generators
- Testing done with virtual machines running real implementations in a dumbbell testbed network
- Attack Injector applies malicious actions
- Performance of target TCP connection identifies attacks



Evaluation

We tested five TCP implementations:

Implementation	Date	Congestion Control
Ubuntu 16.10 (Linux 4.8)	2016	CUBIC+SACK+FRTO+ER+PRR+TLP
Ubuntu 14.04 (Linux 3.13)	2014	CUBIC+SACK+FRTO+ER+PRR+TLP
Ubuntu 11.10 (Linux 3.0)	2011	CUBIC+SACK+FRTO
Debian 2 (Linux 2.0)	1998	New Reno
Windows 8.1	2014	Compound TCP + SACK

Found 11 classes of attacks, 8 of them unknown

Results Summary

Attack Class	Attacker	Impact	OS	New?
Optimistic Ack	On-path	Increased Throughput	ALL	No
On-path Repeated Slow Start	On-path	Increased Throughput	Ubuntu 11.10, Ubuntu 16.10	Yes
Amplified Bursts	On-path	Increased Throughput	Ubuntu 11.10	Yes
Desync Attack	Off-path	Connection Stall	ALL	No
Ack Storm Attack	Off-path	Connection Stall	Debian 2, Windows 8.1	No
Ack Lost Data	Off-path	Connection Stall	ALL	Yes
Slow Injected Acks	Off-path	Decreased Throughput	Ubuntu 11.10	Yes
Sawtooth Ack	Off-path	Decreased Throughput	Ubuntu 11.10, Ubuntu 14.04, Ubuntu 16.10, Windows 8.1	Yes
Dup Ack Injection	Off-path	Decreased Throughput	Debian 2, Windows 8.1	Yes
Ack Amplification	Off-path	Increased Throughput	Ubuntu 11.10, Ubuntu 14.04, Ubuntu 16.10, Windows 8.1	Yes
Off-path Repeated Slow Start	Off-path	Increased Throughput	Ubuntu 11.10	Yes

Summary

- We developed a new, model-guided technique to search for possible attacks on TCP congestion control. This technique uses the congestion control state machine to generate abstract strategies which are then converted into concrete strategies made up of message-based actions
- We implemented this technique in TCPwn, which is able to find attacks on real, unmodified implementations of TCP congestion control
- We tested 5 TCP implementations and found 11 classes of attacks, 8 of which were previously unknown

Check out the code!
https://github.com/samueljero/TCPwn

Questions?

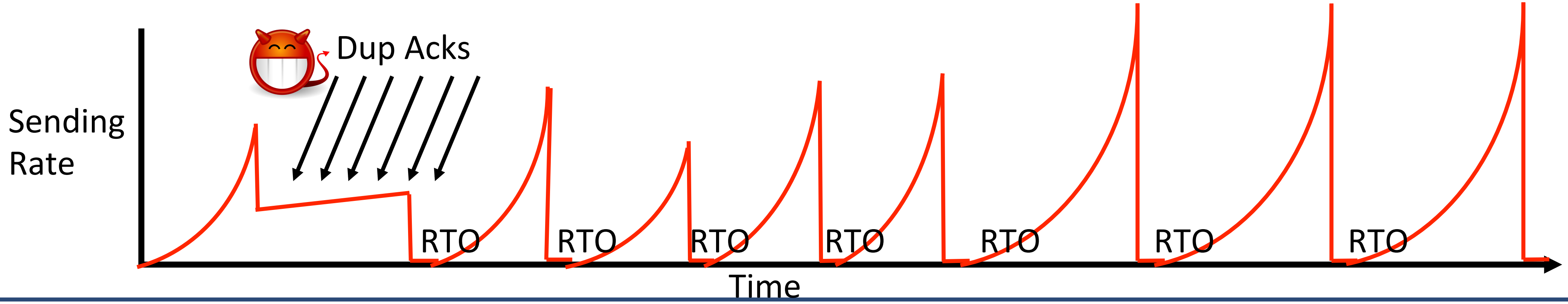
Samuel Jero
sjero@sjero.net

Check out the code!
https://github.com/samueljero/TCPwn

Off-path Repeated Slow Start Attack

- Linux includes adjustable dup ack threshold
  - Based on observed duplicate and reordered packets
- Attacker injects many duplicate acks
  - Increasing dup ack threshold
- Timeout occurs before dup ack loss detection
- Enter Exponential Backoff and then Slow Start
  - Instead of Fast Recovery
- Short 200ms timeout causes throughput to be >= normal
- Competing connections also suffer badly due to repeated losses

Off-path attacker can increase throughput for Linux senders

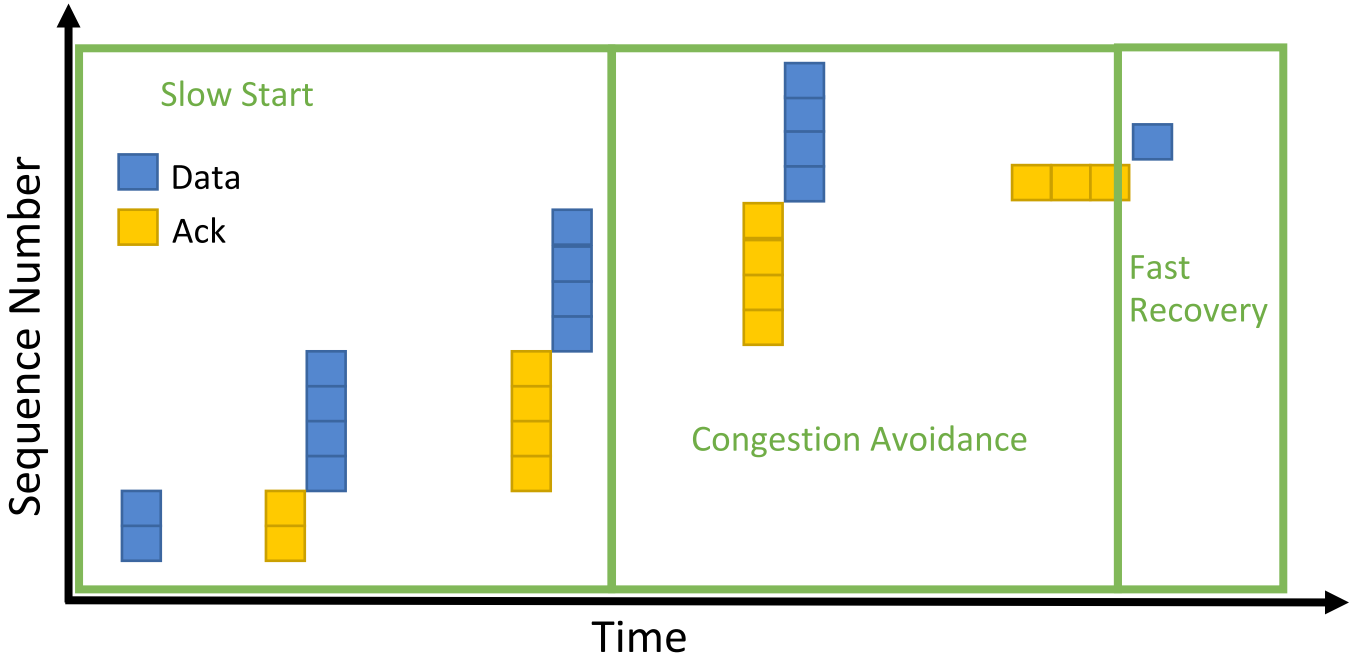


Inferring Congestion Control State

To apply concrete strategies to an implementation, we need to know the sender's congestion control state

- Approximate congestion control state and assume normal application behavior
- Take a small timeslice and observe the bytes sent and acknowledged by the implementation

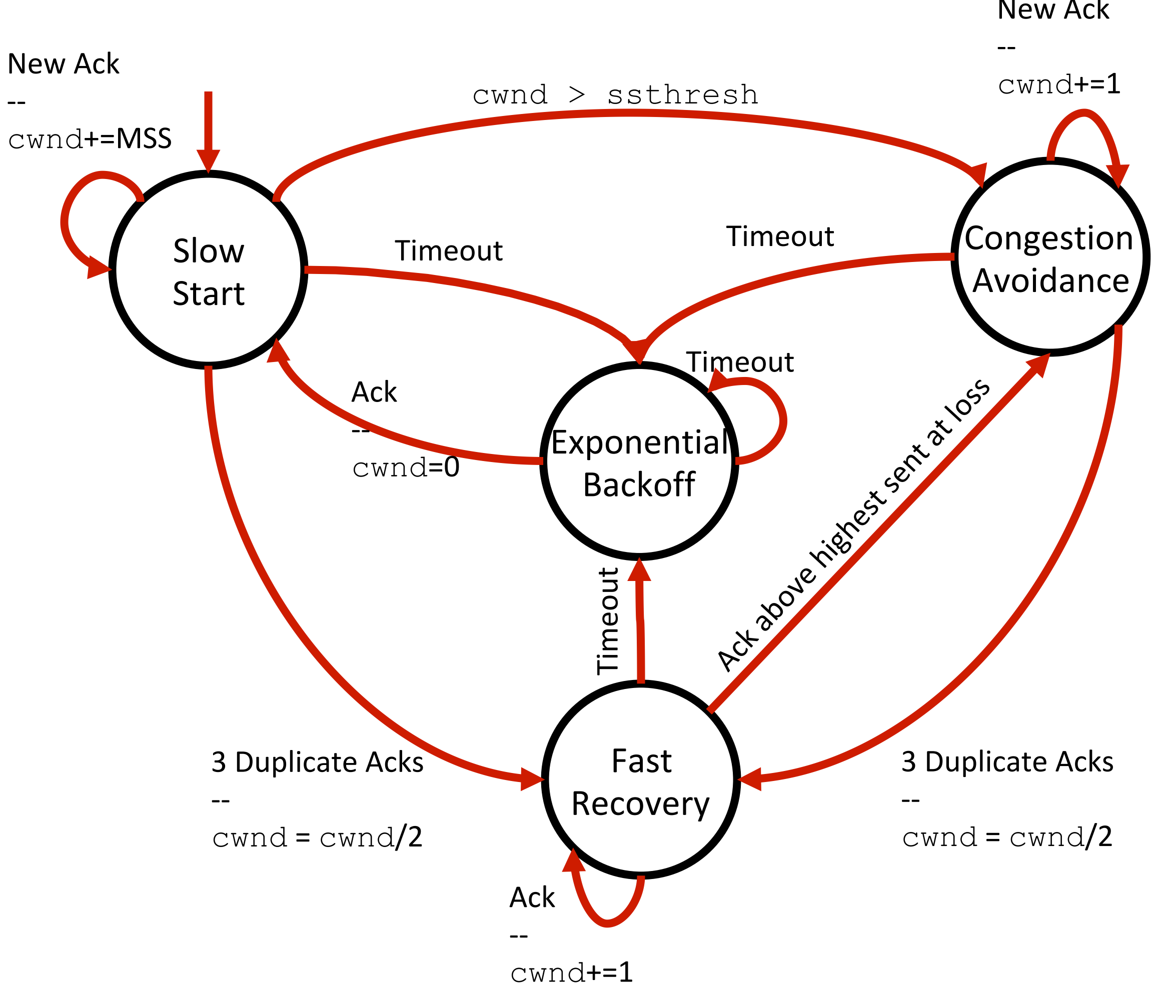
Bytes Sent*2 ≈ Bytes Acked
    State: Slow Start
Bytes Sent ≈ Bytes Acked
    State: Congestion Avoidance
Retransmitted packets or ACK pkts > Data pkts
    State: Fast Recovery
ACK pkts == 0 and Data pkts > 0
    State: Exponential Backoff



More on Congestion Control

- Model as a state machine
  - Input: Acks and Timers
  - Output: Congestion Window (`cwnd`) =sending rate
- Four states:
  - *Slow Start*—Quickly find available bandwidth
  - *Congestion Avoidance*—Steady state sending with occasional probe for more bandwidth
  - *Fast Recovery*—React to loss by slowing down
  - *Exponential Backoff*—Timeout, slow down

New Reno Congestion Control
State Machine



Limitations

- Use of New Reno as model
  - Model limited by ability to infer sender's state from network traffic
  - More precise inference or instrumentation would enable more precise modeling
  - We trade off precision for ease of application to a wide range of implementations
- What about CUBIC, SACK, etc?
  - Most algorithms/optimizations are similar to New Reno
    - This includes: SACK, CUBIC, TLP, PRR
  - We actually tested implementations of these and found attacks
- What about algorithms not similar to New Reno?
  - For example: BBR, TFRC, Vegas
  - Model-based testing still readily generates abstract strategies
  - Need a method to infer sender's congestion control state