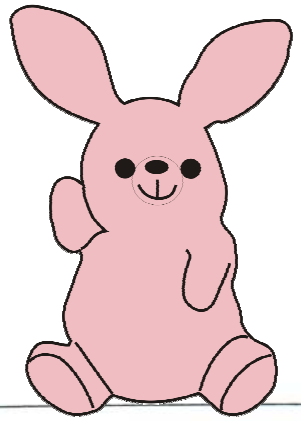


PortBunny
A kernel-based port-scanner



Invent & Verify

Copyright © 2007 Recurity Labs GmbH

A Port Scanner? *Yawn*

- Port scanning is fun for most people
  - Needs random scanning
  - Needs 1337 output
  - Needs 23 different scanning types
- Port scanning is work for some people
  - Needs Accuracy
  - Needs Speed
    - Speed → Time → Money
  - Will use dedicated machines



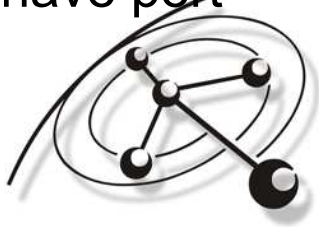
Why not nmap?

- 3 * 255 Hosts in 30 days with nmap
  - I'm actually coming of age
  - Your scanner is not 1337 if it takes 13:37 per host!
  - No, **--disable-waiting-for-things-that-dont-happen** doesn't cut it
- Professionals don't scan hosts that are …

  … powered off

  … disassembled

  … currently being carried around in the office
- Large scale network scanning is application stocktaking, not vulnerability identification
  - Little interest in the one fully filtered host with only port 23420 open
  - Much interest in how many systems in five Class B networks have port 12345 open



And on a more abstract level...

- All discovery methods depend on a single set of information: the list of open, closed and filtered TCP ports
  - OS Fingerprinting
  - Service probing
  - Banner grabbing
- Accordingly, we need this list first, and quickly at that



Our Requirements

- TCP SYN Scanning only, no XMAS trees
- No UDP Scanning
  - UDP scanning is a negative scan method
  - Information value of a UDP scan of a properly firewalled host with UDP services is exactly zero
- Constant access to result data
  - Offloading fingerprinting tasks right when results become available
- Design for embedded use
- Engine design with variable front ends
- Bottom line: Do just one thing, but do it right.

Invent & Verify



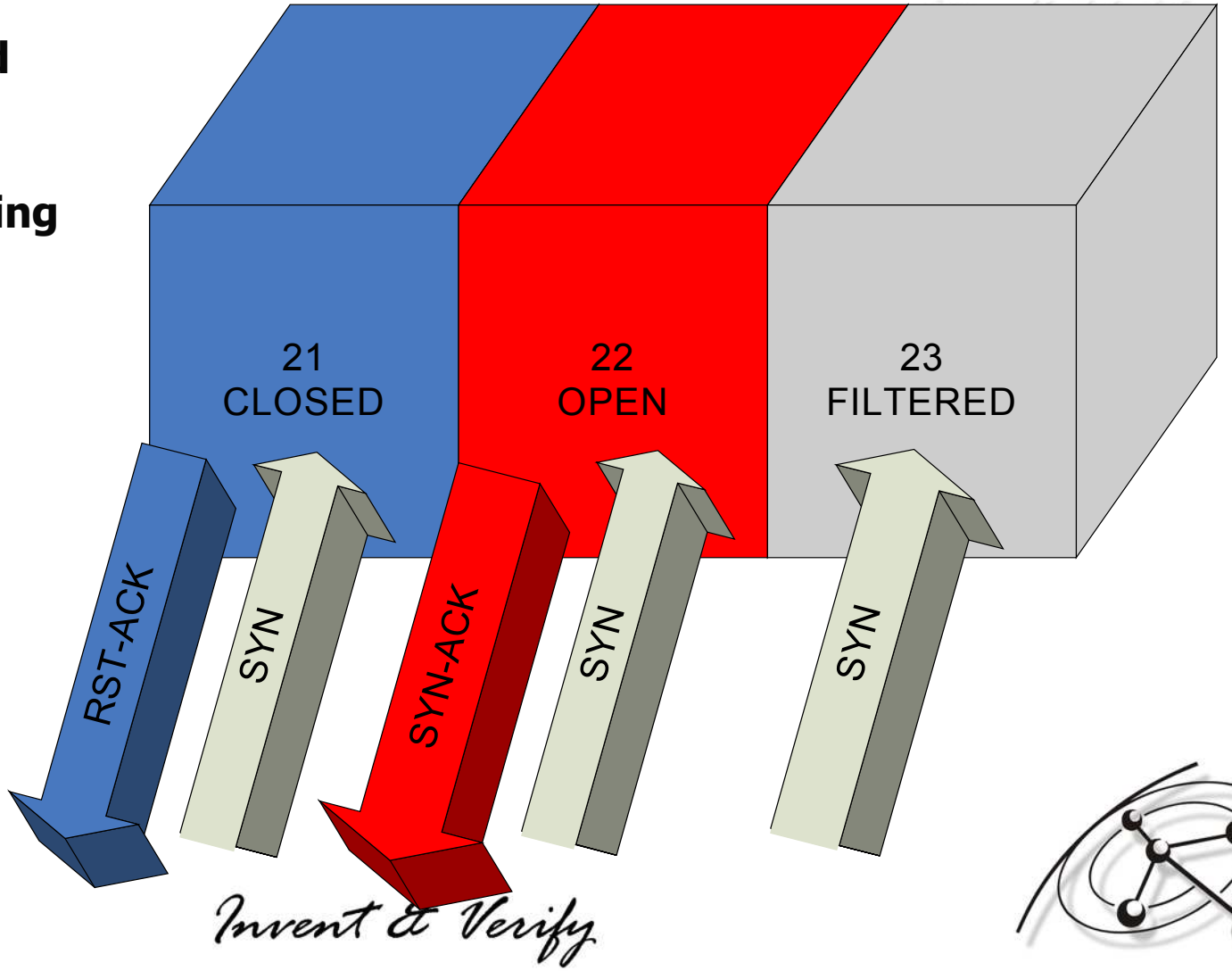
PortBunny

- Portbunny scans faster by sending more
- Portbunny builds a bridge between **TCP congestion control** and port-scanning.
- Portbunny shows that vanilla TCP-SYN port-scans already leave you with lots of room for research.



1. Port-Scanning - Basics

Identify open, closed and filtered ports by sending connection requests and observing responses.



(TCP-SYN or "half-open"-scanning)

Naive port-scanner

```
foreach p in ports_to_scan:
        send_request_to(p)
        get_response()
```

- Won’t quite do it.
- Sending as fast as possible may result in dropped packets or even congestion collapse.
- Open/Closed ports will be falsely reported as being filtered.
- Optimal speed may change over time!



Tell us to slow down, please.

- *Q: Will the network explicitly tell us that we should slow down?*
  A: In general, no.
  - Exception: ICMP source-quenches,
  - Exception: ECN.



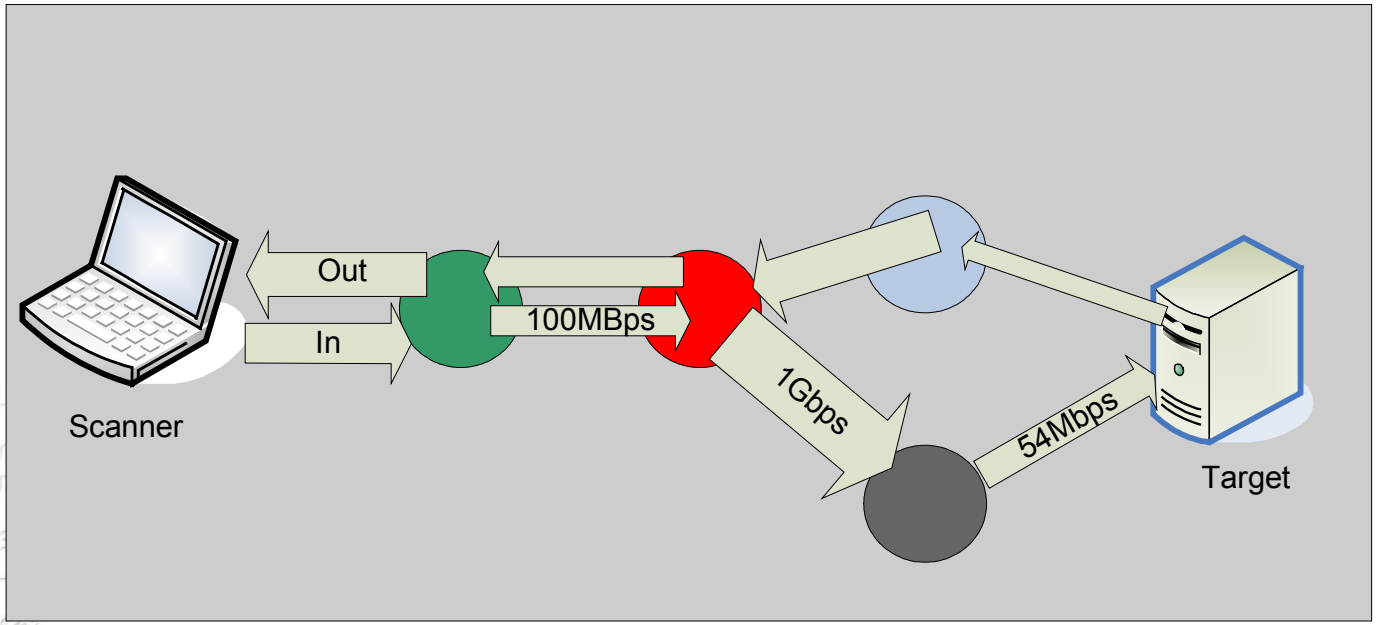
What info do we have?

- If a response is received, we have a round-trip-time.
- Packet-drops can be detected given that we know a certain packet should have provoked an answer.
- That's all.



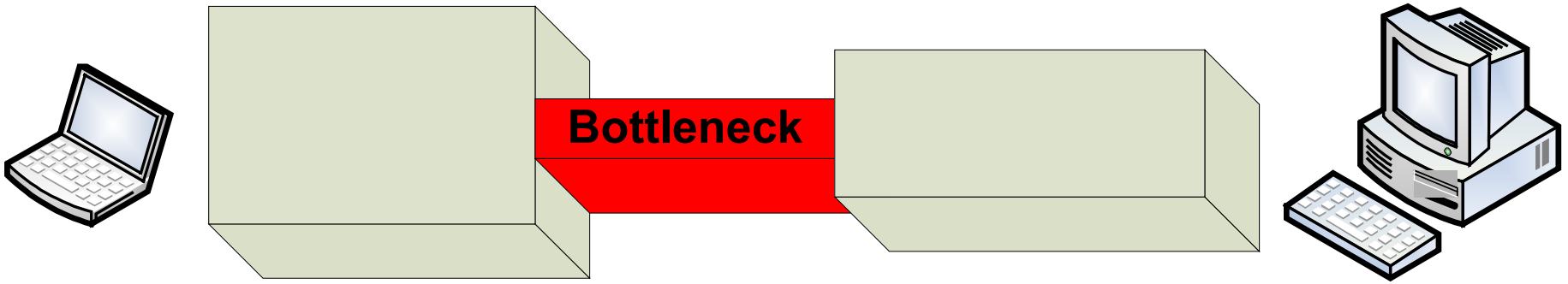
2. A network model

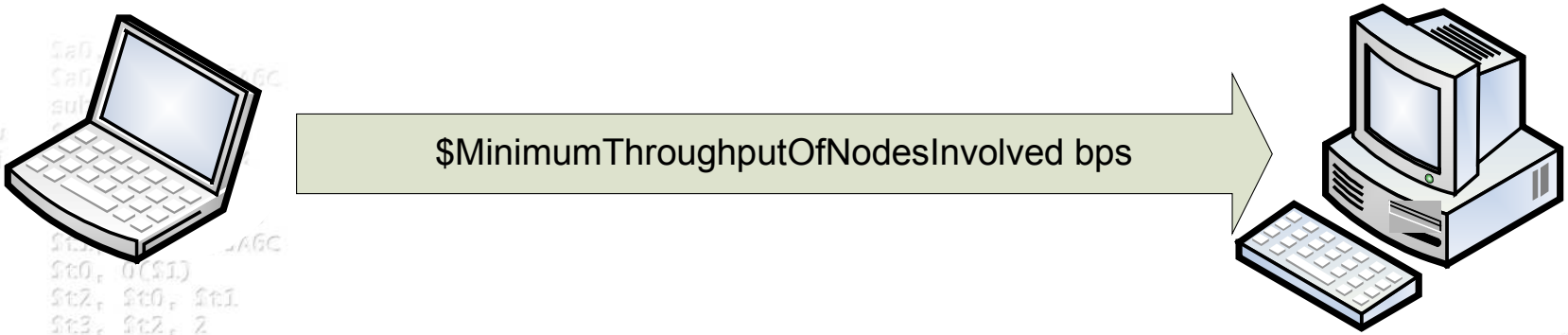
- Edges: Throughput (Delay), Reliability
- Nodes: Queuing-capacity



Simplification

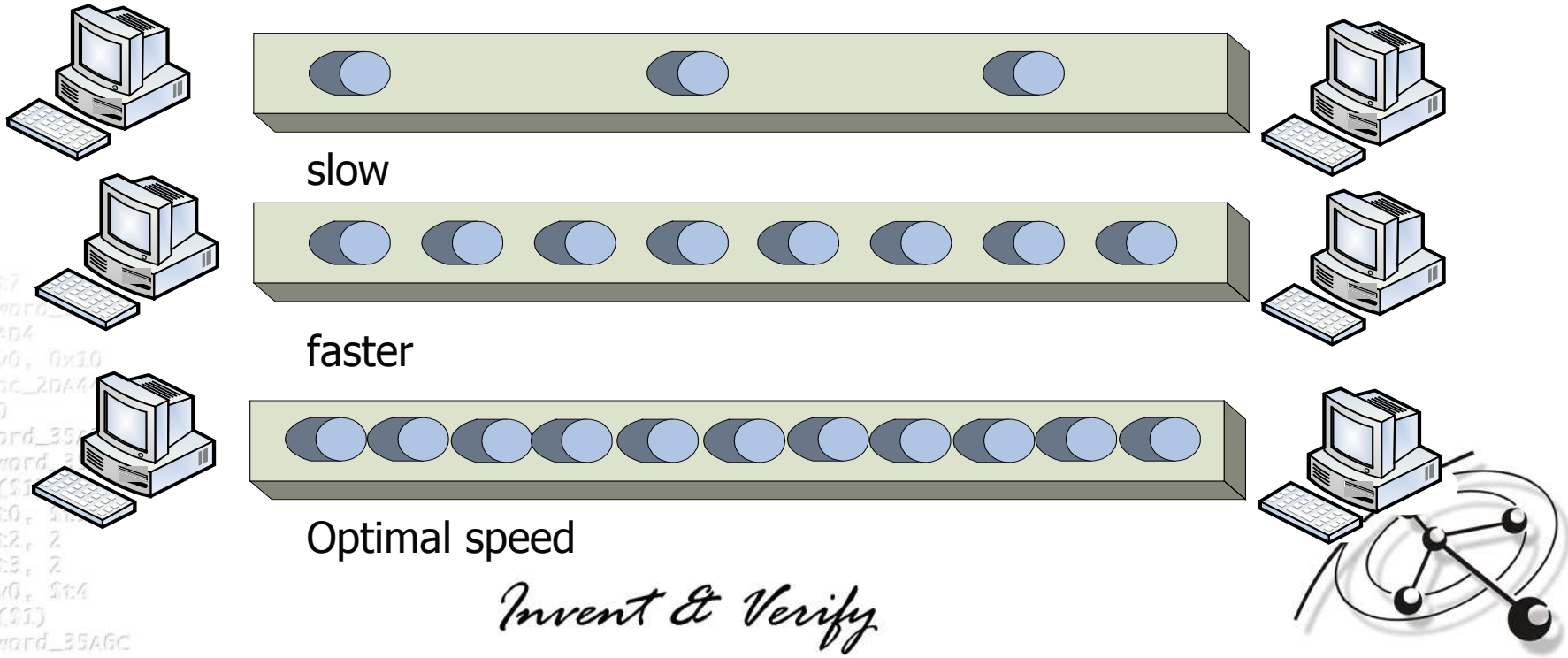
- Model implicitly suggested by the term “bottleneck” and by experience from socket-programming.





Optimal speed

- Speed is the number of packets sent per time-frame.
  Find the optimal delay.



Invent & Verify

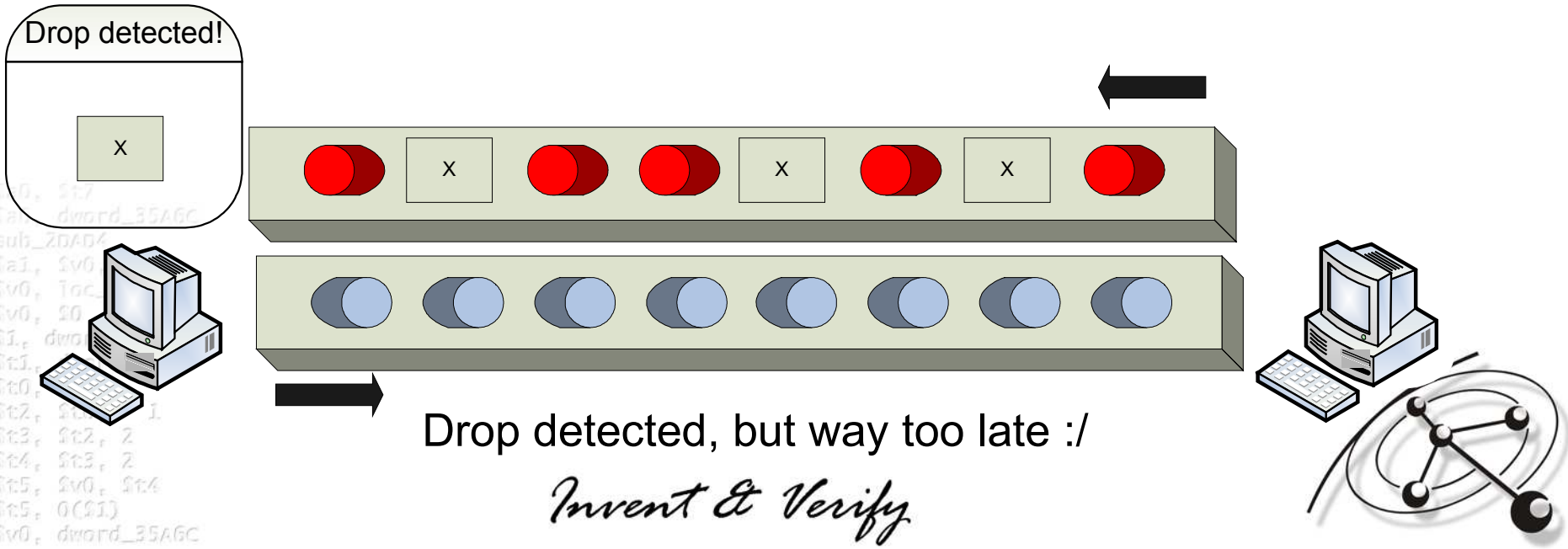
So much for theory...

- … but **finding the optimal delay will fail in practice!**



The round-trip-time problem

- Dropped packets can't be detected before a complete round-trip-time has passed.
- At that time about rtt/delay other packets have already been sent to maintain the "optimal delay".



Queuing capacity

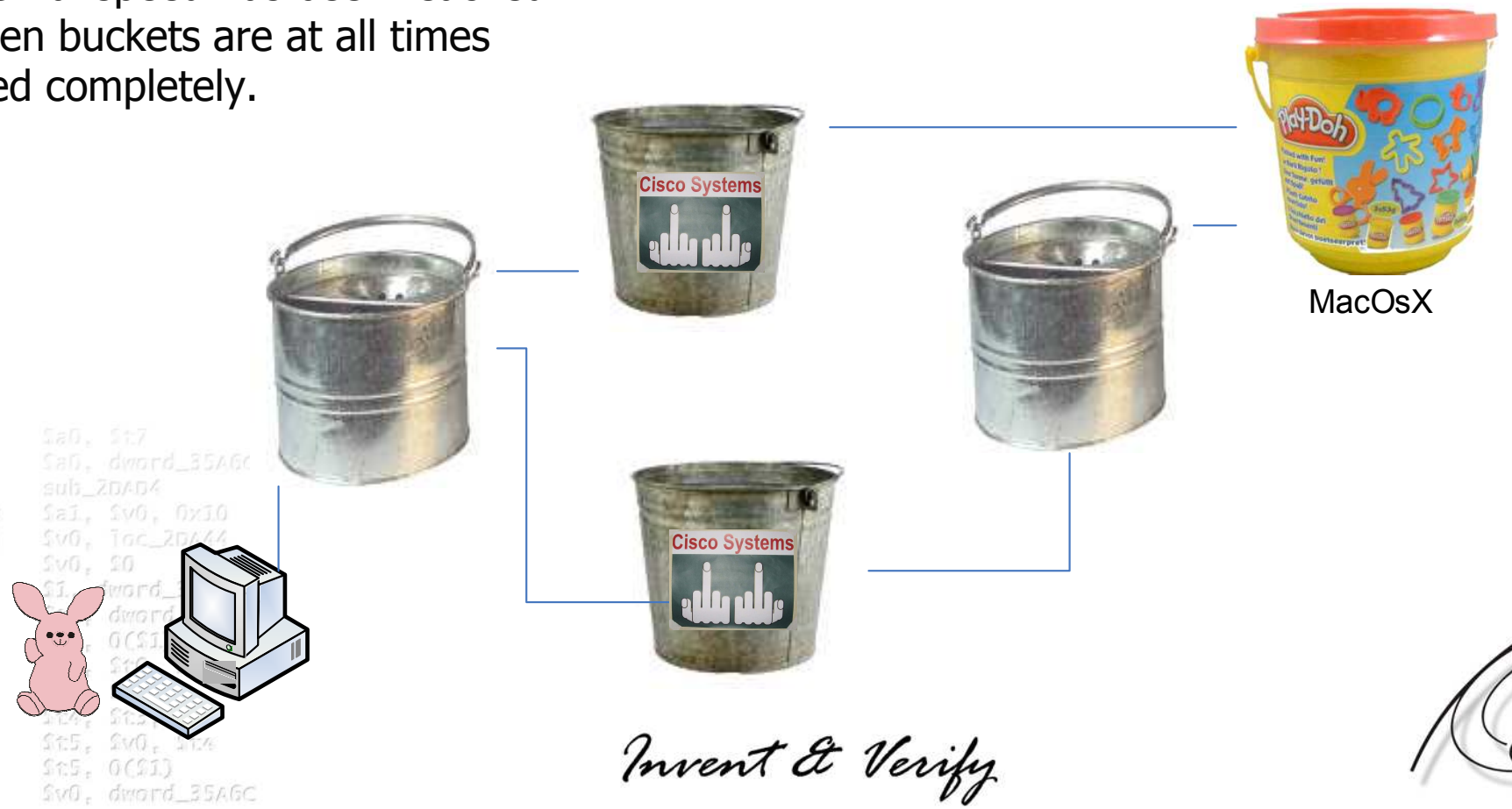
- "You can fire 10 packets at a delay of 0 but that doesn't mean you can do the same with 100 packets." Why?

- The network has limited ability to queue data.
  - This very Important property of the network suggests a new model.



The “bucket-model”

Think of each host as a bucket with a hole at the bottom. The optimal speed has been reached when buckets are at all times filled completely.



Invent & Verify

New model, new question

- Old question:
  "How long should I wait before sending the next packet"

- New question:
  "How much data can be out in the network at once?"



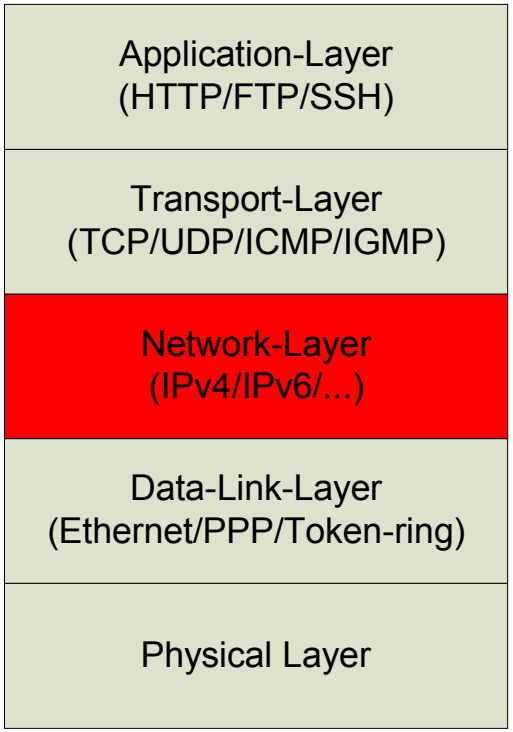
TCP Congestion Control

- TCP congestion control schemes **ask that exact same question!**
- Very active research-field.
- Let's make use of those existing results!



Doesn't that work automatically?

- Why do we have to implement congestion control at all?
- Doesn't TCP provide congestion control to upper layers?
- **No established TCP-connection**
- Control the emission of IP-packets which happen to be TCP-SYNs.





TCP vs. Port-Scanning

TCP	Port-Scanning
Receiver acks packets.	Packets may not produce answers.
Timeouts are error-conditions	Timeouts are not error-conditions
Sequence-numbers are used	No sequence numbers



… in other words:

- The TCP-receiver is cooperative
- A port-scanned host is not cooperative.

- Of course, that doesn't mean we can't force it to be.



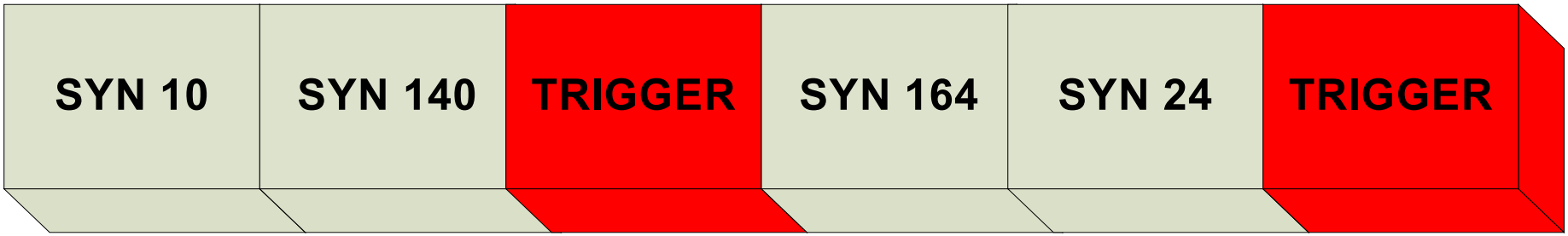
Triggers - forcing cooperation

- Before starting the scan, find one or more packets which trigger a response.
- PortBunny tries the following:
  - ICMP-Echo Requests
  - ICMP Timestamp Requests
  - ICMP Address-Mask Requests
  - TCP-SYN Port 22/80/139/135 …
  - UDP Port …



Inserting triggers into the probe-stream

- Insert these packets into the packet-stream and base your timing-code on the triggers



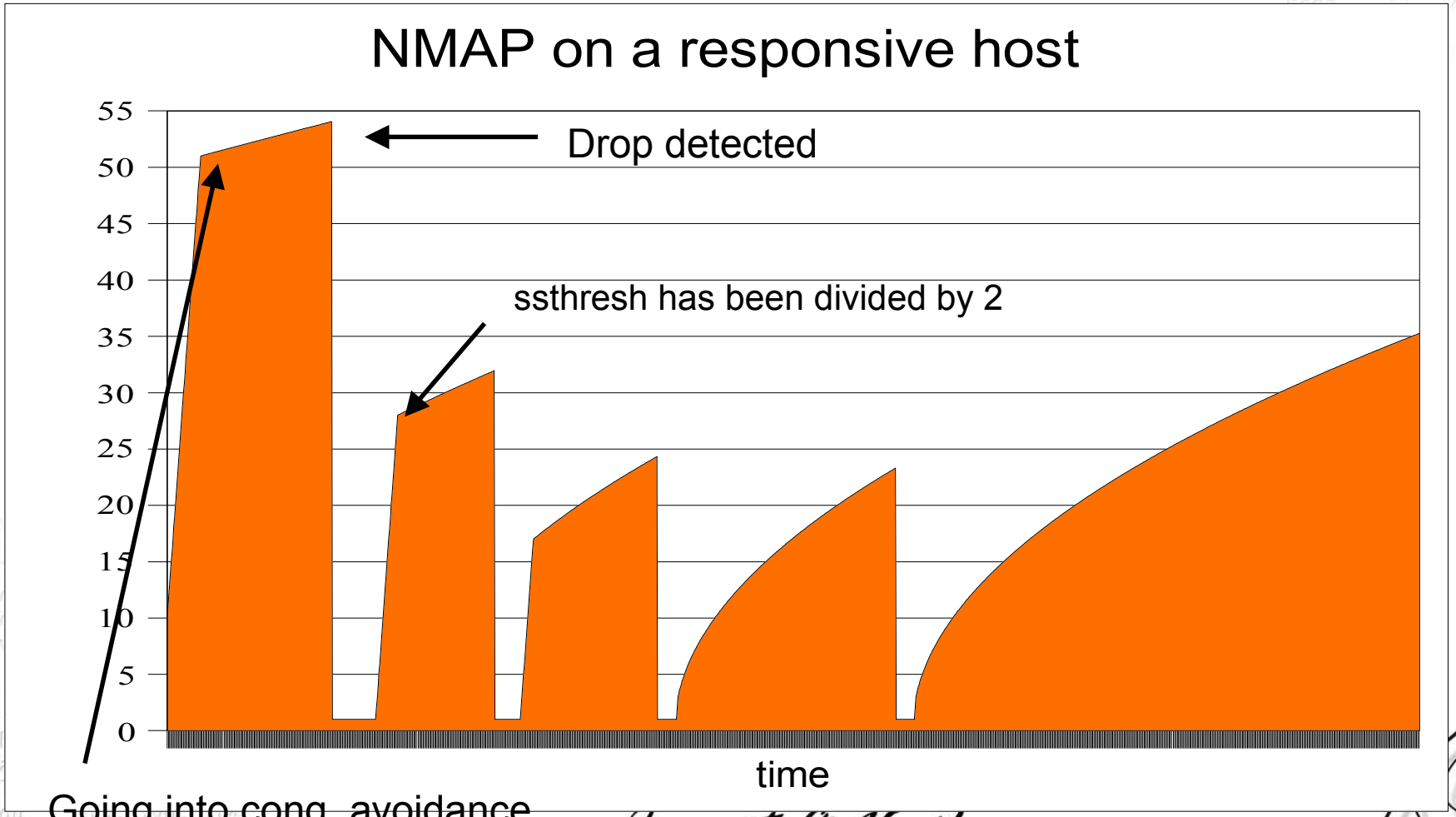


What's that good for?

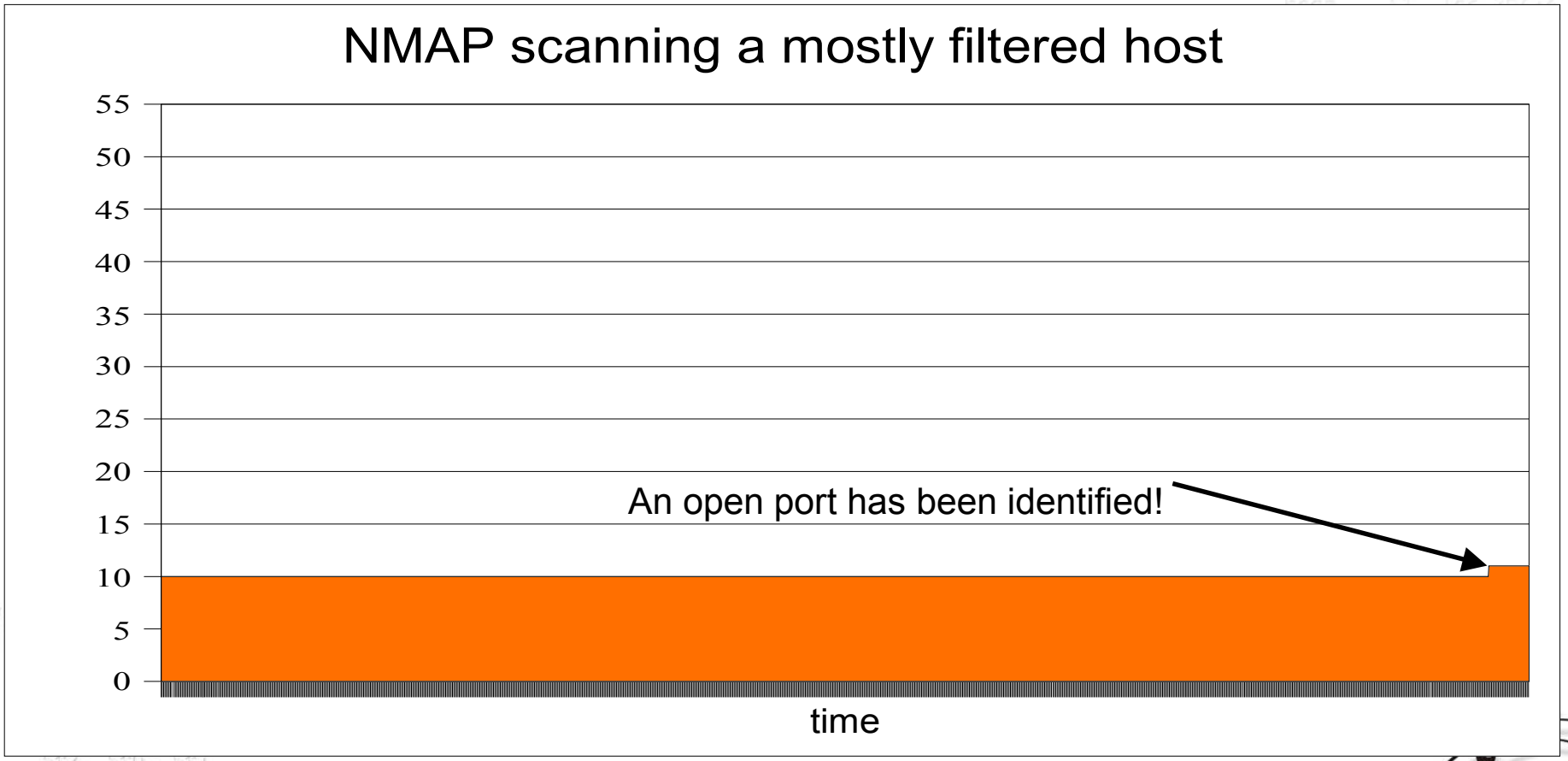
- Trigger-responses now play the same role Acknowledgments play in TCP's congestion control!
- We receive constant information about the network's performance no matter if it is largely filtered or not!
- A timeout is actually a signal of error!



What NMAP Had in Mind



What nmap forgot.



But let's be fair:

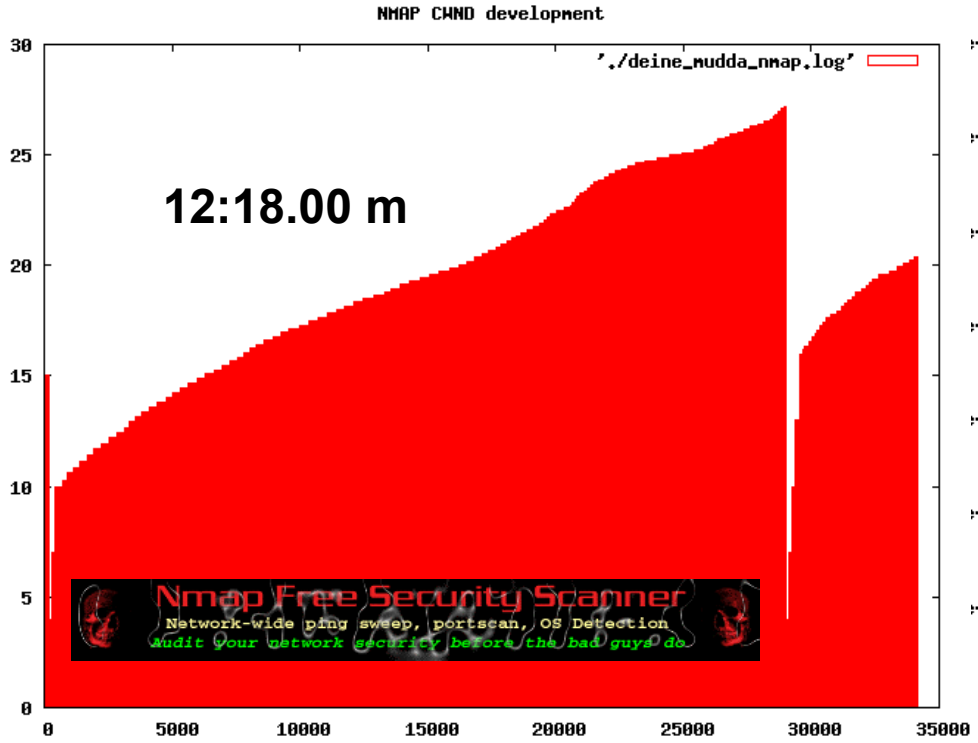
```
/* When a successful ping response comes back, it
counts as this many "normal" responses, because the
fact that pings are neccessary means we aren't
getting much input. */
```

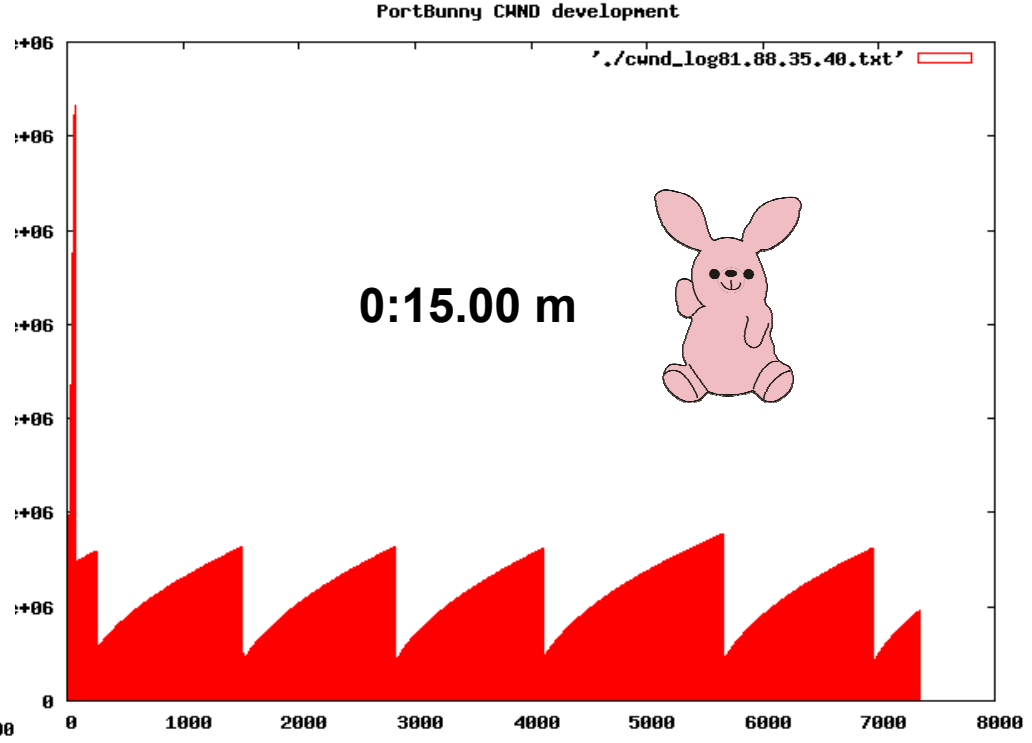
- If a host has not responded in **5 seconds**, a ping is sent.
- A response is then counted as **3** regular responses.
- This is called the "port scan ping"-system



... and then there are filtered hosts ☺

- 65535 ports, mostly filtered, Internet.

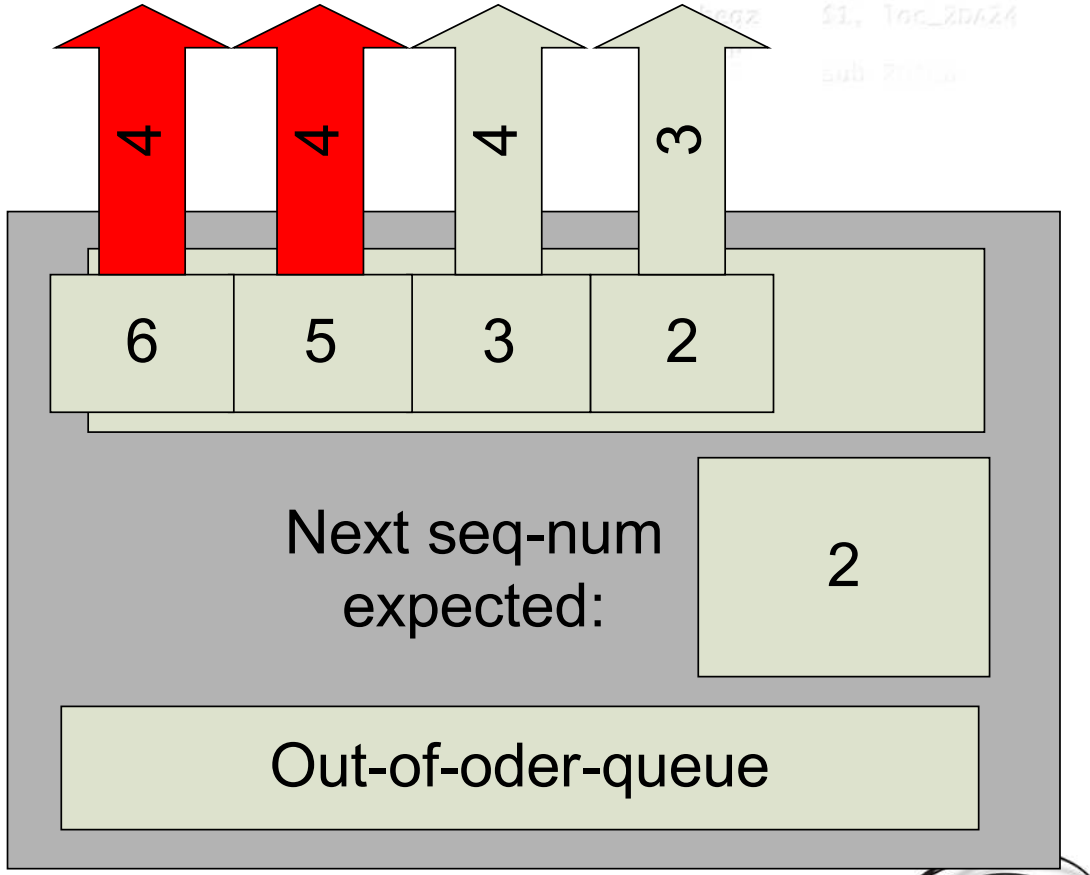




Invent & Verify

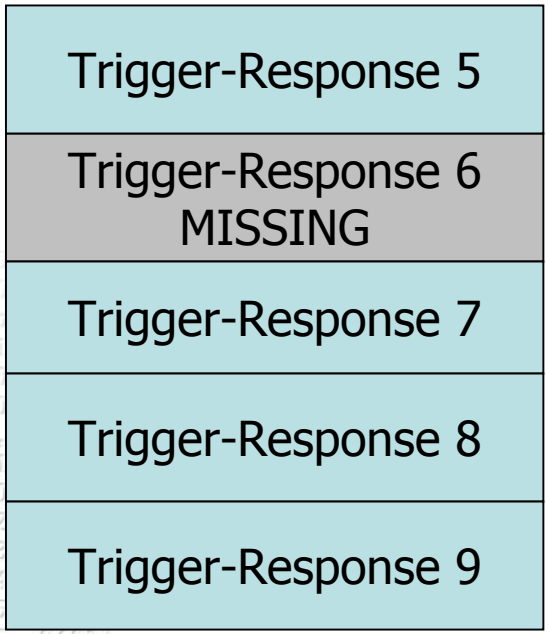
Why mention Sequence-Numbers?

- An Ack is sent by the receiver for each packet
- **Duplicate Acks indicate packet-loss!**
- **Fast-retransmit**



Trigger Sequence-Numbers

- When integrating sequence-numbers into triggers, an algorithm similar to **fast-retransmit** can be implemented:



Example:

- **Responses for 7, 8 and 9 have been received but there's no response for 6.**
- **One can assume that 6 has been dropped even if its timeout-value has not been reached!**

Invent & Verify



Timeout-detection without triggers

```
/* A previous probe must have been lost ... */.
```

- Drops can only be detected after resending
- If a resent probe produces an answer, obviously, the initial probe was dropped.
- Each probe has its own timeout-clock. That doesn't scale well.



Consequence

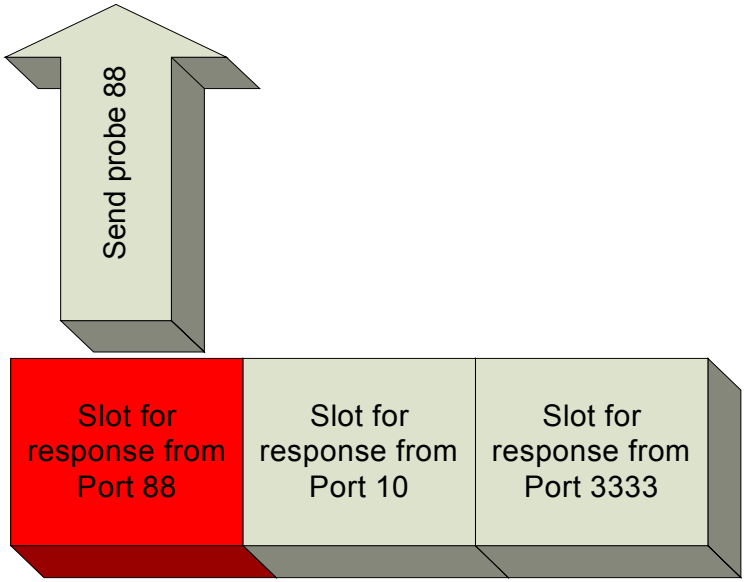
- To stay responsive to drops, probes that may have just dropped must be resent straight away!

- This makes you extremely vulnerable to the “late-responses”-problem

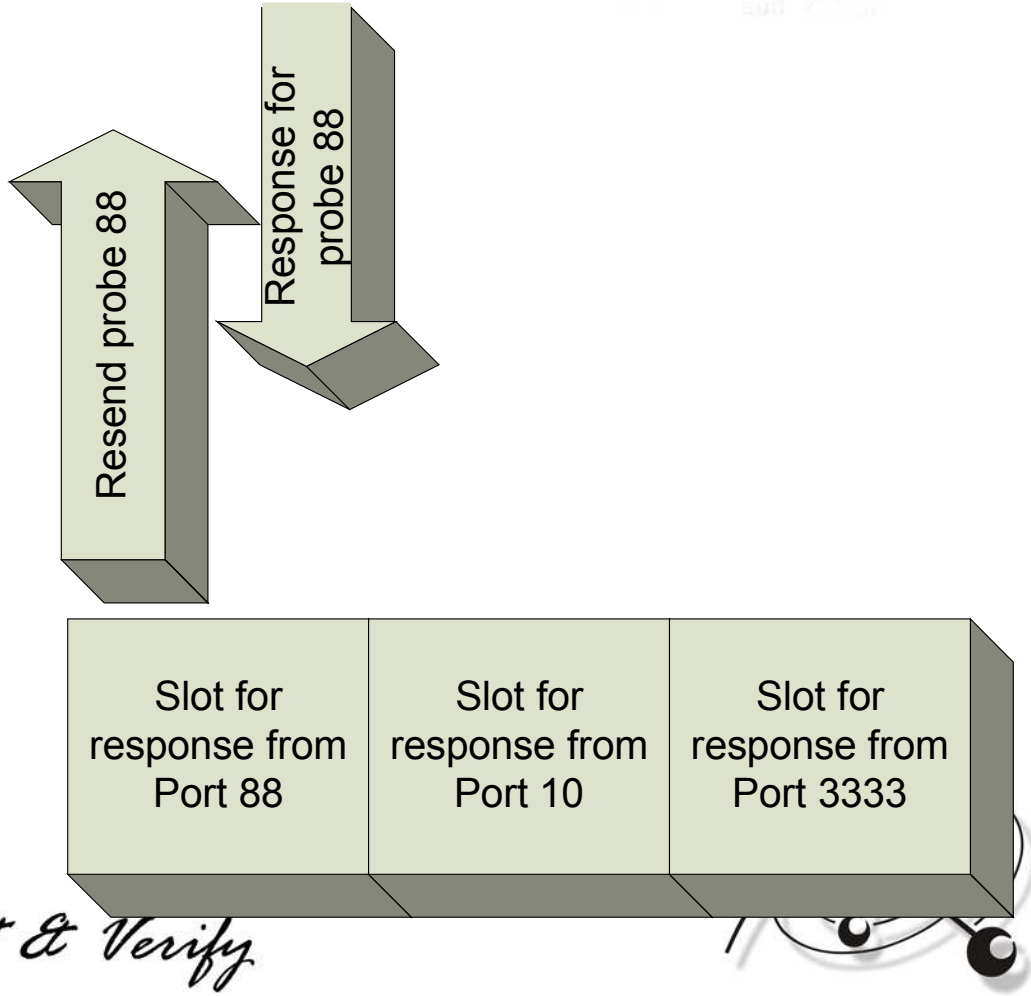


“Late-responses” Problem

(1)



(2)

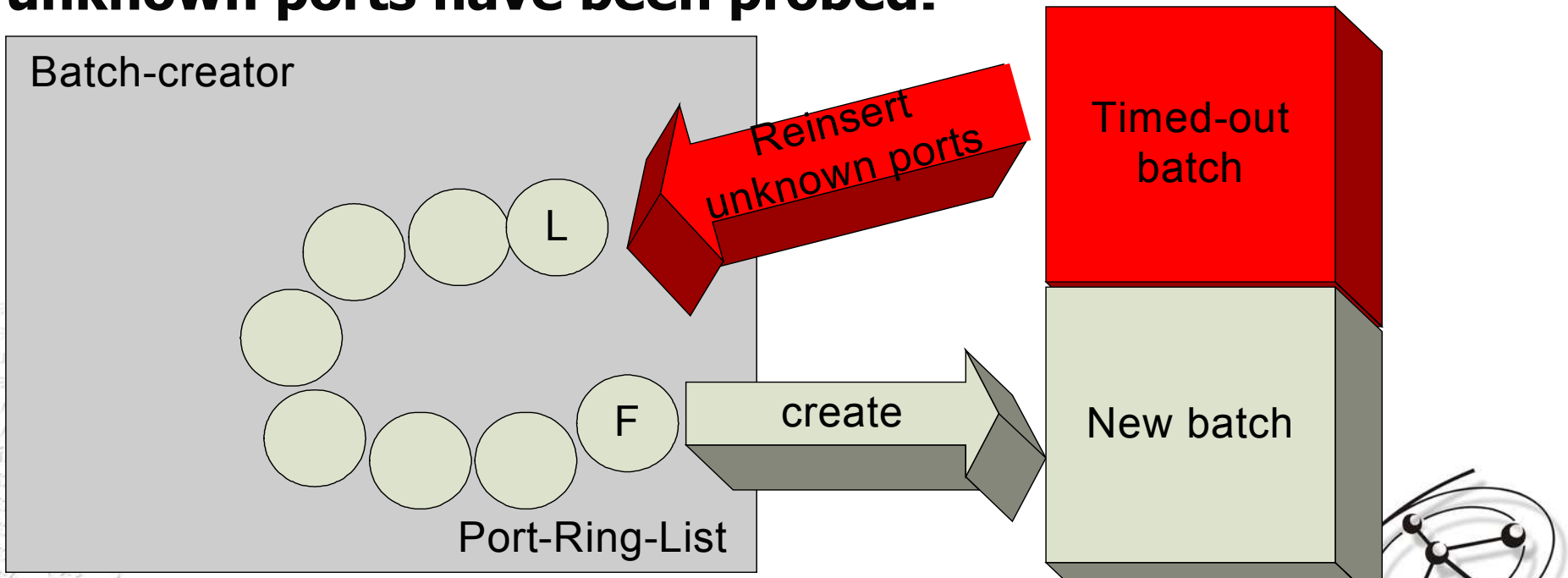


If the approximation of the timeout is too optimistic, responses arrive shortly after the resend has occurred.

➔ **Lots of unnecessary traffic which reduces the scanning-speed.**

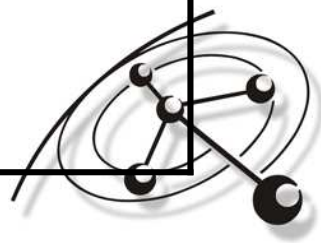
Defeating late-responses (with triggers)

PortBunny does not rely on immediate resends to detect packet-loss!
➔ The probe can be resent after ALL other unknown ports have been probed!



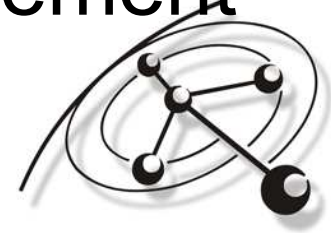
Triggers vs. TCP

TCP	Trigger-based scanning
Receiver acks packets.	Triggers are acknowledged.
Timeouts are error-conditions	Trigger-Timeouts are error-conditions.
Sequence-numbers are used	Sequence-numbers are used for all triggers.



Benefits of trigger-use

- Filtered hosts can be scanned properly
- Packet-drops can be detected much earlier leading to better responsiveness to drops.
- Immediate probe resends are not necessary anymore which helps reduce useless extra traffic.
- Port-Scanning has been ported to the tcp-congestion control domain! We can implement any TCP-congestion-control scheme!



Problems with triggers

- Not all triggers have the same quality:
  - ICMP-triggers and UDP-triggers could be rate-limited while probes aren't.
  - TCP-triggers are the best available triggers.
    - QoS might be a problem, some times
- A host may not respond to any supported trigger.



Fixes

- Try to find TCP-SYN-triggers first and use ICMP and UDP-triggers as a fallback-solution.

- If a TCP-SYN-trigger can be found at scan-time, add it to the list of triggers in use and discard fallback-triggers.

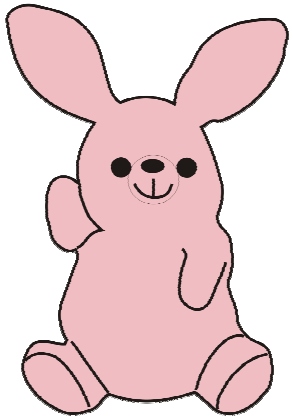


Racing on responsive hosts

- PortBunny sends 10% more data because of the triggers? Can it still compete with the standard tool NMAP on responsive hosts?



VS



Invent & Verify

Numbers and demonstration

- Fresh numbers will be included in the final slide-set, which you can download at http://portbunny.recurity.com



Problems

- The bucket-model is NOT valid for rate-limiting firewalls-configurations!
- We can implement any congestion-control-scheme designed for TCP **but we can't expect the user to know these algorithms** and choose a suited one.



Algorithms implemented:

- Classic TCP-Reno
- TCP-Scalable
  - Slight Reno-improvement for long-distance networks
- TCP-BIC
  - for so called “long-fat pipes”
- TCP-Vegas
  - Experimental pro-active algorithm, which we want to use for WiFi.



We need detection

- The scanner needs to be able to interpret network-conditions and choose a timing-algorithm, which is most suited by itself.
- The scanner is the expert on these issues, not the user.



Trying to detect rate-limits

```
/* If packet drops are particularly bad, enforce a
delay between packet sends (useful for cases such
as UDP scan where responses are frequently rate
limited by dest machines or firewalls) */
```

Translates to: If packet-drops are particularly bad, break the entire timing-concept.

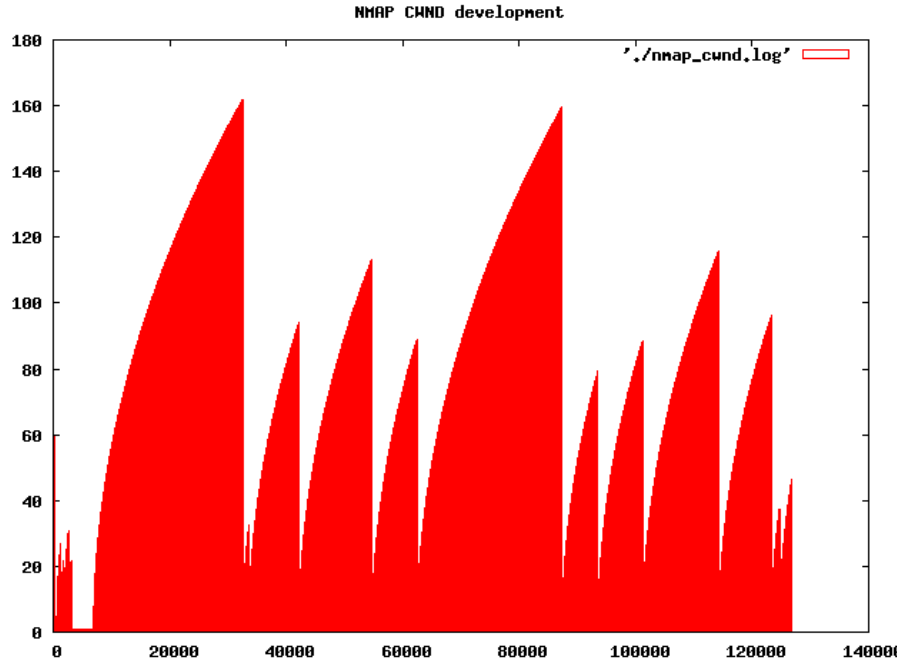
⇒ The CWND will not reflect the number of probes out at once anymore!

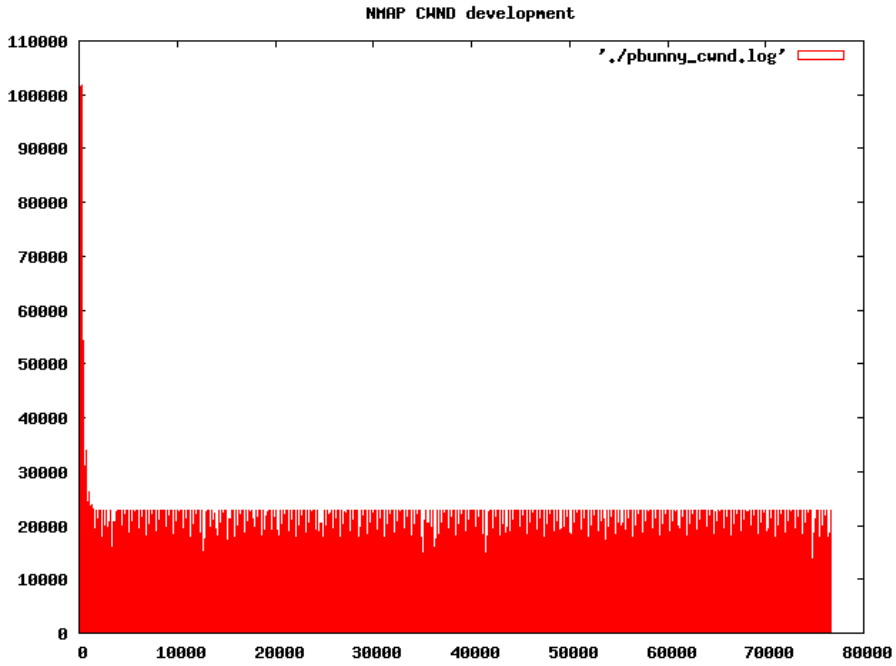
⇒ The self-clocking-property is being ignored!

Invent & Verify



Scanning the IPHONE



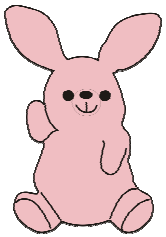




24:41.51 m

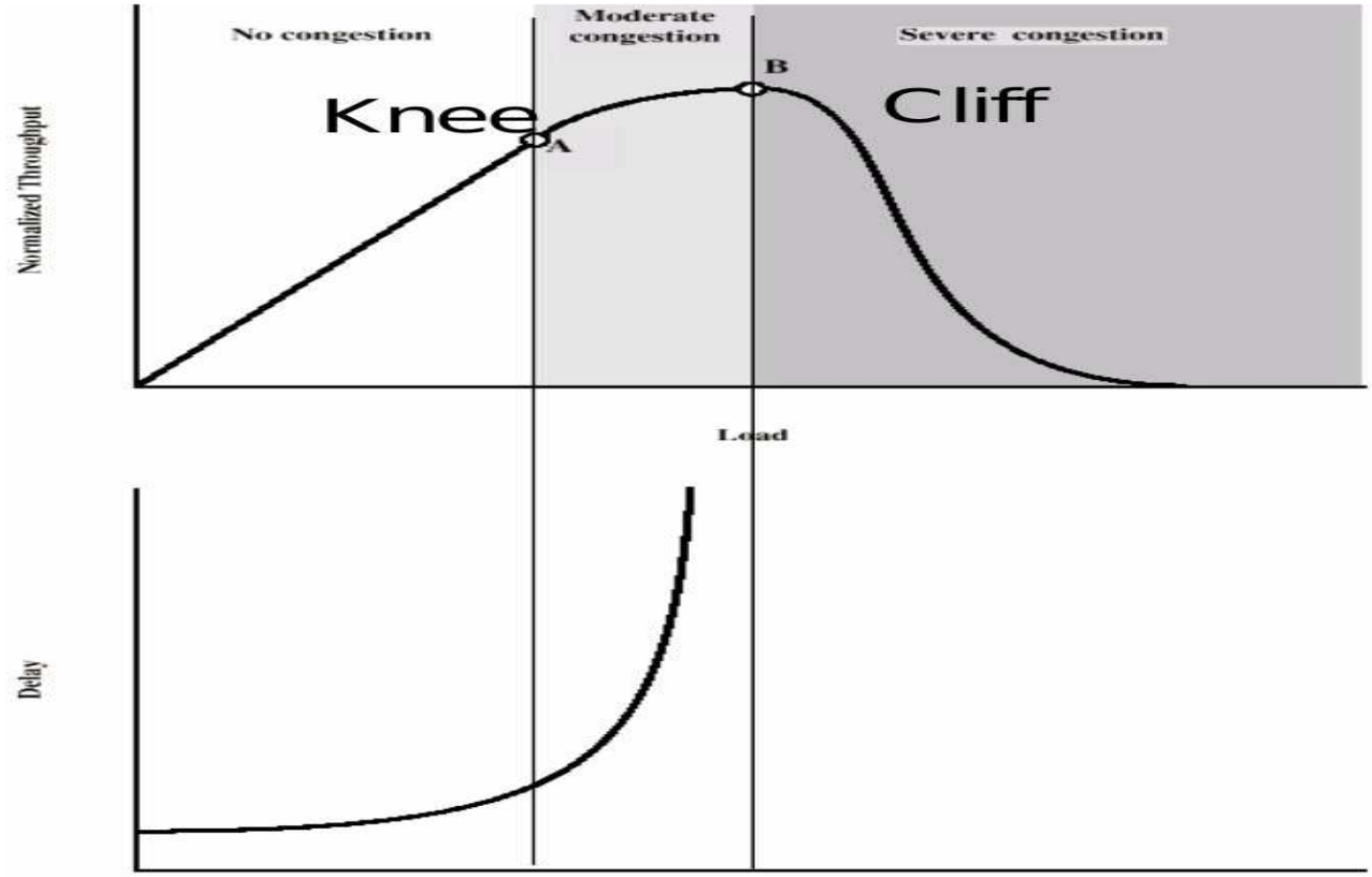
Invent & Verify

7:58.03 m





First approach: RTT-changes



Invent & Verify



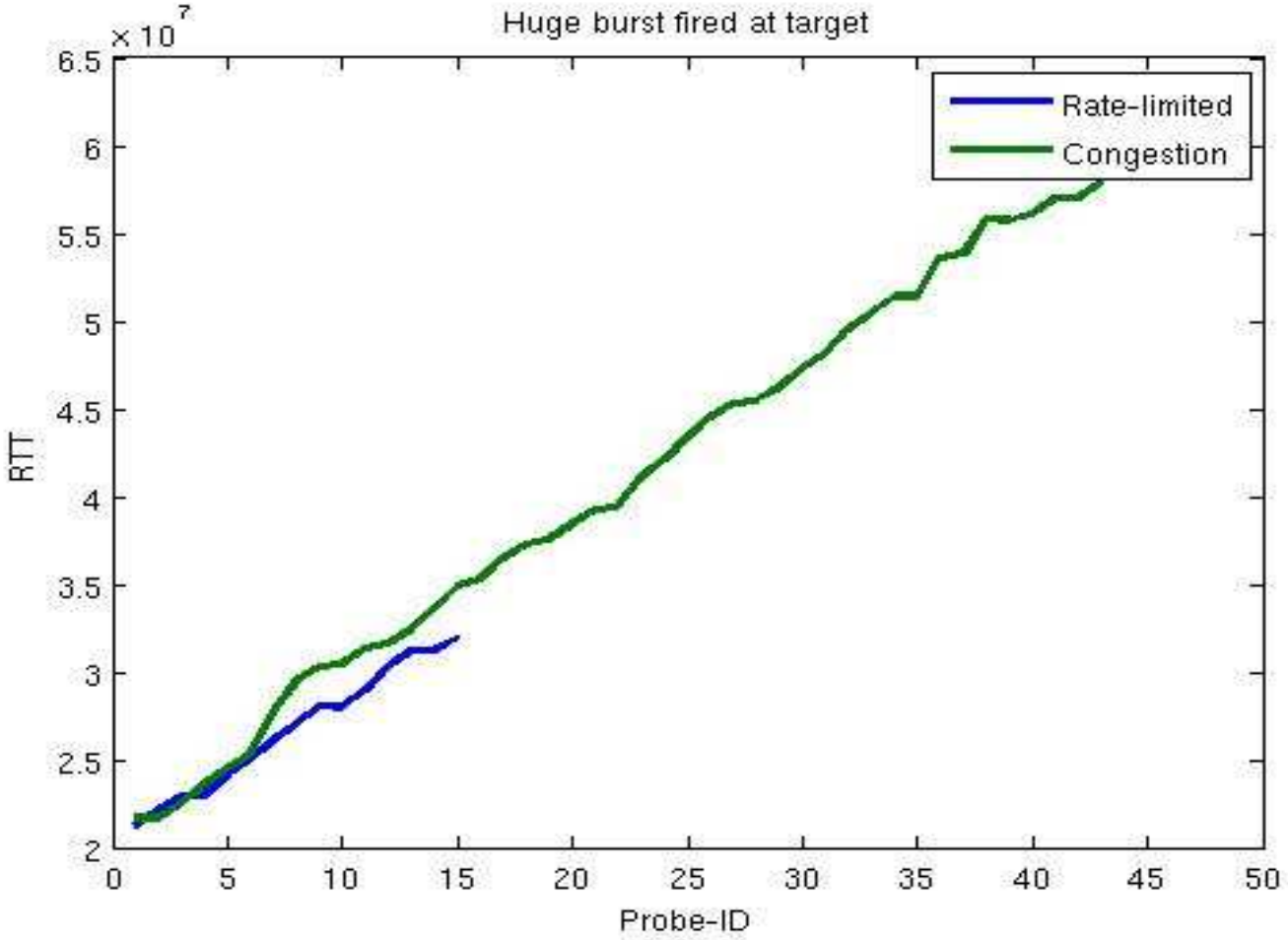
Exponential RTT-increase

- Oh, that's easy! Just send a burst of data, which is big enough and measure RTT.

- If RTT-increase is exponential right before the drop, it's congestion, otherwise it's a rate-limiting firewall.



Nope, doesn't work for bursts



Invent & Verify

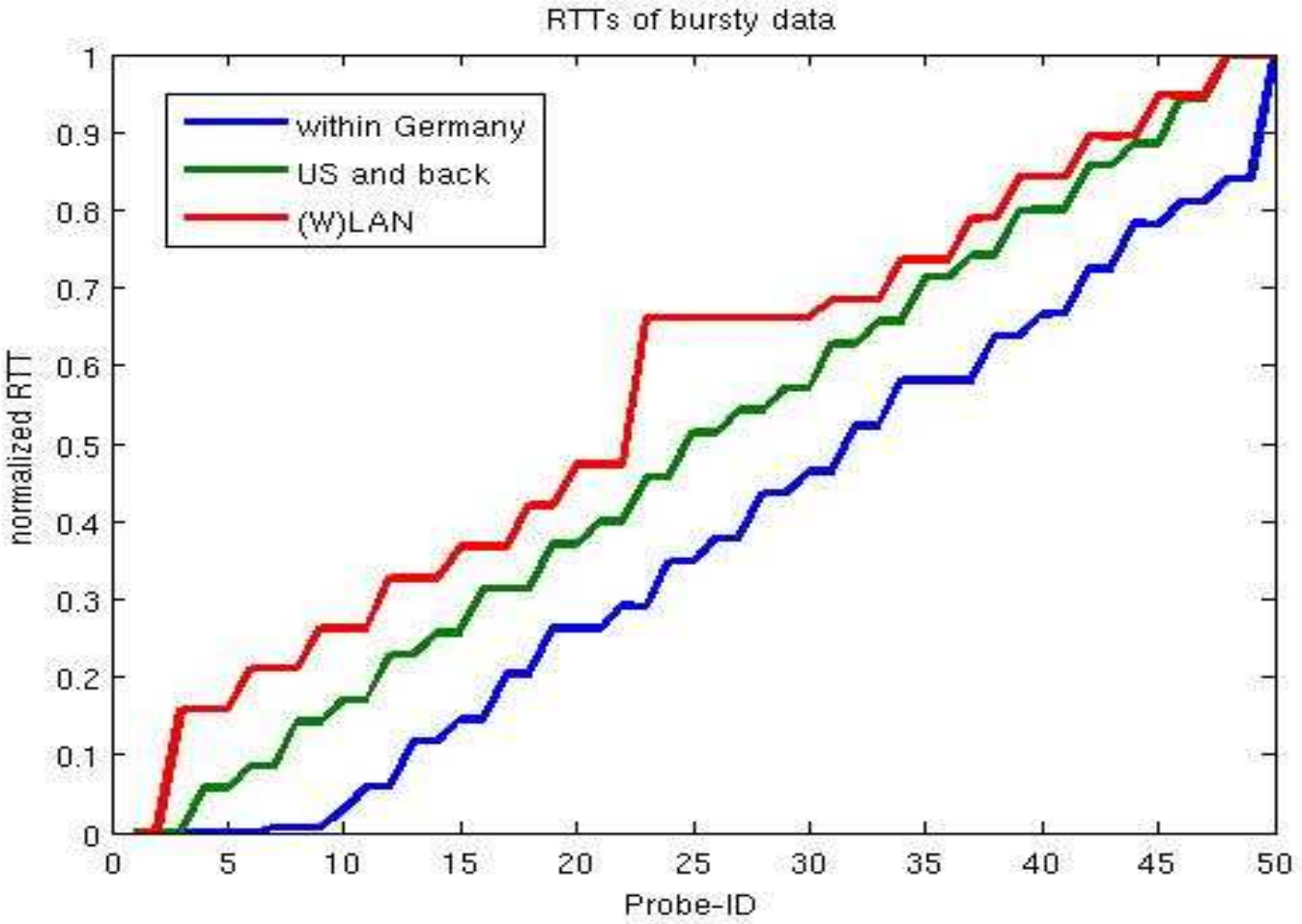


What the hell?

- Q: Where does this linear increase come from?
- A: That's a really pretty illustration of queuing.
- Experiment: Send a burst of 50 triggers to ...
  - (1) the next hop in a 100Mbit-LAN
  - (2) A host within your country over a DSL-Link.
  - (3) A host far, far away, also over a DSL-Link.
- Measure the RTT and normalize it



The “Burst-Response”

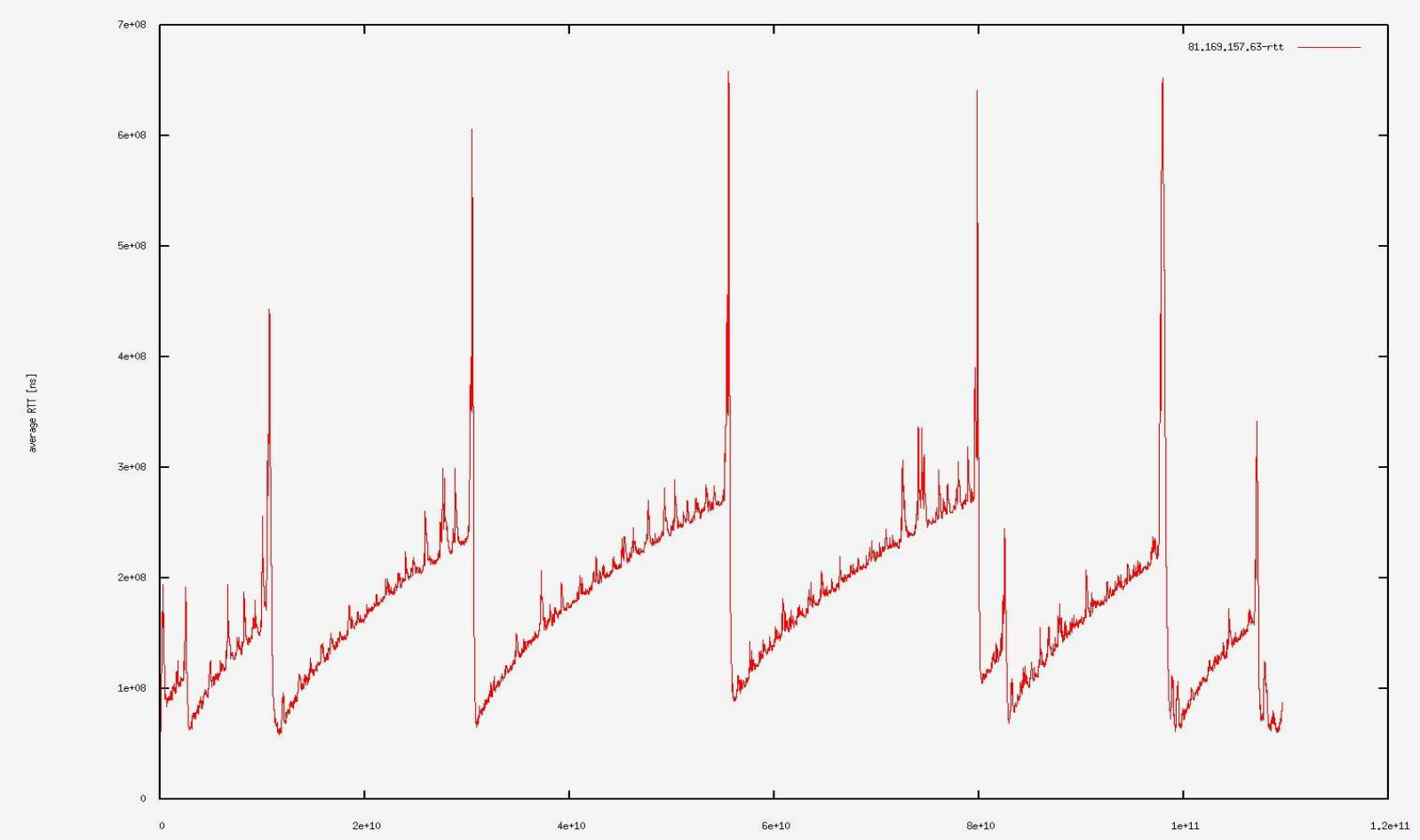


That's all very pretty, but ...

- … now what do we do?
- How will we detect rate-limitation given this unfortunate result?
- Scanning does mean offering data at a certain rate over a longer time-period.
  - Detection during the scan is hard because we're constantly changing the net-load
  - A "one-" or "n-burst" solution would rock…



RTT-Development during scans (Reno)



Decide based on number of drops?

- Of, course, we could just say ”if our timing seems to somehow not work, there's a chance that there's a rate-limitation installed”
- But as we've seen with the IPHONE, that must not be the case -> device could just be a ”dropper”.
- Results of a false decision are disastrous



New idea, limit is artificial!

- Rate-limitation is a totally artificial bottleneck
- It must be possible to reveal this artificial character somehow.
- And this is what we came up with …



Observe: This is a packet...

```
▷ Frame 161 (58 bytes on wire, 58 bytes captured)
▷ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
▷ Internet Protocol, Src: 127.0.0.1 (127.0.0.1), Dst: 127.0.0.1 (127.0.0.1)
▽ Transmission Control Protocol, Src Port: 61373 (61373), Dst Port: tacacs-ds (65), Seq: 0, Len: 0
    Source port: 61373 (61373)
    Destination port: tacacs-ds (65)
    Sequence number: 0    (relative sequence number)
    Header length: 24 bytes
  ▷ Flags: 0x02 (SYN)
    Window size: 0
  ▷ Checksum: 0xaa25 [correct]
  ▽ Options: (4 bytes)
      Maximum segment size: 1460 bytes
```





... and this is, too.

```
Frame 5 (74 bytes on wire, 74 bytes captured)
Ethernet II, Src: Giga-Byt_bf:9a:0c (00:0f:ea:bf:9a:0c), Dst: Cisco-Li_c9:24:57 (00:14:bf:c9:24:57)
Internet Protocol, Src: 192.168.1.208 (192.168.1.208), Dst: 209.85.129.99 (209.85.129.99)
Transmission Control Protocol, Src Port: 41351 (41351), Dst Port: www (80), Seq: 0, Len: 0
    Source port: 41351 (41351)
    Destination port: www (80)
    Sequence number: 0    (relative sequence number)
    Header length: 40 bytes
  ▷ Flags: 0x02 (SYN)
    Window size: 5840
  ▷ Checksum: 0xe209 [correct]
  ▽ Options: (20 bytes)
        Maximum segment size: 1460 bytes
        SACK permitted
        Timestamps: TSval 177528, TSecr 0
        NOP
        Window scale: 6 (multiply by 64)
```



Invent & Verify

Now if the bucket claims...

- … that it can fit only 4 of these:



- … or optionally 4 these:

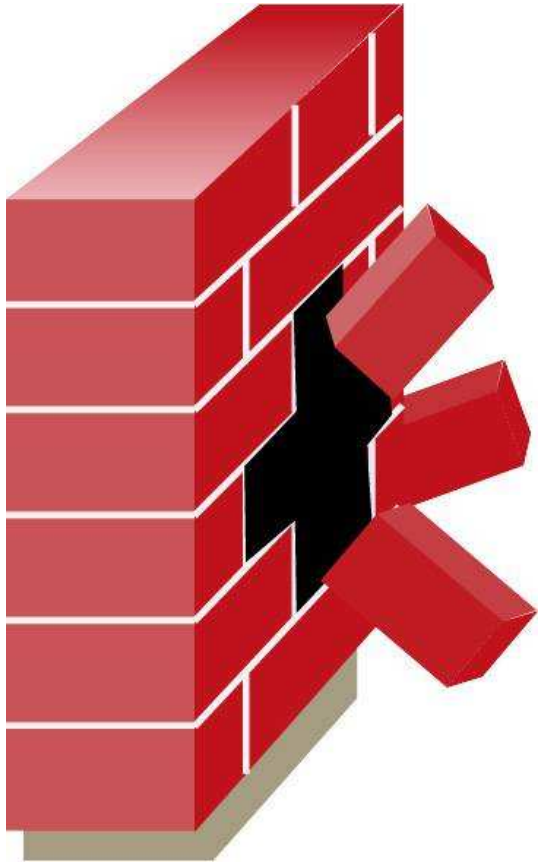




Invent & Verify



… then it's a lousy bucket.

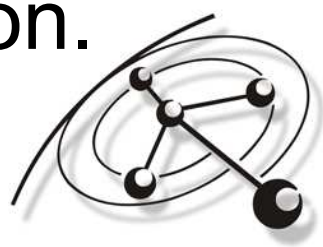


Invent & Verify



Packet-size does not matter!

- Rate-limitation limits **number of packets**, the packet-size does not matter!
- In contrast: congestion is caused by **too much data** in the network
- **Just enlarge the packet** (Add TCP-options)
- If still the same number of packets return, we're obviously dealing with rate-limitation.



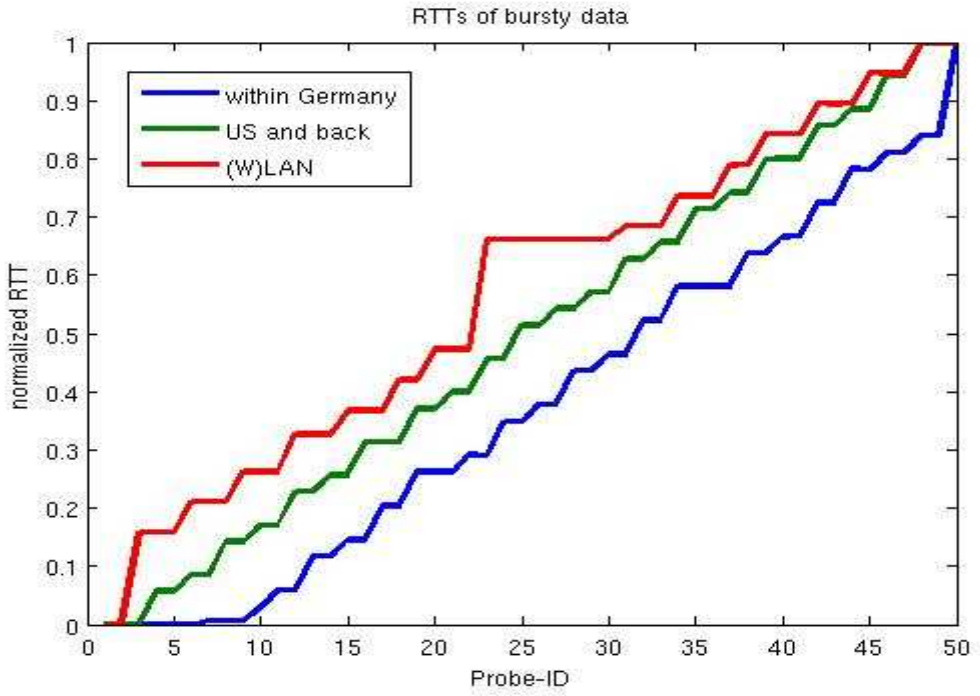
Rate-limit-detector PoC

- [Demonstration]
- Packet-filter is disabled
- Two bursts of pings are sent.
- Packets of the second burst are twice as big as those of the first
- Enable packet-filter
- Send the two bursts again.



… cool, but…

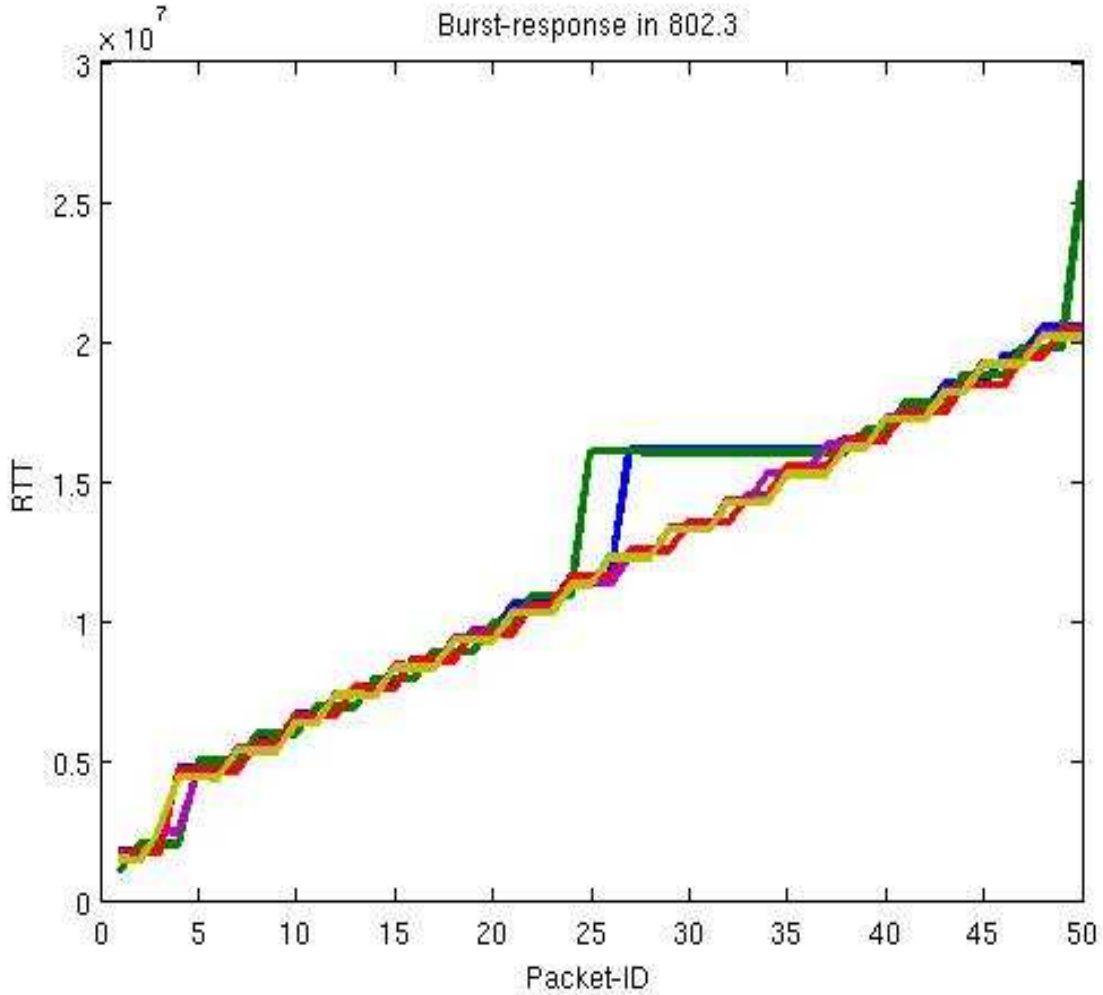
- … this “burst-response” just looks so pretty, is there really nothing we can do with it?



Invent & Verify



Can we detect WiFi-Links?



error [] =

0.0589
0.0737
0.0322
0.0439
0.0421
0.0422

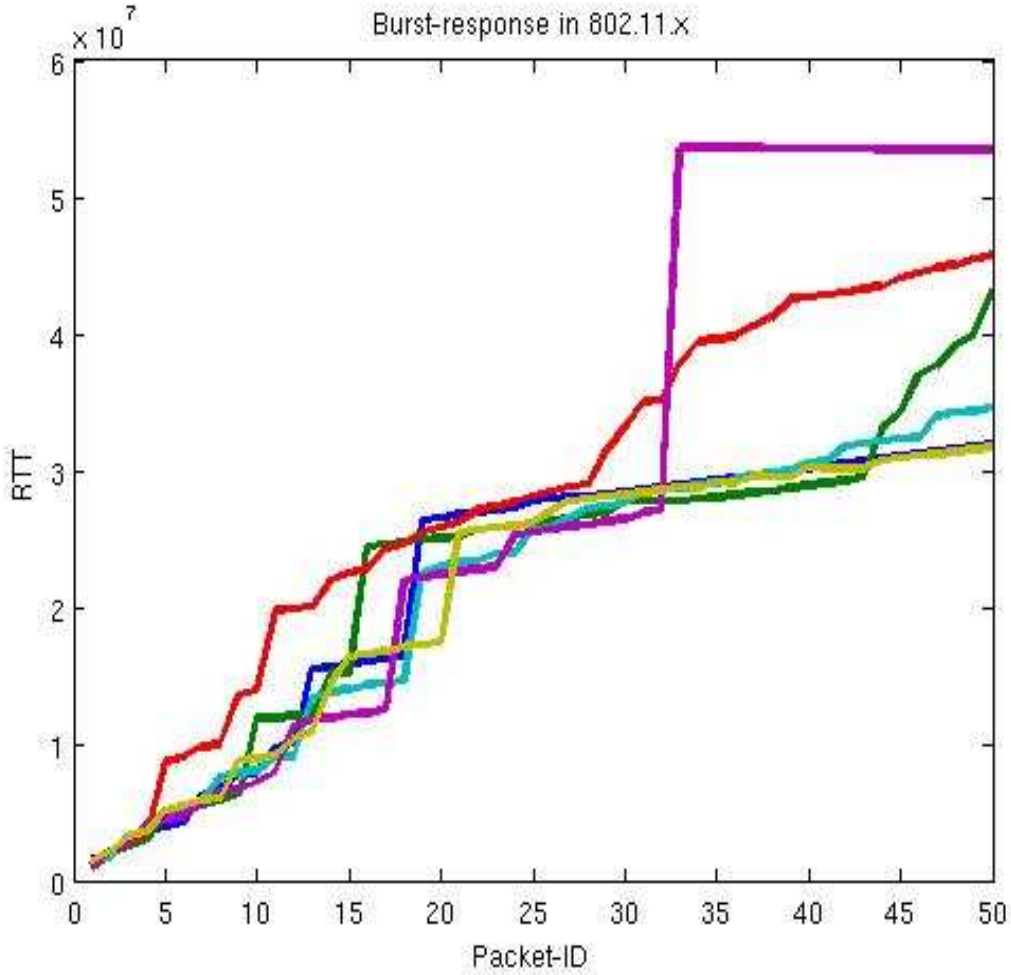
Mean =

0.0488

Invent & Verify



Yes, we can ☺



error [] =

0.1706
0.0823
0.1154
0.1052
0.0935
0.1578

Mean =

0.1208

'Invent & Verify

Error-Calculation

- Given an input-vector ‘s’ of RTTs, the error can be calculated by:
- subtracting min(s) from each element of s
- dividing each element of s by max(s)
- calculating the absolute difference between the linear function (x) and s(x)
- and summing all of these differences up.



Thank you!



Recurity Labs

Fabian ´fabs´ Yamaguchi
fabs@recurity-labs.com

Felix ´FX´ Lindner
fx@recurity-labs.com

Recurity Labs GmbH, Berlin, Germany
http://www.recurity-labs.com

Invent & Verify

