University of Connecticut
Computer Science and Engineering

CSE 4402/5095: Network Security

Denial of Service (DoS)

Amir Herzberg

 |

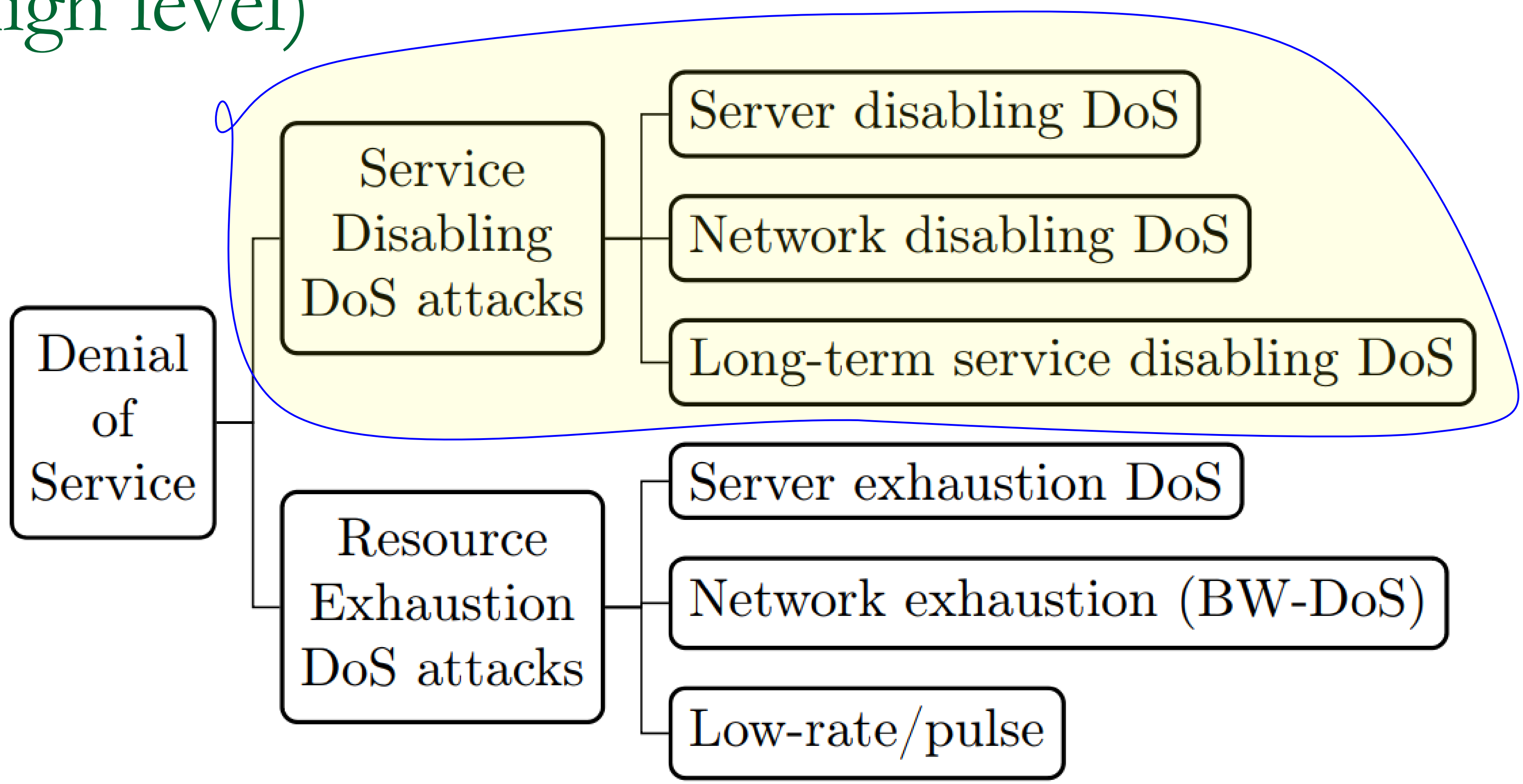
What is Denial of Service (DoS)?

- DoS attack: make a service unavailable
  - Also: Degradation of Service
  - Economic DoS: increasing expenses (costs)
  - Amplification: attacker spends less resources than resources wasted due to attack
- Many types & methods of DoS !

The Denial of Service (DoS) Challenge

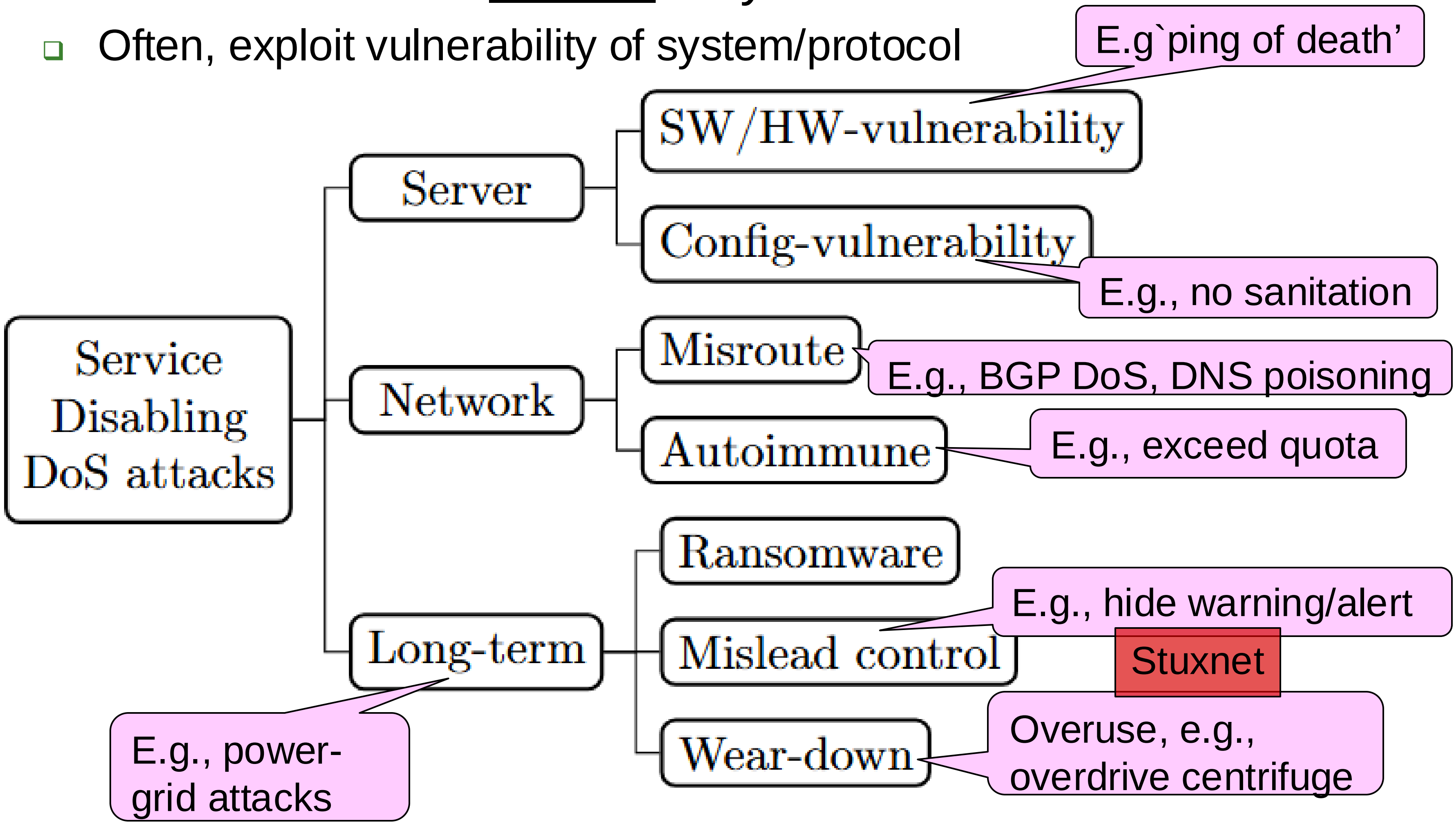
- DoS attacks are a growing, major challenge:
  - Financial goals: blackmail, competition, crypto-mining
  - Other goals: cyber-war, hate/hacktivism, censure/de-anon
  - Or: as part of other attack, e.g., exploit DNS transitivity
    - Or: block DNS → no updates, patches, SPF, blacklist, …
- DoS-facilitating developments:
  - DoS-enabling-technologies
    - Internet-of-Things devices
    - Cloud
  - Offensive cyber-warfare, hacktivism
  - Cryptocurrencies
  - Availability: DoS-Services (aka ‘stress-testing’ ?)

Denial-of-Service attacks: Taxonomy (high level)



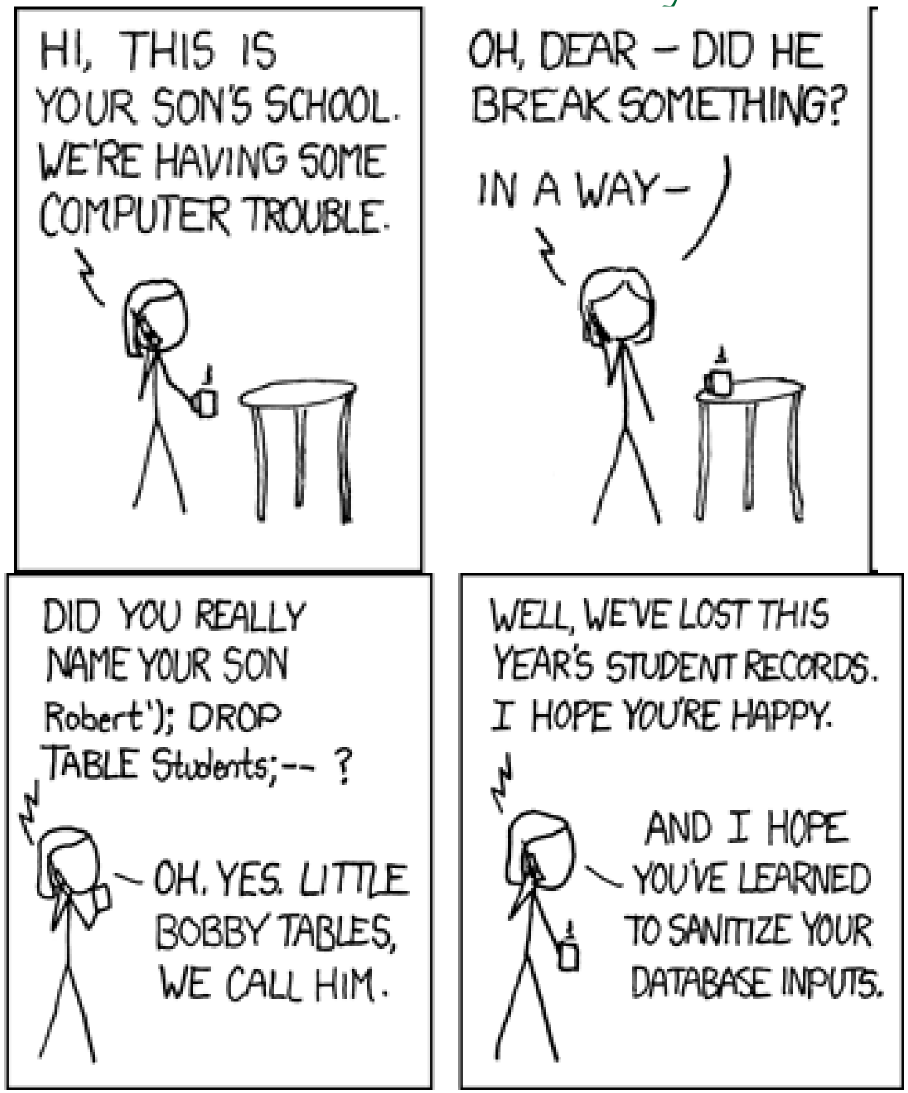
Service Disabling DoS Attacks

- Some DoS attacks disable a system
  - Often, exploit vulnerability of system/protocol



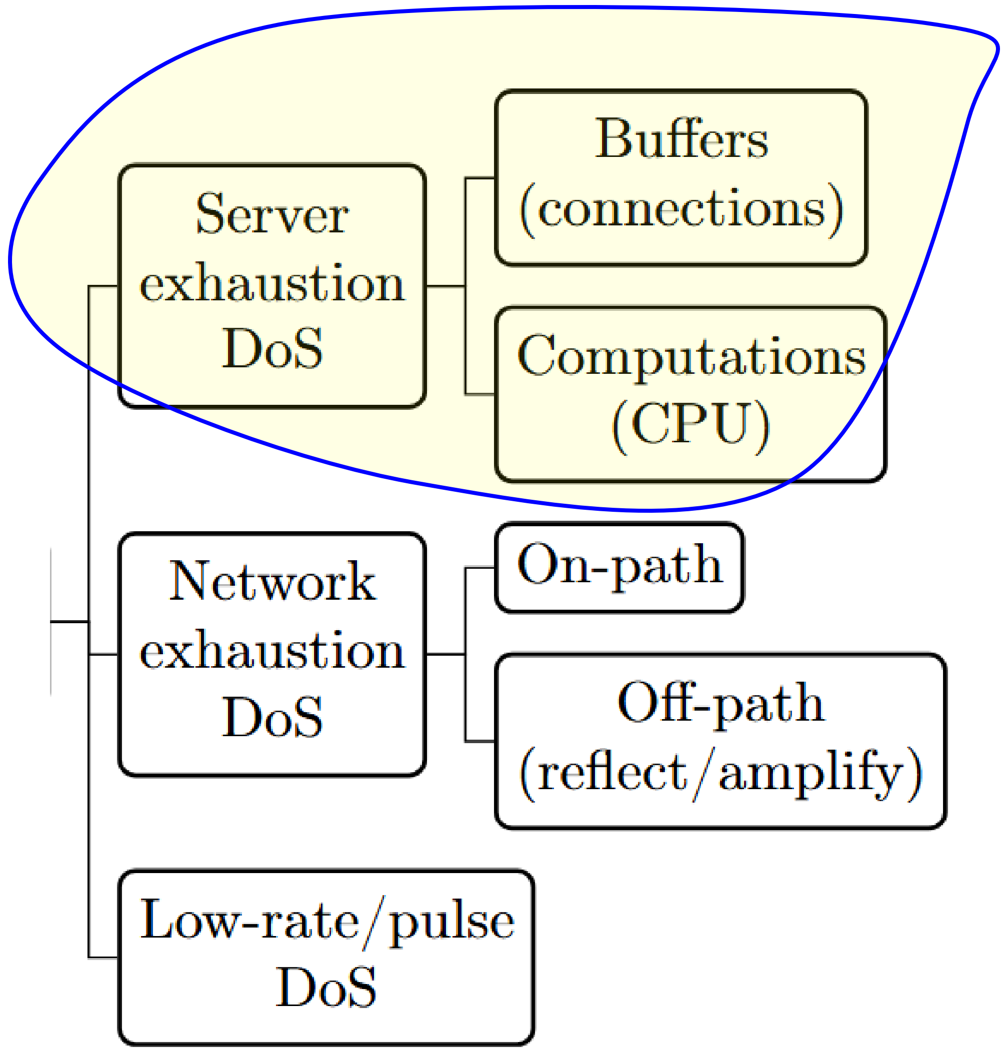
Example: Configuration Vulnerability

- Configuration vulnerabilities include:
  - Vulnerable DNS resolver ➔ route to attacker
  - Vulnerable routing ➔ routing to blackhole
  - Vulnerable web service, e.g, **no sanitation**
  - **Example:** Little Bobby Tables, or: son of Eve

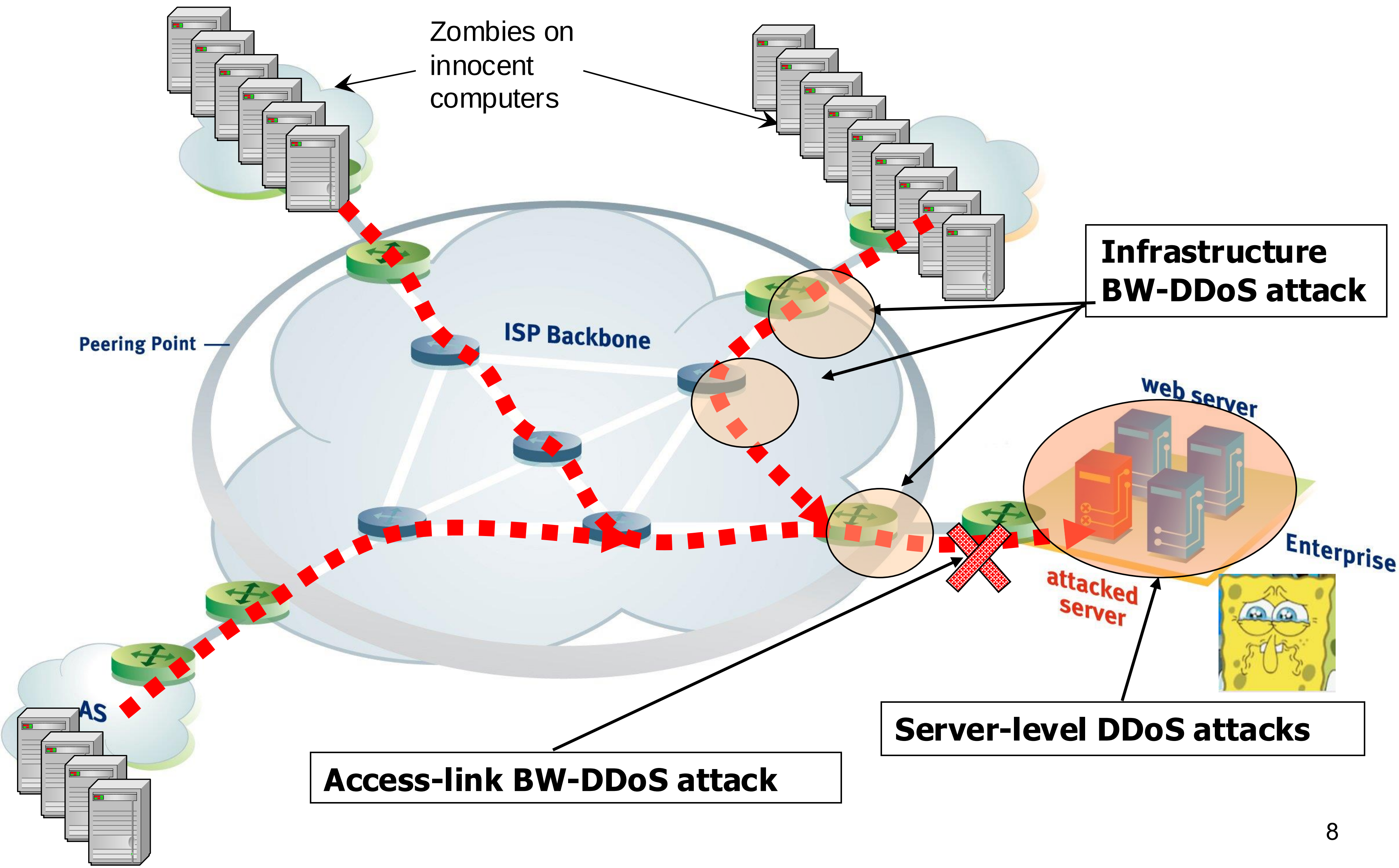


Resource exhaustion DoS attacks

- Server resources: memory, processing, connections,…
- Network resources (BW)
  - Bytes/bits per second
  - Packet per second
- Often by many clients: **Distributed DoS (DDoS) Attacks**

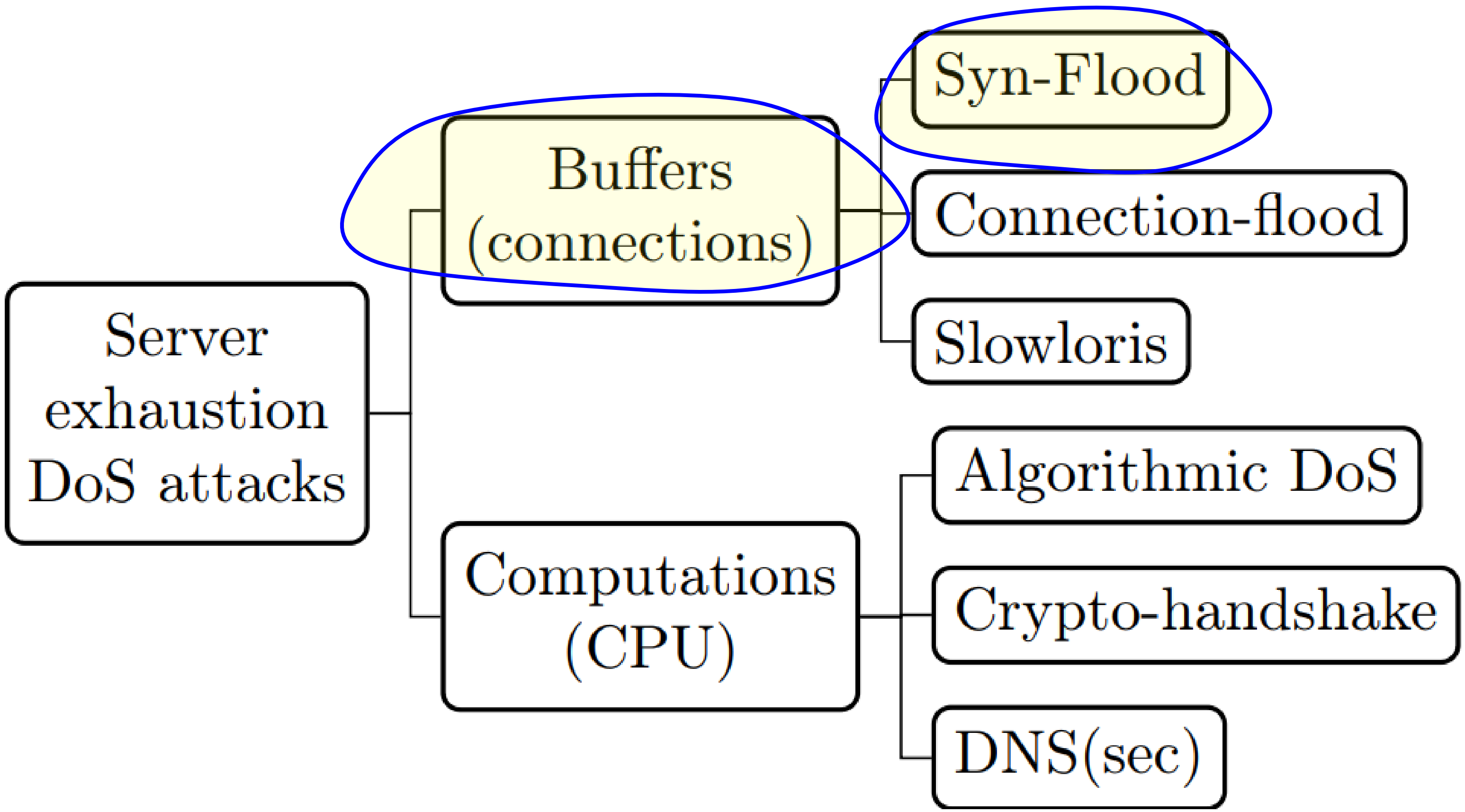


Distributed DoS: exhausting servers/network resources



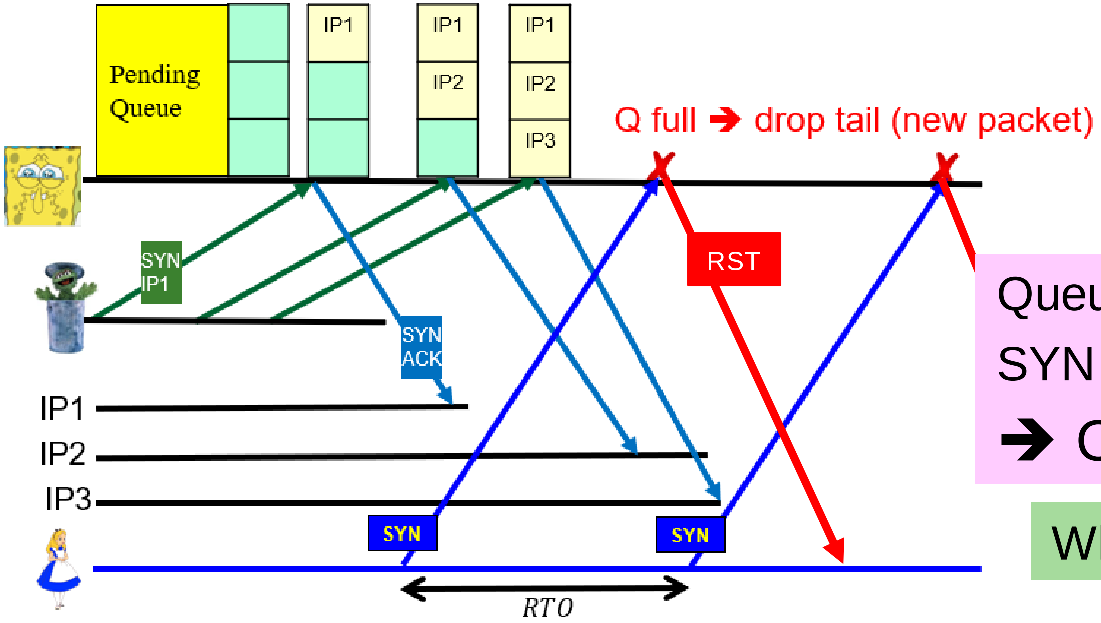
Server resource exhaustion DoS Attacks

- Exhaust server resources:
  - Connections, storage, computation, other



SYN Flood DoS Attack

- Off-path Attacker sends many SYN requests
  - From different (spoofed) src IP addresses:
  - Attacker doesn't receive Syn-Ack – so can't send (legit) Ack [why?]
- Exhausts **servers' pending connections queue**
  - If queue not limited: server crash; most OSes use **drop tail queues**



Queue size: 1000 entries

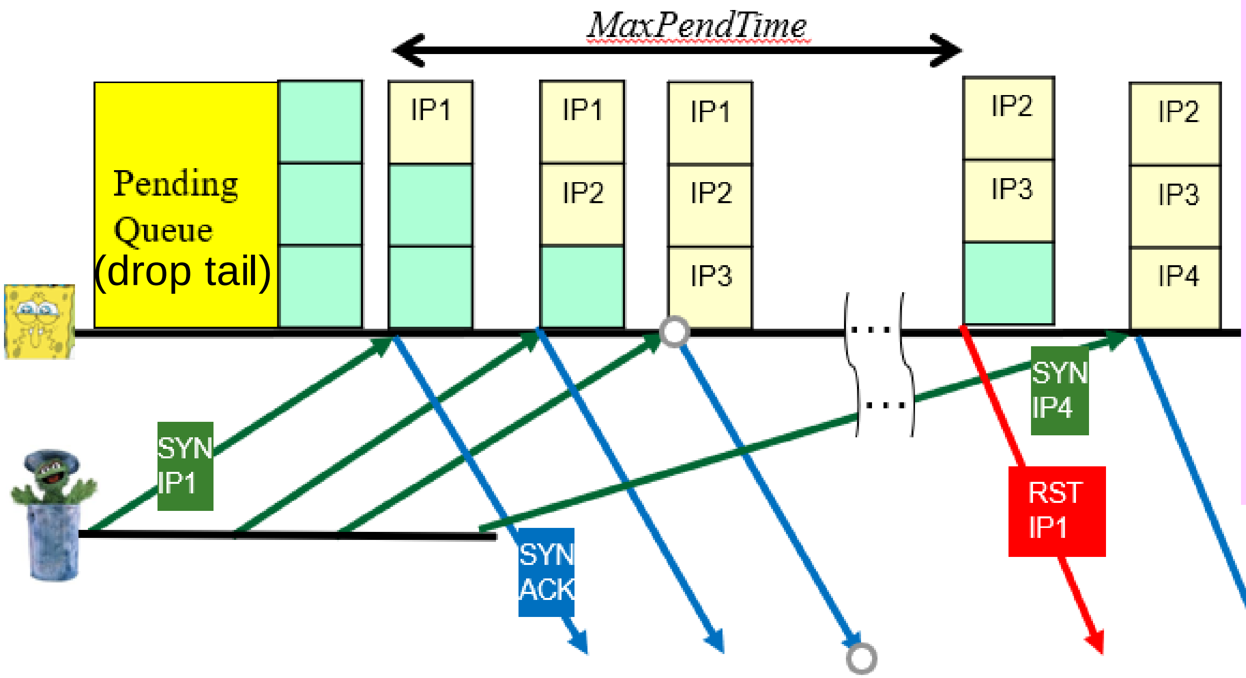
SYN pkt: ~ 60 bytes

➔ Only 60KB to fill queue!

What of pending-Q timeout?

SYN Flood: Timeout for Pending-Q

- Pending-Queue timeout : *MaxPendTime*
- Typical *MaxPendTime* is significant, e.g., 50sec
  - To avoid losing connections due to high delay (RTT)



Queue size: 1000 entries

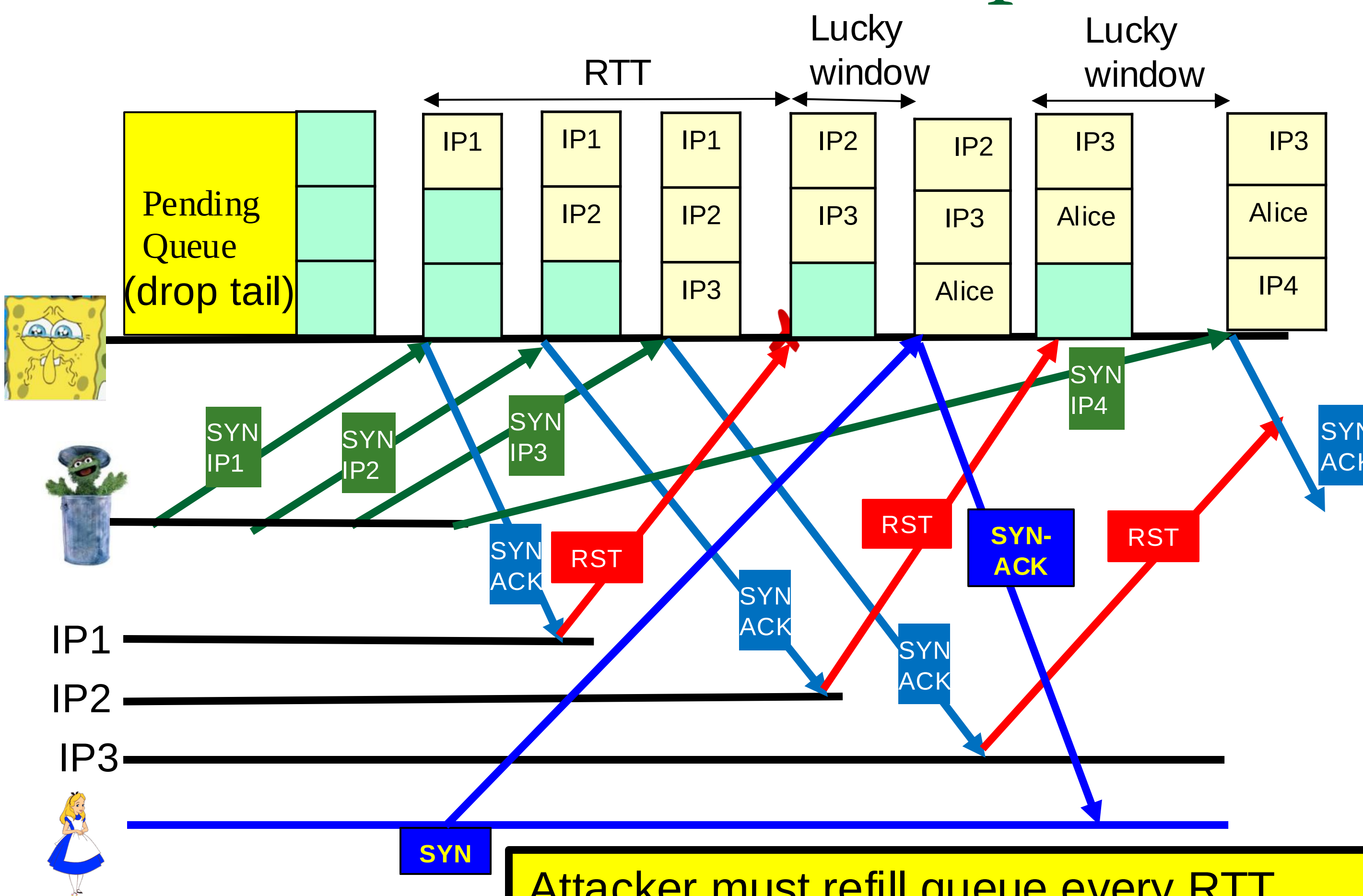
SYN pkt: ~ 60 bytes

To keep Q full, attacker needs:

- No timeout: 60KB
- 50 sec timeout : 1.2KB/s
- 5 sec timeout: 12KB/s

Wait… hosts should respond to unsolicited SYN/ACK with RST, no?

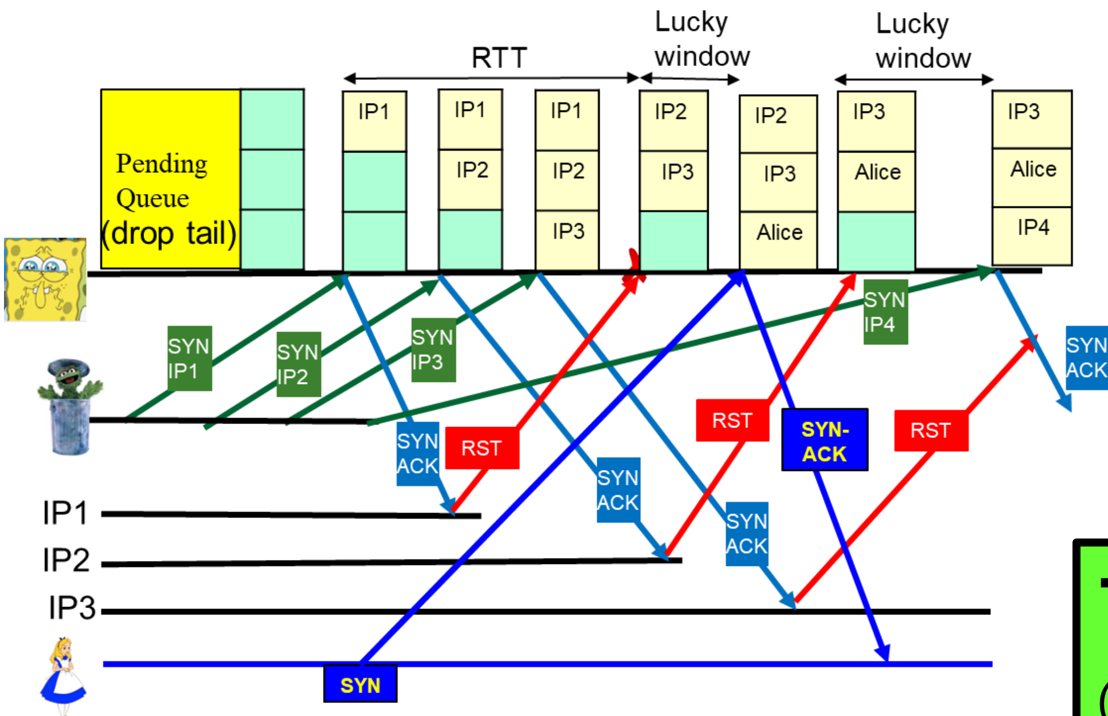
SYN-Flood with RST responses?



Attacker must refill queue every RTT.
Refill packet discarded if queue is full (too early)
Lucky window: time since queue emptied till refilled

SYN Flood with RST-responses

- TCP sends RST upon unsolicited SYN/ACK
- RST arrives after RTT << *MaxPendTime*
  - Typically, RTT is less than 100msec



Queue size: 1000 entries

SYN pkt: ~ 60 bytes

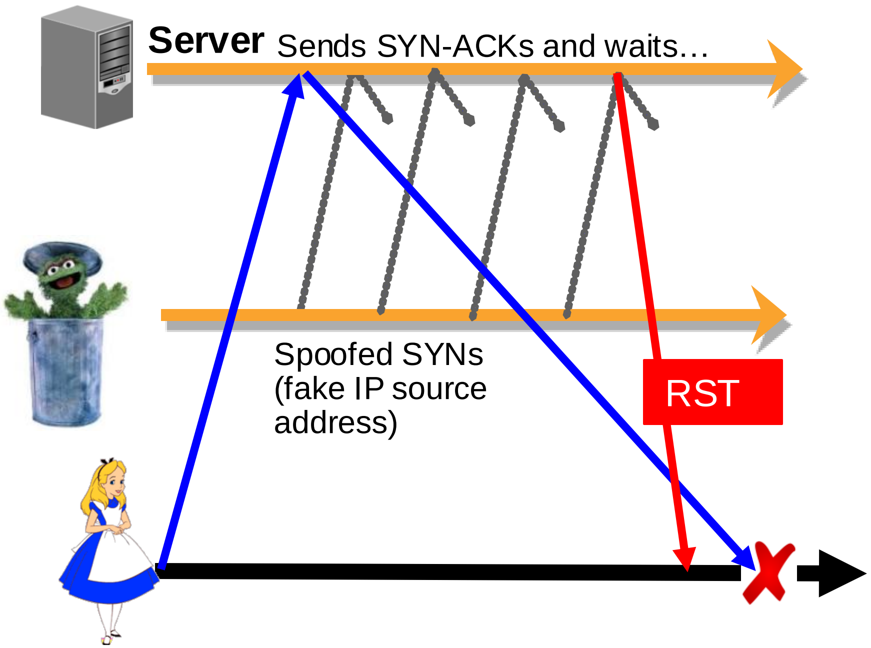
To keep Q full, attacker needs:

- No timeout: 60KB
- 50 sec timeout : 1.2KB/s
- 5 sec timeout: 12KB/s
- **100msec RST: 600KB/s**

➔ Attackers prefer to spoof non-RSTing IP addresses (NATs, FWs, unused IPs, IoT…) [exercise: scan to find such IPs]

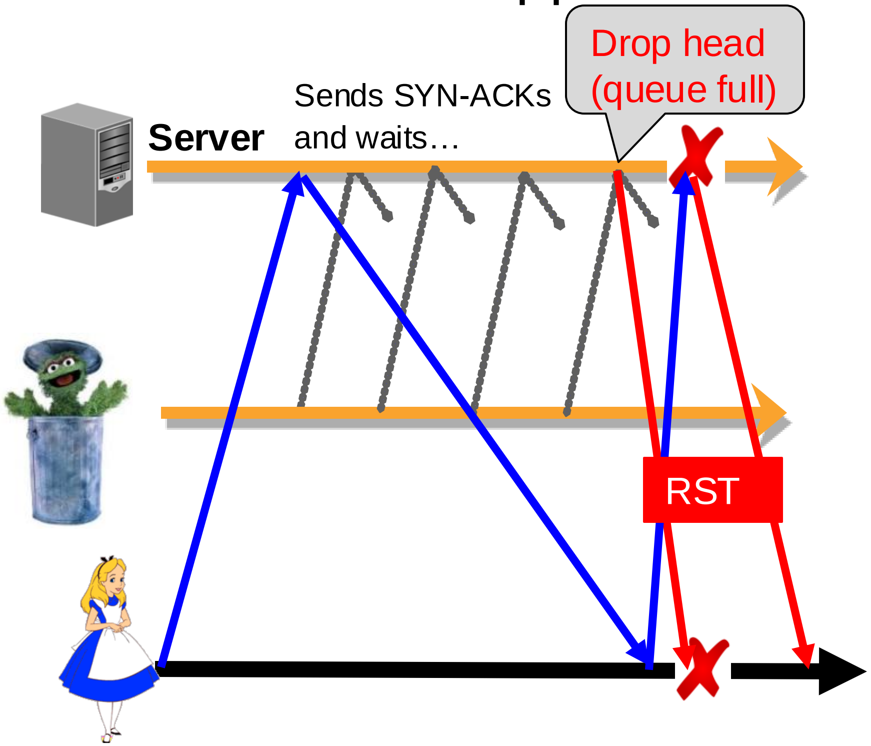
SYN Flood against Drop **Head** Q

- Drop Head Queue: full Q → drop **oldest** packet in Q!
  - Improvements in resistance
  - But, attacker can drop **all** client packets:
    Queue refilled before Ack → oldest connection dropped
  - Example values…
    - RTT: 100msec
    - Queue size: 1000 entries
    - SYN pkt: ~ 60 bytes
  - Rate: 60*1000/RTT=600KBps
    - Improves a lot (like RSTing IPs)!

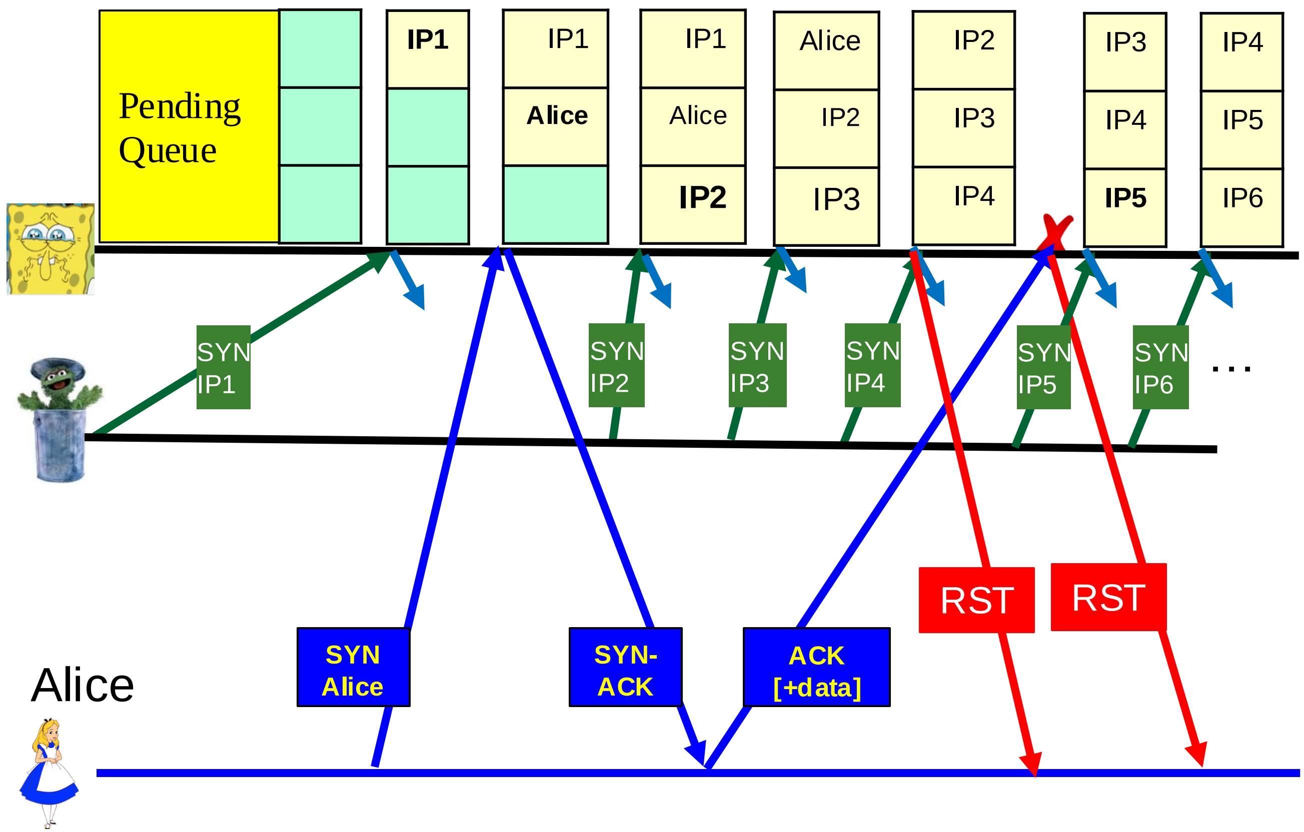


SYN Flood against Drop **Head** Q

- Drop Head Queue: full Q → drop **oldest** packet in Q!
  - Improvements in resistance
  - But, attacker can drop **all** client packets:
    Queue refilled before Ack → oldest connection dropped
  - Example values…
    - RTT: 100msec
    - Queue size: 1000 entries
    - SYN pkt: ~ 60 bytes
  - Rate: 60*1000/RTT=600KBps
    - Improves a lot (like RSTing IPs)!
    - But SYN flood still too easy
      - And no ‘lucky window’

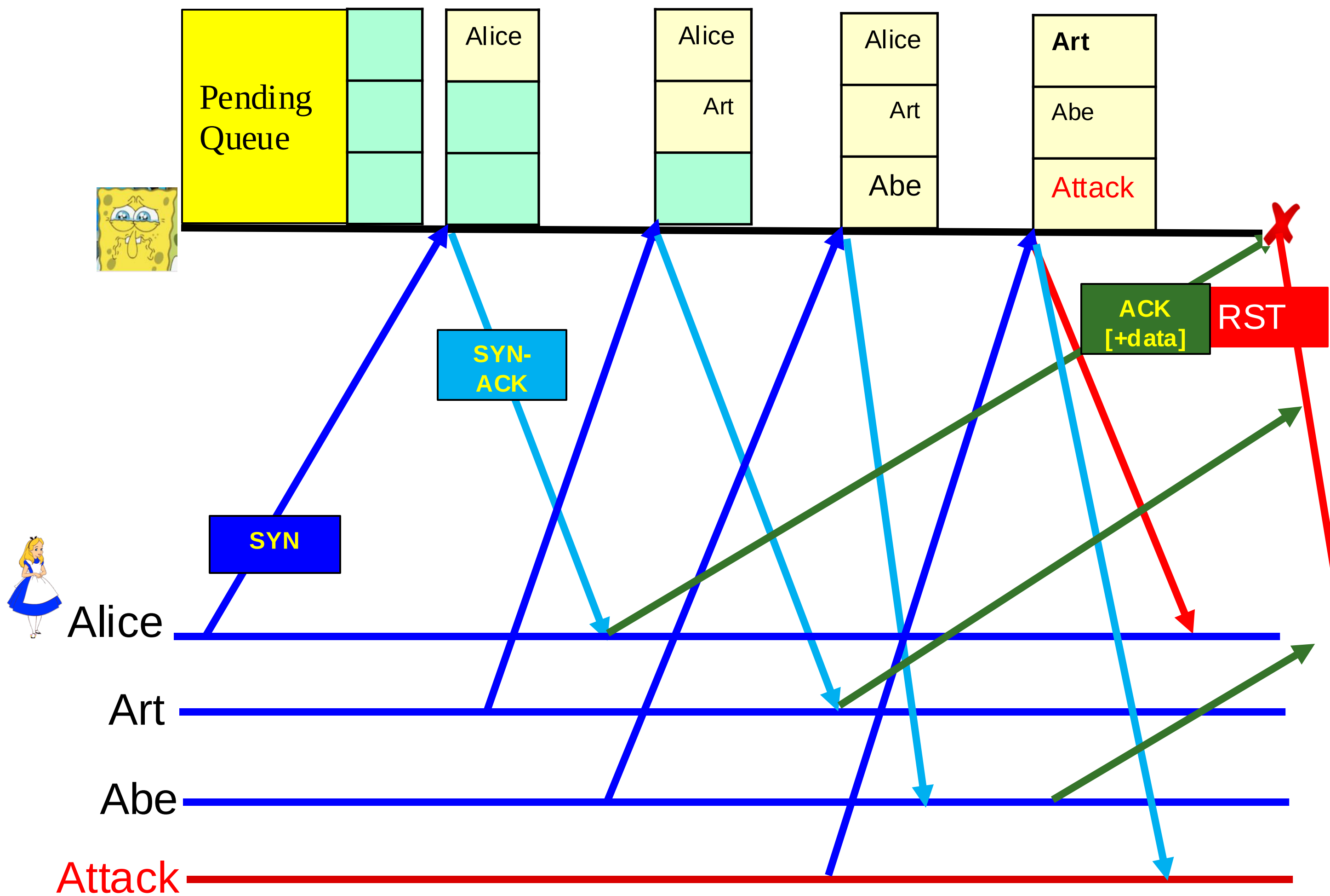


SYN-Flood-Drop-Head-Q

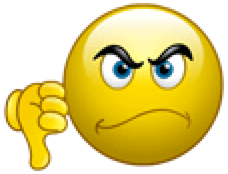


Note: Drop-Head-Q is terrible for high load!

Attacker just needs to add packets so that we keep dropping head…



SYN FLOOD: basic, weak defenses…



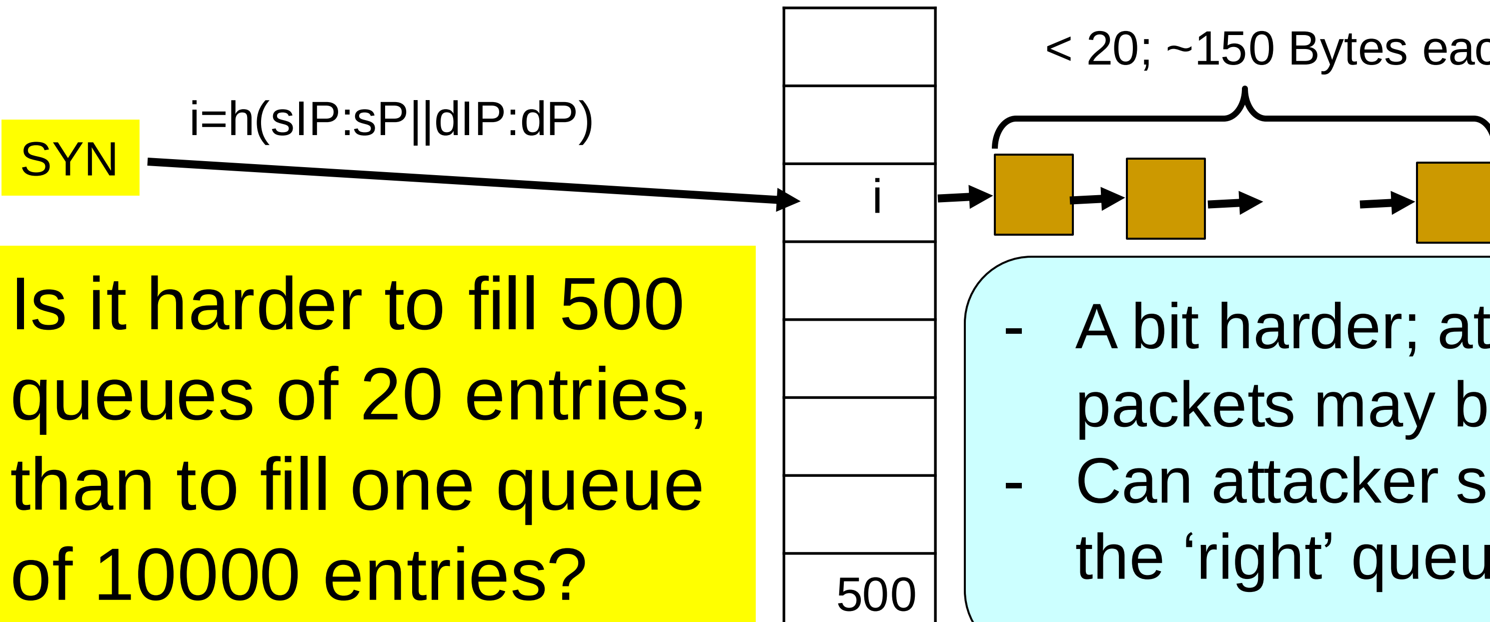
- Reduce SYN Timeout
  - Increases attacker's bandwidth to maintain full queue
  - But only linearly – and with long delay, we'll drop legit queries!
- SYN-Quota: blacklist IPs/subnets with many half-open connections (possibly SYN-flooding)
  - Attacker SYN-floods with spoofed src-IP
  - Attacker SYN-floods with src-IP of victim, causing blacklisting
  - Example of autoimmune DoS attack
  - Victim sending RST may foil the blacklisting, though

SYN FLOOD: better basic defenses

- Reduce state per entry: 96Bytes (vs. 1616B TCB)
  - Allocate TCB only on completed handshake
  - Save client seq-num and connection-options sent in SYN
    - MSS, window-scaling, SACK – and new options?
- Larger pending-connections queue:
  - 1000 entries, 100msec RTT, 60B/pkt ➔ 600KB/s attacker
  - 10000 entries … ➔ 6MB/s ➔ more, but only linearly**!**
  - ➔ slow search in Queue
- Client retransmissions
  - Exists! 10 to 16 retransmissions (drop-tail: SYN, drop-head: ACK)
  - Would retransmissions help or be discarded too?
    - Drop-tail: maybe some will get in the ‘lucky window’ [of drop-tail Q]
    - Drop-head: will increase load, make it easier to cause head-drops!
    - Next, a related defense: **SYN-cache**

SYN-Cache: Hash to Multiple Queues

- Problem: search in long queue is slow
- Idea: instead of one large queue, use **separate queues**
  - Separate queue by hash of `four tuple’ (srcIP:port, destIP:port)
  - One queue with 10000 entries ➔ 500 queues of 20 entries each



Is it harder to fill 500 queues of 20 entries, than to fill one queue of 10000 entries?

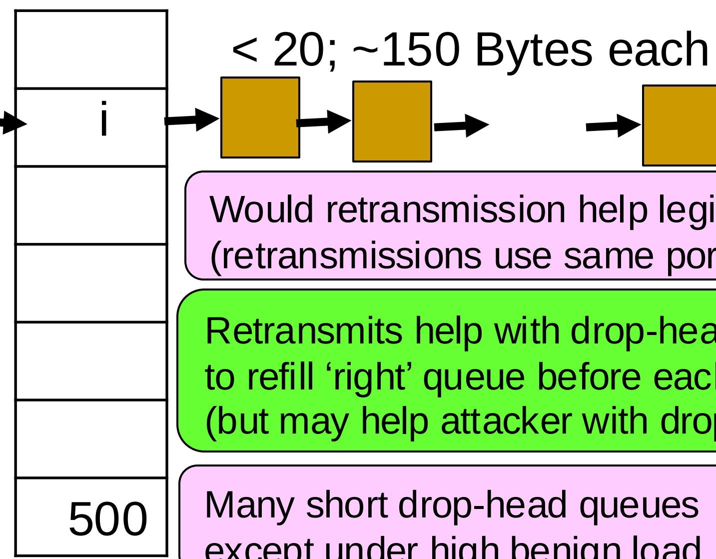
- A bit harder; attack packets may be lost
- Can attacker select the ‘right’ queue?

Syn-Cache: use **secret** random hash

- Problem: attacker selects sIP:sP to select queue(s)
  - To ensure exact number of SYNs per queue (avoid overflow)
  - Even ‘hit’ queue to be used by victim (when ports are predictable)
  - Solution: use keyed hash – more precisely, **PRF**
- Attacker would now **overfill** queues → waste packets!
- 80% queues full ➔ 80% attacker packets lost !!

SYN

i=PRF_k(sIP:sP||dIP:dP)



Would retransmission help legit clients? (retransmissions use same ports)

Retransmits help with drop-head: attacker has to refill ‘right’ queue before each ACK!! (but may help attacker with drop-tail!)

Many short drop-head queues are best – except under high benign load

Q: what if attacker spoofs srcIP:sP **of victim-client**?

Completely Stateless Handshake?

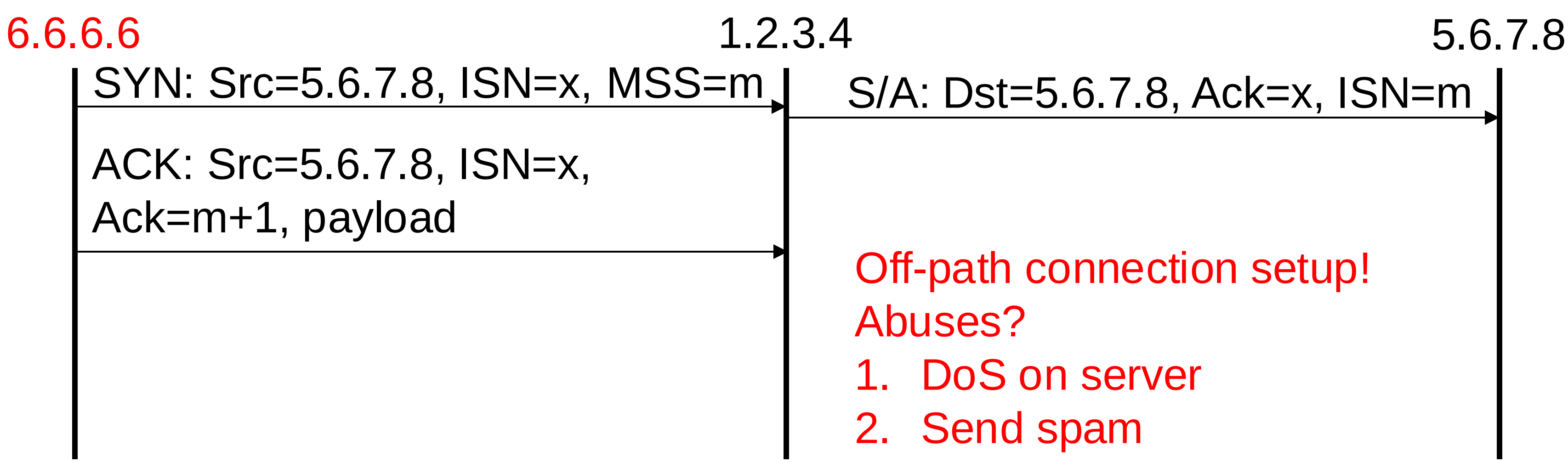
- **DoS defense principle:**
  <u>stateless* until client 'pays or identifies'</u>

… and no CPU-intensive ops

- Challenge: stateless TCP server that:
  - Establishes connection using the client's 'ACK' packet
  - Recovers (all? important? approx?) TCP SYN-only options:
    - **MSS (4B), window scale (3B), SACK permitted (2b)**
  - Connection remains secure against off-path attacker
- Idea 1: use server's ISN as a cookie, to encode options
  - TCP echoes the last received sequence-number in the ACK field
  - Problem 1: server's ISN is only 4B – can't encode 7B+2b !
    - Encode the most important 4B from the options?

SYN-Cookies : strawman design

- Use Server ISN as 4B from the TCP SYN options
  - E.g., only MSS (4B) [no: window scale (3B), SACK-Ok (2b)]
  - Allows server to recover MSS from Ack value in ACK packet
- Vulnerability? Attack?
  - Off-path attacker can predict server's ISN, and…



SYN-Cookies: toward a real design…

- Goal: stateless TCP server that:
  - Establishes connection using the client's 'ACK' packet
  - Recovers (all? important? approx?) TCP SYN-only options
    - Main options: **MSS (4B), window scale (3B), SACK permitted (2b)**
  - Such that connection remains secure against off-path attacker
- Strawman design: encode options as server's ISN
  - Problem 1: server's ISN is only 4B – can't encode 7B+2b !
  - Problem 2: attacker knows options → ISN → hijack connection
- Use part of the ISN to encode, and part to prevent hijack
  - Must be unpredictable to attacker, known to server
  - A function of something… which function? Which inputs?

SYN-Cookies: 2nd strawman

- 32b Server ISN ← (f(MSS)||r):
  - **24 bits** r ← **PRF$_k$**(clientIP:Port || serverIP:Port)
    - **PRF: pseudo-random function**
    - **PRF** key **k** known only to server [and changed?]
    - Probability of guessing valid cookie: ~$\mathbf{2^{-24}}$
  - f(MSS): (8 bit?) encoding of the MSS (Max-Segment-Size)
- Why not just client IP (cIP)?
  - Client and attacker may be behind same NAT (same IP)
  - Server may use same key at multiple IPs, ports
  - Concern: attacker collects r for all client ports before client connects
  - Solution: add the time as input to the PRF
    - But make sure server uses the same time as the client – how?

SYN-Cookies: 3rd strawman

- Server ISN= (T|MSS|r)(32b):
  - **T (5b or 6b)**: time in minutes mod 32
  - **MSS (3b or 2b)**: encodes one of 8 or 4 Max-Segment-Sizes
  - **r=PRF$_k$**(cIP:Port || sIP:Port || T)
    - **PRF** key **k** known only to server [and changed?]
    - Probability of guessing valid cookie: ~2^{-24}
- Small additional concern: unintentional collision
  - 1st connection closes quickly
  - 2nd connection may use same client port
  - A packet from 1st connection will have the same server ISN
  - Solution: randomize using client's ISN!

SYN-Cookies Defense [RFC4987 sec. 3.6]

- Server ISN= client's ISN (32b) + (T|MSS|r)(32b):
  - **T (5b or 6b)**: time in minutes mod 32
  - **MSS (3b or 2b)**: encodes one of 8 or 4 Max-Segment-Sizes
  - **r=PRF$_k$**(cIP:Port || sIP:Port || T)
    - **PRF** key **k** known only to server [and changed?]
    - Probability of guessing valid cookie: ~2^{-24}
- Used when pending Q is full
- Connection not added to Q (no state)
- Connection established upon valid ACK from client
- Timestamp SYN-cookies: more bits of SYN-Options

TCP Timestamps (and timestamp cookies)

- TCP timestamps: a widely-used TCP option [RFC1323]
  - 10 bytes, 4 for Timestamp, 4 for EchoTimestamp
  - Used for RTT measurement: RTTsample=(time-EchoTimestamp)
  - Estimated RTT helps improve performance
    - In particular: optimized retransmission time out
  - Send in SYN packet – and, if received, also in later packets
  - Used (by default) by many systems
- Timestamp SYN-cookies:
  - Use few (9) least-significant bits of timestamp for SYN-options
  - Encode window-scale and SACK options
  - Supported by / default in most modern operating systems

Disadvantages of SYN-cookies

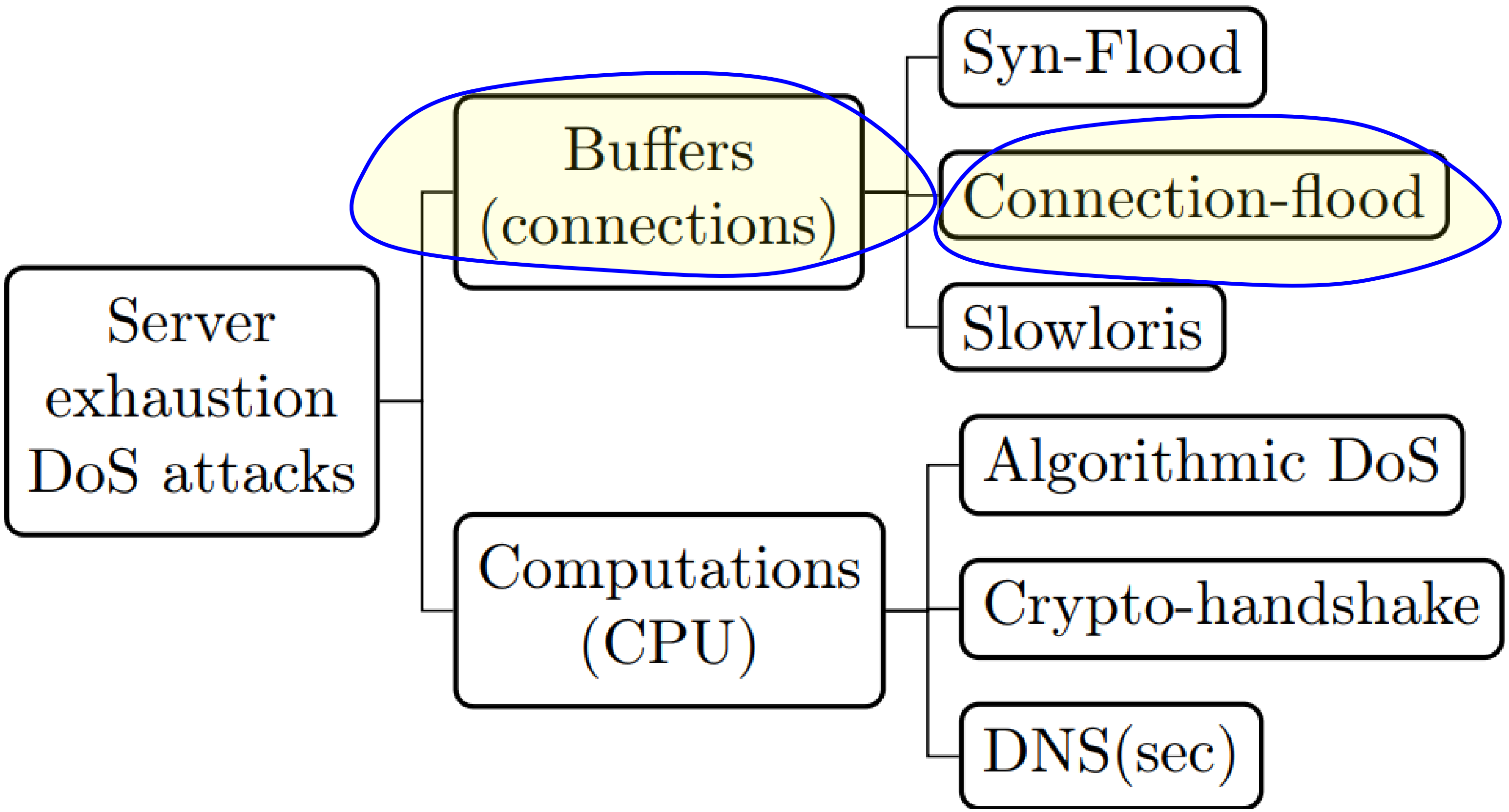
- Lose (most) connections-options:
  - No timestamp: only MSS, with timestamp: also scaling and SACK
  - Lose other options, precision of MSS, scaling, timestamp
- Server can’t retransmit SYN-ACK [if no ACK received]
  - If SYN-ACK is lost, client retransmits SYN [albeit slowly]
  - But if ACK w/o data is lost, **client will not retransmit**
    - E.g., SMTP client sends only Ack w/o data
  - No server retransmit, ack lost ➔ client stuck waiting for server’s 200 OK !
- Some computational overhead for every SYN received
- Makes it easier to guess server’s ISN (hijack/kill connection)
- ➔ So SYN-cookies are usually used only under attack
  - Lab: what is the real behavior of different systems?

More detection-based SYN-Flood defenses

- Detect attack and defend unprotected hosts
- Method 1: route via a protected (TCP/App) proxy
  - Locally or remote (CDN, cloud)
  - Drawbacks: overhead, compatibility (options, etc.)
- Method 2: RST pending connections
  - Or: spoof Ack to server; timeout: RST both ends
  - Fails if attack is within RTT !
- Method 3: **SYN-drop: drop** $x\%$ of SYNs
  - Attack traffic increase by ~ $\frac{1}{(1-x\%)}$; legit: only SYNs
  - Legit connections: slower handshake, and x^{10} fail*

Server resource exhaustion DoS Attacks

- Exhaust server resources:
  - Connections, storage, computation, other



TCP Connection Flooding Attacks

- Attacker establishes connection
  - Valid IP!
  - Large state!
- Exhaust <u>established</u> queue
- Application responsible:
  - Accept
  - Kill

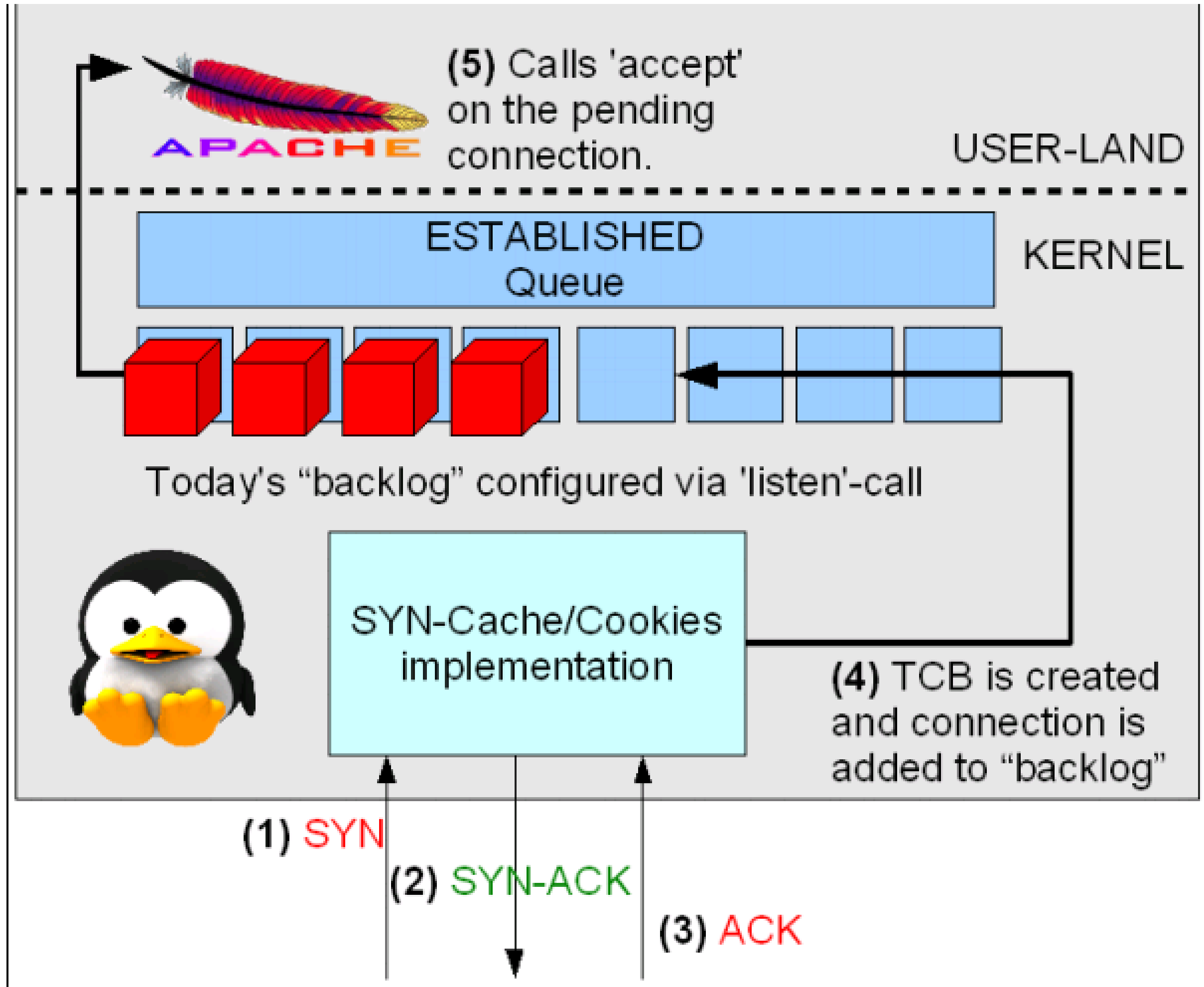


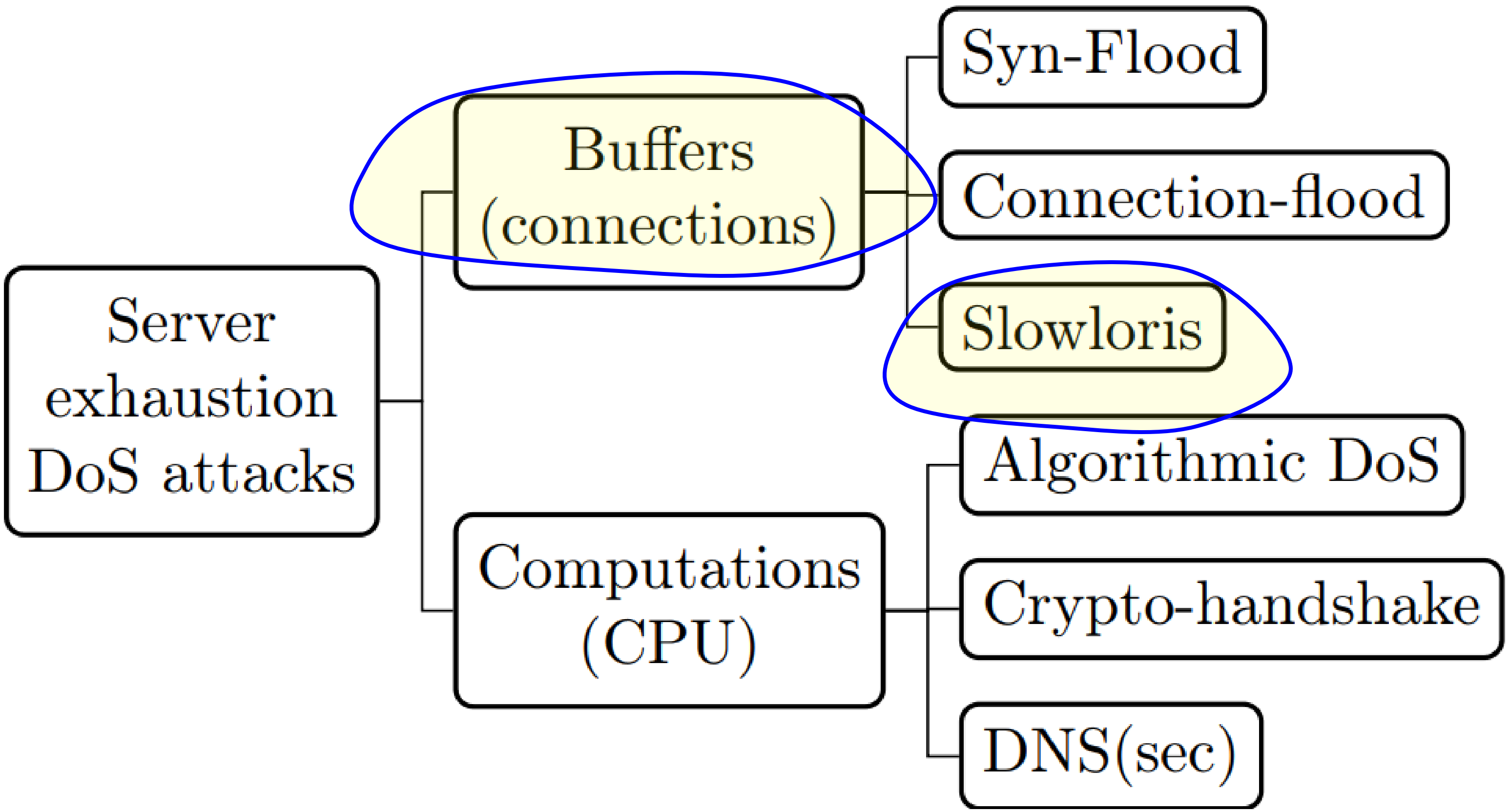
Figure by Fabian (fabs) Yamaguchi, TCP DoS Volunerabilities, online

Connection Flooding : Basic Defenses

- Limit number of concurrent requests, and/or limit resources per request
- Time-out mechanisms: kill (RST) upon…
  - No request, no progress or no acks
- Attacker's response: slow-connection attacks

Server resource exhaustion DoS Attacks

- Exhaust server resources:
  - Connections, storage, computation, other



Slow-Connection Attacks

- Idea: exhaust max# of concurrent requests
  - Esp. good against process-per-req' (web) servers
- Method 1: slow response-reading / close
  - Send empty – or tiny – RcvWin
    - RcvWin can remain empty forever, if ack sent every 'persist timer'
    - So server has to queue response, send slowly…
  - Or: don't close (don't send Ack on Fin)
    - Simple countermeasure: servers RST quickly
- Method 2: slow requests (slowloris)
  - Send request… slowly… few bytes at a time…
  - Sweet but dangerous ☺
  - Easy to deploy: uses standard TCP client
  - In fact, can be done by a cross-site script!!

Defenses from Connections-Exhausting DoS

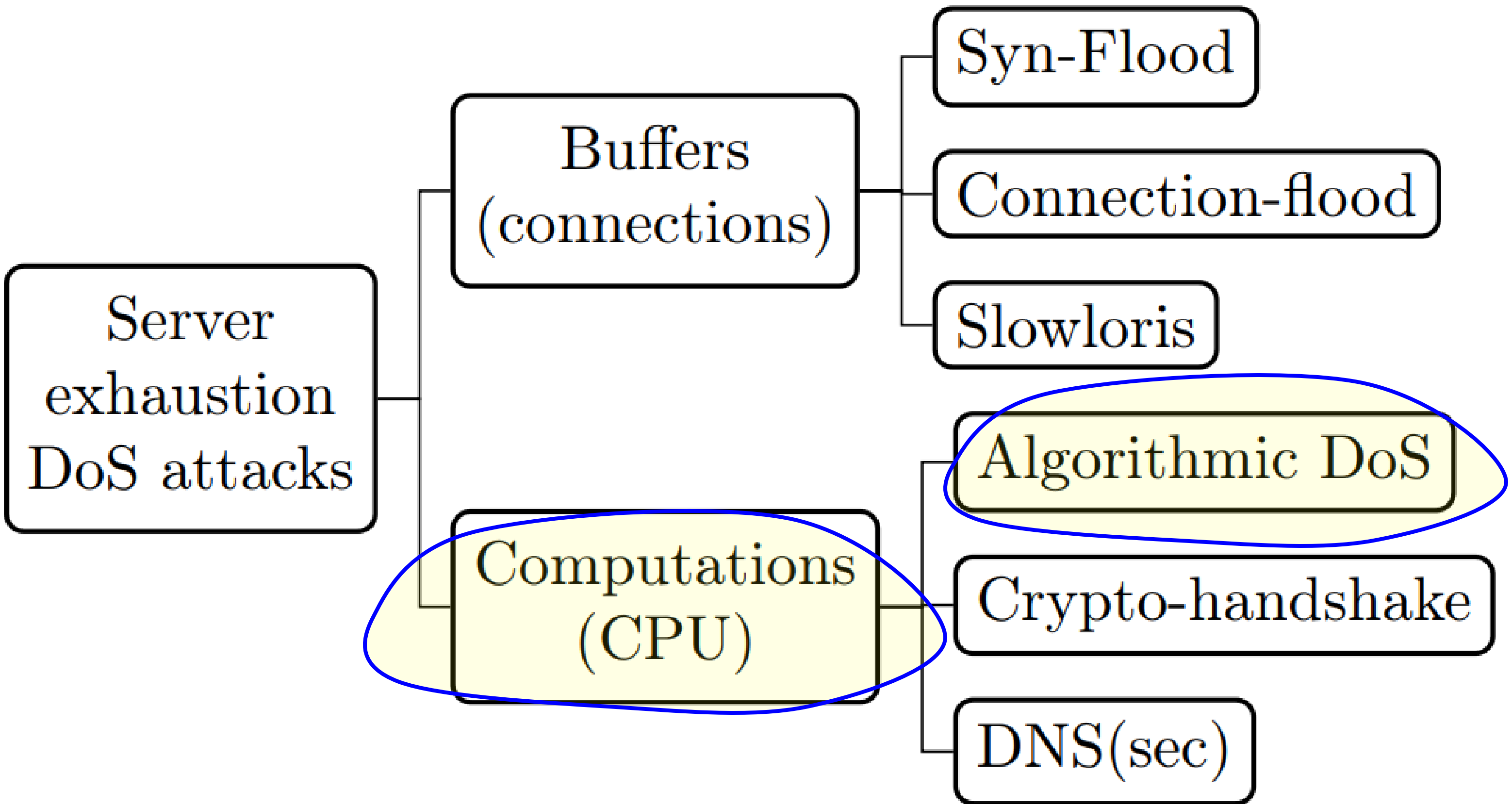
- Limit number of concurrent requests, and/or limit resources per request
- Time-out mechanisms: kill (RST) upon…
  - No request, no/**slow** progress, or no ack/fin
- Priority/service to authenticated clients
- Scale-up: CDN / Cloud
  - Risk: Economic DoS (cost of cloud instances, CDN services)
- Still too many clients? Quotas, CAPTCHs and/or PoWs
  - Same **DoS defense principle:**
    <u>stateless (and no CPU-intensive ops)</u>
    <u>until client 'pays or identifies'</u>
- Let's briefly discuss quotas, CAPTCHs and PoWs…

Quotas, CAPTCHs and PoWs

- Quota per IP and/or client: to limit and to detect
  - Reduce storage with random sampling, bloom-filters/counters
  - Reset counters and blacklist periodically
- Avoid false-positives and fake over-quota attack
  - Clients and IPs are not all alike, e.g. NAT
- Excessive? ➔ CAPTCHA or Proof-of-Work (PoW)
  - For suspect IPs (over-quota?)
  - For everyone: for attack from many IPs
    - Note: not spoofed (since handshake completed)
  - CAPTCHA: often solved by AI better than by humans ☹
    - Or by `CAPTCHA sweatshop employees’ ☹ ☹ ☹

Server resource exhaustion DoS Attacks

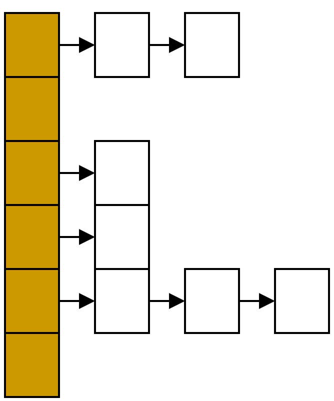
- Exhaust server resources:
  - Connections, storage, computation, other



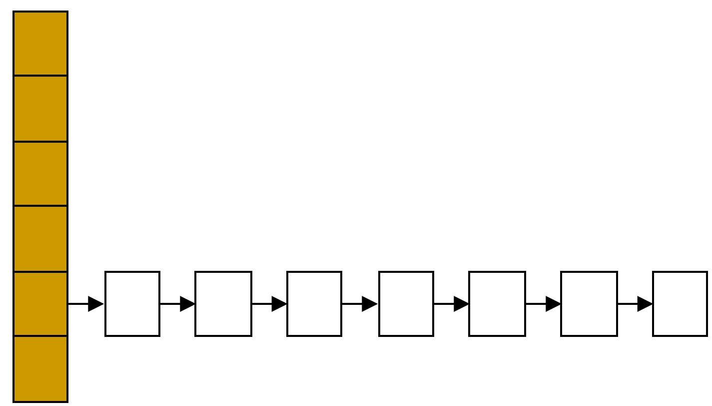
Algorithmic Complexity DoS Attacks

- Server CPU exhaustion algorithmic DoS attacks
  - ***Denial of Service via Algorithmic Complexity Attacks***, Scott A. Crosby and Dan S. Wallach, Usenix 2003
- Attacker induces the worst-case behavior of routers/servers
- Example: hash tables – Average Case O(1) vs. Worst Case O(*n*)
  - Can be done by off-path attacker (if done for out-of-connection packets)

Average case: O(1)



Worst case: O(n)

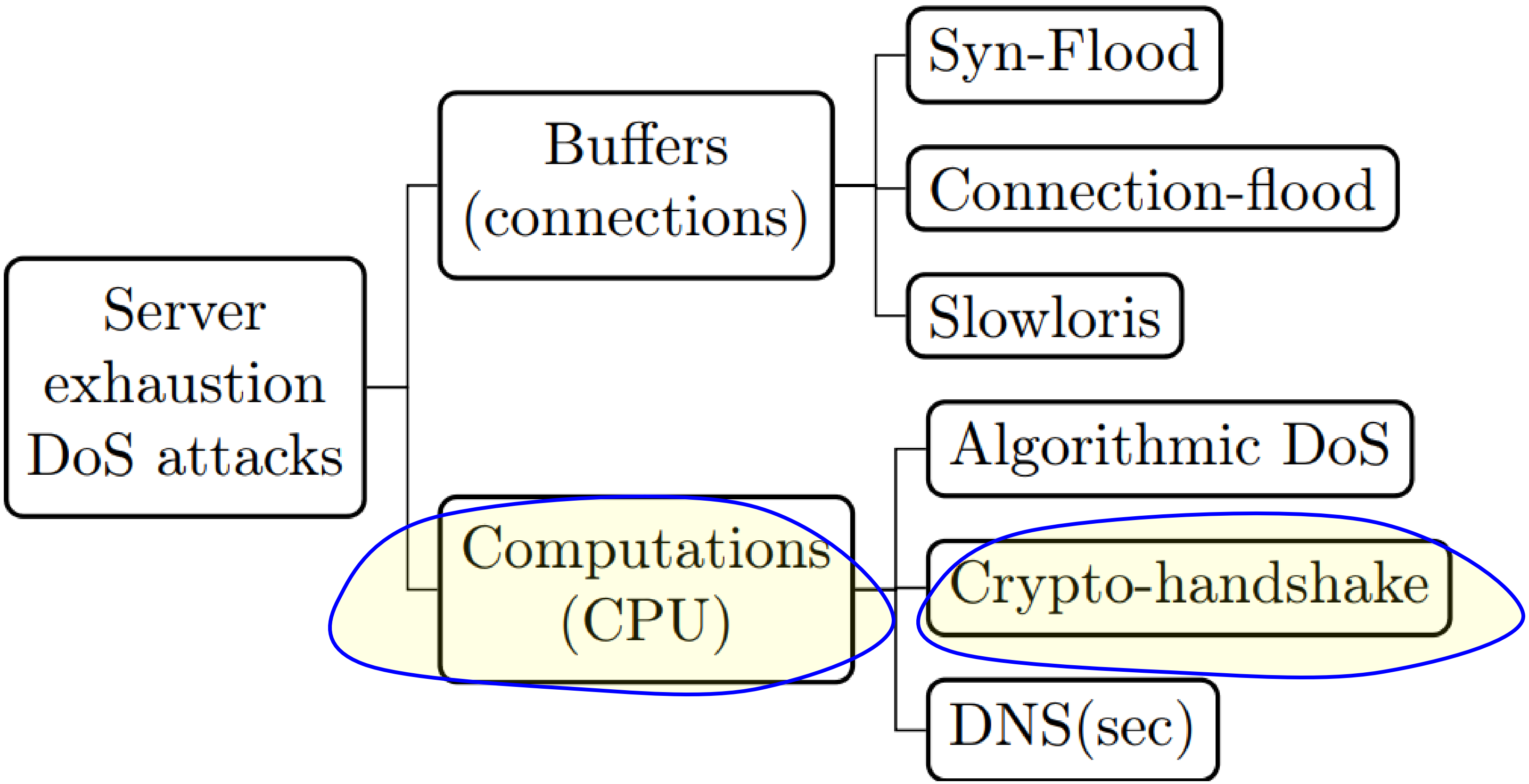


Algorithmic Complexity DoS Attacks

- Hash-based
  - Pre-condition: attacker can compute hash
  - Countermeasures
    - Change to keyed hash
- Cache-based: cause page-misses
- RegExp (ReDoS):
  - `Hard-to-compute' Regular-Expression search
  - Exploits fact that most Reg-Exp engines have exponential worst-case complexity
- Example use: to disable IDS/DPI defenses

Server resource exhaustion DoS Attacks

- Exhaust server resources:
  - Connections, storage, computation, other



Server CPU-exhaustion by Crypto-Handshakes

- DoS concern – mainly crypto-processing-time
  - PK operations (also: state at server, long response sent to client)
  - Recall principle: no CPU-intensive ops until client identifies
- Example - IKEv2 (IPsec's Internet Key Exchange)
  - Handshake begins with a Diffie-Hellman (DH) key agreement, to hide identities from an eavesdropper
  - No connection setup ➔ attacker can be off-path!

Spoofing DoS attacker

g^a mod p, n_A

Victim

g^b mod p, n_B

(by-stander?)

K = f(n_A, n_B, SPIs, g^{ab} mod p)

Two exponentiations!

IKEv2: cookies against off-path CPU-exhaustion

- Recall principle: no CPU-intensive ops until client identifies
- IKEv2 server requires client to echo a cookie before DH

$g^a \ mod\ p, n_A$ →

$Cookie = \underline{PRF_k(srcIP \,||\, n_A \,||\, time)}$ ←

$g^a \ mod\ p, n_A, Cookie$ →

$g^b \ mod\ p, n_B$ ←

- Off-path attacker can't echo cookie ➔ attack fails

Spoofing DoS attacker

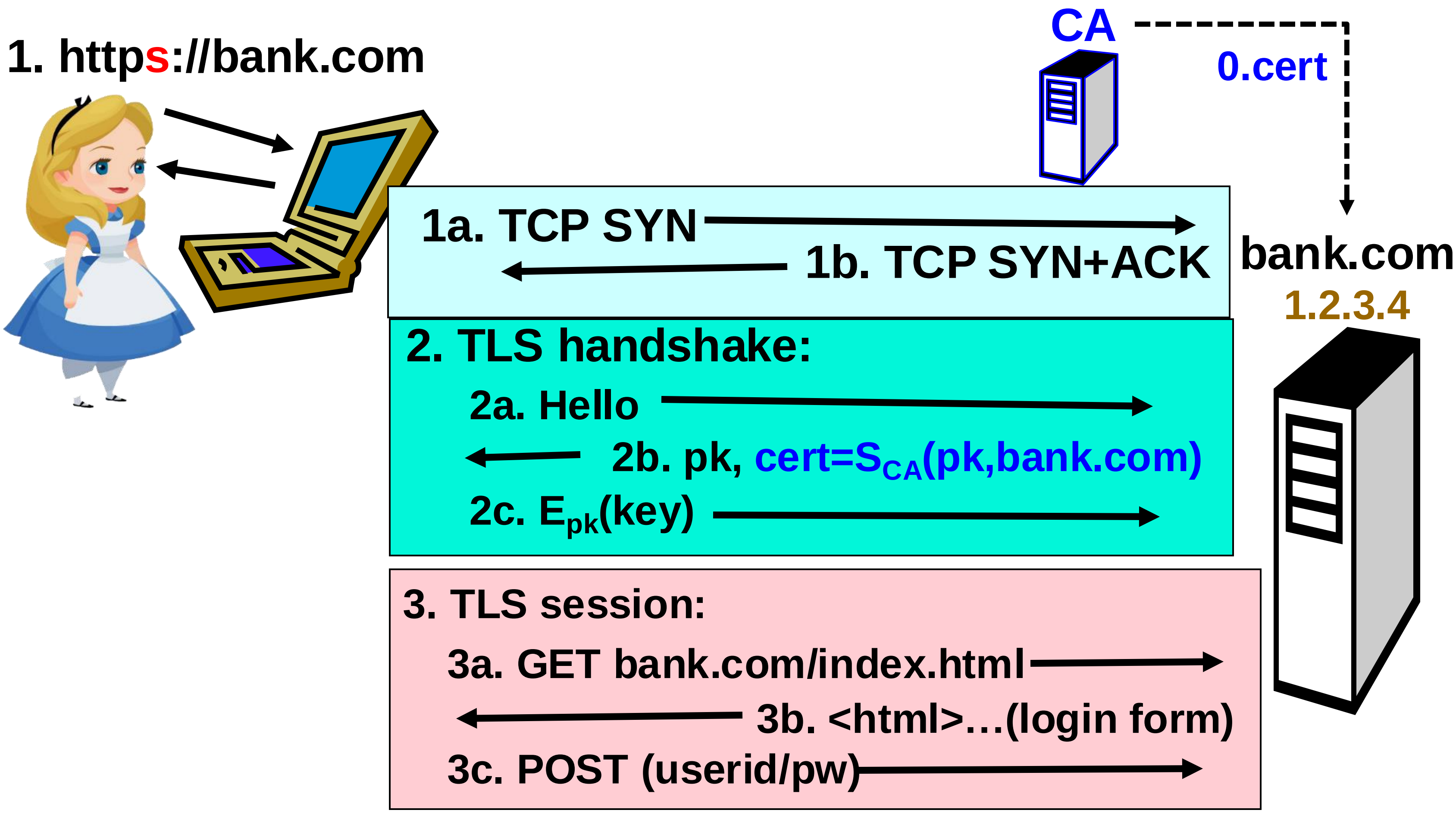
$g^a \ mod\ p, n_A$ →

$Cookie$ ←

Similar attack against TLS – also cannot be done off-path

Web Security with TLS (simplified)

- Off-path attacker can't complete TCP handshake
- **What about cross site attacker?**



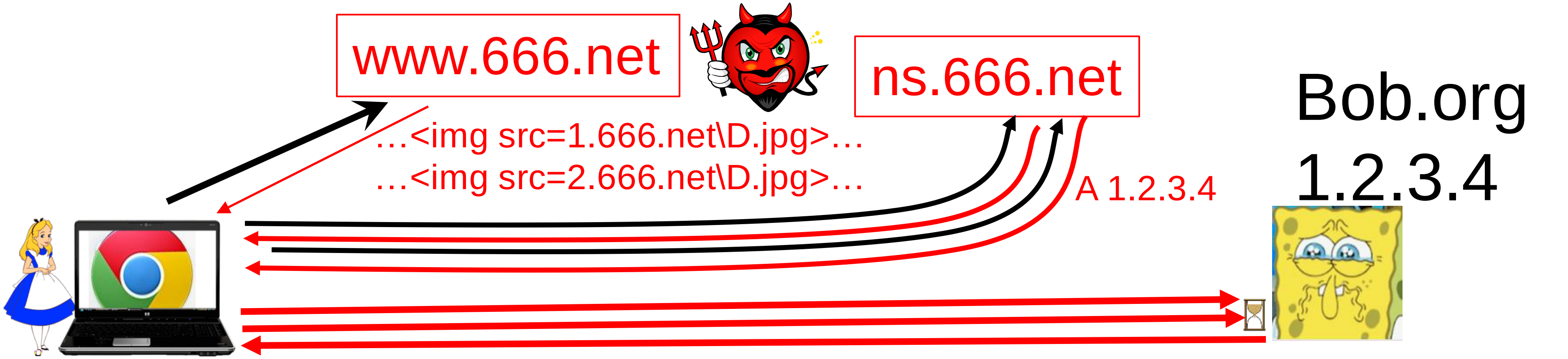
Cross-Site TLS-handshake CPU exhaustion DoS

- Rogue site sends rogue-page to client
  - Page opens many SSL/TLS connections to bob.org (how?)

  - Goal: exhaust server's connections and CPU (TLS handshake)
    - With http/1, browsers open concurrent connections to speed up
      - But: up to 4-8 connections per hostname [depends on browser]
      - But only one public-key TLS handshake, then reuse key
    - With http/2 or 3, only **one** connection (multiplexed streams)
  - Attack is ineffective!!
  - Can attacker fix attack by using different hostnames in requests?

Cross-Site TLS-handshake CPU exhaustion DoS

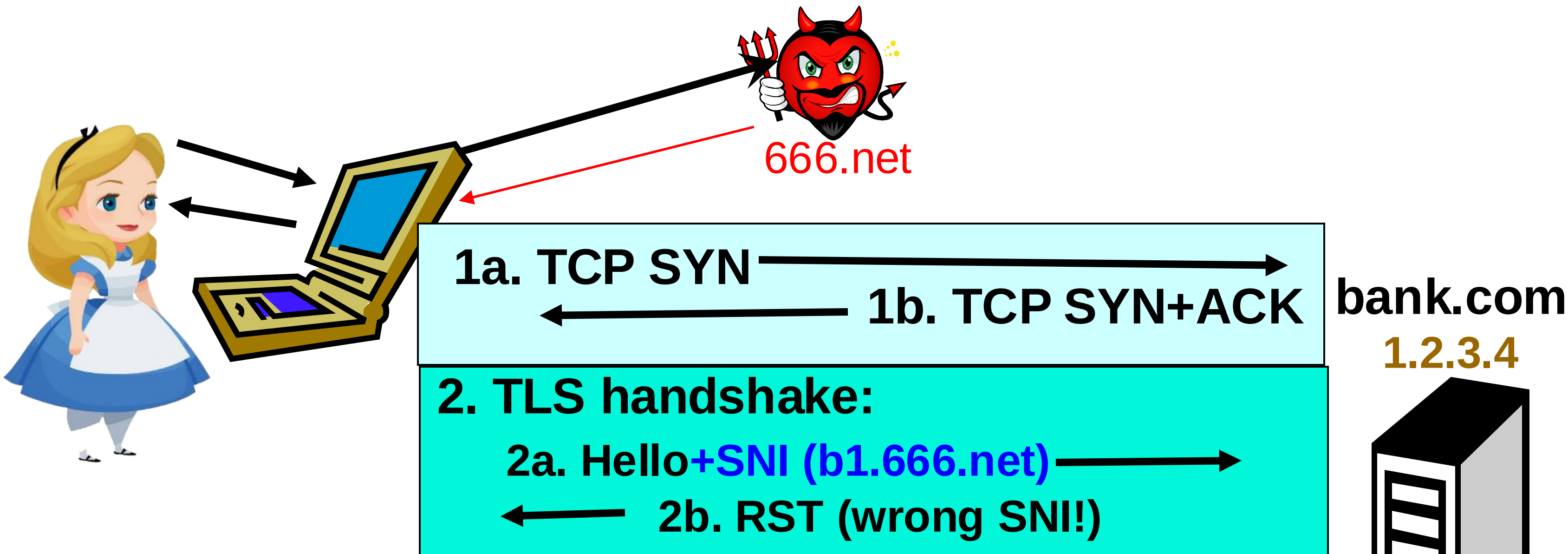
- Rogue site sends rogue-page to client
  - Page opens many SSL/TLS connections to bob.org (how?)

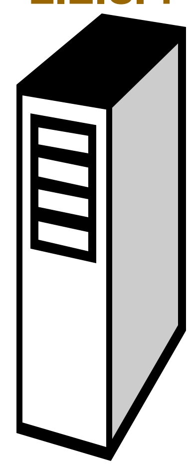


  - Separate hostnames ➔ separate connections, TLS sessions !
  - Wait, wouldn't the web server detect wrong domain and abort??
    - How? Using the http host header? It is sent **after** the TLS handshake
    - But, usually, the wrong domain is detected during TLS handshake, **before** the server does PK operations. [In TLS versions > 1.0]
    - Why and how? Anybody heard the term **SNI**?

SNI foils Cross site TLS-handshake CPU DoS

- Attack page opens many connections: to 1.666.net, 2.666.net, …
- Attacker's NS maps all of them to victim's IP (e.g., 1.2.3.4)
- ClientHello indicates hostname in Server Name Indication (SNI) extension

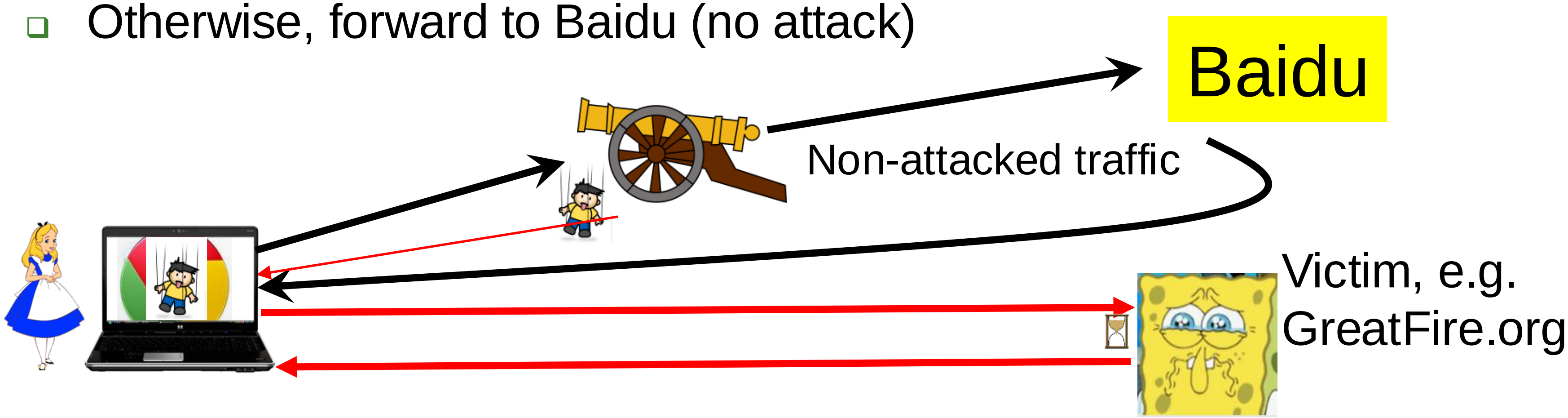




- Attack works for SSL, TLS 1.0, or if client and server support **Encrypted Client Hello (ECH)** extension
  - ECH supported by Chrome, FF, Safari – and important websites incl. CloudFlare
  - ECH motivated to ensure privacy, prevent censorship

The Great Cannon: Cross-Site via MitM

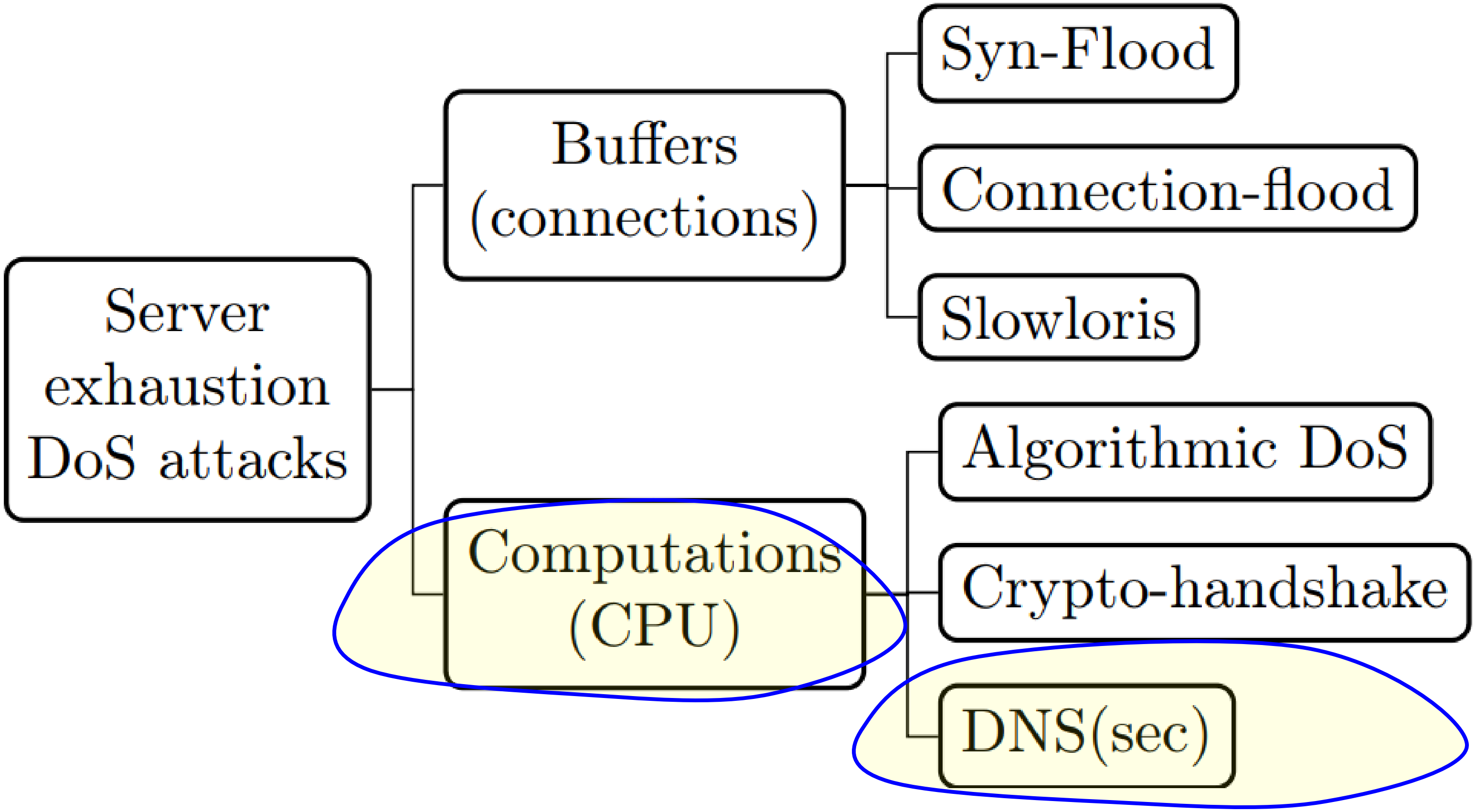
- The great cannon inspects traffic to Baidu website
  - If from desired-IPs, (with x%) drops and sends puppet as response
  - Otherwise, forward to Baidu (no attack)



- Puppet: a mal-script that does DoS attack against victim
  - If victim/client do not deploy SNI **or** both support ECH, use one of the cross-site TLS handshake CPU exhaustion attacks
  - E.g., a loop that repeatedly loads a web page/object from the victim, or performs a search
    - Making sure request is not in client’s or proxy’s cache (how?)

Server resource exhaustion DoS Attacks

- Exhaust server resources:
  - Connections, storage, computation, other



Algorithmic DoS on DNS(sec) Resolvers

- Idea: DNS(sec) queries may cause high load on resolver
- Attacking an open resolver; or: a non-open resolver, by:
  - A zombie using the resolver, or
  - A user using the resolver and visiting attacker's website
  - Connection to SMTP server which uses the resolver
  - A website using the resolver with pw-recovery/domain validation
- Attacks abusing DNSsec (against resolver supporting it):
  - KeyTrap: exploit the redundancy of key-tags
  - Proof-of-Non-Existence (NSEC3) DoS (non-existing subdomains)
    - This attack can prevent attribution!
- Attack abusing DNS (for resolver not supporting DNSsec):
  - NRDelegation: many NS-referrals to non-responding servers
    - Many resolvers have high overhead handling NS-referrals
    - Cause extensive resolver CPU (and network) overhead

Algorithmic DoS on DNSsec Resolvers

- Idea: DNSsec queries may cause high load on resolver
  - DNSsec: via (one or many) signature validations for queries
- KeyTrap: exploit the redundancy of key-tags
  - DNSsec uses 16-bit key-tags to match key, signature and hash
    - RRSIG contain signature; verification key identified by key-tag
    - DNSKEY contain verification key, with its key-tag
    - DS are hash to authenticate key, identifying it with a key-tag
    - RFC: if multiple keys have same key-tag, try them all!
    - RFC: if there are many signatures/hashes, try them all!
  - KeySigTrap: many DNSKEYs and signatures, same key-tag

.	86400 IN DNSKEY 256 3 5 (… …) ; (tag=1127)
com.	86400 IN RRSIG DS 5 1 86400 (… **1127** …)

Algorithmic DoS on DNSsec Resolvers

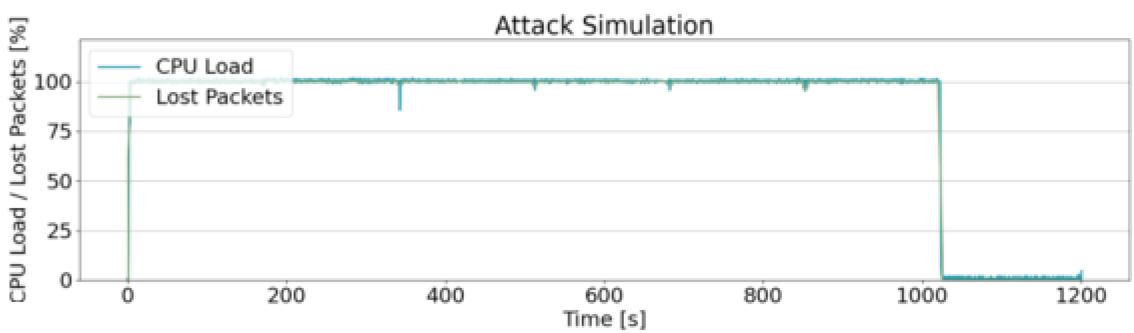
- Idea: DNSsec queries may cause high load on resolver
- KeyTrap: exploit the redundancy of key-tags
  - DNSsec uses 16-bit key-tags to match key, signature and hash
    - RRSIG contain signature; verification key identified by key-tag
    - DNSKEY contain verification key, with its key-tag
    - DS: hash authenticates delegated key, identifying it with a key-tag
    - RFC: if multiple keys have same key-tag, try them all!
    - RFC: if there are many signatures/hashes, try them all!

666.org	86400 IN DNSKEY 256 3 5 (…<pk1> …) ; (tag=1127)
666.org	86400 IN DNSKEY 256 3 5 (…<pk2> …) ; (tag=1127)
666.org	86400 IN DNSKEY 256 3 5 (…<pk3> …) ; (tag=1127)
1.666.org	6400 IN RRSIG DS 5 1 86400 (… 1127 666.org <sig1>…)
1.666.org	6400 IN RRSIG DS 5 1 86400 (… 1127 666.org <sig1>…)
1.666.org	6400 IN RRSIG DS 5 1 86400 (… 1127 666.org <sig1>…)

How many Signature validations?

Algorithmic DoS on DNSsec Resolvers

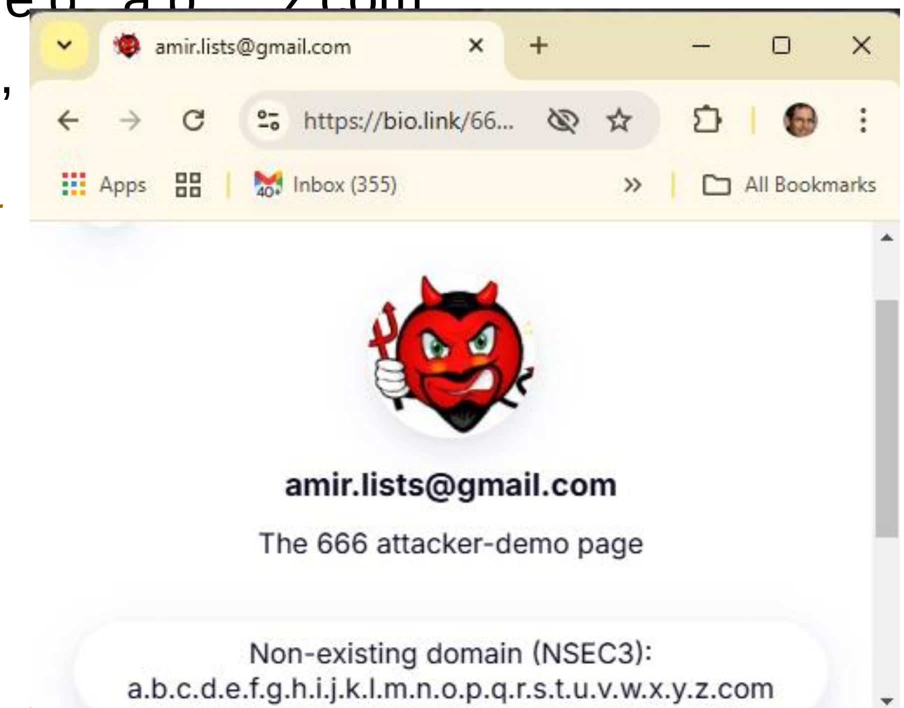
- Idea: DNSsec queries may cause high load on resolver
- KeyTrap: exploit the redundancy of key-tags
  - DNSsec uses 16-bit key-tags to match key, signature and hash
    - RRSIG contain signature; verification key identified by key-tag
    - DNSKEY contain verification key, with its key-tag
    - DS are hash to authenticate key, identifying it with a key-tag
    - RFC: if multiple keys have same key-tag, try them all!
    - RFC: if there are many signatures/hashes, try them all!
  - KeySigTrap: many DNSKEYs and signatures, same key-tag



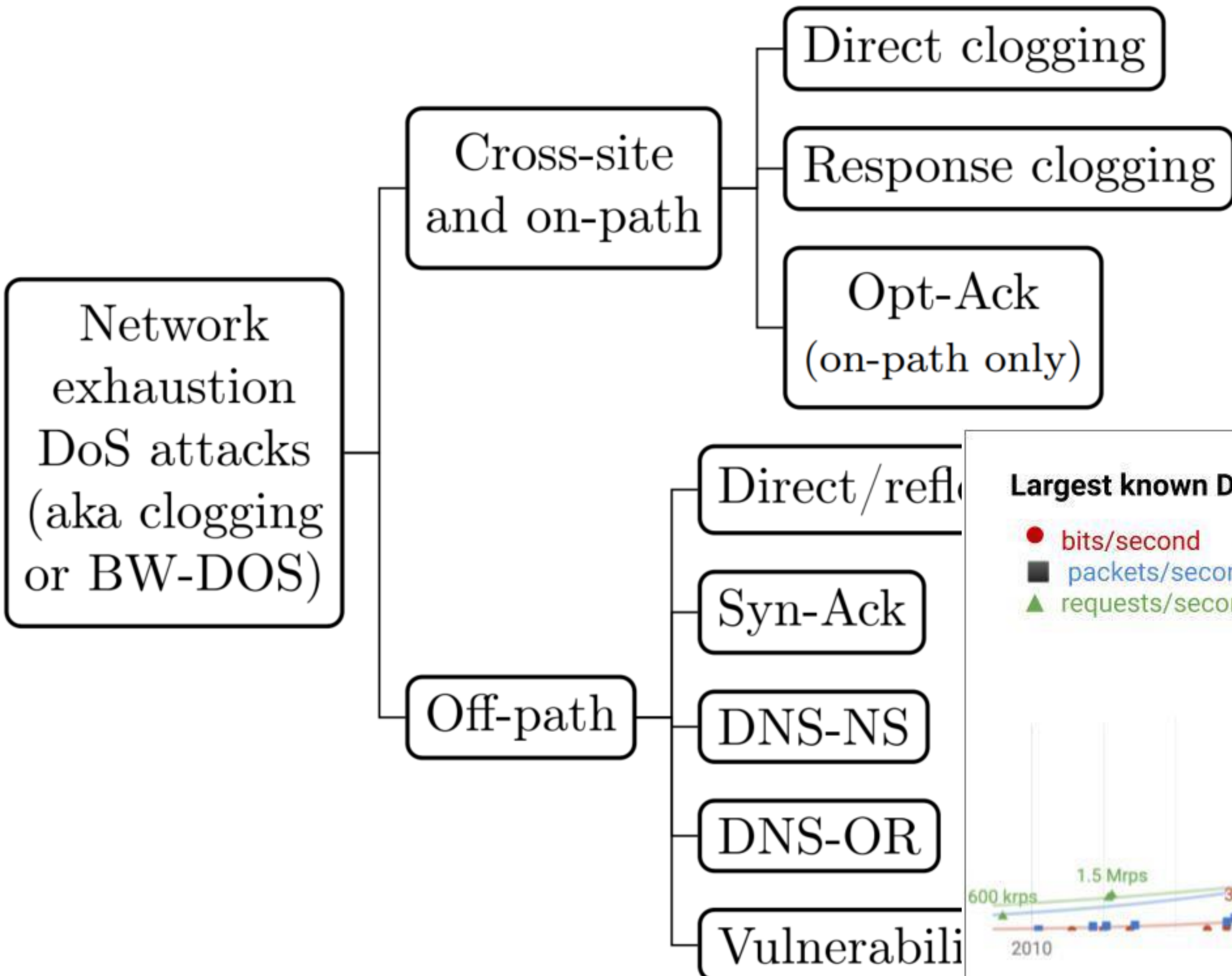
KeySigTrap attack on Unbound with single request.

Another Algorithmic DoS on DNSsec Resolvers

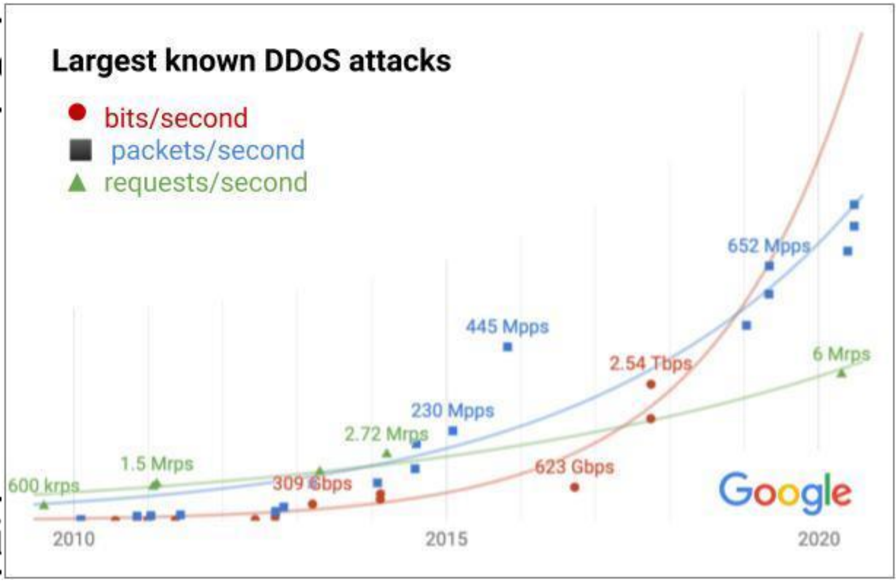
- Proof-of-Non-Existence (NSEC3) algorithmic DoS attack
- Query for a ‘deep’ subdomain: a.b.c….z.666.com
- Resolver should validate NSEC3 for any suffix (e.g., x.y.z.666.com)
- Use large salt, hash-iterations to increase overhead
- Or: queries to benign domains, e.g., a.b.…z.com
  - To further prevent attribution,
    host site in a legit domain
    - E.g., http://bio.link/666net



Network Exhaustion DoS attacks
aka **Clogging** or Bandwidth-DoS (**BW-DoS**)



Victims (routers, servers, switches) are more limited in req/sec than in pkts/sec, and even more than in bits/sec



Clogging → High Delay and Loss-rate

- Consider a link under BW-DoS (clogging)
- Traffic (in) rate R_{in} > R_{out} output rate
  - Hence, (at most) R_{out} out of R_{in} is delivered
  - Loss probability: 1 - R_{out} / R_{in}
    - Attack even better using short packets; output rate more restricted, and higher probability of short (attack) packets to fill queue, long (legit) packets more likely to be discardedR_{in} > R_{out}
    - This is simplified analysis. More precise analysis, using queuing theory, shows losses also when R_{in} < R_{out}.
- Queue (Q) is mostly full ➔ delay $\cong \frac{|Q|}{R_{out}}$
- What's the impact on end-to-end application performance?
  - Assume running TCP
  - QUIC has similar behavior

Impact of Congestion on **TCP** Transfer

- TCP BW bounded by **congestion control** and **reliability**
- Simplified bound on End-to-End (E2E) bandwidth:

$$R^{E2E} < \frac{MSS}{E2Ed} \cdot \sqrt{\frac{2}{3p}}$$

- Under BW-DOS:
  - Loss probability and delay dominated by congested link
  - Losses are common: $p \rightarrow 1$
  - E2Ed $\cong \frac{|Q|}{R_{out}}$ where Q and R_{out} are of congested link
  - Typically, $R^{E2E} << R_{out}$!!

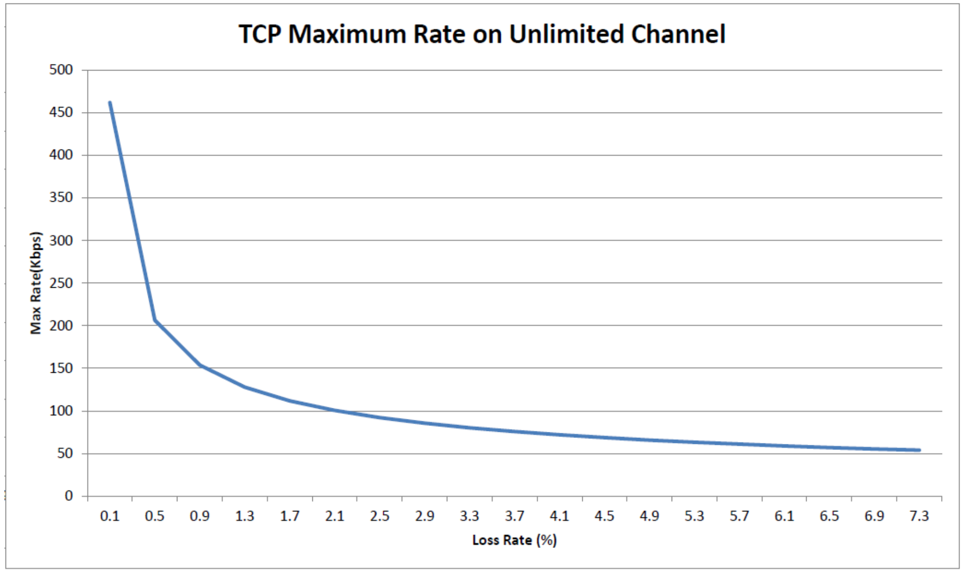
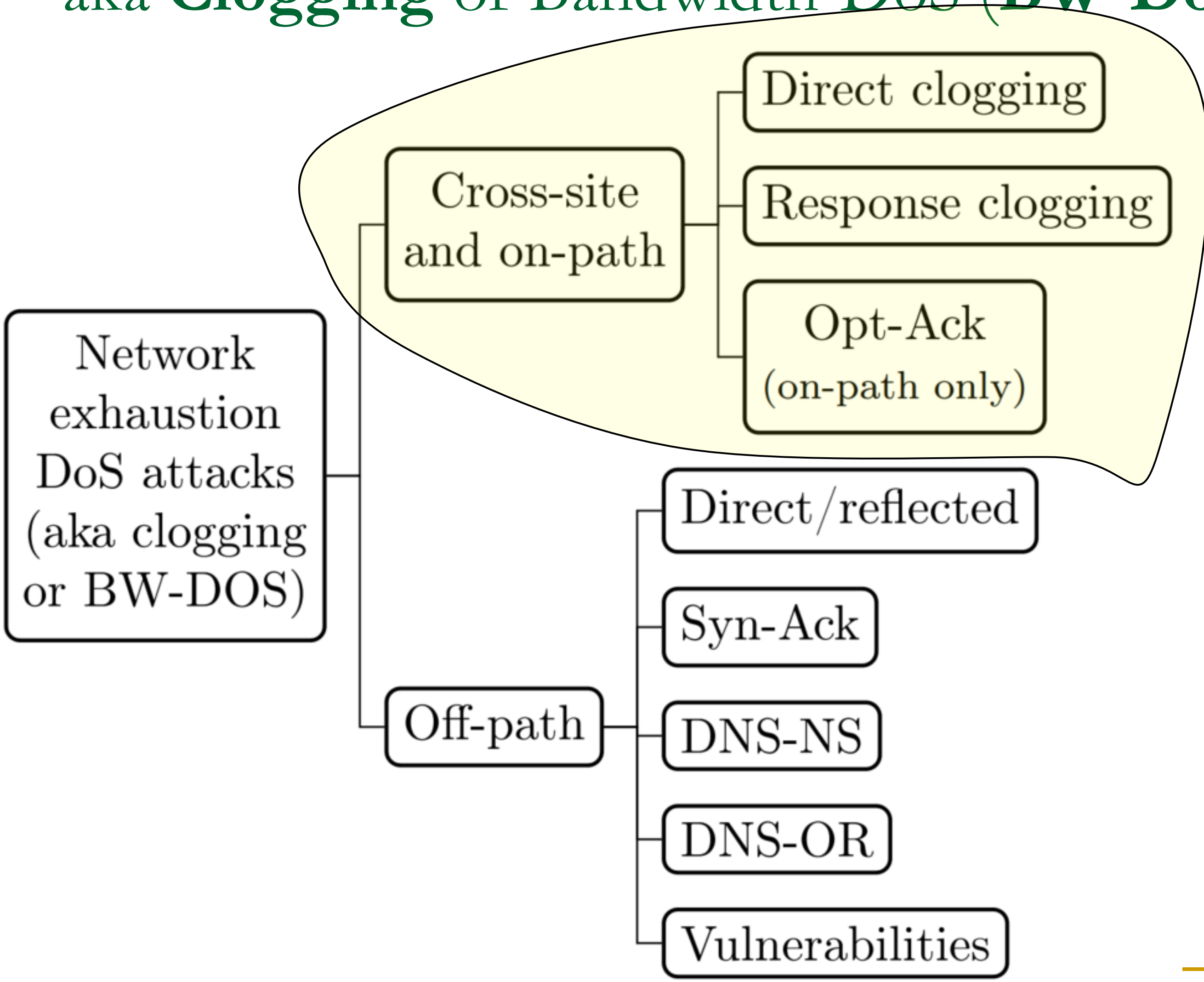


Figure 5.4: Maximum TCP Throughput under losses. Calculated with MSS=1460 bytes and RTT=0.1 sec.

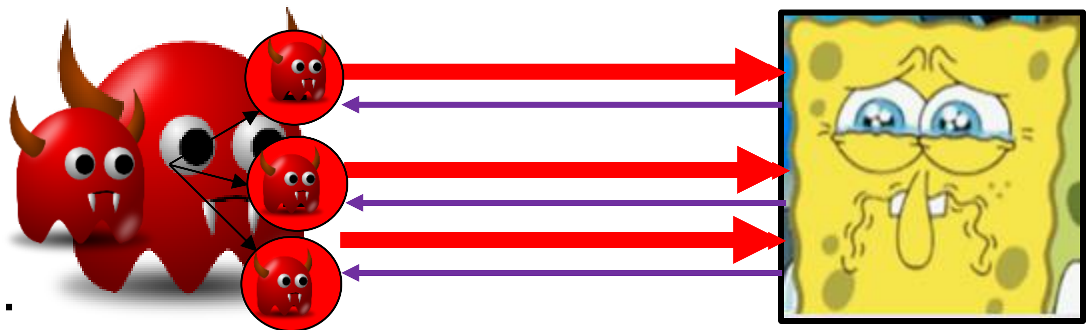
Network Exhaustion DoS attacks

aka **Clogging** or Bandwidth-DoS (**BW-DoS**)



Direct Clogging BW- DDoS Attacks

- Direct Clogging attacks: attacker sends traffic to victim
- On-path Direct Clogging (using attacker's IP) ?
- Pros (vs. off-path):
  - 😎 Easy coding
  - 😎 Any ISP [vs.?]
  - 😎 Can do handshake
- Cons (vs. off-path):
  - ☹ Blacklisting & filtering (quotas)
  - ☹ Attribution



Bots may be pwned PCs, phones, **IoT devices** or **cloud instances**

- **Cross-site clogging:** clog by visitors of attacker's site
  - Similar advantages, and site can't blacklist / filter
  - Limited to visitors of attacker's site; browser limits to 4-8 connections, TCP limits by congestion control.

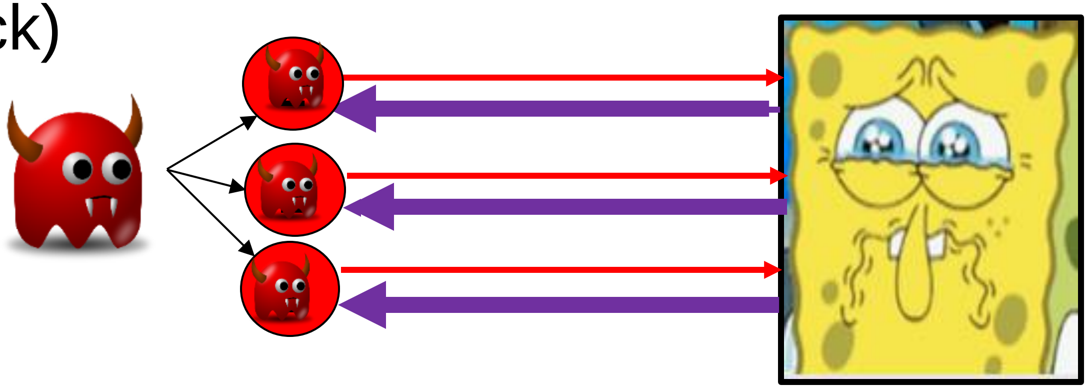
A variant uses DNS to avoid these limits…

Cross-site **DNS** Request Clogging DoS

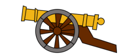
- Browsers (almost) don't limit number of DNS resolutions
- Variant 1: attacker maps different domains to victim
  - Same as in Cross-Site TLS-handshake CPU exhaustion
    - Often less impact – but works even when TLS attack fails
- Variant 2: attacker cause traffic to victim from resolver, by causing browser to resolve x1.y.666.net, x2.y.666.net
- Response: referral (NS) to victim's IP addresses
  - Up to 13 NS records for y.666.net, all with glue to victim's IP
  - Or, multiple IP addresses for each NS name, in same subnet
- Request fails (it's not a DNS server!); often, fails silently → resend (10 to 16 times)
- Repeat: queries to other subdomains of this domain

Response-Clogging BW-DDoS Attacks

- Up-link (from bots to Internet/server) usually has much less BW than down-link (from Internet/server to bots)
- Attack can be more effective by generating large responses
- Again, can be done by bots – or by visitors of attacker's site (cross site attack)
- What is better?



Response Clogging: On-path vs. Cross-site

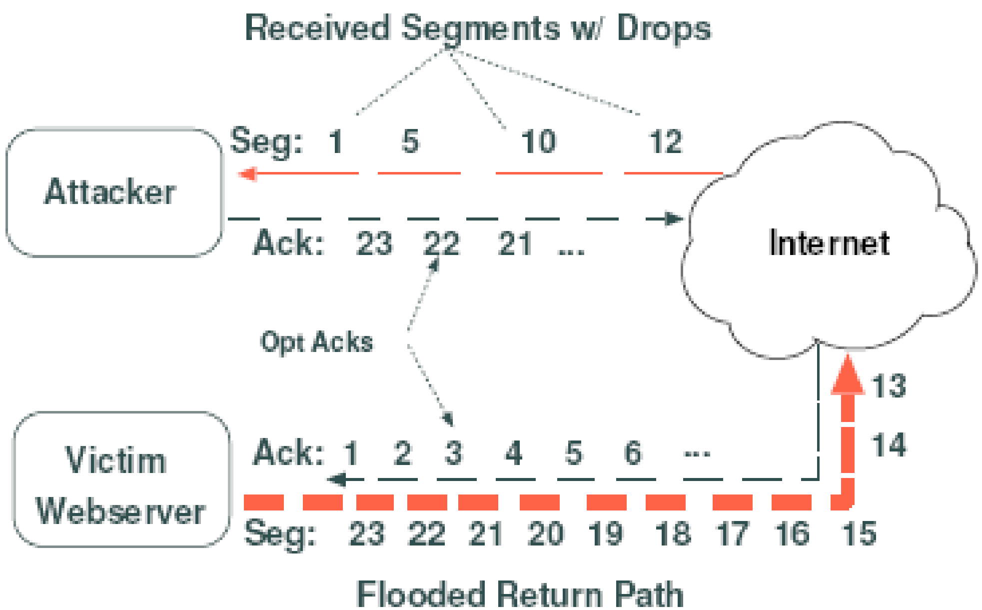
- Cross-site Response Clogging: rogue website [or Great  Cannon] causes browser to request many objects from victim website
  - 😎 Pro: use customer's BW; can't filter attacker's IP
  - ☹ Cons: need to attract customer to rogue site; attribution of attack site
    - Use 'refer-policy: same-origin' to make it harder to identify website
  - ☹ Limited by browser to 4-8 connections
  - ☹ Limited by TCP's congestion-control – slow down upon congestion
    - Can cause much traffic from victim's DNS server, if using DNSsec [how?]
- On-path Clogging (using IP of attacker or of bot/zombie):
  - ☹ Cons: blacklisting & filtering, use attacker's or bot's BW
  - 😎 Pros: no need to attract customer; attribution is only of (bot's?) IP, not of attacker's domain/site
  - 😎 Bot is not limited by browser, TCP: can send at line speed
    - But what about responses? Are these limited by congestion control??

On-path **Response** Clogging from TCP server

- Goal: congest the **response** path (from server to bot)
  - Using only limited bot resources (CPU, BW)
  - In particular, not requiring bot to receive all of the response traffic
- Challenge: server runs TCP (i.e., congestion, flow control)
- How? Bot tricks server into sending faster than path allows:
  - Bot signals huge receive window ➔ server's flow control ineffective
  - Bot sends **Opt-Ack**s: circumvents server's congestion control

Opt-Ack: response-clogging breaking congestion control

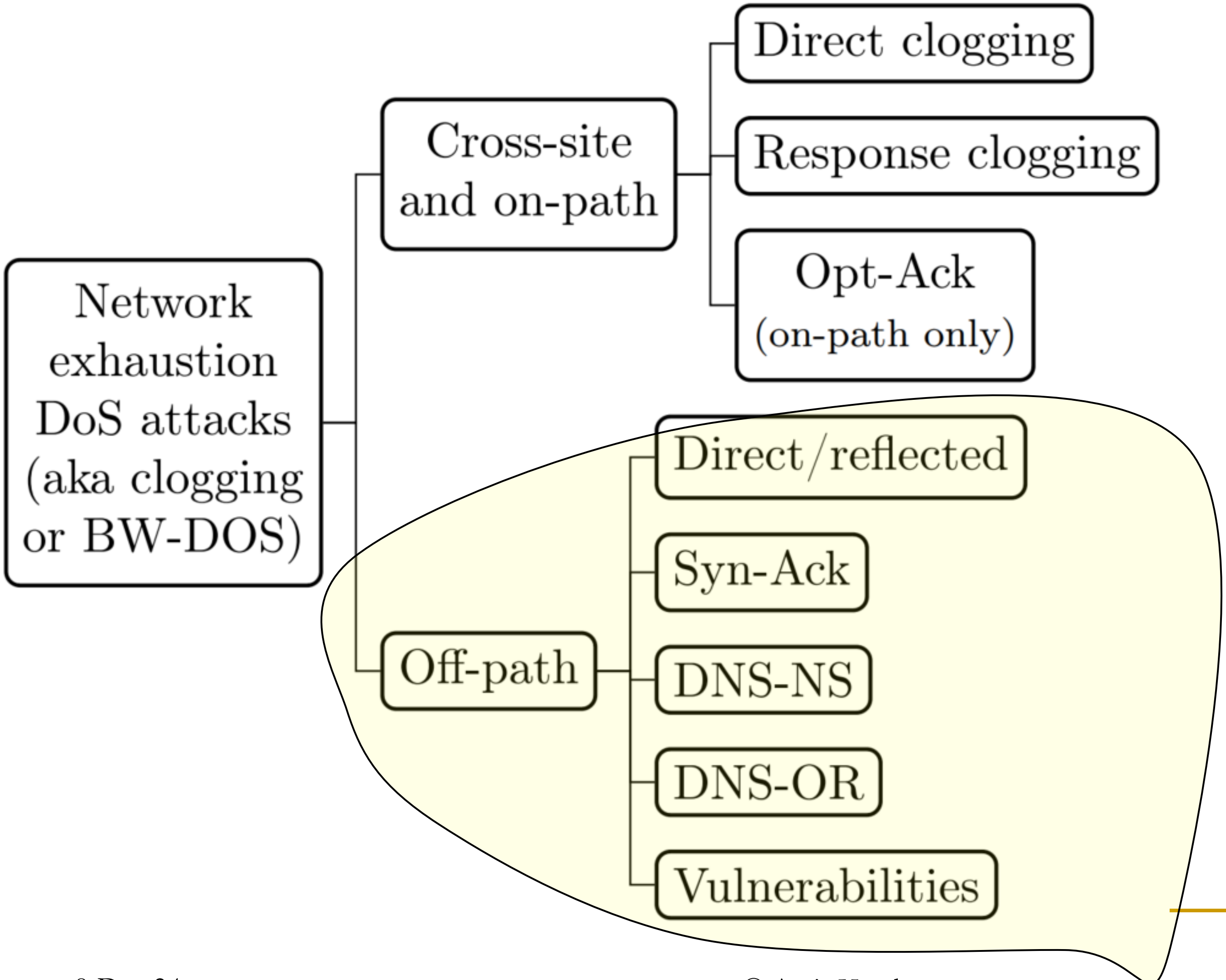
- **On-path** clogging of response path
- Attacker (bot) request lots and lots of data from server
- Server limits sending rate by congestion+flow controls
- Attacker tricks server to send faster than attacker receives:
  - Signals huge receive window ➔flow control ineffective
  - Sends Ack incorrectly:
    - TCP: cumulative Ack
    - Attacker: optimistic Ack, i.e., Ack for packets not yet received!!
    - TCP ignores ack for not-yet-sent packets
- High amplification:
  - Theoretical: huge…
  - Measured: 261



Defenses against Opt-Ack

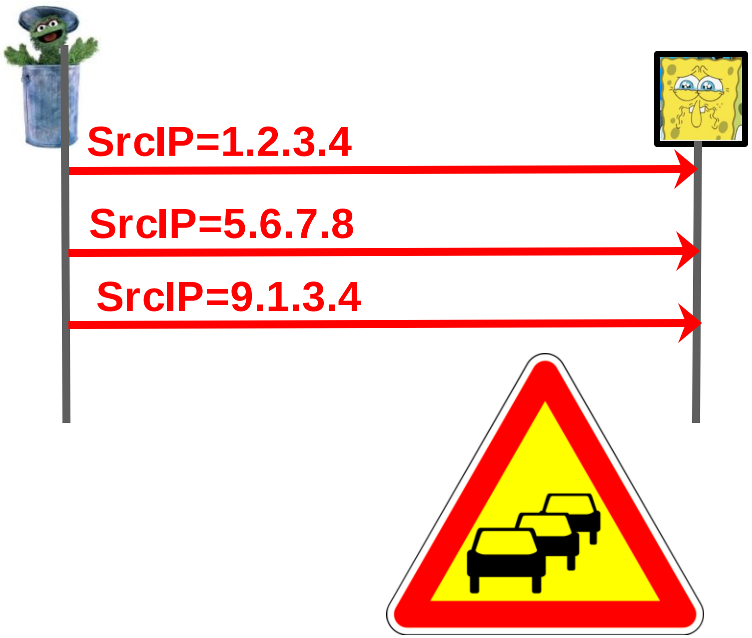
- Server rate-limitations (per client)
- Server drops some packets (randomly)
  - Detect misbehaving client: commulative-ack…
  - Problem: rate reduction (false positive)
    - Send `skipped packet' after sending `enough' later packets
      - Identify attack by not receiving correct number of dup-acks
      - Send `enough' later packets since a few dup-ack packets may be lost
    - Server will not reduce window (not real loss!)
    - May lose notification of `real' event
- Or: use ECN signaling
  - Skipped

Network Exhaustion DoS attacks
aka **Clogging** or Bandwidth-DoS (**BW-DoS**)

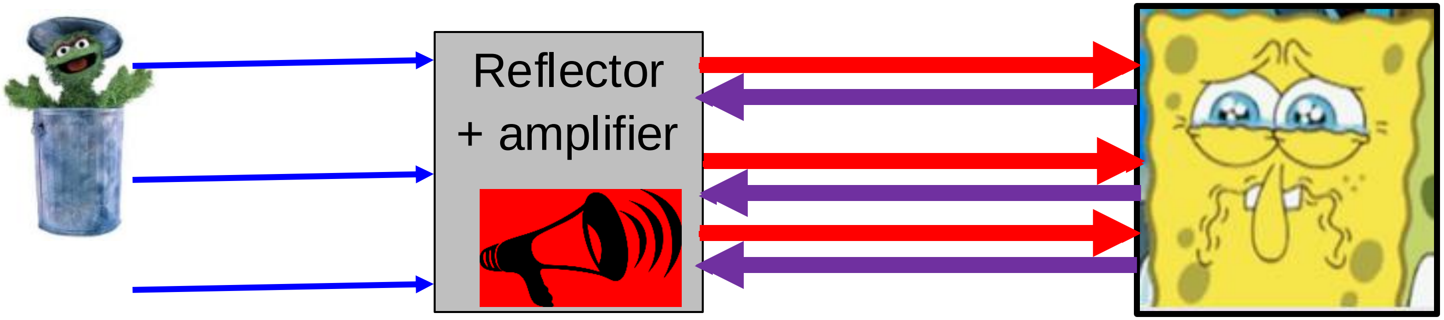


Off-path Direct Clogging (BW-DoS)Attacks

- Off-path clogging (using spoofed IP)
  - 😎 Pros: avoids per-IP filtering, attribution*
  - ☹ Cons: requires admin/root
  - ☹ May be filtered by ingress-filtering and other uRPF/SAV filters
  - ☹ No interaction (does not receive responses)



Off-path Reflection/Amplification Attacks

- **Reflected** off-path clogging is harder to prevent, trace back, filter
  - Traffic from reflector isn't even spoofed (it uses valid src IP)
- 

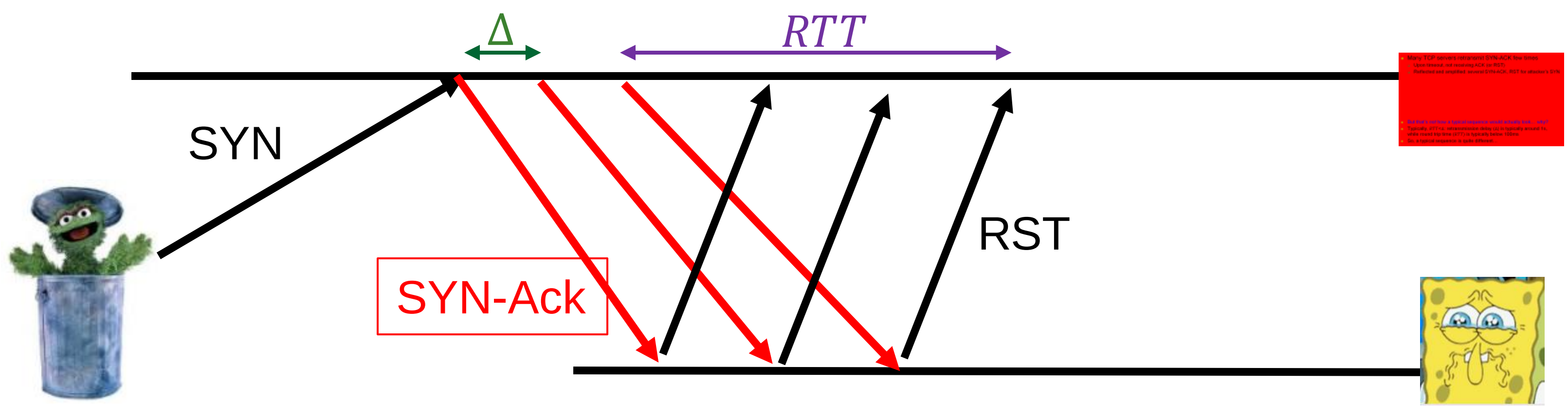
- Even better: reflection with response-clogging
- Much better: **amplifying** BW-DoS (clogging) attack

Q: A packet(s) that Oscar can send to cause reflection (usually amplification too) from any TCP server?

A: SYN-ACK

The SYN-ACK BW-DoS Attack

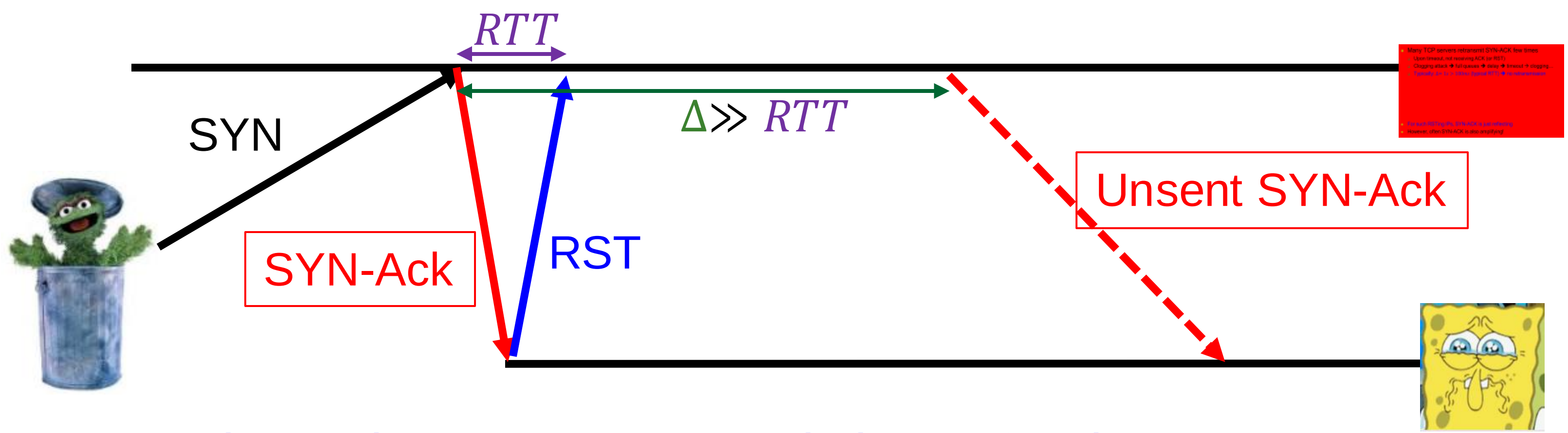
- Most TCP servers retransmit SYN-ACK few times
  - Upon timeout, not receiving ACK (or RST); helps if ACK is lost
  - Reflected and amplified: several SYN-ACK, RST for attacker's SYN



- But that's *not* how a typical sequence would actually look… why?
- Typically, RTT<Δ: retransmission delay (Δ) is typically around 1s, while round trip time (RTT) is typically below 100ms
- So, a typical sequence is quite different…

The SYN-ACK BW-DoS Attack

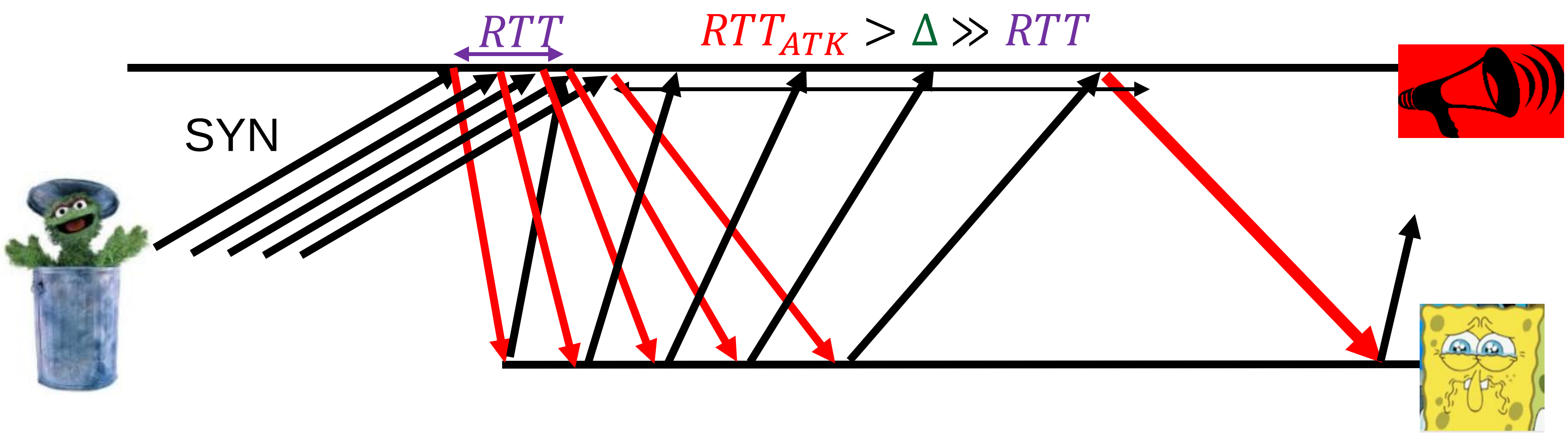
- Many TCP servers retransmit SYN-ACK few times
  - Upon timeout, not receiving ACK (or RST)
  - Clogging attack ➔ full queues ➔ delay ➔ timeout → clogging…
  - Typically: $\Delta = 1s > 100ms$ (typical RTT) ➔ no retransmission



- For such RSTing IPs, SYN-ACK is just reflecting
- However, often SYN-ACK is also amplifying!

Amplifying SYN-ACK BW-DoS Attack

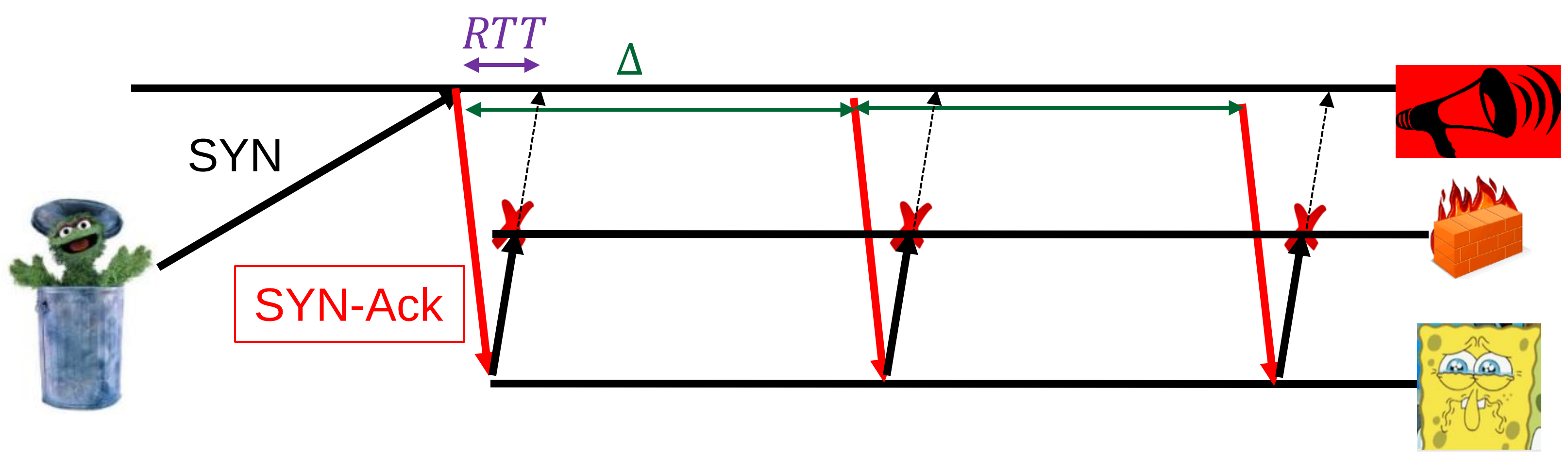
- Many TCP servers retransmit SYN-ACK few times
  - Upon timeout, not receiving ACK (or RST)
- When would SYN-ACK also be amplifying?



- Option 1: attack traffic causes full queues ➔ $RTT_{ATK} > \Delta$ (retransmit)
- Option 2: RST is dropped (why would RST be dropped??)

Amplifying SYN-ACK BW-DoS Attack

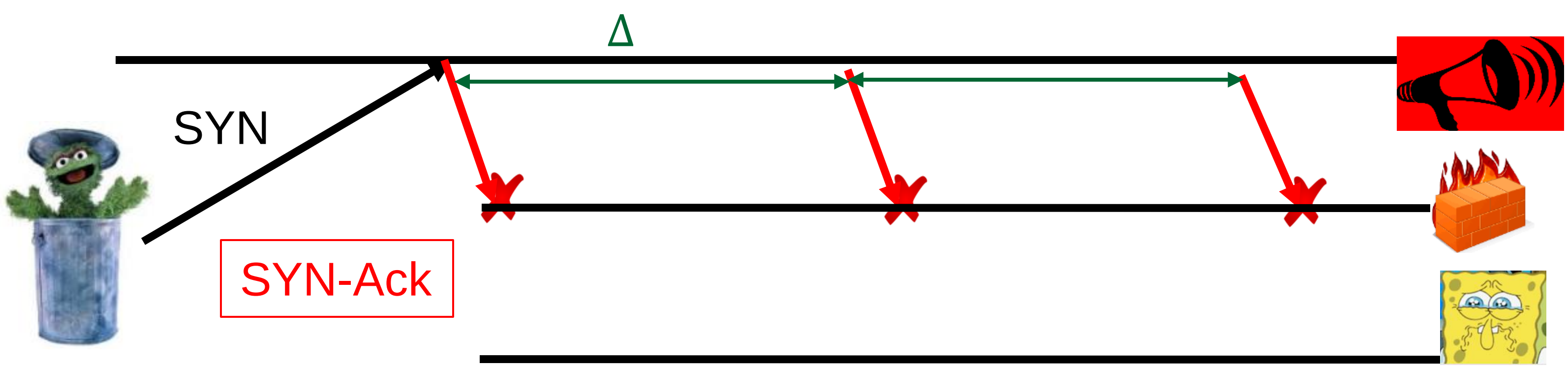
- Many TCP servers retransmit SYN-ACK few times
  - Upon timeout, not receiving ACK (or RST)
- When would SYN-ACK also be amplifying?



- Option 1: attack traffic causes full queues ➔ $RTT_{ATK} > \Delta$ (retransmit)
- Option 2: when RST is dropped (by FW/ISP, to save upstream BW)
- Option 3: ?

Amplifying SYN-ACK BW-DoS Attack

- Many TCP servers retransmit SYN-ACK few times
  - Upon timeout, not receiving ACK (or RST)
- When would SYN-ACK also be amplifying?
  - Typically, 10 or 16 attempts to send SYN-ACK (amplification)



- Option 1: attack traffic causes full queues ➔ $RTT_{ATK} > \Delta$ (retransmit)
- Option 2: when RST is dropped (by FW/ISP, to save upstream BW)
- Option 3: when **SYN-ACK** is dropped/ignored: blackholed or filtered (NAT/FW; DMZ, DNS server: allow only incoming connections)
  - Advantage: prevents response clogging

What is a 'good' amplifying service?

- Reflecting: does not validate source-IP
  - Mostly: stateless **UDP** services
    - Also: ICMP, some broken TCP implementations,…
  - Not deploying effective anti-spoofing filtering (SAV, uRPF)
- Availability:
  - Many servers and locations, high bandwidth
- **Hard to filter**
- **High Amplification:** |responses|>|requests|
  - **Number and/or length** of responses vs. requests
  - Response (upstream) clogging: (almost) only for SYN-ACK?
  - **'Amplification factor' ?**

Amplification Factors

- Packet amplification factor (PAF):

$$PAF = \frac{\#Packets_{AMP \to Vic}}{\#Packets_{Zombies \to AMP}}$$

- Bandwidth amplification factor (BAF):

$$BAF = \frac{|Payload|_{AMP \to Vic}}{|Payload|_{Zombies \to AMP}}$$

- Header-inclusive BAF (HiBAF):

$$HiBAF = \frac{|Packets|_{AMP \to Vic}}{|Packets|_{Zombies \to AMP}}$$

- Simplified: Amplification factor <u>should</u> also consider losses, backoff
  - Losses: attack traffic dropped due to congestion
  - Backoff: e.g., TCP's congestion control response to loss

Reflecting-Amplifying UDP Services

- Often abused: DNS, NTP and SSDP (why?)
  - From 'Amplification Hell', Rossow, NDSS'14 [so: outdated]

Protocol	#Amplifiers	BAF			PAF	Method
		All	Top 50%	Top 10%		
SNMP v2	4,832,000	6.3	8.6	11.3	1.00	GetBulk request
NTP	1,451,000	556.9	1083.2	4670.0	3.84	Request client statistics
DNS NS	255,819 (*1404)	54.6	76.7	98.3	2.08	ANY lookup at author. NS
DNS OR	7,782,000	28.7	41.2	64.1	1.32	ANY lookup at open resolv.
NetBios	2,108,000	3.8	4.5	4.9	1.00	Name resolution
SSDP	3,704,000	30.8	40.4	75.9	9.92	SEARCH request
CharGen	89,000	358.8	n/a	n/a	1.00	Character generation request
QOTD	32,000	140.3	n/a	n/a	1.00	Quote request
BitTorrent	5,066,635	3.8	5.3	10.3	1.58	File search
Kad	232,012	16.3	21.5	22.7	1.00	Peer list exchange
Quake 3	1,059	63.9	74.9	82.8	1.01	Server info exchange
Steam	167,886	5.5	6.9	14.7	1.12	Server info exchange
ZAv2	27,939	36.0	36.6	41.1	1.02	Peer list and cmd exchange
Sality	12,714	37.3	37.9	38.4	1.00	URL list exchange
Gameover	2,023	45.4	45.9	46.2	5.39	Peer and proxy exchange

Reflecting-Amplifying UDP Services

- Most abused @ 2014: DNS, NTP and SSDP (why?)

Protocol	#Amplifiers	BAF			PAF	Method
		All	Top 50%	Top 10%		
~~SNMP v2~~	4,832,000	6.3	8.6	11.3	1.00	GetBulk request
NTP	1,451,000	556.9	1083.2	**4670.0**	3.84	Request client statistics
DNS NS	255,819	54.6	76.7	**98.3**	2.08	ANY lookup at author. NS
DNS OR	7,782,000	28.7	41.2	**64.1**	1.32	ANY lookup at open resolv.
~~NetBios~~	2,108,000	3.8	4.5	4.9	1.00	Name resolution
SSDP	3,704,000	30.8	40.4	**75.9**	9.92	SEARCH request
Char						n request
QOT						
~~BitT~~						
~~Kad~~						
Qual						ge
~~Stea~~						ge
~~ZAv~~						xchange
~~Salit~~						
~~Gam~~						hange

More recent:

- **Memcached**: amplification up to 51,000
  - A simple, UDP caching / file download protocol
  - Adversary spoofs requests for huge files (it uploaded)
- **Mitel PBX-to-Internet gateway: 3/2022**
  - One packet generates 4,294,967,296 packets
  - 53Mpps, 23GBps, from 5 minutes to hours…

TCP Amplification? (beyond SYN-ACK)

- Some (buggy?) TCP stacks allow off-path amplification!
- TCP state machine is complex, underspecified, buggy
  - E.g., in Listen state, how to handle packet with SYN+FIN ?
- SYN-FIN Amplification:
  - Off-path attacker sends a spoofed SYN+FIN pkt
    - Standard-conforming TCP never sends SYN+FIN packet
      - Normally, FIN received in Established state, causing server to send back FIN+ACK and wait for final ACK (LAST-ACK state)
    - Standard does not define how to handle such SYN+FIN
  - Most implementations drop SYN+FIN, but others:
    - Send SYN+ACK, send FIN+ACK, wait for ACK (not received)
      - Retransmit FIN+ACK on timeout (multiple times → **amplification**)
      - Until `connection time-out` …
  - Other buggy TCP implementations amplify even more….

(broken) TCP amplifiers can be bad!

Protocol	# Responsive	# Amplifiers with amplification factor > 20	> 50	> 100	> 500	> 1,000	> 2,500
FTP	152,026,322	2,913,353	3,500	1,868	1,032	937	847
HTTP	149,521,309	427,370	15,426	6,687	1,596	649	347
NetBIOS	82,706,193	12,244	2,449	1,463	873	811	783
SIP	154,030,015	22,830	5,158	3,913	3,289	3,123	2,889
SSH	141,858,473	87,715	4,611	2,141	1,275	1,176	1,082
Telnet	126,133,112	2,120,175	16,469	7,147	2,008	1,393	994

Luckily, it is relatively easy to filter
Syn+Fin and other malformed TCP packets,
and (relatively) easy to remove buggy servers

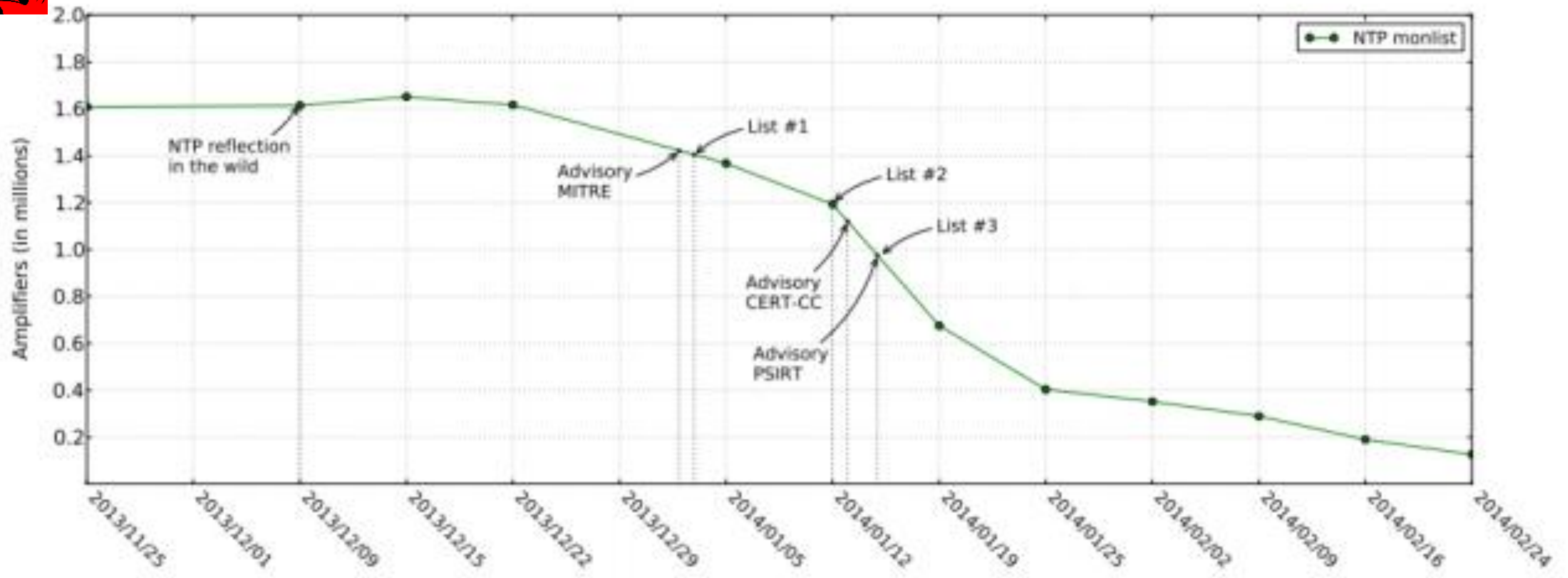
Header-inclusive BAF (HiBAF)

- NTP, DNS and SSDP have (also) high HiBAF!

Protocol	All			Top 50%			Top 10%			PAF	PL(B)
	BAF	HiBAF	Ratio	BAF	HiBAF	Ratio	BAF	HiBAF	Ratio		
SNMP v2	6.3	4.5	0.71	8.6	6.03	0.7	11.3	7.81	0.69	1	82
NTP	556.9	72.13	0.13	1083.2	137.92	0.13	4670	649.38	0.14	3.84	8
DNS NS	54.6	20.13	0.37	76.7	27.73	0.36	98.3	35.16	0.36	2.08	22
DNS OR	28.7	10.73	0.37	41.2	15.03	0.36	64.1	22.9	0.36	1.32	22
SSDP	30.8	23.61	0.77	40.4	29.91	0.74	75.9	53.19	0.7	9.92	80
CharGen	358.8	6.26	0.02	n/a	n/a	n/a	n/a	n/a	n/a	1	1
QOTD	140.3	2.85	0.02	n/a	n/a	n/a	n/a	n/a	n/a	1	1
BitTorrent	3.8	3.16	0.83	5.3	4.23	0.8	10.3	7.79	0.76	1.58	104
Kad	16.3	7.95	0.49	21.5	10.32	0.48	22.7	10.86	0.48	1	35
Quake 3	63.9	15.64	0.24	74.9	18.22	0.24	82.8	20.07	0.24	1.01	15
Steam	5.5	2.75	0.5	6.9	3.28	0.48	14.7	6.19	0.42	1.12	25
ZAv2	36	9.67	0.27	36.6	9.82	0.27	41.1	10.94	0.27	1.02	16
Gameover	45.4	27.52	0.61	45.9	27.8	0.61	46.2	27.97	0.61	5.39	52

Defending against Amplification DDoS

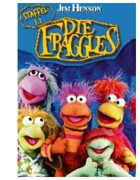
- De-Amplification of 'Buggy services'
  - A 'patched' server/service that does not amplify (as much)
  - The community patched 'buggy' NTP, TCP servers effectively!
- Specific product vulnerabilities (e.g., Mitel's): even faster



Defending against Amplification DDoS

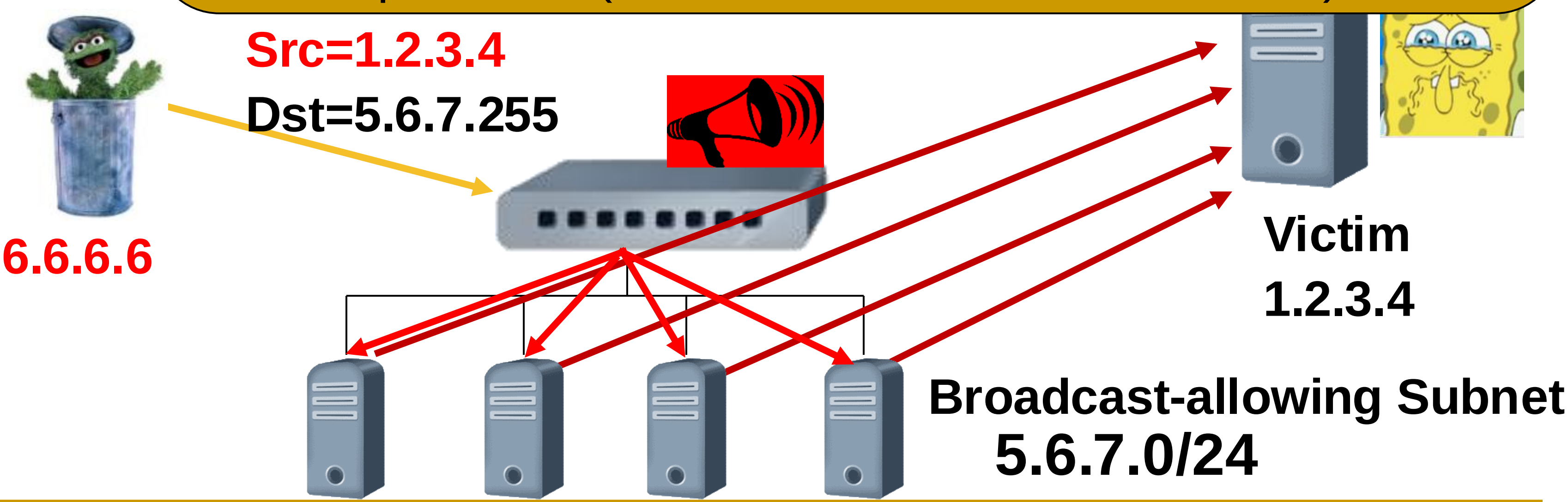
- De-Amplification of 'buggy services'
- De-Amplification of 'buggy network configurations'
  - Using 'standard filtering'
  - Example: the Smurf (ICMP)  and Fraggle (UDP) **Broadcast Amplification** attacks 

Broadcast Amplification BW-DoS attacks

- Abuse broadcast-allowing subnets:
  - Variants: Smurf (ICMP)  Fraggle (UDP) 
  - Osca
    sub
  - Rou
  - Rec
  - **Amp**

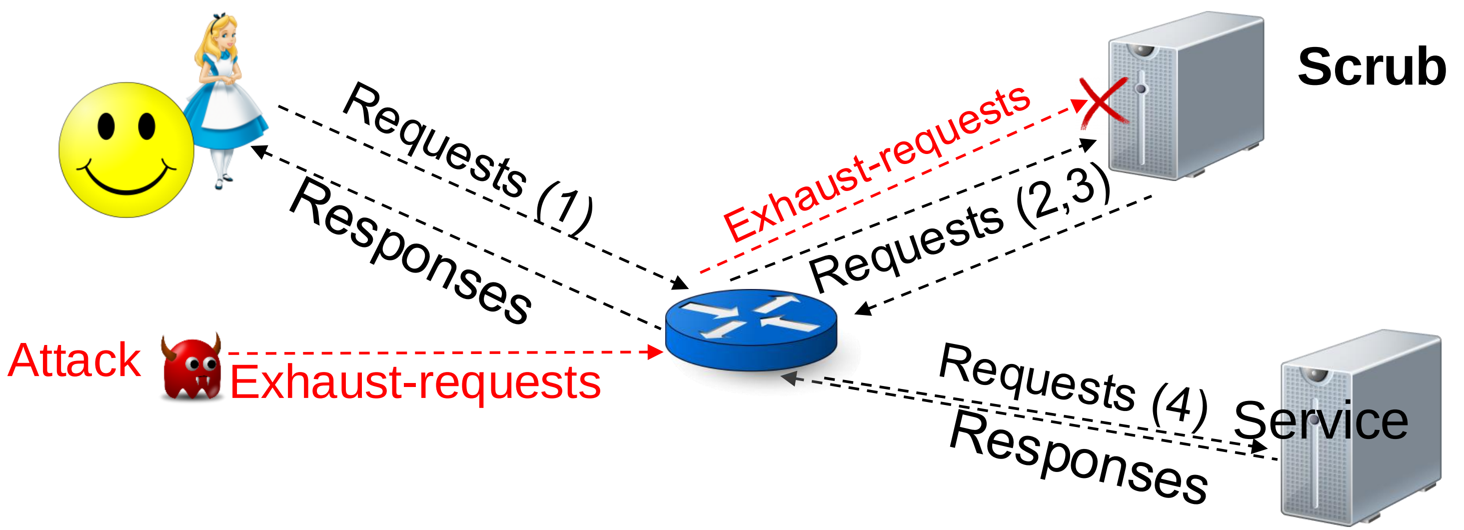
Amplifies by number of hosts in network (hundreds?), but…
Now, (almost?) all networks filter incoming broadcasts.

Amplification may work with few networks that do not. For several years, there were blacklists of networks that allowed such amplification (I doubt these are still maintained)



Amplification DDoS: Victim Defenses

- Main victim-based defenses: Filtering, and/or **Route to 'scrubbing service' and/or via CDN**
- Scrubbing service against BW-DDoS:
  - Works against bottleneck in path from router to host
  - **But not if bottleneck is in input queue to router!**
    - Let's zoom in on the network topology



Scrubbing of Amplification DDoS: How?

- Many amplification DDoS attacks are easy to scrub:
  - Drop/quota UDP (+ICMP?) traffic (except to services in use)
    - Possibly, only drop large and/or fragmented packets
  - Or: block traffic from **known amplification ports (services)**

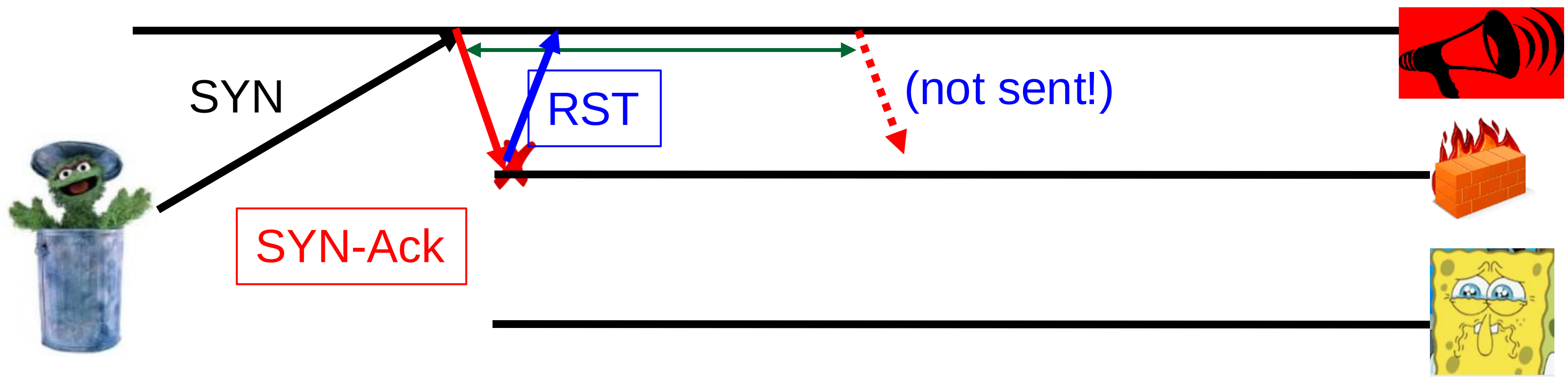
Amplifier	SSDP	NTP	Mem -cached	Mitel	DNS	Chargen	Bad TCP	Syn/Ack
Protocol	UDP						TCP	TCP
Src port	1900	123	11211	10074	53	19	(filter by flags)	Many
Amplify:	31	557	51000	$4 \cdot 10^9$	55*	359	25	10
Amplifiers?	Many					Few	Many	Many

* Amplification of up to 154 measured

Scrubbing is harder, if the protocol is in use: Syn/Ack and DNS

Scrubbing Amplification DDoS: How? (2)

- Many attacks are easy to scrub:
  - Drop/quota UDP (+ICMP?) traffic (except to services in use)
    - Possibly, only drop large and/or fragmented packets
  - Or: block traffic from **known amplification ports (services)**



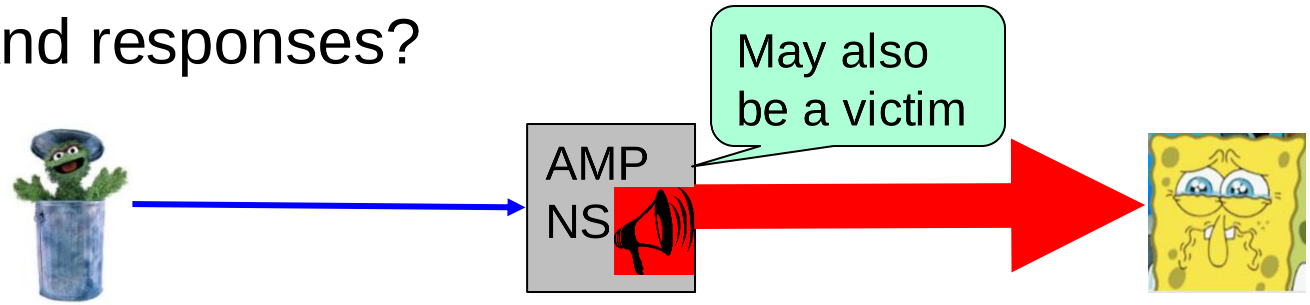
  - **Scrubbing is harder, if the protocol is in use: Syn/Ack and DNS**
    - **Stateful** FW / NAT can (should?) send RST for unsolicited Syn/Ack
    - **But what about DNS ?**

DNS Reflection-Amplification Attacks

- DNS: widely used, critical protocol ➔ hard to filter
  - Especially when victim is itself a DNS resolver (or hosts one)
  - Attacks are sometimes simply against the amplifier itself !
  - Quotas are ineffective:
    - Requests use spoofed src IP
    - Too many name servers and open resolvers to maintain quotas
- **DNS-NS: DNS Name Server Amplification**
- **DNS-OR: DNS Open Resolver Amplification**

DNS Name Server Amplification

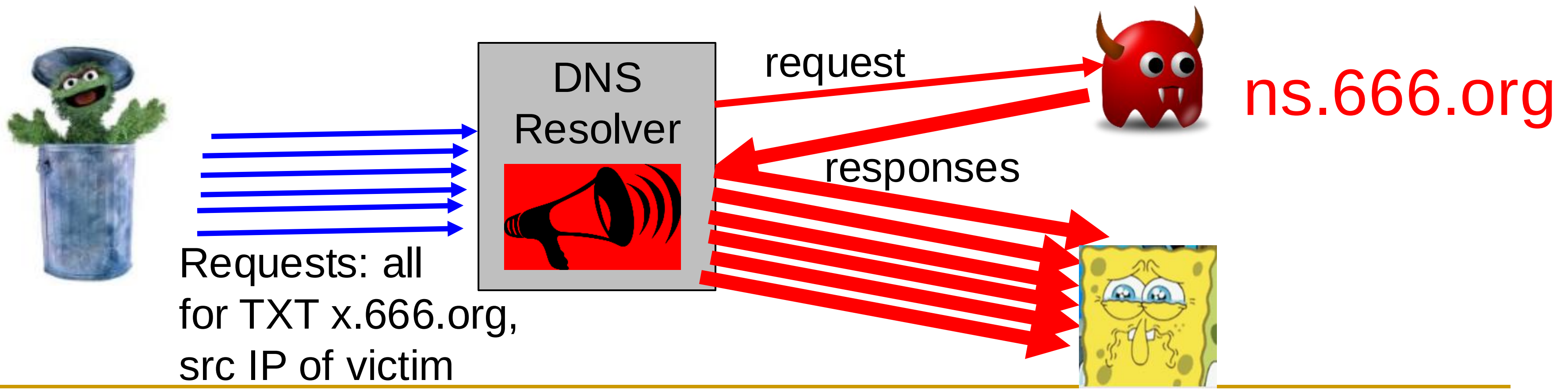
- DNS: reflects (UDP, stateless) & **amplifies**
- DNS requests can be short (64 bytes including headers!)
- And responses?



- Originally (w/o EDNS0): 512B
- With EDNS0 extension: up to 4096B (some even 8KB)
  - Motivation for EDNS0: DNSsec, ‘ANY’
- Few responses are <u>that</u> long
- But few suffice… any NS with DNSsec RRs will do !
  - So, we’ll discuss defenses; but first, open-resolver amplification

DNS Open Resolver Amplification

- Millions of DNS Open Resolvers…
  - Huge (bandwidth) but protected: OpenDNS, Google Pub-DNS…
  - Millions (low-bw but unprotected): home routers, etc…
- **Allow attacker to control <u>response</u>**
  - I.e., send max length response supported by resolver
    - Large variance in support; max measured amplification was 154
  - Response is <u>cached</u> – negligible 'cost' of sending



Example: Spamhaus Attack

- March 2013, 300GBps targeting Spamhaus.
- One attacker: teenager from London
  - A spammer, of course
- 10 compromised servers
- 3 networks that allow IP spoofing
- 9GBps DNS requests to 32,000 open DNS resolvers
- Worldwide disruption of Internet exchanges and services

- There are millions of open resolvers
- But many cap responses at 512B (no EDNS0 support)
- Attacker will scan to find the ‘good amplifying open resolvers’
- Current scans use Attacker’s own IP → allow attribution ??

Defenses from DNS off-path amplification

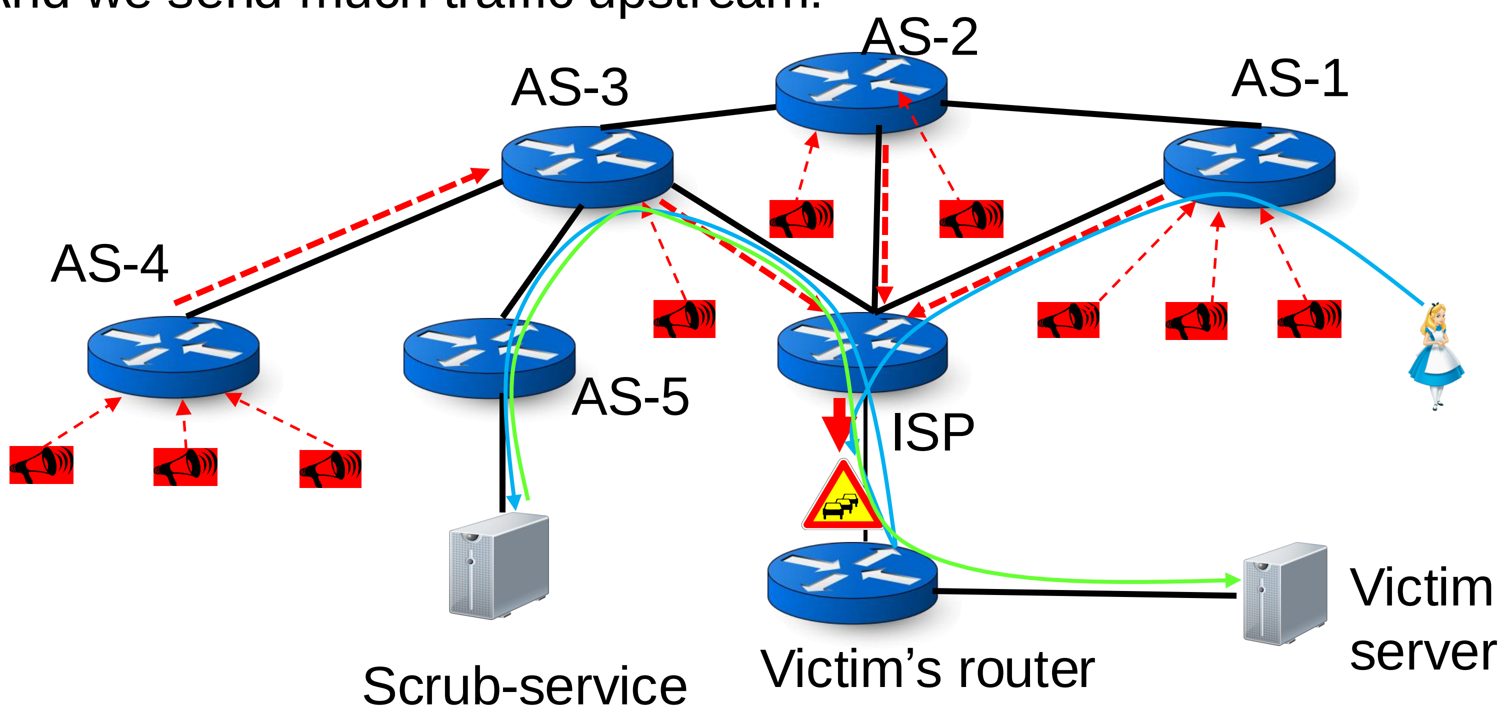
- DNS is needed and stateless filtering may prevent use
- Victim recipient's defense options:
  - Per-IP Rate-limit for DNS traffic (quotas)
    - Hard: many amplifiers (name servers, open resolvers), false positives
    - Open Resolver Project: 5M @ 15.3.2020
  - Filter DNS over UDP; performs DNS queries over TCP (or DoH or DoT)
- Name-server and (open) resolver defenses:
    - Goal: do not amplify – save resources of self and victims
  - Don't be open! allow requests only from known/legit resolvers
  - Quotas per IP/Prefix/origin-AS
  - Apply uRPF (and/or other anti-spoofing defenses)
  - Send long responses only over TCP (or DoH or DoT)
  - Challenge-response mechanism, e.g., CNAME-based (hidden)

Scrubbing defenses

- Common defense against server and BW exhaustion DoS attacks:
  - Upon detection of attack (or simply always)
  - Send traffic to a ‘scrubbing service’ that blocks known attacks
  - Also, often, run service in a scalable, harder-to-attack cloud/CDN environment
  - Often, these are provided by the same entity
- Let’s first discuss for BW-DoS…

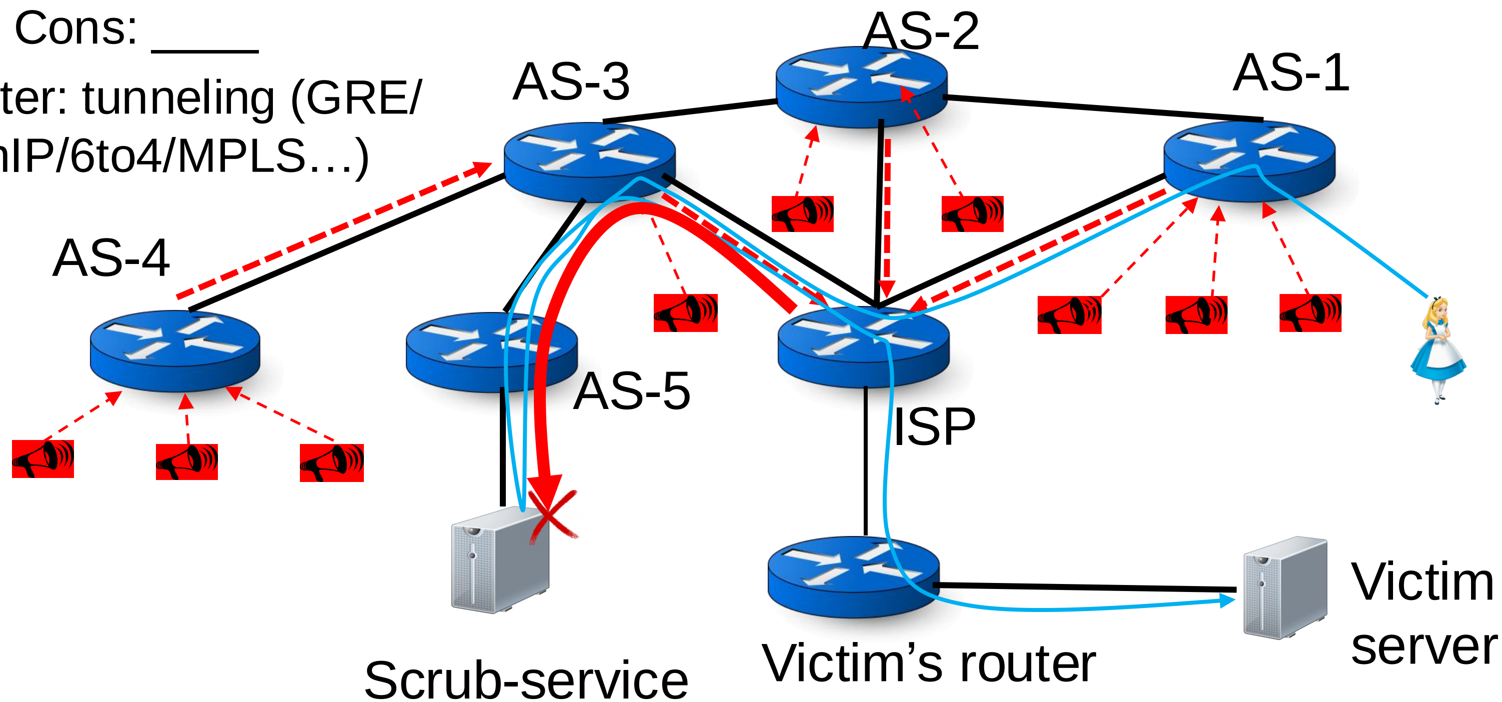
Anti-Clogging Service Deployment Challenge

- Typically, clogging is in access link (ISP to customer)
- Rerouting to scrub-service could make things worse:
  - Legit traffic would still need to pass bottleneck from ISP to router
  - In fact, need to pass it twice
  - And we send much traffic upstream!



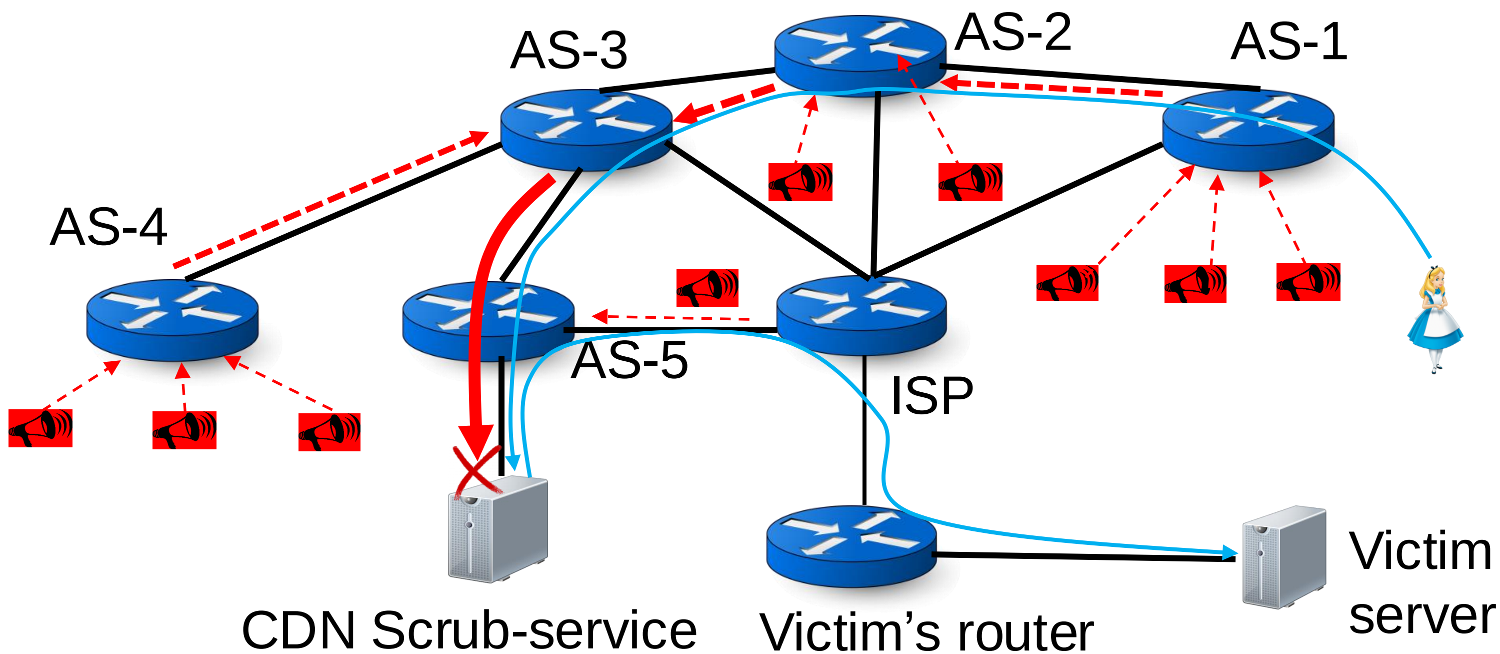
Anti-Clogging Services Deployment Solution (1)

- Typically, clogging is in access link (ISP to customer)
- Solution 1: ISP routes to scrub-service
  - Requires significant support from ISP; and ISP hit **twice** with pre-scrub traffic (receive, then relay to scrubber), once with post-scrub
  - ISP should allow legit-traffic from scrub-service; **how**?
    - IP? Cons: ____
    - Better: tunneling (GRE/ IPinIP/6to4/MPLS…)



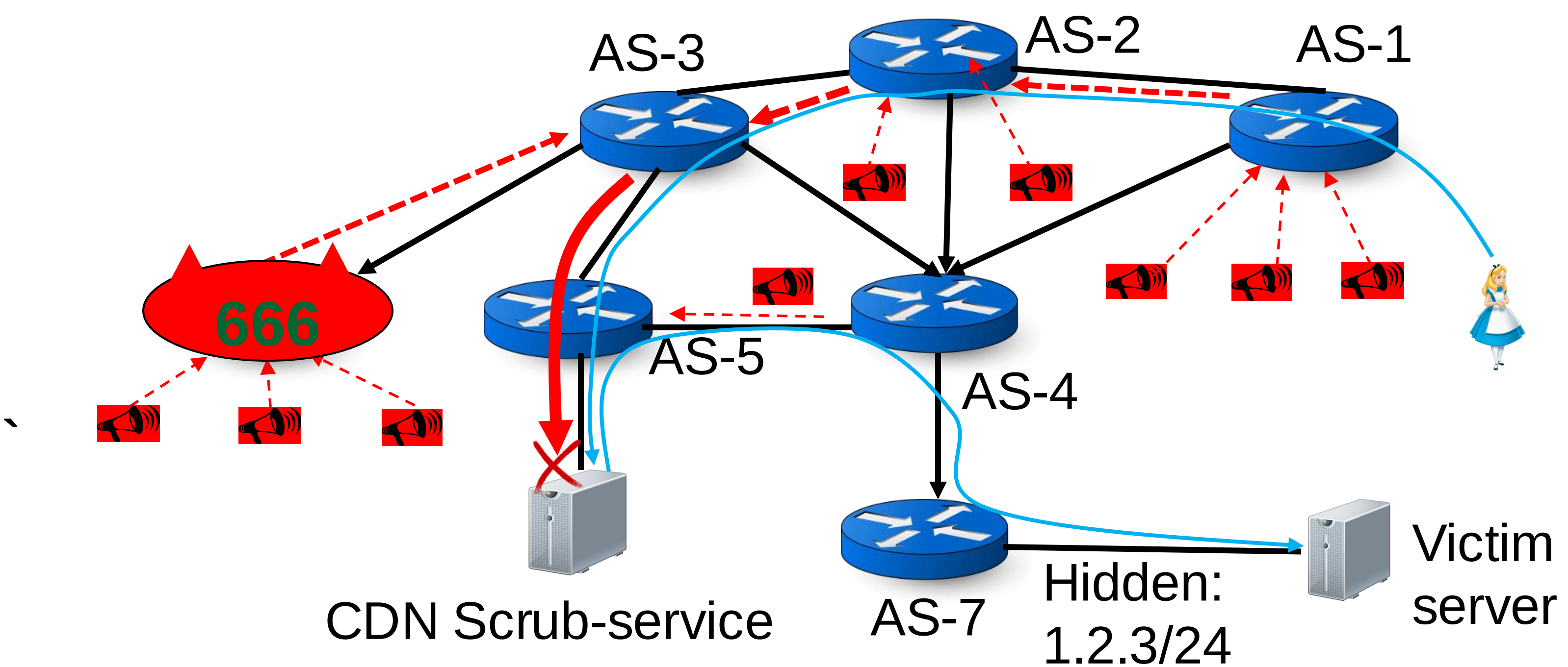
Anti-Clogging Services Deployment Solution (2)

- Typically, clogging is in access link (ISP to customer)
- Solution 2: redirect traffic to scrubber (use DNS/BGP), tunnels scrubbed traffic to hidden IP of victim
  - Pro: no need for support from ISP
  - Cons: (1) overload, costs to scrubber, (2) attacker may find victim's IP



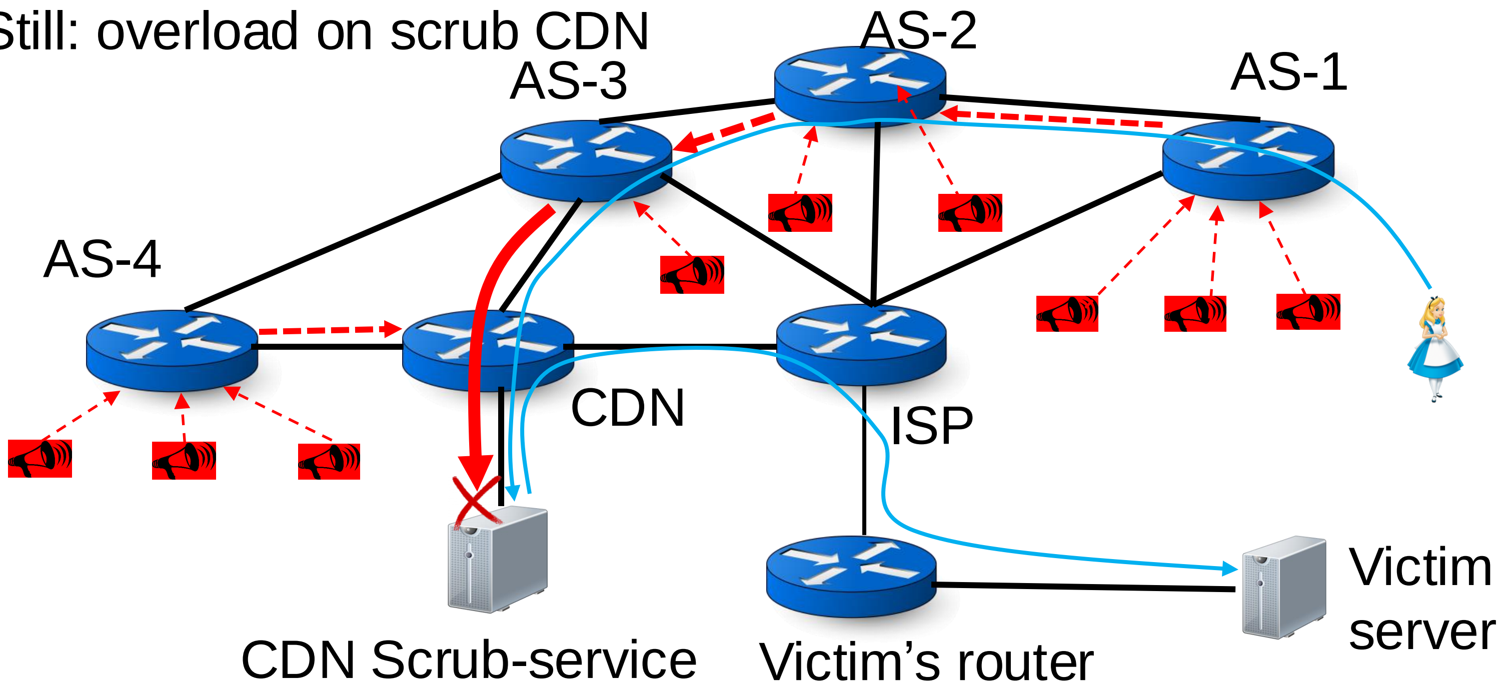
Anti-Clogging Services Deployment Solution (2)

- Solution 2: redirect traffic to scrubber (use DNS/BGP), tunnels scrubbed traffic to hidden IP of victim
- Exercise 1: show how attacker (AS666) can find hidden IP
- Exercise 2: how victim (AS-7) can prevent this



Anti-Clogging Services Deployment Solution (2*)

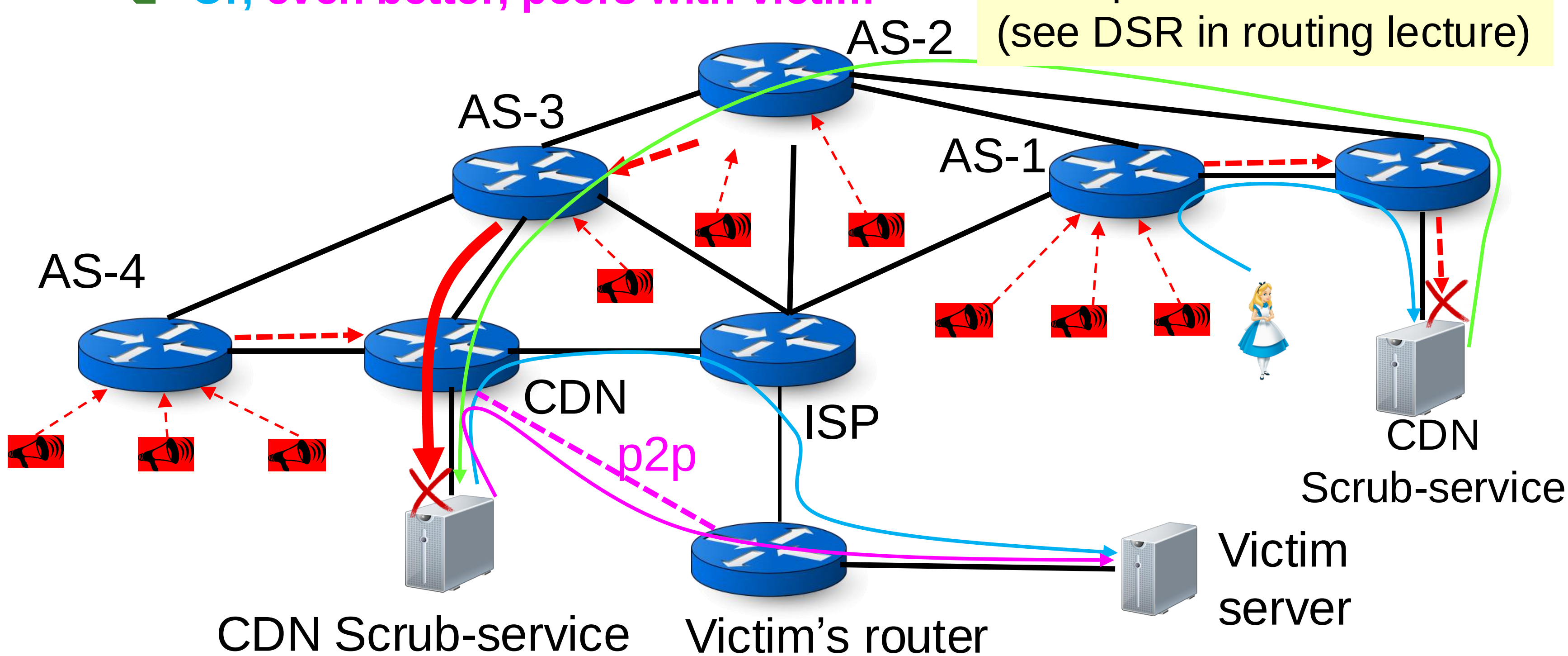
- Solution 2*: redirect traffic to **scrubbing CDN** (use DNS/BGP), tunnels scrubbed traffic to hidden IP of victim
  - **CDNs peer with many ISPs → reduced costs**
  - Announce hidden prefix only to CDN (community/prepend), **or simple filtering by ISP avoids need of hidden IP**
  - Still: overload on scrub CDN



Anti-Clogging Services Deployment Solution (2**)

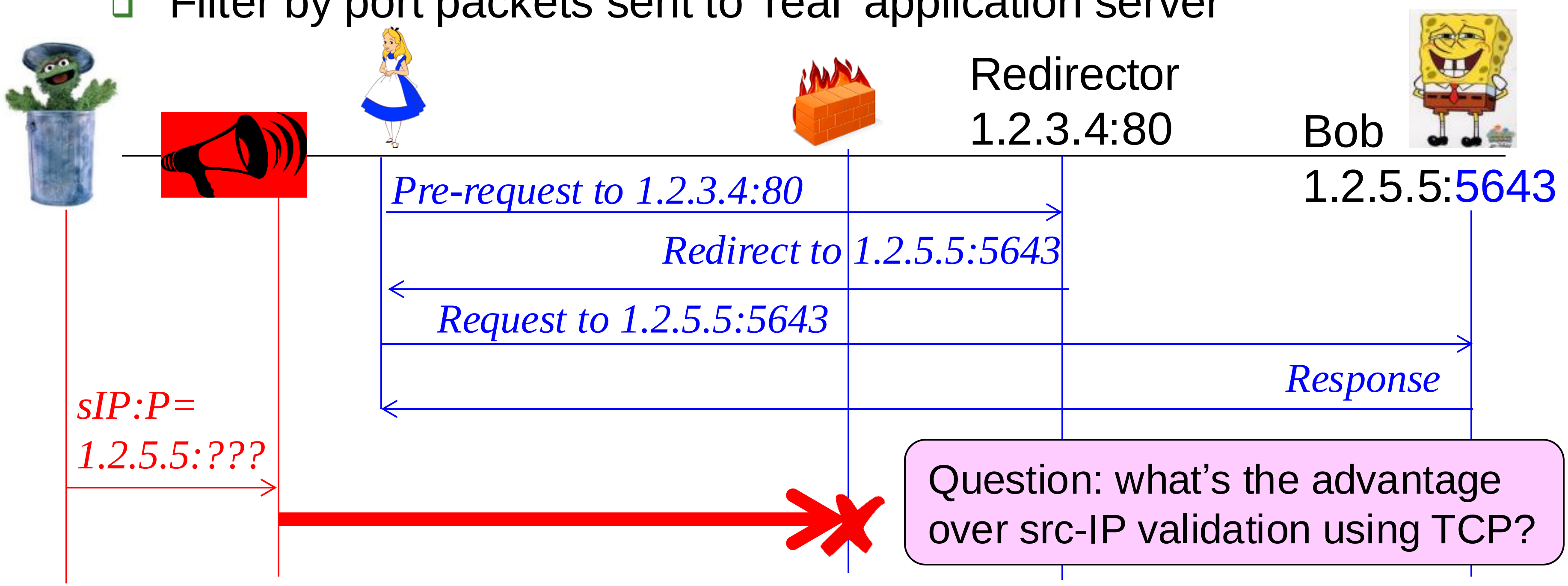
- Solution 2**: CDN receives victim's traffic (use DNS/BGP), scrubs, tunnels legit traffic to (hidden) IP of victim
  - CDNs have many Points of Presence → further reduces costs
    - Remote CDN PoPs tunnels to CDN PoP which peers with ISP
      - Or, even better, peers with victim

Note: potential uRPF issue (see DSR in routing lecture)



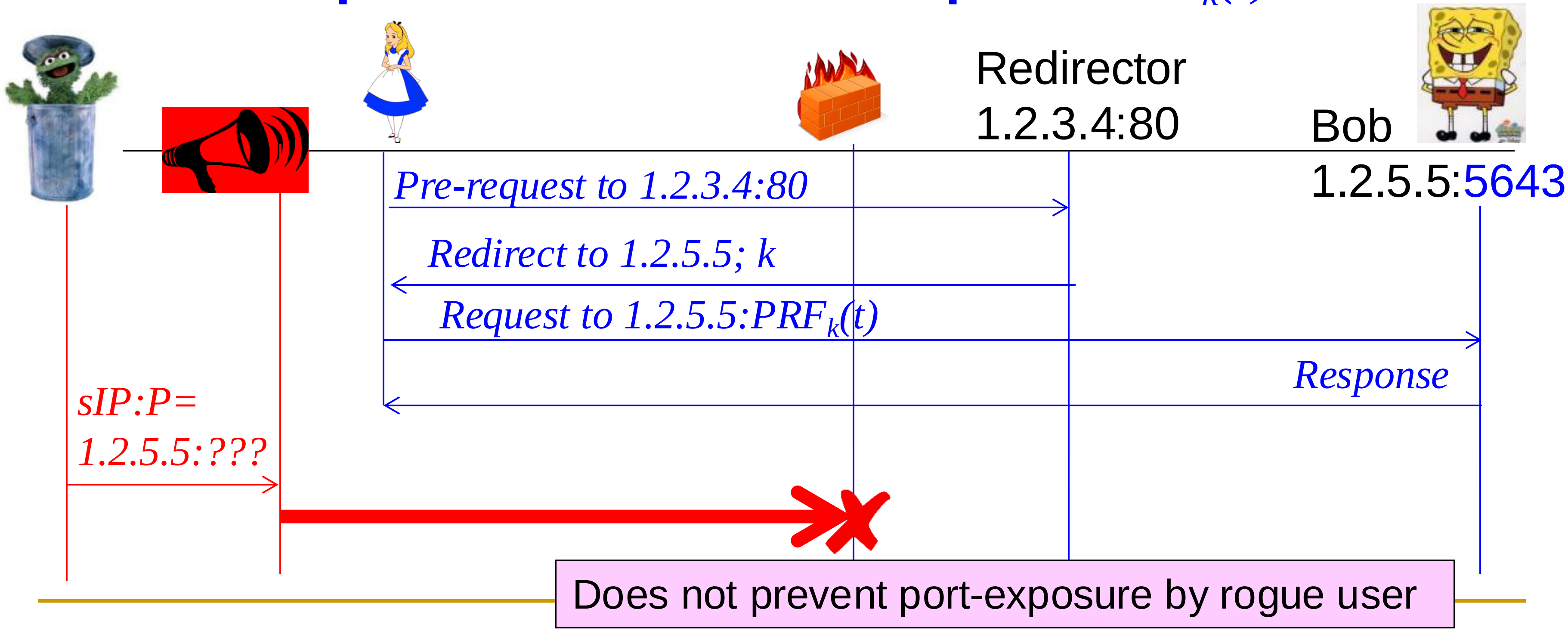
Source-IP Validation with Stateless Filtering

- Clouds, ISPs allow **few free** stateless filtering rules
- Can we use these free rules against off-path BW-DoS ?
  - Idea: allow direct access only to simple (DNS/HTTP) redirector
  - Redirector sends ‘real’ IP:port of ‘real server’
  - Filter by port packets sent to ‘real’ application server



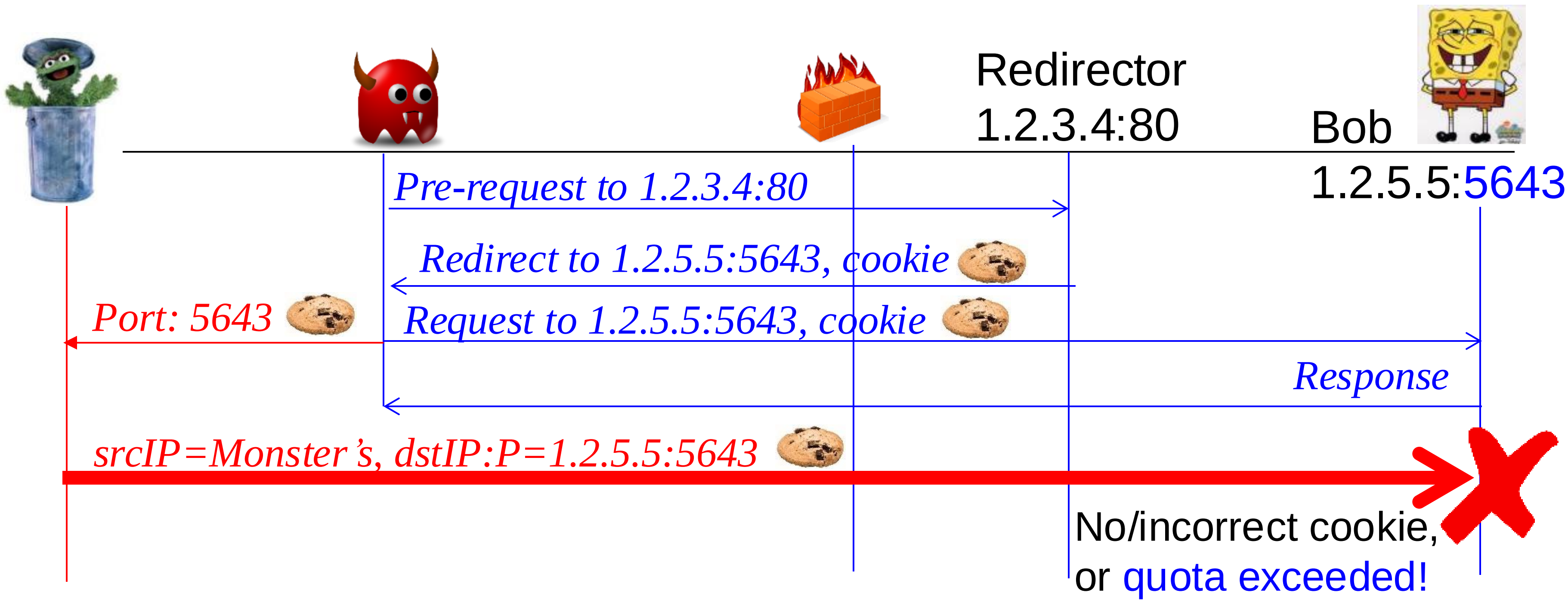
Change 'secret port', same pre-request

- Concern: attacker may (eventually?) find 'secret port'
  - By scanning all ports, detecting responding (non-filtered) one
  - Or by clogging different ports, detecting impact on 'real' port
- How to change port – without another pre-request?
- Solution: '**pseudo-random secret port**': $PRF_k(t)$



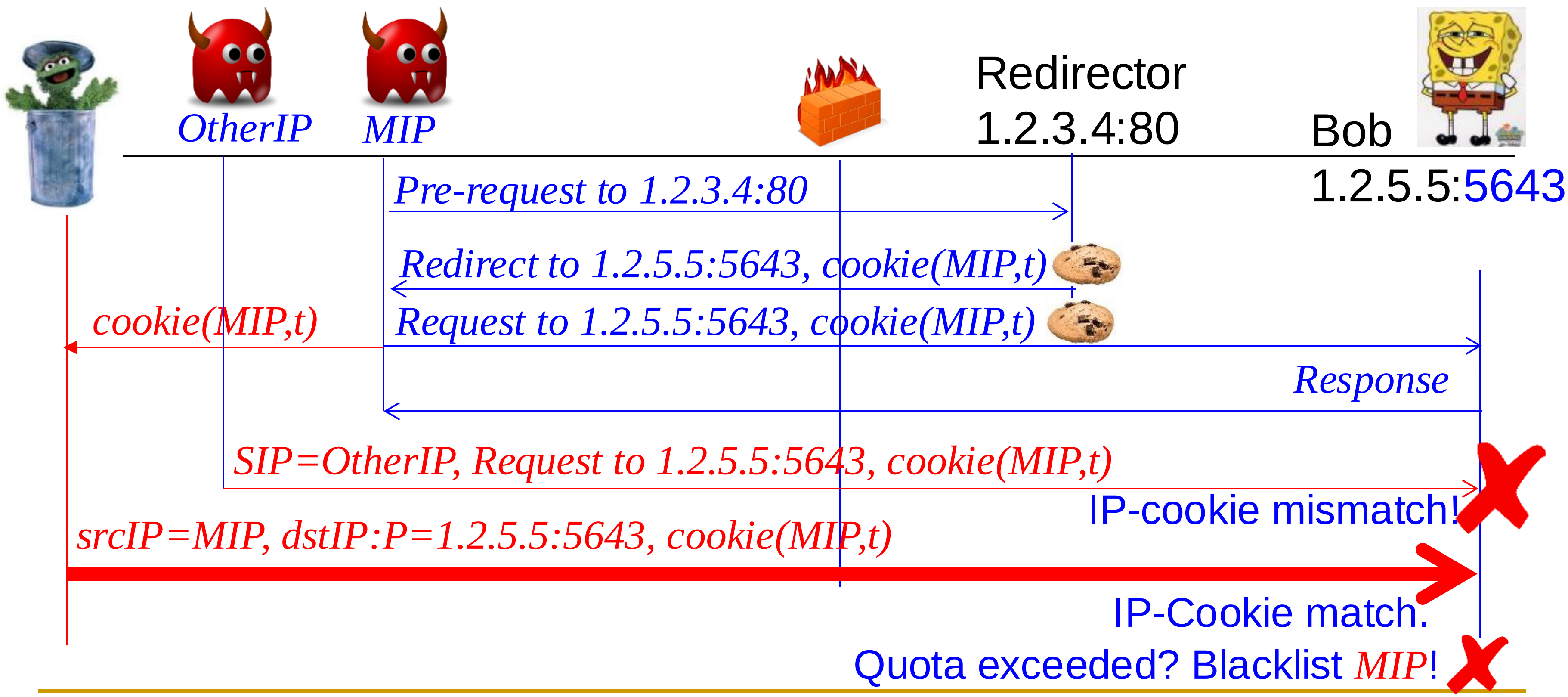
Using Cookie to detect Rogue User

- A rogue user may expose port, then send from many IPs
- Defense: Per-user cookie + quotas ➔ **detect** port exposure, **prevent** excessive use by one user (insider)
- Q: how to link btw cookie & user/IP – without huge table?



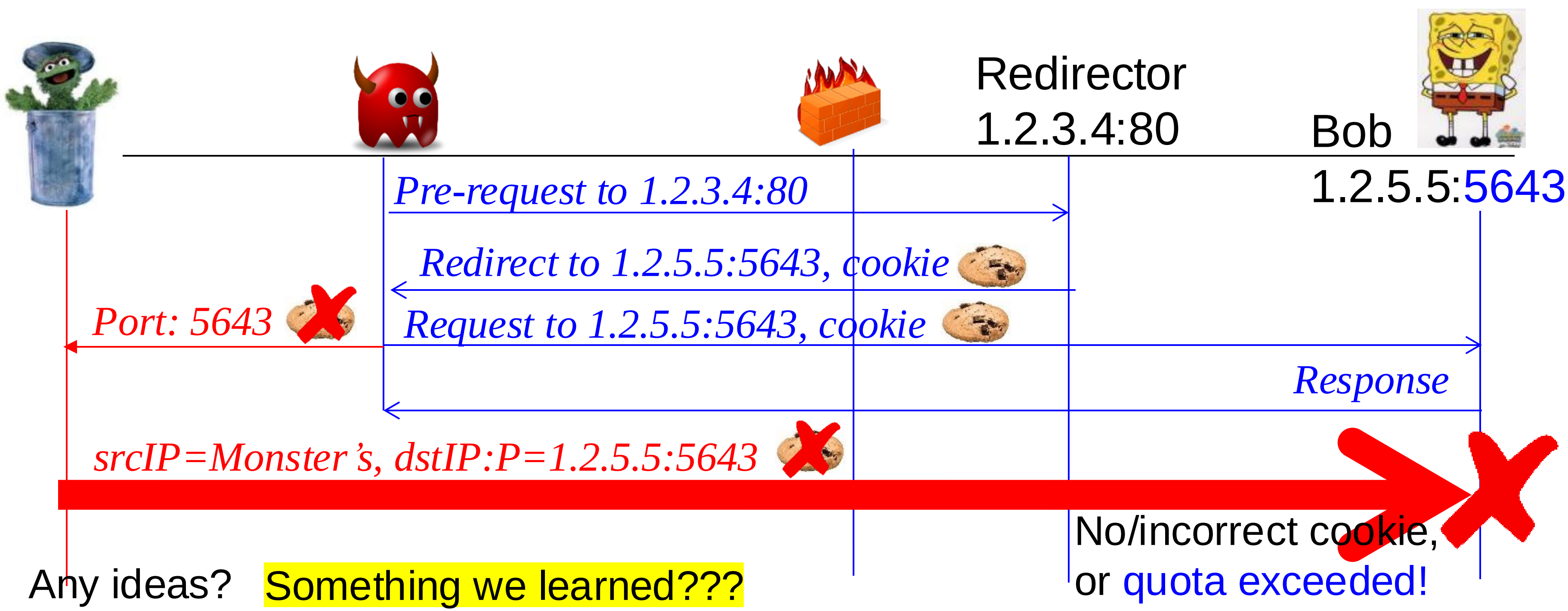
Source-IP Validation with IP-linked Cookie

- Goal: link btw cookie & client's IP – **without state**
- Solution (cont'): Cookie identifies client's IP, date/time *t*
  - $cookie(MIP,t)=t||MAC_{kB}(MIP||t)$



Cookies do not prevent clogging of port!

- A rogue user may expose port, then send from many IPs
- Defense: Per-user cookie + quotas ➔ **detect** port exposure, **prevent** excessive use by one user (insider)
- **Insider** may still clog port (of Bob or redirector)

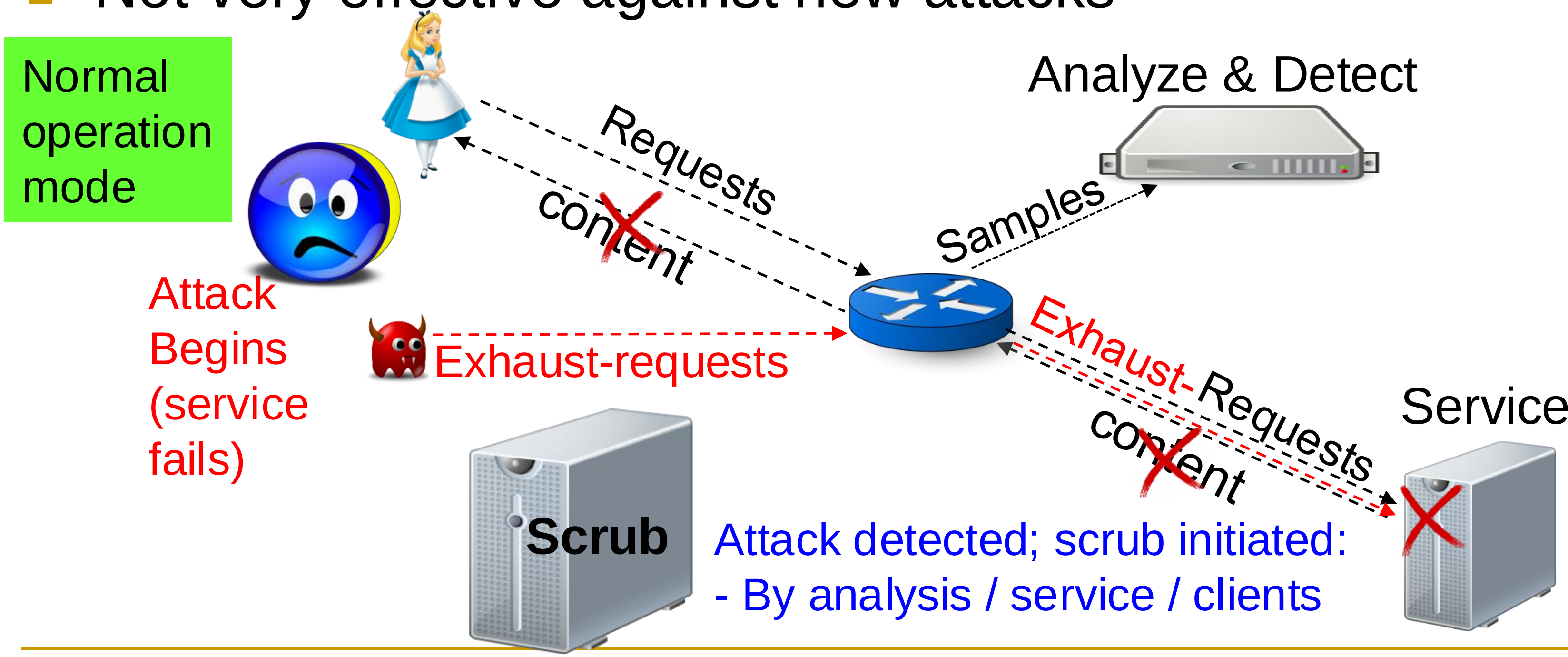


Insider Isolation against DoS on secret port

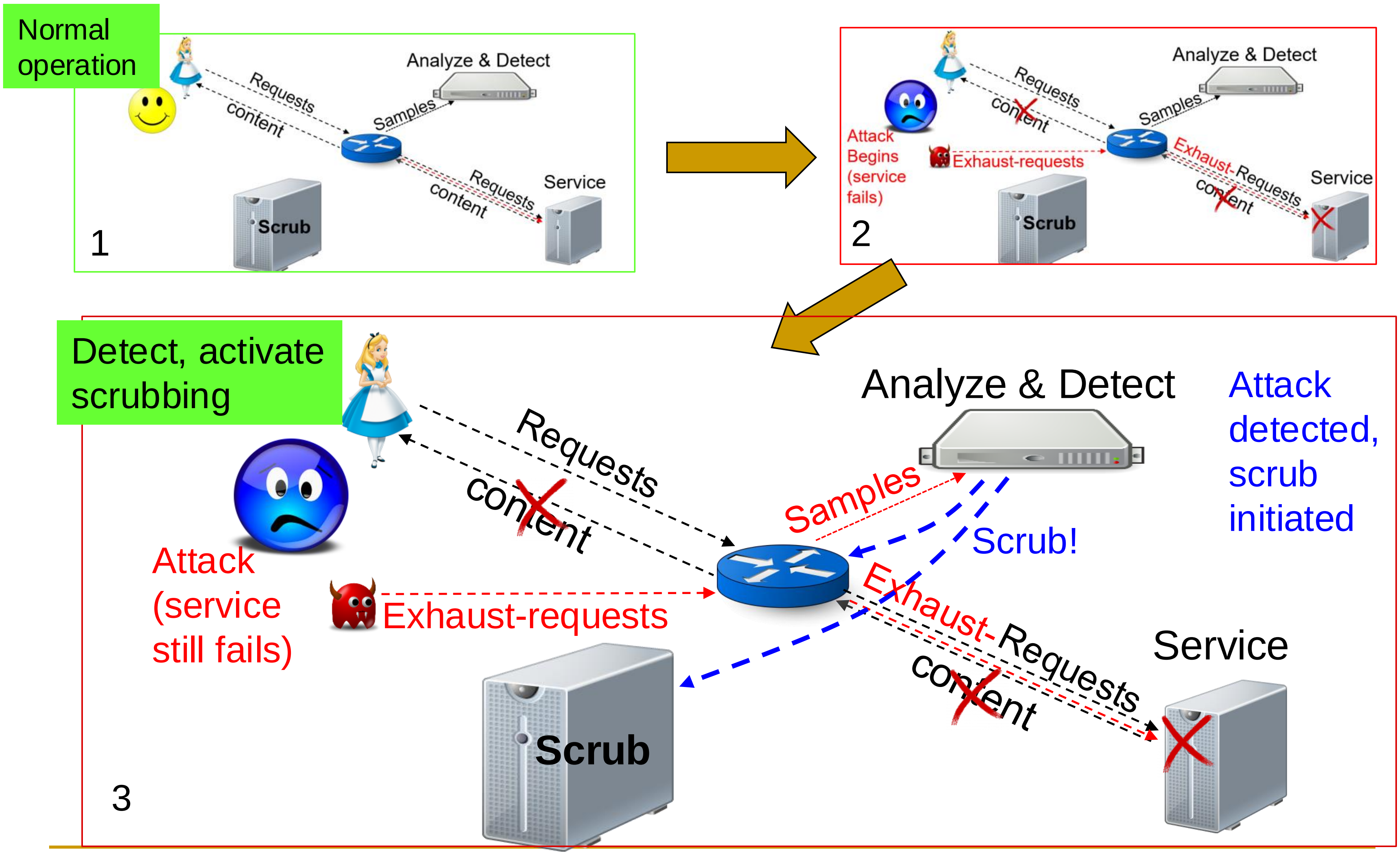
- Use multiple secret IP:ports, assign to sets of users
- Isolate users whose secret IP:port is clogged
  - We discuss below the insider-isolation problem; but first – **scrubbing against server exhaustion attacks**

Scrubbing against Exhaustion DoS (1)

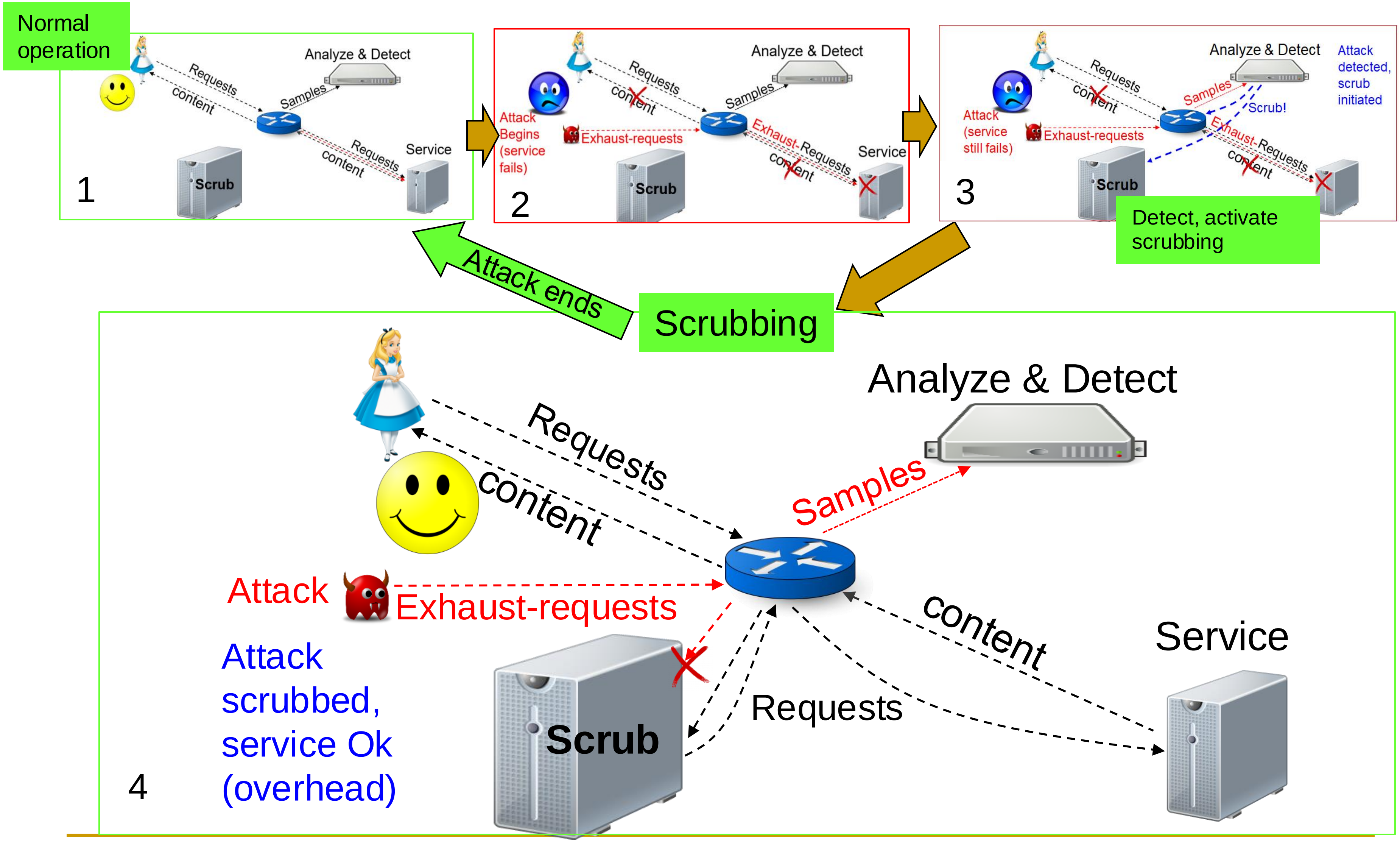
- Learn, detect attack patterns, then ‘scrub’ attack
  - Server and network exhaustion attacks
- Good results for known attacks, often by ‘scrub service’
- Computationally-intensive ➔ scrub only detected attacks
- Not very effective against new attacks



Scrubbing against Server Exhaustion DoS (2)

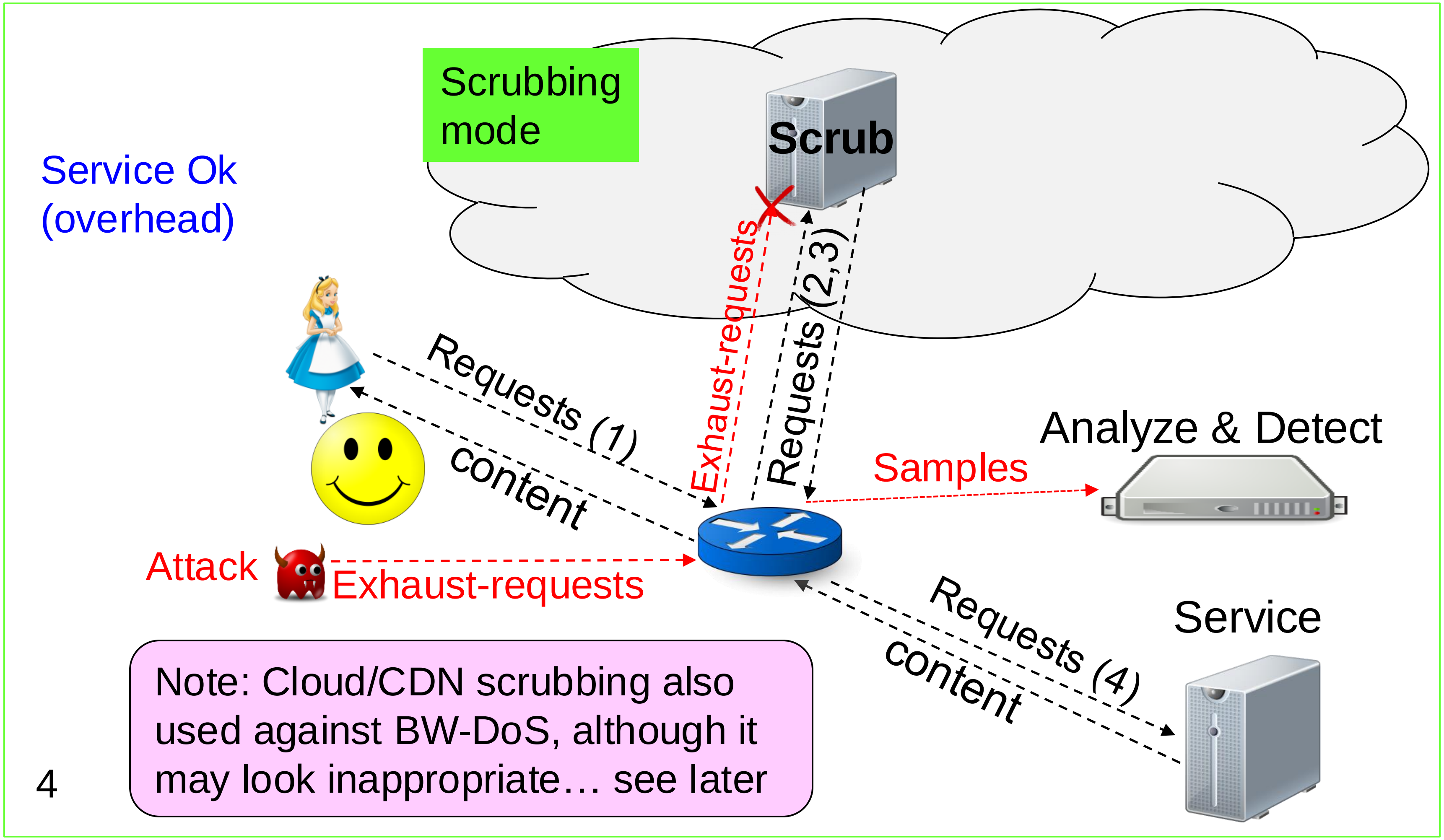


Scrubbing against Server Exhaustion DoS (3)



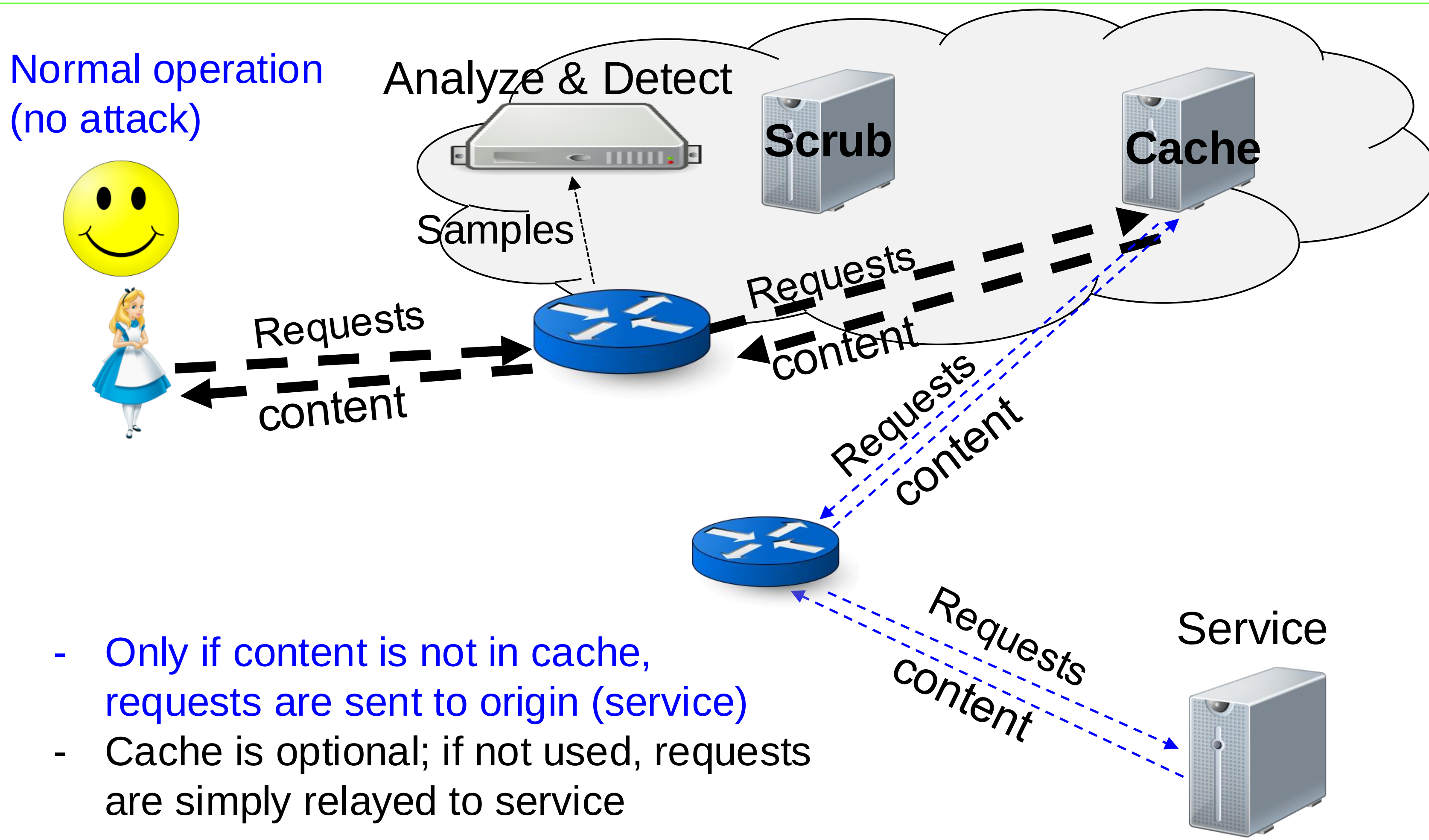
Cloud/CDN DoS-Scrubbing Architectures

Scrub in cloud/CDN, then relay to origin



Cloud/CDN DoS-Scrubbing Architectures

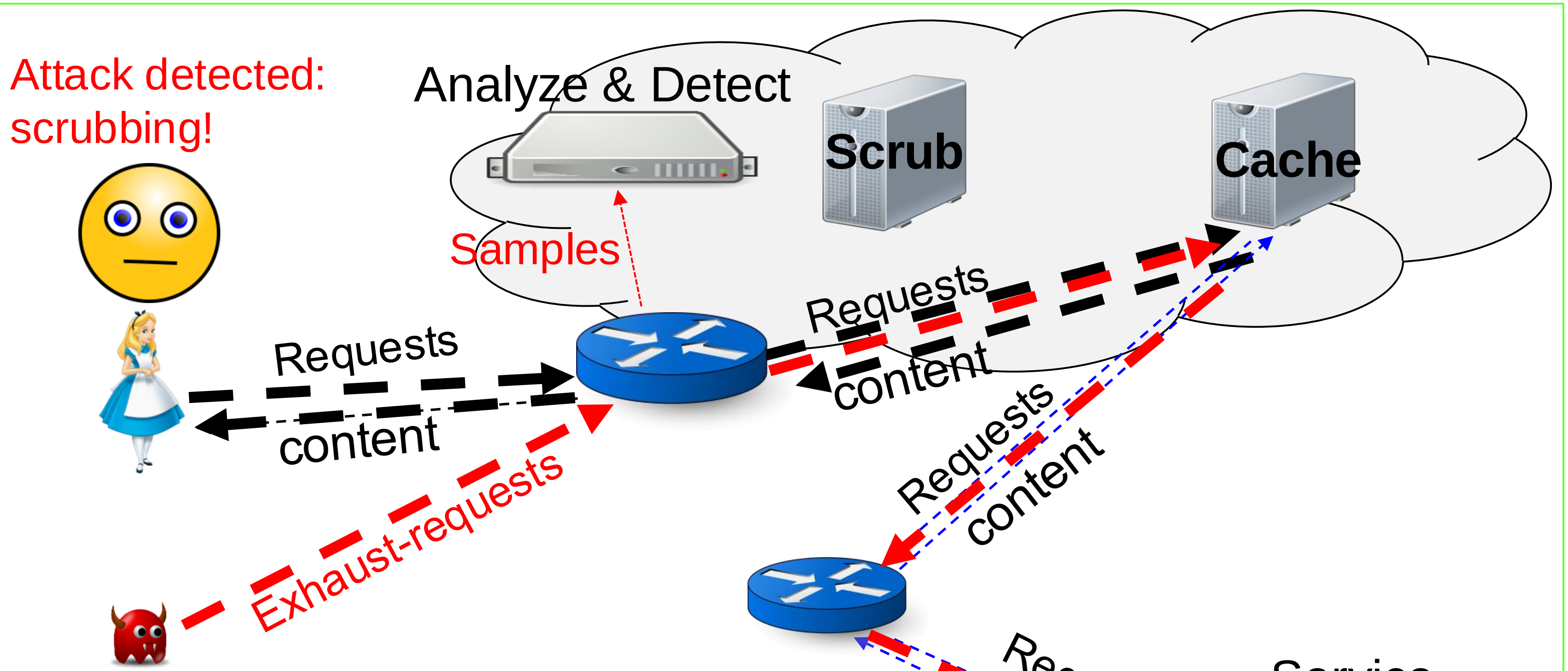
Cloud/CDN caches (and detect DoS), then relay to origin



- Only if content is not in cache, requests are sent to origin (service)
- Cache is optional; if not used, requests are simply relayed to service

Cloud/CDN DoS-Scrubbing Architectures

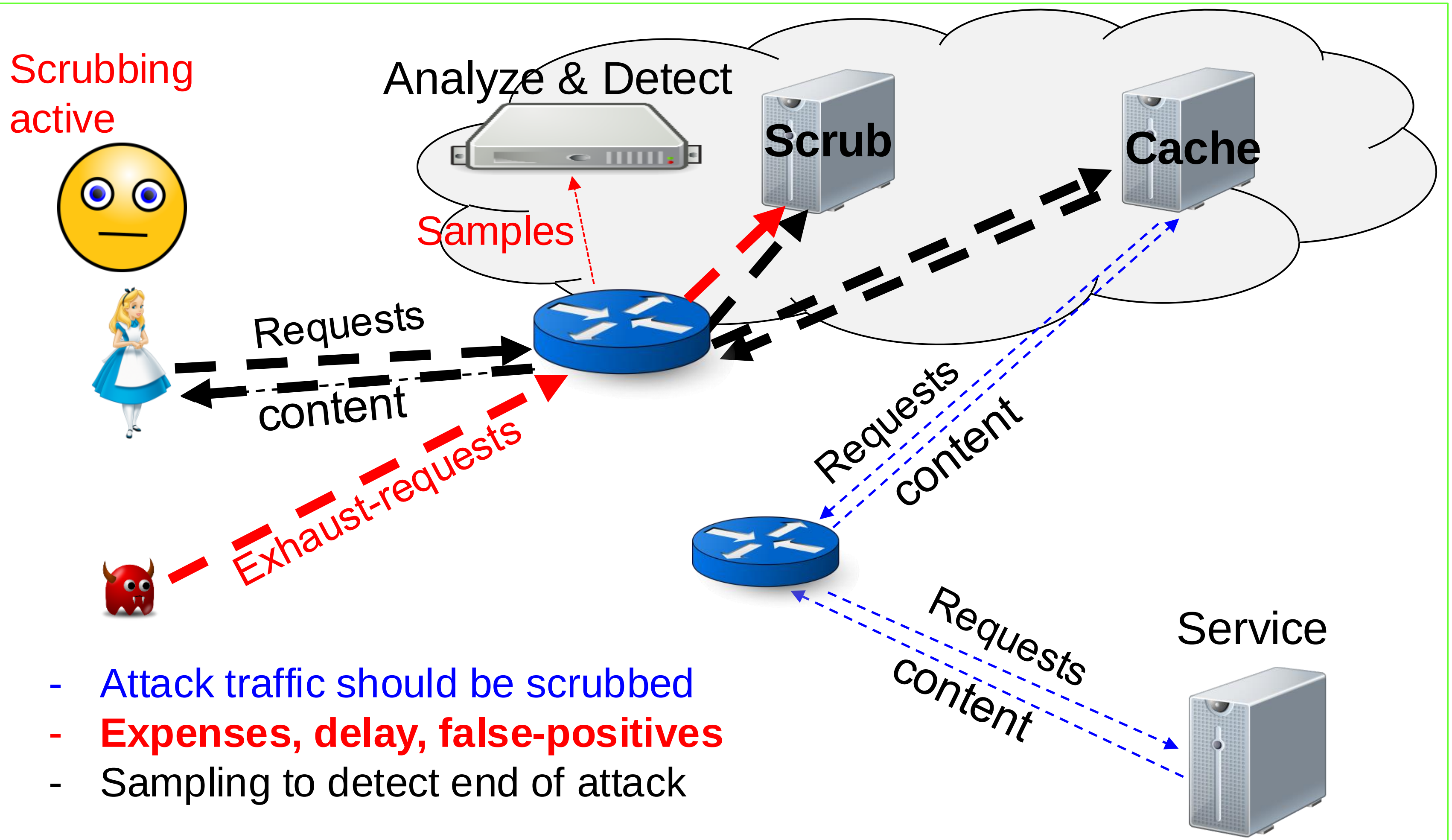
Attack ➔ detection ➔ activate scrubbing



- When content is not in cache, requests are sent to origin (service)
- Attack requests sent to origin, but should be detected, scrubbing invoked…

Cloud/CDN DoS-Scrubbing Architectures

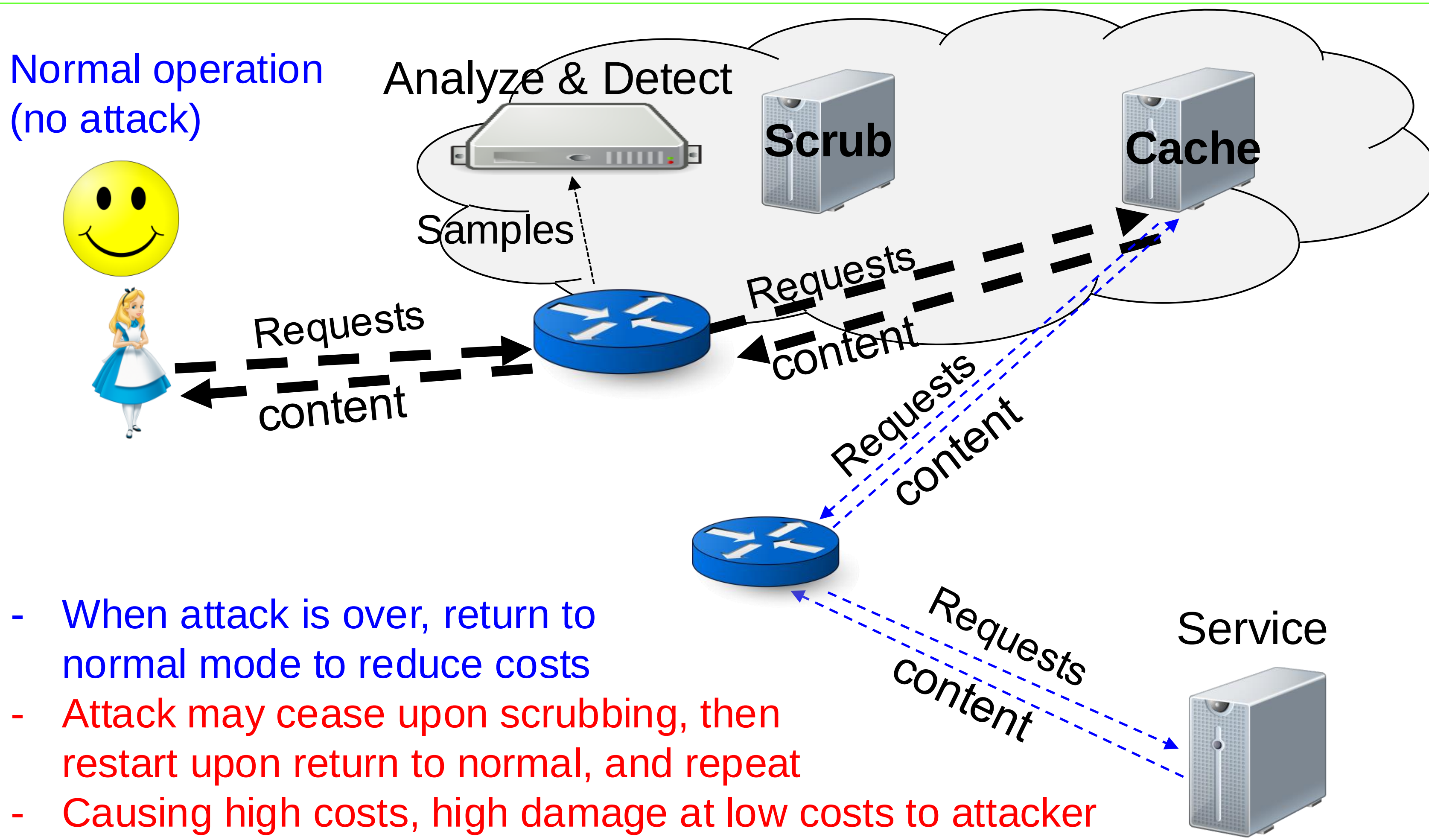
Scrubbing active: scrub, cache, detect end of attack



- Attack traffic should be scrubbed
- **Expenses, delay, false-positives**
- Sampling to detect end of attack

Cloud/CDN DoS-Scrubbing Architectures

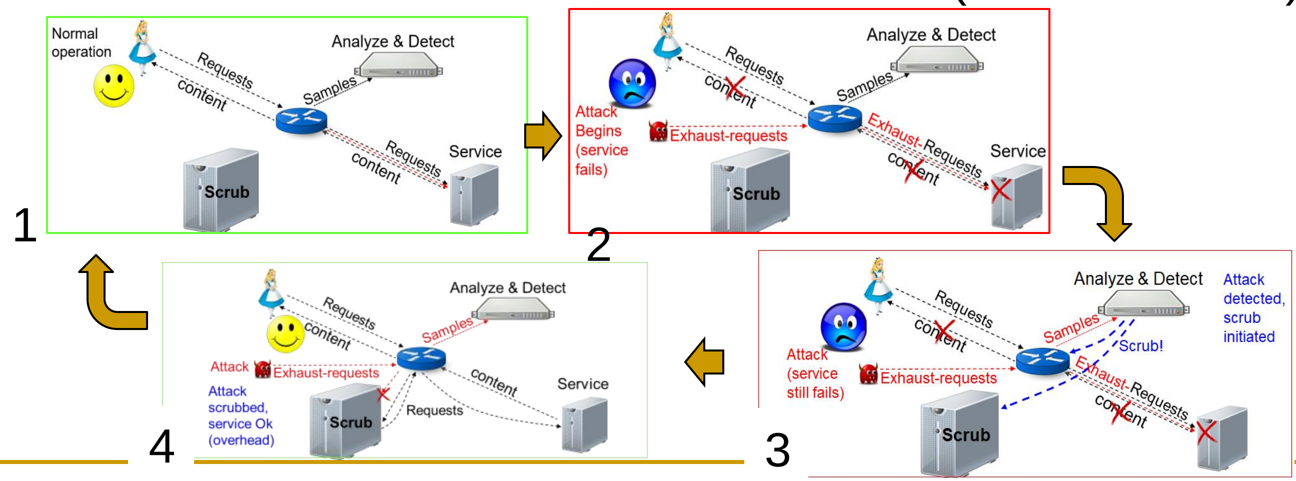
Attack ended ➔ return to normal (no-scrub) operation



- When attack is over, return to normal mode to reduce costs
- Attack may cease upon scrubbing, then restart upon return to normal, and repeat
- Causing high costs, high damage at low costs to attacker

Pulse-Wave BW-DoS Attacks

- Exploit reactions to BW-DoS and to server/net failures
  - Re-route via scrubbing service
  - Scale-up: additional instances/resources (cloud/CDN)
  - Re-route to avoid clog, e.g., use of alternate name server
- Goals:
  - Cause damage during (repeated!) transition periods
    - Transition periods (and down time) can be quite long (minutes)
  - May force victim to use additional resources for long time
    - Cause waste of resources when not attacked (Economic DoS)



Server Exhaustion DoS defenses beyond-scrubbing

- How to defend when scrubbing fails (ZD, stealthy atks)?
- Scale-up, over-provide (cloud/CDN)
- Provide (better) service to users with good **reputation**
- Different **reputation** mechanisms:
  - Whitelist / blacklist / …
  - Based on source IP (of specific user, of network)
    - Applicable mainly since most services provided only over TCP (so??)
  - Based on connecting AS
  - Authenticated / 'paying' users [password, cookies, CAPTCHA…]
- Challenge: dealing with 'insiders'
  - Authenticated / 'paying' users - that still attack the system!
  - But first let's look

Insider DoS attacks

- Insider: rogue authorized user/server
  - I.e., exploit/circumvent reputation defenses
- Different attacks:
  - Service disabling DoS
  - Server exhaustion DoS
  - Network exhaustion DoS
  - Economic DoS
  - … and other attacks
- **Insider Isolation**: isolate attacking insiders to minimize harm to legit users
  - And with minimal costs

The Insider Isolation Problem

- Intuitive goal: service to ‘good’ users with least cost
  - Cost: number of servers in all rounds
    - Server can serve (any number / up to x) benign users
  - Harm: number of benign users <u>not</u> serviced (in all rounds / per round)
- Fixed set of n users; out of them, m are **insiders** (attacking)
- For **rounds** $r = 1, 2, \ldots, R$:
  - Algorithm determines number of servers s_r, and maps users to servers
  - **Attacker** receives mapping **of insiders**, decides who will attack
    - Always attacking adversary model: insiders always attack
  - Servers with attacking insiders fail; other servers serve all their users
  - Algorithm learns (only) which servers failed and which were Ok
- Goals: minimize average **cost <u>and</u> harm**

Example operation of insider isolation alg.

- $n = 10$ users, $m = 3$ insiders: 1 2 3 4 5 6 7 8 9 10
- First round, algorithm splits to (say) four servers:
  - S1:[1 2] , S2: [3 4] , S3: [5 6 7], S4: [8 9 10]
  - All insiders attack, servers 2, 3 fail; cost=4, harm=2
- Merge Ok servers (S1, S4), 'shuffle' failed servers:
  - S1:[1 2 8 9 10] , S2: [3 5 6] , S3: [4 7]
  - 7 doesn't attack, server 3 fails: cost=3, harm=1
- Merge Ok servers (S3, S1), split failed server S2:
  - S1:[1 2 8 9 10 4 7] , S2: [3 5] , S3: [6]
  - All attack, detect-and-ignore 6 (insider); cost=3, harm=7
  - (more rounds…)
- **Cost**=4+3+3=14, **Harm**=2+1+7=10

The Insider Isolation Problem

- Intuitive goal: service to ‘good’ users with least cost
  - Cost: number of servers in all rounds
    - Server can serve (any number / up to x) benign users
  - Harm: number of benign users <u>not</u> serviced (in all rounds / per round)
- Fixed set of n users; out of them, m are **insiders** (attacking)
- For **rounds** $r = 1, 2, \ldots, R$:
  - Algorithm determines number of servers s_r, and maps users to servers
  - **Attacker** receives mapping **of insiders**, decides who will attack
    - Always attacking adversary model: insiders always attack
  - Servers with attacking insiders fail; other servers serve all their users
  - Algorithm learns (only) which servers failed and which were Ok
- Goals: minimize average **cost <u>and</u> harm**
  - Exercise : algorithm that ensures zero harm (with high costs)
  - And: algorithm that has low costs (but high harm)

The Insider Isolation Problem

- Intuitive goal: service to ‘good’ users with least cost
  - Cost: number of servers in all rounds
    - Server can serve (any number / up to x) benign users
  - Harm: number of benign users <u>not</u> serviced (in all rounds / per round)
- Fixed set of n users; out of them, m are **insiders** (attacking)
- For **rounds** $r = 1, 2, \ldots, R$:
  - Algorithm determines number of servers s_r, and maps users to servers
  - **Attacker** receives mapping **of insiders**, decides who will attack
    - Always attacking adversary model: insiders always attack
  - Servers with attacking insiders fail; other servers serve all their users
  - Algorithm learns (only) which servers failed and which were Ok
- Goals: minimize average **cost <u>and</u> harm**
- **Challenge: Strategic attacker** (may not attack in some rounds)
- **Easier: always-attacking insider model** (insiders always attack)

Protag algorithm against always-attacking-insiders

All insiders identified, removed in $\log n$ rounds

Cost: $O(R + m \log n)$

Harm (total!): $O(n \log m)$

- Initially (round $r = 1$) : a single server $s_{1,1}$
- Round $r = 1, \ldots, R$:
  - Attacker decides which insiders attack
  - Merge users from all non-failed servers into $s_{r+1,0}$
  - For failed servers with one user: user is an insider
  - For other failed servers $s_{r,i}$: split users to servers $s_{r+1,2i}$, $s_{r+1,2i+1}$
- Example against always-attacking-attacker:
  - $n = 16$ users; say users $\{3,11\}$ are **insiders** (malicious)
  - Round 1: split: $s_{2,1} = \{1,2,3,..,8\}$ and $s_{2,2} = \{9,10,11 \ldots 16\}$
  - Round 2: split: $\{1,2,3,4\}$, $\{5, \ldots 8\}$,$\{9,10,11,12\}$,$\{13, \ldots 16\}$
  - Round 3: merge: $\{5, \ldots 8, 13, \ldots 16\}$, split: $\{1,2\}$, $\{3,4\}$,$\{9,10\}$, $\{11,12\}$
  - Round 4: merge: $\{1,2,5, \ldots 10, 13, \ldots 16\}$, split: $\{3\}$,$\{4\}$, $\{11\}$, $\{12\}$
  - Round 5: merge: $s_{6,0} = \{1,2,4,5, \ldots 10, 12,13, \ldots 16\}$; insiders: 3,11
  - Only one server from now on!

Protag algorithm vs. strategic insiders

  - Initially (round $r = 1$) : a single server $s_{1,1}$
- Round $r = 1, \dots, R$:
  - Attacker decides which insiders attack
  - Merge users from all non-failed servers into $s_{r+1,0}$
  - For failed servers with one user: user is an insider
  - For other failed servers $s_{r,i}$: split users to servers $s_{r+1,2i}, s_{r+1,2i+1}$
- Exercise: attacker causing $O(Rn)$ harm
  - Hint (or challenge?): with a single attacker, $m = 1$!
  - Strategy: attack every 2nd round ➔ total harm: $R \cdot \frac{n}{2}$
- Exercise: cause cost $O(rm)$, for $m < \log n$
- Few algs obtain better cost, harm in simulations
  - Limited provably-secure positive results – more research required

Summary: DoS is a Challenge!

- Breaking is easier than fixing
- Systems designed for benign environments
- Predicting impact on performance is hard
- Can we define, prove security against DoS?
- This seems still very much whack-a-mole
  - Many providers, products, protocols, attacks, defense
- So… it's a challenge

Denial-of-Service Presentation: License