POLITECNICO DI MILANO

Traffic Measurement

Measuring Network Traffic

Know your network

- traffic measurement makes the network more observable
- (traffic engineering makes the network more controllable)

Packet counting and logging provides input for

- capacity planning
- accounting
- traffic analysis

Purpose of Traffic Measurement

Capacity Planning

- Determine the traffic matrix
- Over short time scales give input for traffic engineering
- Over long time scales upgrading link capacity

Accounting

- Verification of Service Level Agreements (SLA)
- Identification of most significant flows for network upgrade or peering agreements

Traffic Analysis

- Classification and Identification of traffic types
- Detection of possible attacks

Traffic Matrix

Assume that a provider's network has N links to/from other networks (external links).

The traffic matrix is the table of the traffic between all the N(N-1) pairs of external links

- about N^2 variables
- note that it also changes over time and the provider might want per-application breakdown

Simple routers implement per-interface counting

- number of packets and bytes incoming and outgoing from each link
- about N coefficients.

With N equations and N^2 variables the system is undetermined.

Need more counters per-interface (e.g. per-IP-prefix)

Masurement is hard

The basic building block for measurement is counting packets/bytes

- many counters (~millions)
- multiple counters to update per packet
- high speeds
- large widths (64 bit per counter)

Example

- 1M counters of 64 bit = 64Mbits of memory
- updating 2 counters every 8 ns = 16 Gbit/s memory throughput
- high speed requires fas SRAM (which is expensive)
- cheap DRAM is too slow

Reducing the need for fast RAM using cheaper RAM

Many long counters = a lot of fast memory

How to save fast memory?

- remember that locality is bad, no gain in caching the most recently used counters

Keep short versions of all the counters in fast memory

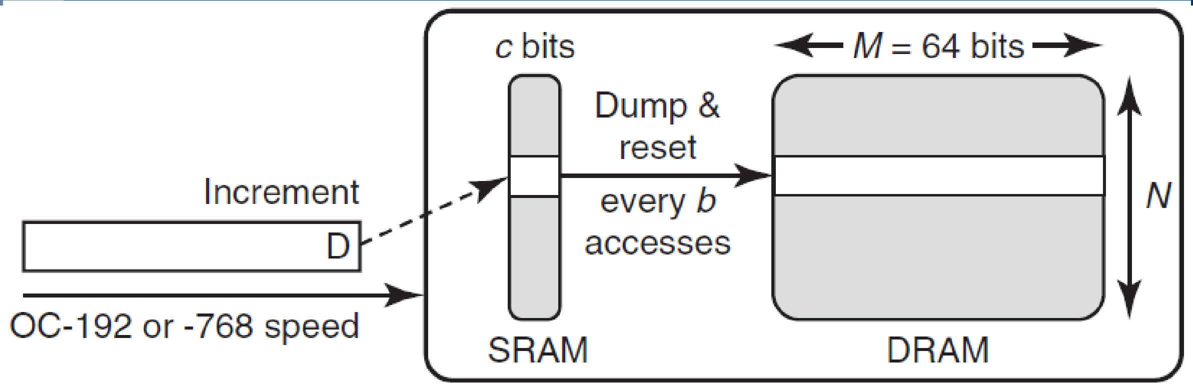
- e.g. 16 bit per counter

Keep long versions in cheap (slow) memory

Periodically, choose one counter from fast memory, add to the corresponding counter in slow memory and reset

For correctness every counter must be backed up before it overflows

Largest Count Algorithm (LCF)



If fast memory is *b* times faster, you can move a counter every *b* packets

What counter to move?

- Ideally the one with the largest number (Largest Count First algorithm)
- with LCF   c ≈ log log b N
- implementation can be tricky, the algorithm to find the largest number needs time and memory

Simplified algorithm:

- let cj = the largest updated counter in the last cycle of b accesses
- if cj >= b update cj
- else update any other counter with value at least b

Randomized Counting

Trade accuracy for efficiency

When packet arrives, increment counter with probability 1/c.

If counter has value *x*, the expected number of counted packets is *xc*

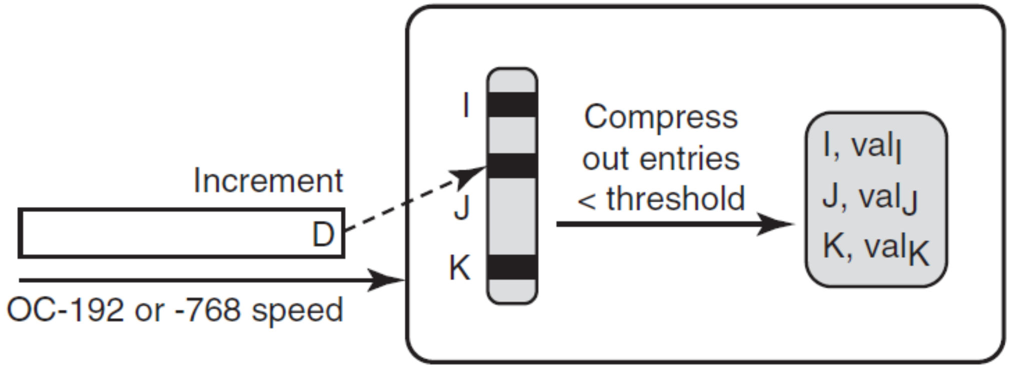
Note that standard deviation is a few *c*, so

- when x >> c, precision is high
- when x ≈ c or smaller, numbers are useless

Randomized counting works when we count flows that are likely to be relevant

Measurement of unfrequent events is inaccurate

Threshold Aggregation



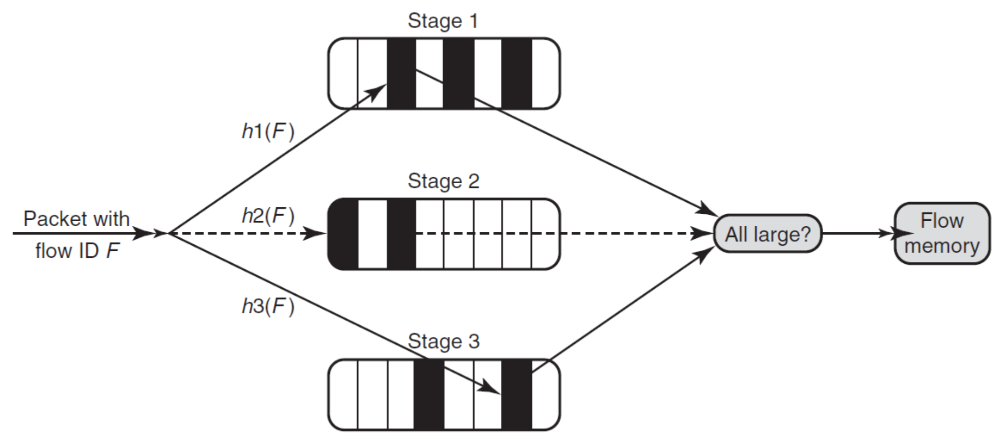
Keep only the counters above a given threshold

How to detect the elephants?

- keep all counters
- backup only large counters

Does not save memory

Threshold aggregation



Compute the packet’s flow ID (e.g. the 5-tuple that defines a TCP socket)

Hash the ID and add to the corresponding counter

If the counter is above the threshold, then track it

There are false positives:

- a small flow fall in the same counter as a large flow
- several small flows can make a counter pass the threshold

- Solution: use multiple independent hash functions
  - false positives are reduced (not eliminated)

Threshold aggregation example (1)

Input

- link capacity v=100 Mbit/s link
- N=100k flows
- measurement interval T=1s

Output

- flows above 1% of the link capacity during T

Design choices

- each stage has 1000 buckets
- threshold is 1 Mbit/s
- 4 independent stages

Threshold aggregation example (2)

Calculate the probability that a 100-kbit/s flow is above the threshold

- For each stage, the flow is above the threshold if the other flows in the same counter add up to 900 kbit/s
- Worst case is if all the traffic from other flows splits in

$$x = \frac{99999 \text{ flows}}{900\text{kbit/s}} = 111 \text{ buckets}$$

- The probability of false positive is at most 11,1% at each stage
- With 4 stages the probability of false positive is at most 0.016%

Flow Counting

Instead of counting the packets, count the number of distinct flows according to a given pattern that generate traffic above a given threshold

- Examples: detection of attacks, identification of bandwidth hungry applications
- Naive method: hash table containing all the observed S/D pairs
  - Requires too much memory

Flow Counting / Probabilistic Counting

Trades accuracy for memory

Basic idea:

- look for uncommon patterns in flow IDs
- if uncommon patterns are found, then there are a lot of flows

Implementation:

- hash the flow ID
- count the number of consecutive zeros starting from the LSB
- x is the maximum observed number of consecuteive zeros
- the estimate for the number of flows is 2^x

Better results using an array of independent hash functions

IPFIX

The IETF standard ipfix (formerly Cisco Netflow) provides per-flow counting:

- a flow is a TCP connection or a UDP dialogue
- data can be aggregated (e.g. per transport port)
- collected data are sent to a “manager” workstation

Two problems

- processing overhead
  - measuring increseas the workload of the router
- collection overhead
  - reporting increases the usage of bandwidth and the workload of manager workstations

IPFIX – Packet Sampling

We already discusses some techniques to reduce processing overhead

A commonly used tecniche in practice is packet sampling (known as sampled netflow)

Count only a packet every N

Common values for N range from N=16 to N=1000

Considerable inaccuracies

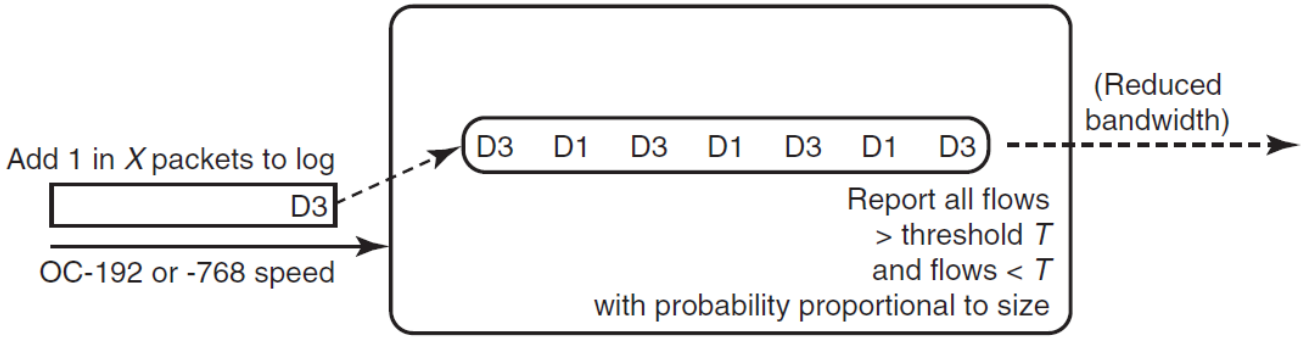
- entire flows can be undetected
- only useful for aggregated measurements over long reporting periods

IPFIX - Sampled charging

To reduce collection overhead only a subset of the measures is reported to the manager

Reports a flow of size s

- if s is above a given threshold
- with a probability proportionally to s if s is below the threshold



Add 1 in *X* packets to log
D3
OC-192 or -768 speed
D3 D1 D3 D1 D3 D1 D3
(Reduced bandwidth)
Report all flows > threshold *T* and flows < *T* with probability proportional to size

Saves bandwith but also gives information about small-sized flows

Otherwise a lot of traffic from individually small flows would be undetected

Trajectory Sampling

If sampling happens independently at each router, we cannot follow a packet's route in the network

- some router counts it, some router ignores it

With trajectory sampling, the same packet is either counted by all the routers or ignored by all the routers

We can also track the path of a packet in the network

Take a string of bits from immutable packet fields. Call this string *m*

- IP addresses and payload ore ok
- TTL and MAC addresses are not ok

Hash them with the function *h(m)*, equal in all the routers

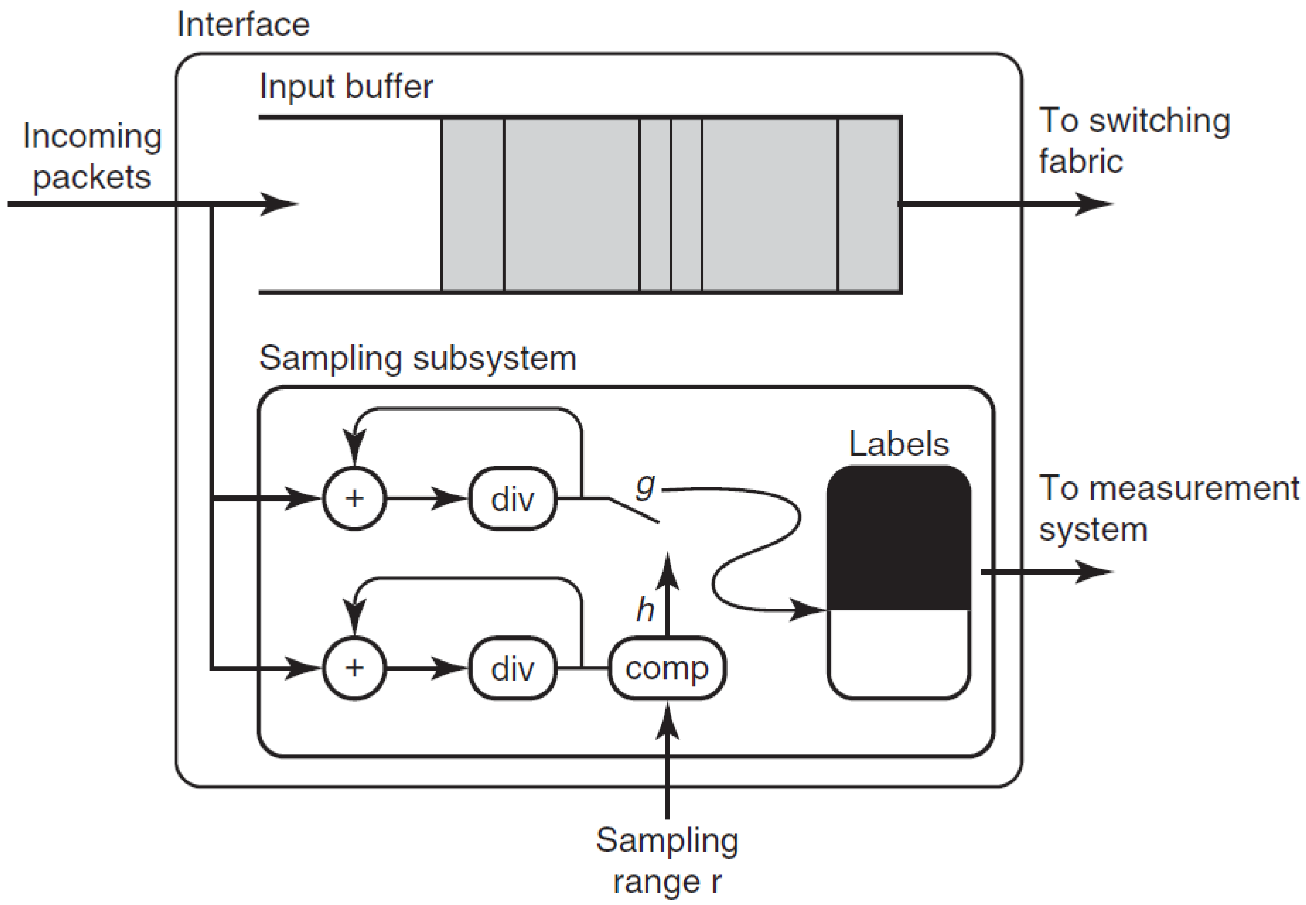
- *g(m)* is a number in the range $0 \le h(m) < R$

Choose $B < R$. If $h(m) < B$ the packet is counted.

A given packet is counted in all the routers with probability B/R.

If the packet is sampled, store its label g(m) in the log

Trajectory Sampling



IP traceback

Suppose we have found a malicious packet with a spoofed IP source address. We want to trace it back to the subnet of origin.

Useful against DoS from a single attacker using asymmetric attacks

- Common before 2000, now replaced by DDoS

Still a building block for more sophisticated response or forensics

Why not simply filter spoofed addresses?

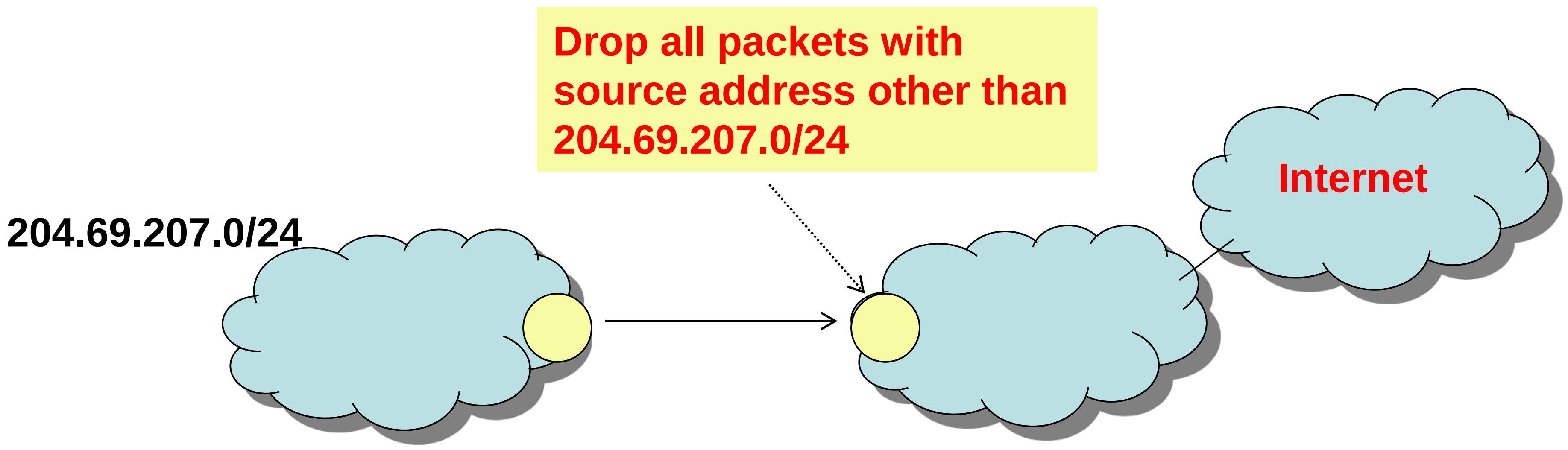
- RFC2827 suggest ingress filtering, but it is feasible only at the edges
- In the core can check Unicast Reverse Path Forwarding, but requires symmetric routing

Ingress Filtering

- **RFC 2827:** Routers install filters to drop packets from networks that are not downstream

Feasible at edges

Difficult to configure closer to network "core"


Drop all packets with source address other than 204.69.207.0/24
204.69.207.0/24
Internet

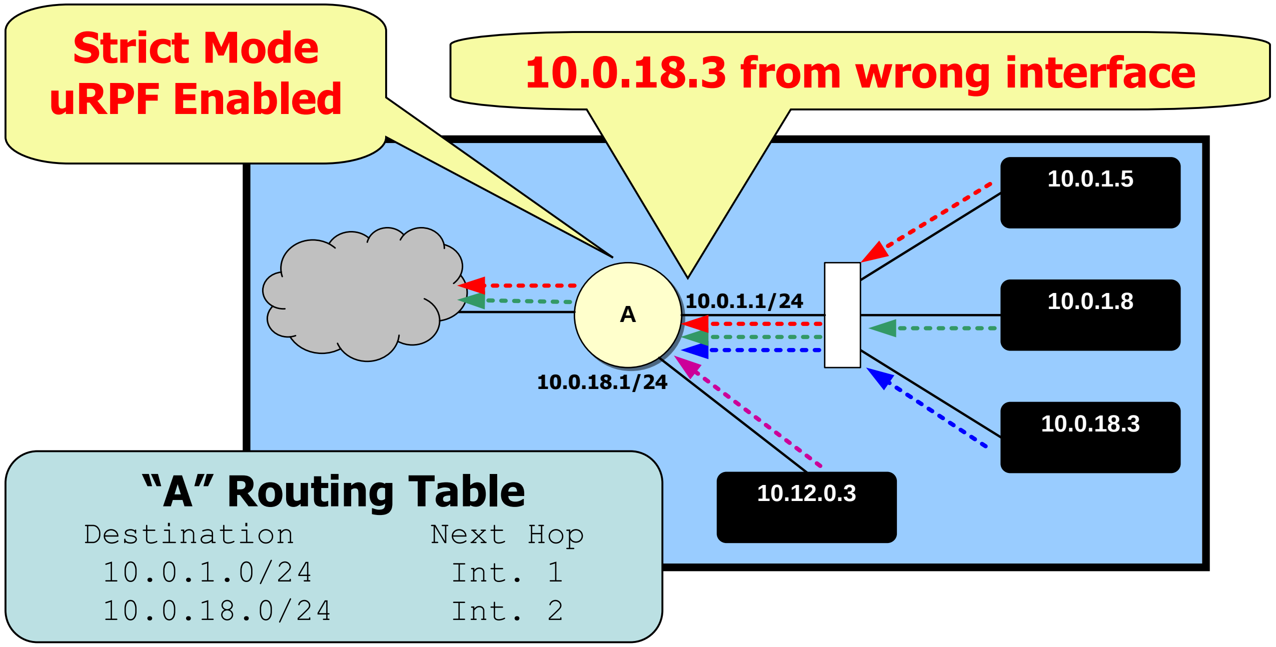
uRPF Checks

Unicast Reverse Path Forwarding

- Cisco: “**ip verify unicast reverse-path**”

Requires symmetric routing

Accept packet from interface only if forwarding table entry for source IP address matches ingress interface

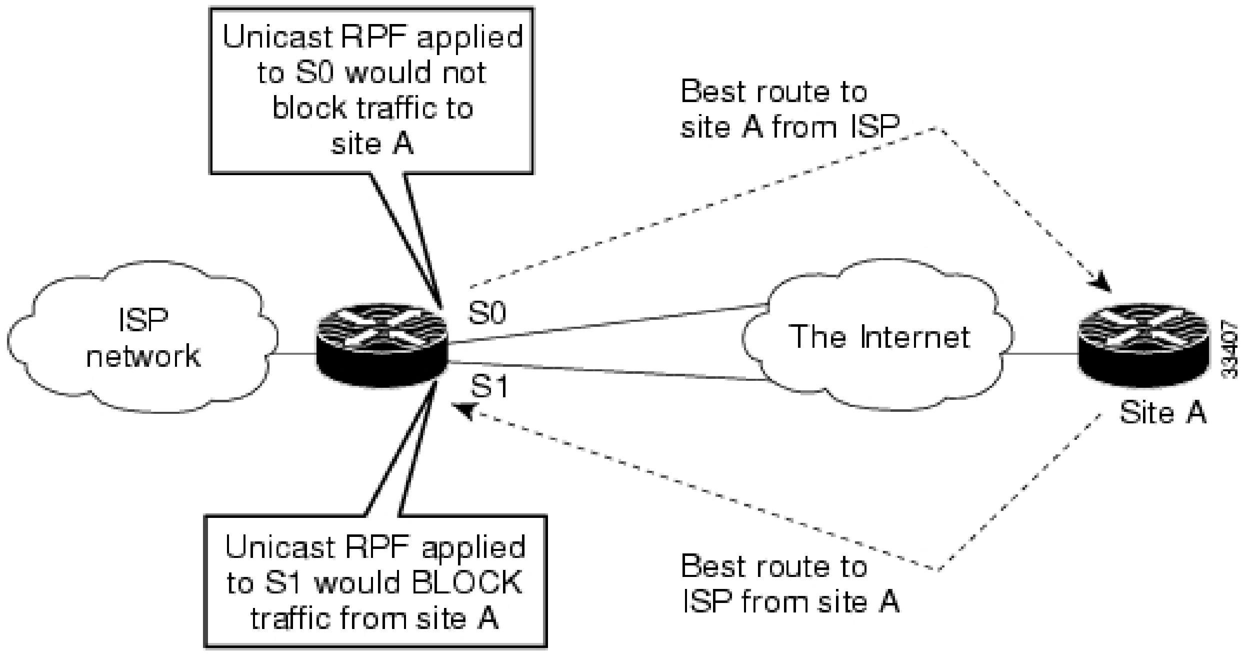


“A” Routing Table

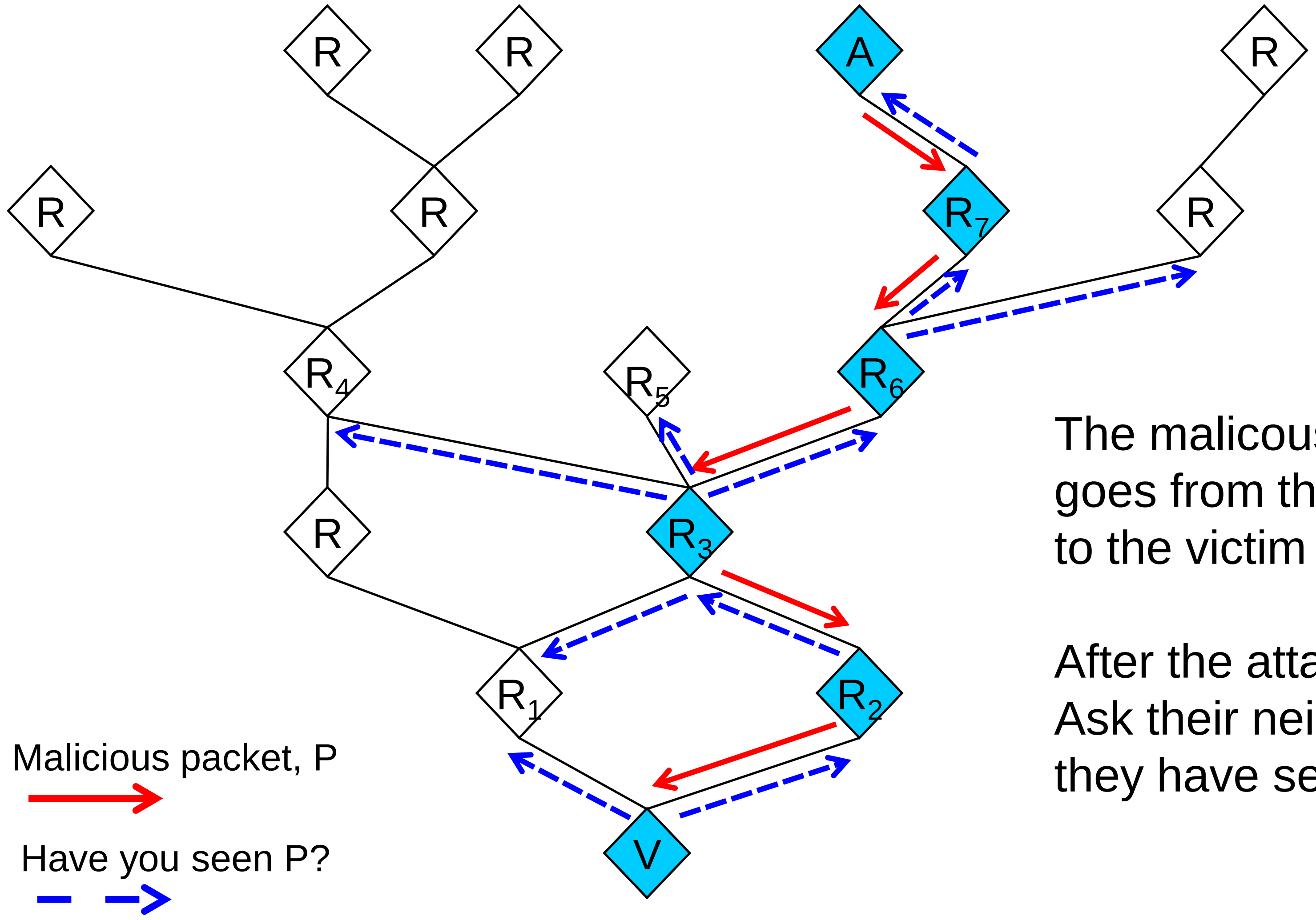
Destination	Next Hop
10.0.1.0/24	Int. 1
10.0.18.0/24	Int. 2

Problems with uRPF

Asymmetric routing


Unicast RPF applied to S0 would not block traffic to site A
Best route to site A from ISP
ISP network
S0
The Internet
S1
Site A
33407
Unicast RPF applied to S1 would BLOCK traffic from site A
Best route to ISP from site A

IP Traceback via logging



The malicous packet P goes from the attacker A to the victim V.

After the attack the nodes Ask their neighbors if they have seen P

Logging Challenges

Attack path reconstruction is difficult

- Packet may be transformed as it moves through the network

Full packet storage is problematic

- Memory requirements are prohibitive at high line speeds (OC-192 is ~10Mpkt/sec)

Extensive packet logs are a privacy risk

- Traffic repositories may aid eavesdroppers

Packet Digests

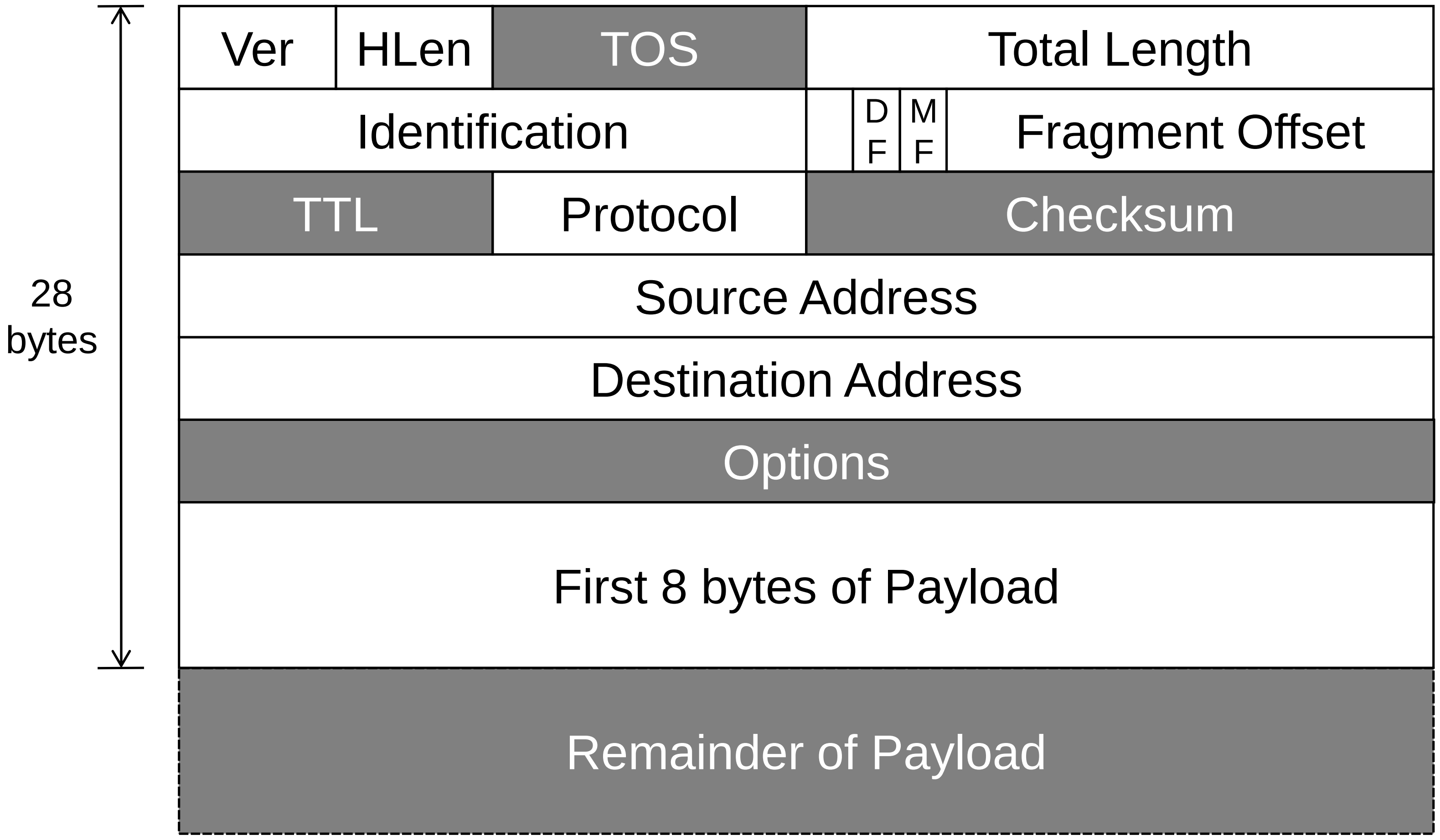
Compute hash(P)

- Invariant fields of P only
- 28 bytes hash input, 0.00092% WAN collision rate
- Fixed sized hash output, *n*-bits

Compute *k* independent digests

- Increased robustness
- Reduced collisions, reduced false positive rate

Hash input: Invariant Content



Hashing Properties

Each hash function

- Uniform distribution of input -> output

  H1(x) = H1(y) for some x,y -> unlikely

Use k independent hash functions

- Collisions among k functions independent
- H1(x) = H2(y) for some x,y -> unlikely

Cycle k functions every time interval, t

Digest Storage: Bloom Filters

Fixed structure size

- Uses m = 2^b bit array
- Initialized to zeros

Insertion

- Use *n*-bit hash as indices into bit array
- Set to '1'

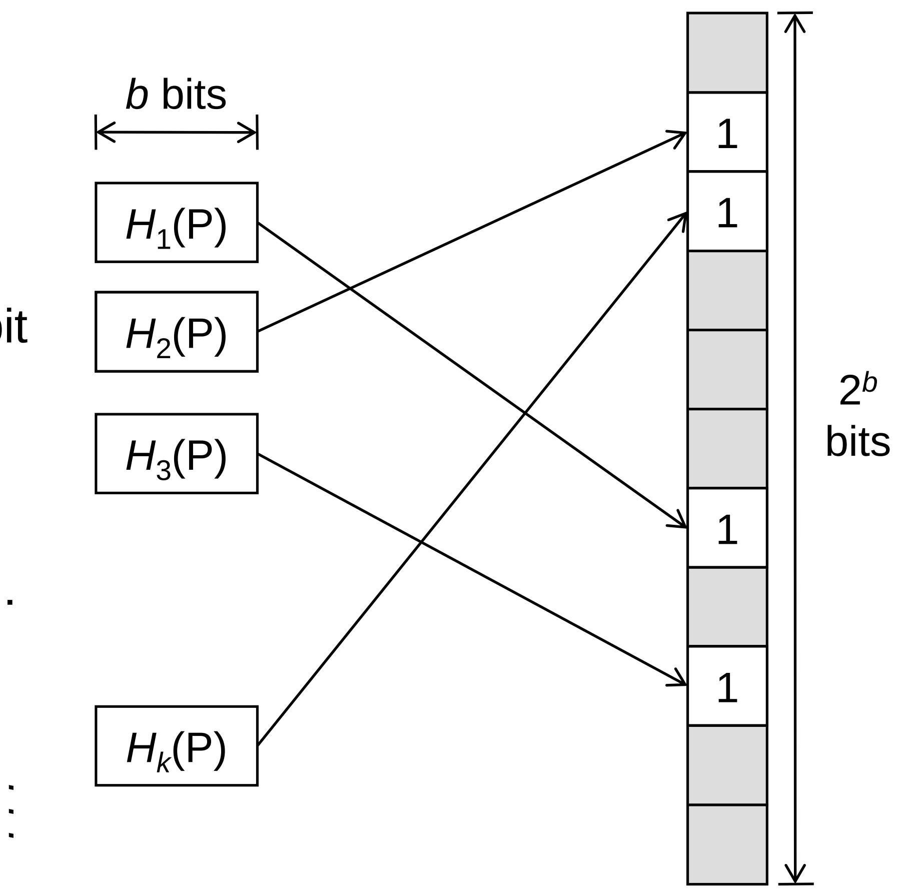
Membership

- Compute *k* hashes, d_1, d_2, etc…
- If (filter[d_i]=1) for all i, router forwarded packet

False positive rate

- If there filter contains n packets

$(1-(1-1/m)^{kn})^k \approx (1-\exp(kn/m))^k$


b bits
H_1(P)
H_2(P)
H_3(P)
H_k(P)
1
1
1
1
2^b bits