Bad Packets Come Back, Worse Ones Don't

Petros Gigis
University College London
London, UK

Mark Handley
University College London
London, UK

Stefano Vissicchio
University College London
London, UK

Abstract

ISPs may notice that traffic from certain sources is entering their network at an unexpected location, but it is hard to know if this represents a problem or is just normal spoofed background noise. If such traffic is not spoofed, it would be useful to generate alerts, but alerting on background noise is not useful.

We describe Penny, a test ISPs can run to tell unspoofed traffic aggregates arriving on the wrong port from spoofed ones. The idea is simple: when receiving new traffic at unexpected routers, drop a few TCP packets. Non-spoofed TCP packets ("bad packets") will be retransmitted while spoofed ones ("worse packets") will not. However, building a robust test on top of this simple idea is subtle. We show how to deal with conflicting goals: minimizing performance degradation for legitimate flows, dealing with external conditions such as path changes and remote packet loss, and ensuring robustness against spoofers trying to evade our test.

CCS Concepts

• **Networks** → **Network management**; **Network algorithms**; • **Security and privacy** → **Network security**.

Keywords

Traffic testing, ISPs, Internet routing, IP spoofing, TCP, BGP

ACM Reference Format:
Petros Gigis, Mark Handley, and Stefano Vissicchio. 2024. Bad Packets Come Back, Worse Ones Don't. In *ACM SIGCOMM 2024 Conference (ACM SIGCOMM '24), August 4–8, 2024, Sydney, NSW, Australia.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3651890.3672259

1 Introduction

An Internet Service Provider (ISP) has many reasons to want to understand the traffic entering its network: for example, *where* traffic enters the ISP network dictates who pays whom. It is simple to measure specific subsets of traffic at border routers, but the obtained data is typically not actionable because traffic may be spoofed. Can we provide *reliable actionable* alerts when ingress traffic does not conform to the ISP's expectations? Doing so would help detect routing misconfigurations, suggest changes to policies and commercial agreements, and protect against security attacks.

Consider the example in Figure 1. AS1 has two customers, AS2 and AS4, and one provider AS3. Traffic from AS2 entering AS1 over the direct link generates revenue, while any traffic received via AS3 costs money. If AS1 sees traffic originated from AS2 entering at an unexpected ingress point, as in Figure 1b, then there are many scenarios where it would attempt to change the path. AS2 may have deliberately chosen to forward some of its traffic via another provider, for economic or traffic engineering reasons. But knowing whether this is the case may be important for AS1, for example, if it violates the commercial agreement between AS1 and AS2. Alternatively, the traffic path may be caused by a misconfiguration: AS2 may be unintentionally filtering or deprioritizing AS1's route. If so, AS1 can contact AS2, and fix the traffic path. Even in the absence of misconfigurations, AS1 may still be able to influence AS2's route choice, either technically such as by fine-tuning AS path prepending on its BGP announcements, or negotiating cheaper peering rates. Unfortunately, AS1 often cannot tell if the received traffic actually originates from AS2, or if it is source-spoofed, as in Figure 1c.

Reconstructing inter-domain traffic paths is generally not possible from control plane information. AS1 may check if the source IP of the received traffic is owned by an AS in the customer cone of AS3, but this provides no usable information if the traffic is spoofed. Public route monitors [14, 34, 35] provide an incomplete coverage, and are not be helpful to detect all routing problems. For example, in Figure 1b, looking glasses and monitors do not report anything unusual because they do not cover the forwarding router at AS2. Even worse, traffic taking unexpected paths is simply not detectable at all via the control plane in many cases: for example, routes do not reflect the actual forwarding paths in the presence of BGP deflections [19], static routes or policy-based routing [11]. In the end, the ground truth lies in the actual data-plane traffic.

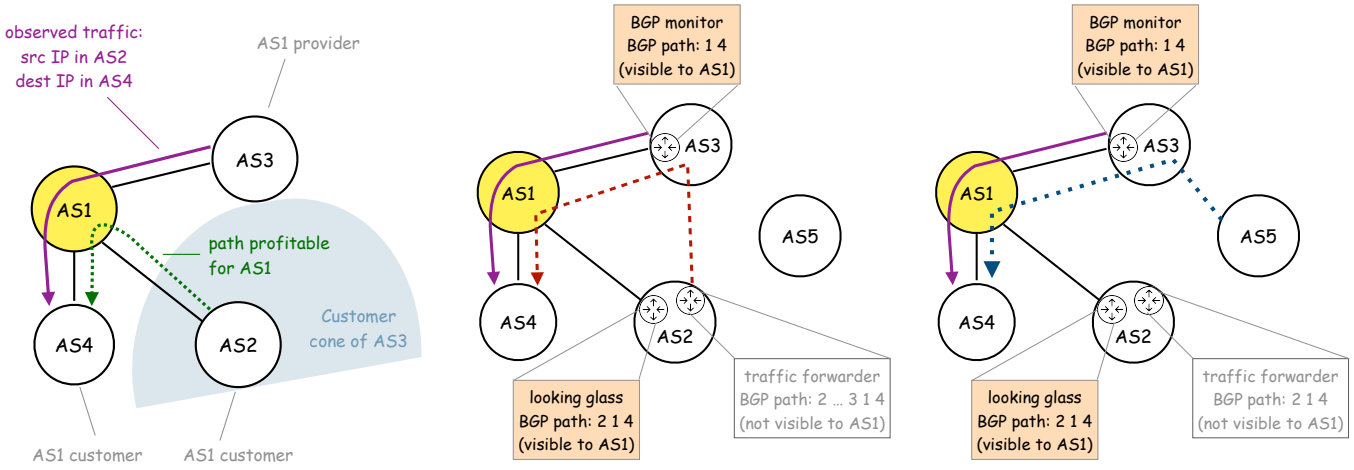
Yet, ISPs *cannot distinguish spoofed and non-spoofed traffic using passive measurements*, such as those extracted from packet counters or traffic mirroring. Even if the ISP looks at the transport headers, spoofed traffic can entail sequences of TCP packets with consistently increasing sequence numbers that look like any normal TCP flow. If the ISP sees packets both from a host A to a host B and from B to A, AS1 may be able to identify non-spoofed flows, but this is very rarely the case because of path asymmetry [4, 10, 44, 45]. How, then, can an ISP check ingress points for non-spoofed traffic?

In this paper, we present Penny, a traffic checker that interacts with transiting packets to assess if they are part of *closed-loop* and hence non-spoofed TCP flows. By closed-loop, we refer to a TCP flow where the two endpoints actually exchange packets with each other, reacting to received ACKs. The basic idea is simple: when new unexpected traffic is seen, Penny drops the occasional TCP packet and waits for its retransmission [1]. Most likely, the dropped packet will be retransmitted if it belongs to a closed-loop TCP flow, but it won't be resent if the traffic is source-spoofed, as a spoofing sender cannot know the drop occurred.

ACM SIGCOMM '24, August 4–8, 2024, Sydney, NSW, Australia

ACM ISBN 979-8-4007-0614-1/24/08
https://doi.org/10.1145/3651890.3672259

[1] Why "Penny"? The phrase "the penny dropped" means to finally understand something, but also "a bad penny always comes back".



(a) AS1 receives traffic from a provider (AS3) rather than from a customer (AS2).

(b) Option 1: actual unprofitable path, not detectable in the control plane.

(c) Option 2: spoofed traffic, not worth to be further investigated.

Figure 1: Case where ISP AS1 needs to assess if the observed traffic is closed-loop or not. Control plane information is not sufficient even when remote ASes host BGP monitors and looking glasses: BGP routes visible to AS1 are the same in Figures (b) and (c). We assume that AS1 does not see any traffic from AS4 to AS2 because inter-domain paths are often asymmetric [4, 10, 44, 45]: path asymmetry is *not* what makes the path unprofitable to AS1.

It is far from obvious, however, that such a simple idea can be turned into a practical and reliable traffic test in real ISPs. Indeed, ISPs have very little visibility on the many factors affecting their observations of packets and retransmissions, including external packet loss, remote routing changes, and the specific TCP implementation used by end hosts. Also spoofers may mimic closed-loop flows.

Penny relies on statistically evaluating the likelihood that closed-loop flows generated the observed retransmissions, missing retransmissions and duplicated packets. Its statistical core is reinforced with mechanisms to deal with several practical aspects, including timers and quirks of TCP, external network conditions, the impact of the test on the performance of probed flows, and the presence of spoofers aware of our test and actively trying to bypass it.

We evaluate Penny with ns-3 simulations of a wide range of traffic and network conditions. Penny always identifies closed-loop flow aggregates while causing minimal performance degradation. This remains true for different TCP implementations and any evaluated network condition that allows progress of the tested closed-loop flows. It correctly identifies spoofed traffic that actively tries to be identified as closed-loop. Finally, when legitimate flows are mixed with spoofed traffic from the same prefix, Penny's goal is to detect that unspoofed traffic is present – see, e.g., Figure 1. We show that Penny correctly finds the unspoofed flows in traffic aggregates even when the vast majority (e.g., 90%) of the packets is spoofed, though doing so requires extra analysis time compared to when all traffic is unspoofed or all is spoofed. Our implementation of Penny and code to reproduce its evaluation is publicly available [18].

We finally stress that the example in Figure 1 is far from being the only case where detecting unspoofed ingress traffic is valuable for ISPs. Appendix A describes additional scenarios, ranging from other examples of unprofitable paths to BGP hijack detection and support for new services.

2 Overview

It is easy to raise alerts, but what makes an alert *actionable*?

- An alert identifies a genuine anomaly; to do so requires an idea of what *normal* is.
- The false positive rate must be very low, even in the presence of malicious traffic generated with the explicit goal of triggering alerts.
- False negatives (missed anomalies) are allowable, but if the system is to be useful they should be rare for large anomalies.

For Penny, traffic is anomalous if it is *genuine* traffic entering a network at an *unexpected* entry point. This in turn raises two questions: what is genuine and what is unexpected?

Genuine traffic follows the path dictated by routing from its stated IP source to its IP destination, even if routing has made a mistake. In contrast, spoofed traffic may be following the routed path to the destination, but as it did not come from the location indicated by its source address, it cannot be used to check whether routing is performing properly. The fact that traffic is genuine says nothing about whether it is *good*. Genuine traffic may be actively malicious, but if it follows the routing and arrives at an unexpected entry point, we want it to raise an alert. Penny aims to detect genuine traffic and distinguish this from the spoofed background noise.

What makes an entry point unexpected? Penny is just a traffic checker, so does not really care about *why* an entry point is unexpected, but such a checker is only of use in a context where we can define what is expected. Traffic can be unexpected simply because an operator sees something odd and asks Penny "is that traffic genuine?". But Penny can also operate in the context of broader automated monitoring. Penny can be run in the background at a low rate to build up a map of normality as seen from genuine traffic, and then later identify changes from that normal baseline. Alternatively,

Penny can by triggered by a passive monitor (eg. using *sflow*[38]) that identifies traffic that does not match configured routing policies. Due to spoofing, such passive monitors raise too many false alarms to be actionable, but Penny can automatically check each and alert only on those caused by genuine traffic. Results can then be cached to avoid continuous retriggering. So how does Penny identify genuine traffic? Broadly, Penny looks for closed-loop TCP flows as an indicator that a traffic aggregate contains genuine flows. Due to routing asymmetry, Penny will usually only see one direction of a flow, but if it can probe TCP flows and prove they are closed-loop, then it can raise an alert.

To summarize: *the goal of Penny is to raise an actionable alert if the traffic under test contains closed-loop flows that are symptomatic of an unexpected interdomain path.*

2.1 Getting access to traffic

To check unexpected traffic, operators first need to have access to it. In principle, any tests could be performed within the ingress border router, but traditional routers offer only limited traffic monitoring capabilities – e.g., a few, predefined counters [9] and constrained access to their software (e.g., Cisco and JunOS SDKs). However, by its very nature, any router can redirect packets. In Penny, we thus opt for redirecting traffic for analysis by an external box.

Different ISPs have different PoP architectures, but we envision a deployment of the general form shown in Figure 2. The checker is the key component that actually tests ingress traffic. It can be asked to analyze traffic sourced from specific IP prefixes (e.g., those owned by AS2), entering at an unexpected port of a border router (e.g., R3 in Figure 2). To do so, it configures the border router to redirect the traffic to be tested so that it follows the solid red lines. Source-based forwarding can be achieved using route-maps in traditional routers [12] or with custom logic in programmable ones, such as Broadcom's Trident-series switches or Intel's Tofino. The prefixes to be checked simultaneously are chosen so that the redirected traffic does not overwhelm the network bandwidth and resources of the checker. Different implementations of Penny's checker are possible, ranging from dedicated hardware devices to general-purpose servers processing packets in software. Future implementations may also run directly on border routers.

2.2 How to test traffic?

For testing to be effective for the use case above and those in Appendix A, it must detect closed-loop traffic with high probability. How can the checker tell if the received traffic is closed-loop? There are a few options.

Passive traffic monitoring. Inferring the nature of a flow is relatively simple if we see packets both from client to server and server to client. We can match packets in the two directions, checking that sequence numbers and ACK numbers correspond. Unfortunately, Internet paths are usually asymmetric. An ISP only seeing packets in one direction cannot infer that closed-loop flows are not present.

Passively observing traffic flowing in a single direction cannot be relied on in general. Checking that sequence numbers consistently increase will spot simplistic spoofers, but cannot detect slightly less naive ones that increase sequence numbers of the sent packets without waiting for ACKs.

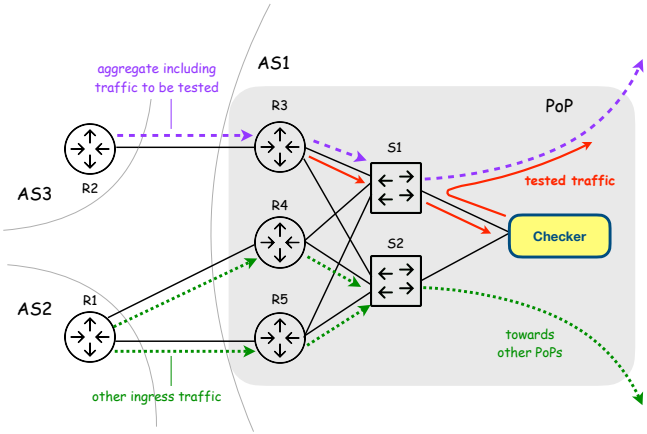


Figure 2: When asked to test unexpected ingress traffic, Penny's checker in the ingress PoP configures the receiving router (e.g., R3) to redirect just the traffic to be tested.

Interposing in traffic flows. To enable effective passive observation of flows, we can force the checker to receive all packets of the analyzed flows in both directions by performing network address translation (NAT). Suppose that we receive a SYN packet sent from host A to host B and we wish to monitor the flow. The checker can rewrite A's source IP address in the SYN packet to an address of the checker. B will send its response to the checker, which then swaps the destination address with that of A before forwarding back to A. Port translation may be needed too.

While this approach provides a reliable test, it has unintended consequences. NATing traffic in the middle of the network prevents geo-location and geo-blocking mechanisms and disrupts protocols such as SIP that embed IP addresses discovered using STUN [15] to enable local NAT traversal. NATing is also comparatively expensive - the checker must continue rewriting for the entire duration of the flow, even after it comes to a conclusion, or the flow will die. Perhaps worse, if routing between the source and destination changes, existing flows may now bypass the NAT, reach the destination without being rewritten, and will be reset.

Actively probing senders. A checker could take an active role and probe the traffic sender, for example by sending a SYN to it. If it receives a reply on the same ingress port as the tested traffic, it can conclude that non-spoofed traffic from that source network can indeed be received at that ingress port. While this approach has the advantage of not interfering with user traffic, it is also unreliable. Typically only TCP servers using public IP addresses would reply to the probes sent to them. Even for servers, ECMP routing may route one connection from a server via one path and a different connection from the same server via a different path.

Dropping packets. We would like a general and reliable approach to distinguish closed-loop from spoofed traffic while causing minimal impact on legitimate applications. None of the techniques above fulfills these requirements.

Dropping the occasional TCP packet and checking for its retransmission has potential. It is general because all TCP implementations

must retransmit lost data to ensure reliability. It should have limited impact on applications because all TCP implementations must effectively deal with low levels of packet loss to ensure reasonable performance in the Internet. We refer to this approach as the *packet drop test*. Hereafter, we describe the design and implementation of Penny's drop test. Section 8 discusses a possible integration with the other techniques presented above.

3 Packet Drop Test: Basics

Consider a traffic checker that sees TCP packets sent from host A to host B, but no packet from B to A. How can we build a reliable, robust and practical packet drop test?

First of all, we should only drop packets containing data: pure ACKs don't advance the sequence space, and they are cumulative, so we don't expect them to be resent.

Also, the signal provided by a *single* data packet is weak and noisy. When dropping a packet from a closed-loop flow, we may receive no retransmission – e.g., if the sender fails, or the retransmission follows a path that bypasses the monitor. When dropping a source-spoofed packet, we may also receive a "retransmission", if some packets are duplicated by the source or in the network.

A better approach is to drop a few packets until we gain enough confidence on the nature of the traffic. For a closed-loop flow, we expect to see retransmissions for most packets we drop; likely, we will also see a few unsolicited duplicates that are retransmissions of packets lost between the checker and the destination. For spoofed traffic, we rarely expect to see a retransmission, unless the traffic includes many duplicates.

Penny drops. Each received packet is dropped with some low probability p_{drop}. TCP struggles to make progress with more than 10% packet loss [36]. To limit the performance degradation of legitimate flows, p_{drop} should thus be significantly less than 10% – say, in the range between 0.1% and 5%. Section 5.4 discusses why this range is practical, and how operators can tune the value of p_{drop} depending on their goals and operational constraints. Section 6 evaluates the impact of our test on legitimate flows.

For each received packet, we classify it as a new one, a duplicate of a previous one, or a retransmission of a packet we dropped. To assess if the tested traffic is closed-loop, we then rely on a statistical model that combines observations of the overall number of packets received so far, retransmissions, and unsolicited packet duplicates.

Penny statistical model. The model, summarized in Figure 3, is based on comparing the relative probability of the two competing hypotheses that the observed flow is closed-loop (H1) and that it is not (H2). When we observe a retransmission of a packet we dropped, we gain confidence on the validity of H1 and lose confidence on the validity of H2 – and vice versa if we don't observe a retransmission. Penny computes the individual probabilities of H1 and H2 being valid, and checks if one dominates the other.

$P(H1)$ is the probability that the observed traffic is generated by a TCP sender always retransmitting non-ACKed packets, given a low but non-zero chance p_{noRTX}, such as 5%, that a retransmission is sent but Penny misses it. Internet devices do not typically load-balance packets of the same flow on different paths, and inter-domain routing is normally stable at the timescale of our tests. If during a test the path of a tested flow changes and permanently avoids the checker, the test will be inconclusive – this is fine. However, we also wish the test to cope with the sort of transient glitch that can happen during BGP reconvergence. For this reason, $P(H1)$ is p_{noRTX} raised to the number of times we don't see a retransmission for a packet we drop. If we see retransmissions for all the dropped packets, $P(H1)$ remains equal to 1. If we don't see one retransmission, $P(H1)$ is equal to p_{noRTX}, which is not low enough to allow a conclusion without more data. For further non-retransmitted packets, $P(H1)$ decreases geometrically with p_{noRTX}.

$P(H2)$ is the probability that a spoofing sender could match the observed retransmission behaviour by blindly duplicating some fraction of packets. $P(H2)$ depends on the fraction of unsolicited duplicates f_{dup} we measure, and the number n_{RTX} of retransmissions we see of dropped packets: likely, traffic is not closed-loop if we observe too many duplicates, or only see a few retransmissions. For example, $P(H2) = 1$ if no dropped packet is retransmitted.

We reach a conclusion when one hypothesis is 100 times more likely than the other, or if we receive an excessive number of unsolicited duplicates, incompatible with a packet loss rate that would allow any TCP flow to progress.

From theory to practice. The test described so far rely on an over-simplified abstraction of TCP. The devil, as always, is in the detail. Sections 4 and 5 describe a practical implementation of the packet drop test. Figure 4 provides an overview.

Hypotheses

$H1$: hypothesis that the flow under test is closed-loop
$H2$: hypothesis that the flow under test is not closed-loop

Parameters

p_{drop}: probability of dropping a TCP data packet
p_{noRTX}: assumed probability that we do not observe a retransmitted packet within a closed-loop flow

Measurement counters

n_{RTX}: number of observed retransmissions for packets we dropped
n_{noRTX}: number of packets we dropped for which we didn't observe a retransmission
f_{dup}: fraction of observed packets with one or more duplicates

Probabilities

$P(H1) = (p_{noRTX})^{n_{noRTX}}$
$P(H2) = (f_{dup})^{n_{RTX}}$
$P(genuine) = P(H1)/(P(H1) + P(H2))$

Procedure: For every received packet of the flow under test, update counters and possibly drop the packet, with probability p_{drop}. Whenever $P(genuine) > 0.99$, conclude that the flow is closed-loop. Whenever $P(genuine) < 0.01$ or $f_{dup} > 0.15$, conclude that the flow is not closed-loop. Stop dropping packets as soon as dropped packets are enough to reach a conclusion.

Figure 3: Overview of Penny statistical model.

4 Packet Drop Test at Runtime

We now detail how to run the approach described in Section 3 on live traffic, using the example in Figures 5-6 as support.

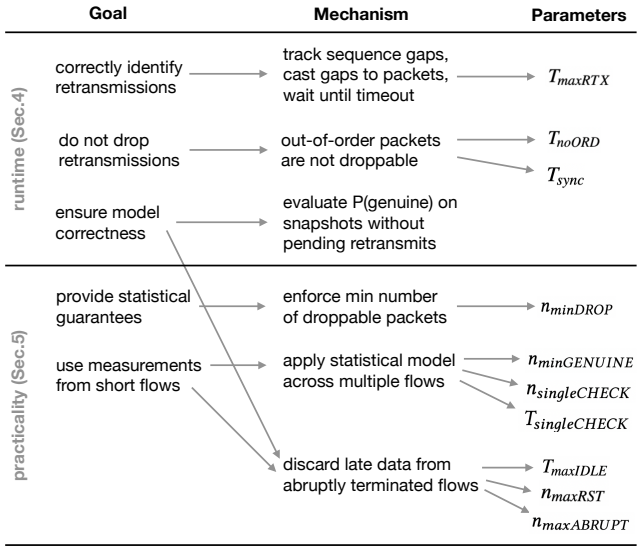


Figure 4: Overview of Penny drop test design.

4.1 Retransmissions

Our test relies on collecting evidence of retransmissions. How can we detect retransmissions, though?

In its simplest incarnation, a retransmission is a TCP packet identical to one that we have already seen. However, TCP does not guarantee that retransmitted packets are identical to non-ACKed packets. Indeed, the bytes to retransmit can be resent in smaller packets, for example if the path MTU changes; or, conversely, they can be merged with fresher application data to form bigger packets [32], such as packet P8 in Figure 5.

Definition. From the above examples, it is evident that we should track the TCP sequence numbers of the received packets, and in particular the gaps in such sequence numbers. For brevity, we refer to such gaps as *sequence gaps*.

Dropping a packet creates a specific sequence gap: the start of the sequence gap is the first byte of the dropped packet, and the gap length is the number of bytes in the payload of the dropped packet. For example, dropping P5 in Figure 5 creates a gap spanning sequence numbers between 40 and 50.

We then define retransmissions by casting closed sequence gaps to packets. For any packet we drop, we can indeed say that the packet is retransmitted when all its bytes are resent.

Our *gap-to-packet casting* mechanism works as follows. Every time we receive a packet, we check if the bytes in the packet fill a sequence gap opened by one of our drops. If so, we classify the packet as a retransmission, and discard the gap. Otherwise, the packet is either a duplicate if it includes bytes lower than the current highest sequence number for the flow, or a new packet otherwise.

To complete the above definition, we need to address two corner cases of *packets not perfectly fitting inside or outside any gap*, such as packets P7 and P8 in Figure 5. First, we may receive a packet that covers only part of a gap. In this case, Penny resizes the gap, and does not classify the packet neither as retransmission nor as a duplicate. Second, we may receive a packet that includes bytes both in a gap and outside it, such as P7 in Figure 5. In this case, we break down the packet into two: one sub-packet including only the bytes within the gap, and another only including the bytes outside the gap. If a packet spans multiple gaps, we break it down into multiple sub-packets. Finally, we apply our gap-to-packet casting to each sub-packet. A retransmission is recorded only when an entire sequence gap is closed – e.g., after P8 in Figure 5.

Computation. We now know how to classify every received packet as retransmitted, duplicate or new. When do we decide that a dropped packet is *not* retransmitted, though?

Clearly, we need a timeout after which we mark a dropped packet as not retransmitted. However, we cannot make assumptions on the exact retransmission time: the choice of when a sender retransmits any packet depends on both the sender's congestion control algorithm and its implementation. For example, the Linux implementation of TCP Reno has a min retransmission timeout (RTO) of 200ms, but the Windows implementation uses a default min RTO of 300ms. Other implementations and congestion algorithms such as [2, 8, 20] may also influence the retransmission time.

In Penny, our top-level concern is to classify retransmitted and non-retransmitted packets *correctly*. We thus mark a dropped packet as not retransmitted only after a conservative timeout T_{maxRTX}. By default, T_{maxRTX}=3 seconds, to cover cases of high-RTT flows for which the first couple of retransmissions are dropped between the sender and Penny. Operators can change the value of T_{maxRTX} to achieve different tradeoffs between test speed and accuracy.

4.2 Droppable packets

Our statistical model is based on probabilistically dropping data packets. To limit its performance impact, we should refrain from dropping retransmissions, as doing so would slowdown subsequent retransmissions, and may eventually stall data transmission.

To avoid dropping retransmissions of packets dropped by Penny or downstream of it, we can just remember the sequence numbers already seen. However, when a packet is dropped between the sender and Penny, its retransmission would look like a new packet, except that its sequence number is out-of-order.

Definition. We denote any packet that Penny can drop as *droppable packet*. To avoid dropping retransmissions, we classify a packet A as droppable if upon its reception, A has the highest sequence in its flow: this guarantees that A is not out-of-order. For example, packet P3 in Figure 5 is not droppable. Duplicates, such as packets P4 and P7 in Figure 5, are also not droppable. We then apply our statistical model (shown in Figure 3) to droppable packets only.

The above definition implies that *any out-of-order packet is not droppable*, including those not triggered by a loss. This has a risk: if we receive most packets out of order (e.g., because the source sends them this way, erroneously or deliberately), very few packets will be droppable, and we might not be able to run our test. We explicitly flag these abnormal cases by tracking the number of out-of-order and in-order packets. If their ratio exceeds an *out-of-order threshold* T_{noORD}, set to 80% by default, we mark the test as inconclusive.

Computation. How can we compute droppable packets on live traffic? If we monitor a flow from its start, we can track the sequence

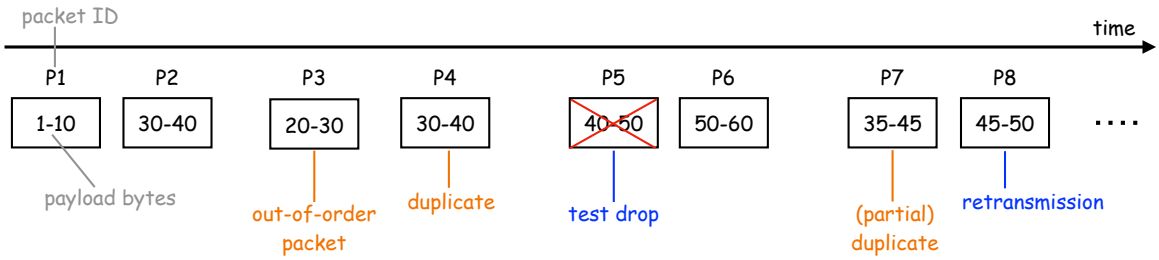


Figure 5: Illustration of some of the cases that the packet drop test must handle in practice for data packets in a flow.

numbers from the TCP handshake, and directly apply the definition of droppable packets. Unfortunately, we may also need to test *already-started flows* such as closed-loop flows already exchanging data, or spoofed traffic that does not simulate the TCP handshake.

Computing droppable packets for already-started flows requires some care. The size, content and evolution of the TCP congestion window heavily depend on the specific implementation used by the sender amongst the many deployed nowadays [8, 20, 21]. So, it is hard if not impossible for a device in the Internet core to quickly infer the congestion window and packets in-flight of crossing flows, and compute the highest sequence number for already-started flows.

We again adopt a conservative approach. We collect sequence numbers of observed packets for a *flow sync time* T_{sync} much longer than the expected RTT of any Internet connection – i.e., 3 seconds by default. After T_{sync}, we should have seen a few windows of packets, and it is thus safe to consider the highest sequence number observed at that time as the highest sequence number in the TCP flow.

4.3 Evaluating the hypotheses

The packet drop test is based on evaluating probabilities of competing hypotheses. Figure 3 provides a static view of how those probabilities are computed, but it does not account for the time needed to collect the counters used in the formulas.

A correct implementation of the packet drop test must ensure the consistency of the counters whenever the hypotheses H1 and H2 are evaluated. This is not completely trivial because different counters are updated at different rates, due to factors outside our control. For example, in Figure 5, we can assess that P5 is retransmitted only after receiving P8; in the meanwhile, P6 and P7 would change f_{dup}.

Counting retransmissions is generally slower than observing new packets, and counting non-retransmitted packets is much slower still (see Section 4.1). So, if we just run our statistical analysis every time we receive a packet, we incorrectly consider fewer retransmitted packets than we should, and even fewer non-retransmitted ones.

Definition. At any time t during a packet drop test, we define *pending retransmissions* as the dropped packets for which at t, we are still assessing if they are retransmitted or not. In other words, a pending retransmission is a packet for which we have not received a retransmission yet, and the retransmission timeout T_{maxRTX} defined in Section 4.1 is not expired. For example, packet P5 is a pending retransmission when P6 is received in Figure 5.

We use pending retransmissions to identify the values of counters used in the evaluation of H1 and H2. Namely, we evaluate the two hypotheses on *sequence numbers delimited by the same pending retransmission*. This ensures that the hypotheses are assessed on packets which are all classified as new, duplicate, retransmitted or non-retransmitted. In our running example, for instance, we evaluate H1 and H2 with counters for packets with sequence number up to 40, because P5 is a pending retransmission.

Computation. To track counter values delimited by pending retransmissions, we use *counter snapshots* defined as follows. Whenever we drop a packet, we take a snapshot of observed, duplicate and retransmitted packets at the time of the drop. When created, each snapshot includes at least one pending retransmission. Whenever we receive a retransmission or a retransmission timeout expires, we update all the snapshots including the corresponding pending retransmissions – e.g., updating the number of duplicates and decreasing the number of pending retransmissions. We also compute the set S of counter snapshots including no pending retransmission anymore, and we evaluate H1 and H2 on the counters in the most recent snapshot in S. If the test reaches a conclusion, we stop and output the result. Otherwise, we discard all the snapshots in S, and keep running the test.

Figure 6 provides an illustration. After dropping P5 we instantiate a counter snapshot. Later on, the reception of P8 closes the sequence gap created by the drop of P5, and triggers the update of the snapshot: the counter of pending retransmissions is set to 0, the number of retransmissions is set to 1, and the counters of droppable and duplicate packets are increased by 1, to account for P6 and P7, respectively. Since the snapshot includes no pending retransmissions anymore, we then compute P(genuine) with the counters in the snapshot.

5 Ensuring Practicality

Section 4 describes a runtime implementation of our test. In this section, we further evolve Penny's design to deal with a range of real-world factors and network conditions.

5.1 Dealing with worst-case traffic patterns

In Penny, the goal is to correctly detect any closed-loop TCP flow included in the aggregate under test. Traffic that evades, deliberately or by chance, our statistical analysis (e.g., because they include more than 15% packet duplicates or many non-droppable packets) would only improve Penny's performance, as it causes the drop test to focus on closed-loop flows. Misclassifying closed-loop traffic as spoofed is also not disastrous: when this happens, Penny fails silent, not raising any alert, which neither improves nor worsens the status quo.

The worst-case scenario for Penny is instead represented by spoofed traffic with a number of duplicates just below f_{dup}, as

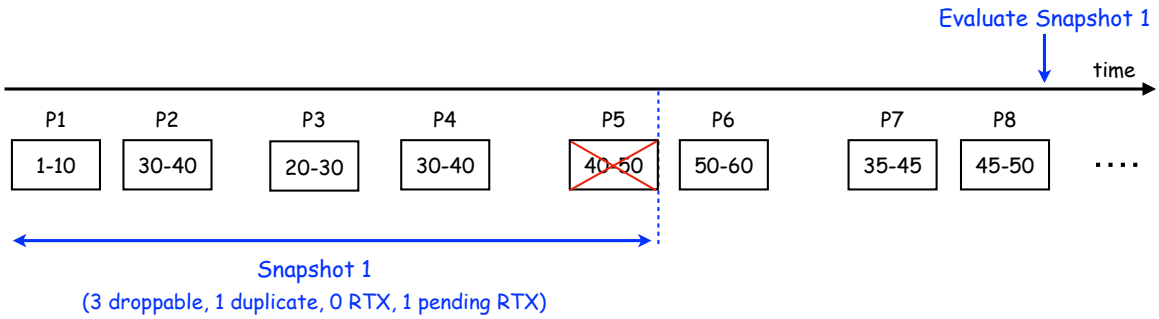


Figure 6: Illustration of counter snapshots in our reference example.

this has the maximum probability of being reported as closed-loop. Doing so would trigger false alarms, and possibly induce operators to ignore future alerts – see again Figure 1.

Figure 3 defines no constraint on the counter values, and hence on the number of statistical samples, to evaluate P(genuine). However, statistics computed on very few samples are more likely to be biased. For example, we experimentally confirm that spoofed flows can be confused with closed-loop ones if the sender duplicates 2 packets in the first 12, and we drop exactly these 2 packets.

To prevent such cases, we add a parameter $n_{minDROP}$, corresponding to the *minimum number of dropped packets* that we must classify as retransmitted or not retransmitted before concluding the test. That is, we never conclude a test before dropping at least $n_{minDROP}$ packets and assessing if they are retransmitted or not. The value of $n_{minDROP}$ directly controls the tradeoff between test accuracy vs its speed and intrusiveness: higher values of $n_{minDROP}$ increase our confidence in the outcome of the test, but they also make the test slower and potentially more impactful to end users.

To decide the value of $n_{minDROP}$, we propose a methodology based on (i) analytically computing the probability to misclassify spoofed flows as closed-loop, and (ii) deriving the value of $n_{minDROP}$ from operators' preferences for such probability. We now exemplify our methodology, and use it to define the default $n_{minDROP}$ value.

Figure 7 shows the analytically computed worst-case probability of misclassifying spoofed traffic as closed-loop (y axis) against the number of droppable packets (x axis), when p_{drop} is 5%. We take p_{drop}=5% as an example, but all the following observations also apply to other values of p_{drop}. Fundamentally, indeed, Figure 3 shows that droppable and dropped packets do not contribute to any of the probabilities evaluated by Penny, except for f_{drop} which is however a fraction. We empirically confirm that the relationship between misclassification probability and dropped packets remains the same for lower values of p_{drop}, as detailed in Appendix B.

Each curve in Figure 7 corresponds to the highest misclassification probability when the sender randomly duplicates a certain number of packets, up to 45. We don't consider more than 45 duplicates because with more duplicates, we would always exceed f_{dup} (i.e., 15%). All curves have a similar shape, peaking at a relatively low x value, and quickly decreasing afterwards.

From Figure 7, we can set $n_{minDROP}$ according to operators' preferences. By default, we aim to never report spoofed flows as closed-loop in order to avoid false alarms (see, e.g., Figure 1). So we set the default $n_{minDROP}$ to the conservative value of 12, which maps to an error probability lower than 10^{-4}, as for example shown in Figure 7 at x=240 droppable packets. Operators can set a lower (resp., higher) value of $n_{minDROP}$ if they want to reduce (resp., increase) the test duration in exchange for weaker (resp., stronger) correctness guarantees.

5.2 Dealing with short flows

Section 5.1 raises the question of how to handle short flows: if we run a test on a flow that terminates before we drop $n_{minDROP}$ packets from it, our test will be inconclusive.

While ignoring short flows may be fine in some cases, it does not work well in general. First, we may simply have no long flows – e.g., if all the flows we receive contain a single HTTP GET. Second, only testing long flows (if any) may misrepresent the nature of the received traffic. For example, if there are many closed-loop short flows and one spoofed long flow, testing only the long flow would lead us to wrongly conclude that all the traffic is spoofed.

Fortunately, we can *extend our approach to test arbitrary traffic aggregates*, including both short and long flows that share the same source prefix IP addresses. The key observation here is that each packet is a separate sample in our statistical model, independent from

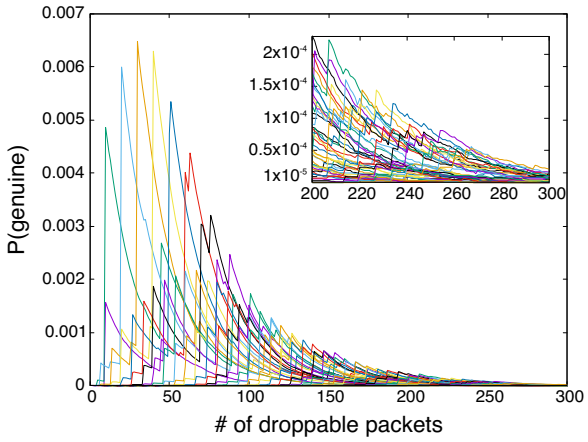


Figure 7: Worst-case probability to misclassify spoofed traffic as closed-loop when p_{drop} is 5%. Each curve corresponds to a fixed number of duplicates between 2 and 45.

any other sample or previous drop. We can thus aggregate counters across multiple flows, and use our statistical model, unmodified.

When aggregating counters, we must however preserve the property that H1 and H2 are evaluated on consistent sets of packets. To do so, we apply the same intuition described in Section 4.3 across all tested flows. More precisely, every time we drop a packet from any flow, we compute a counter snapshot with observed, duplicate and retransmitted packets from the entire flow aggregate. Whenever we receive a retransmission or a retransmission timeout expires, we update all the snapshots, and re-evaluate P(genuine) on the most recent snapshot with no pending retransmission after the update.

Applying the packet drop test to flow aggregates is not exactly the same as applying it to individual flows. Checking aggregates speeds up the test and makes it less intrusive, as collecting samples from multiple flows is faster and reduces the drops per flow. However, the semantics of the test also change: when applied to flow aggregates, the test does not relate to any specific flow, but evaluates if the overall traffic mostly behaves as belonging to closed-loop flows.

For cases where some flows are closed-loop and others are not, we *deliberately bias our test towards detecting closed-loop flows*, as those flows are the most important to detect in our use cases (e.g., see Figure 1). Namely, when our test concludes that an aggregate is closed-loop, we directly report the traffic as closed-loop. If instead the aggregate appears not to be closed-loop, we perform packet drop tests on individual flows: we check flows until we reach a number $n_{singleCHECK}$ of test results, or we exceed a timeout $T_{singleCHECK}$. We finally report the traffic as closed-loop if we find a minimum number $n_{minGENUINE}$ of closed-loop flows. By default, we set $n_{singleCHECK} = 100$ and $T_{singleCHECK} = 10$ seconds, to collect results on a reasonable number of flows, and set $n_{minGENUINE} = 2$, to avoid being misled by a single flow.

This approach tends to find closed-loop flows even if the majority of the traffic is spoofed, as we show in Section 6. We may still misclassify traffic if the only closed-loop flows are short and most of the traffic is spoofed. We further discuss this limitation in Section 8.

5.3 Dealing with abruptly interrupted flows

Penny can easily classify flows as terminated if it observes FIN packets for them. Unfortunately, any packet stream received by an ISP may also stop abruptly – e.g., if BGP paths change, an endpoint crashes, or an attacker stops spoofing.

For such *abruptly interrupted flows*, Penny suddenly stops seeing any packet. This raises two problems. First, we don't know how long to wait for new packets before deeming the flow as terminated. Second, if we are waiting for retransmissions, we don't know if the source would have sent them should the flow have continued.

A special case of abruptly interrupted flows entails TCP *RST packets*. RST packets were originally designed to signal loss of state at one of the flow endpoints, but modern TCP implementations also employ these packets to expedite the termination of a connection [1][2]. So, receiving a RST packet introduces uncertainty on pending retransmissions.

Counter snapshots (see Section 4.3) ease dealing with abruptly interrupted flows, including those with RST packets. We indeed classify a flow as terminated when we see a RST packet, or when a conservative timeout $T_{maxIDLE}$ (i.e., 30 seconds by default) expires. When this happen, we update all the counter snapshots by discarding any pending retransmission and observed packet for that flow.

We also include a threshold $n_{maxABRUPT}$ for the maximum ratio of abruptly interrupted flows per aggregate, set to 80% by default; and a threshold n_{maxRST} for the max number of RST packets per flow, set to 2 packets by default. The thresholds are meant to flag suspicious cases, expected to be rare for closed-loop traffic. We deem a test as inconclusive when either of the two thresholds is exceeded.

5.4 Configuring the packet drop rate

Armed with the full design of Penny, we are now ready to properly discuss the tradeoffs involved in setting p_{drop}, the probability that Penny drops any droppable packet.

Penny stops dropping packets as soon as it collects enough evidence that the tested traffic is highly likely to be either closed-loop or spoofed (Figure 3). In practice, each drop test is expected to drop very few packets. We find that Penny generally reaches a conclusion on a traffic aggregate after *dropping exactly 12 packets in total per test*. A lower bound comes from setting $n_{minDROP}$ =12, but across all our experiments with closed-loop, spoofed and mixed traffic we saw no case where Penny required more drops. This is also consistent with the experiments in Section 6.1.

Penny is designed for infrequently testing unusual traffic, not for continuously monitoring the same traffic aggregate. Given the low number of packets dropped from any anomalous traffic aggregate, there are few critical constraints on the value of p_{drop} other than allowing the flows under test to progress. Thus, we envision that any value between 0.1% and 5% may be practical for p_{drop}.

What should operators consider when setting the per-packet drop probability? In practice, p_{drop} affects: (i) the test speed, (ii) the likelihood that the test can be applied to short individual flows, and (iii) the worst-case per-flow performance degradation. Higher values of p_{drop} generally reduce the test duration and increase its applicability to individual short TCP flows, but also slightly increase the performance degradation caused to such individually tested TCP flows. We quantify this effect further in Section 6.2.

When applied to a flow aggregate with at least a few tens of flows, a Penny test likely drops only one packet per flow for any value of $p_{drop} \leq 5\%$, so any performance degradation is very limited. We verify this in Section 6.2. In such cases, around twelve flows lose a single packet, whether p_{drop} is 0.1% or 5%, so higher values may be preferred because they give quicker results. In practice, for high rate aggregates, the test speed may be limited by Penny's need to wait for retransmissions or abruptly interrupted flows. With timeouts on the order of 30 seconds, there can be little benefit from using p_{drop} greater than 0.1% for any aggregate faster than about 5Mbps.

When applied to individual flows or to aggregates with only a few flows, performance degradation may be more of a concern. Our evaluation shows that Penny has a relatively small impact on the completion times of individual TCP flows even when $p_{drop} = 5\%$, as it carefully selects which packets to drop and stops dropping after a few packets. An operator can easily configure Penny to apply different p_{drop} values to aggregates and to individual flows so as to balance performance concerns against test duration.

[2]we experimentally confirmed that Chrome and MacOS Safari commonly use RST packets to terminate TCP connections for video streaming.

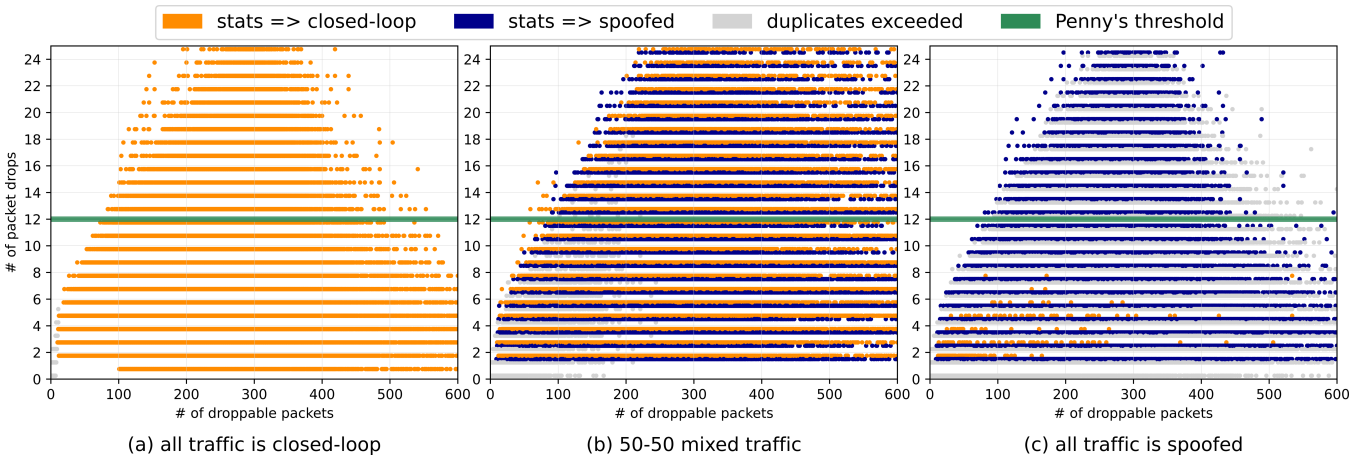


Figure 8: Accuracy of Penny's statistical model. Outcomes below y=12 (green horizontal line) are ignored by Penny because of the $n_{minDROP}$ parameter. Penny correctly classifies closed-loop and spoofed traffic aggregates. For mixed traffic, Penny either classifies the aggregate as closed-loop, or triggers a test to individual flows.

6 Evaluation

We evaluate Penny in ns-3 [40]. We implement the packet drop test in C++, create an ns-3 switch running our test implementation, and simulate TCP flows and spoofed traffic crossing this switch. In our experiments, we try to choose settings that are deliberately unfavorable to Penny. Notably, the traffic tested by Penny has a high RTT (i.e., 100 ms), competes with established background TCP flows, and we optionally add extra loss so that not all retransmissions are caused by Penny. Our results thus measure accuracy for flows relatively slow to retransmit, and performance impact on flows likely to be in slow start when the first packet is dropped.

6.1 Accuracy

We want to see that Penny never mischaracterizes aggregates containing only closed-loop flows, or those only with spoofed traffic. Also, closed-loop flows should be detected even when they are mixed with large amounts of spoofed traffic from the same source prefix.

Figure 8 shows the results of experiments on different types of aggregates, network settings (e.g., 10-300 Mbps links) and Penny drop probabilities (between 1% and 5%). In Figure 8a all traffic is closed-loop, in 8c it is all spoofed, whereas in 8b it is a mix of the two. Each point shows the preliminary decision from Penny's statistical model—*closed-loop*, *spoofed*, or *duplicates exceeded*—after a specific number of droppable packets have been observed (x-axis) and after Penny has dropped a specific number of packets (y-axis). Penny will only make a decision above the green line.

The packet drop test is always correct in the absence of spoofed traffic. We run 10,000 experiments, each testing an aggregate consisting of 100 closed-loop TCP flows with randomized start times and no random packet loss on any link. The precise number of flows does not greatly matter here, because Penny makes a decision based on drops and retransmissions striped across all the flows in the aggregate. In all these experiments, Penny drops exactly $n_{minDROP}$ =12 packets, and correctly classifies the aggregate as closed-loop. This observation is also consistent with Figure 8a, where only orange points are positioned above the green line.

To verify that external packet loss does not affect Penny's accuracy, we repeat the experiments with random link losses in addition to Penny's drops. Since TCP struggles to make progress when the loss rate exceeds 10%, we experiment with random link losses of 1%, 3%, 6%, 9%, and 12%, upstream of Penny (it will see gaps, then out-of-order packets), downstream of Penny (it will see duplicates), or both. Penny always correctly classifies the traffic as closed-loop.

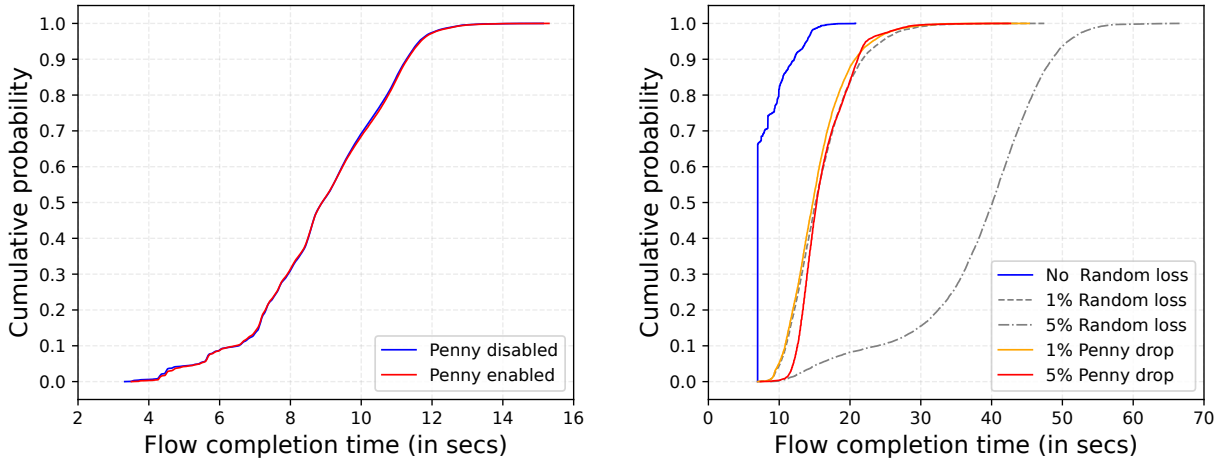
We also re-run a sample of the above tests with different TCP congestion control algorithms including LinuxReno, NewReno and Cubic, as well as different retransmission strategies, such as enabling and disabling SACK. We also test both RED and tail-drop queuing. The results remain the same.

The packet drop test is always correct in the absence of closed-loop traffic. We run 10,000 experiments with aggregates comprising only spoofed traffic. We mimic 1,000 flows using different source-destination addresses and with increasing sequence numbers. To try and defeat the test, spoofing sources duplicate every packet with a probability of 14.9%, just below Penny's f_{dup} threshold of 15%. This is a *worst-case scenario*, since it maximizes the chances that blind duplicates look to Penny like retransmissions.

When testing the traffic aggregate, Penny always reaches a conclusion after dropping exactly $n_{minDROP}$ = 12 packets. It always reports the tested traffic as spoofed, or it hits the threshold for an excessive number of duplicates[3]. Figure 8c shows that there are incorrect decisions below the 12-drops line, where the spoofer's duplicate strategy would have defeated the test, but none above.

What can we conclude from this result? First, it is important that Penny never finds a closed-loop aggregate when there are no closed-loop flows. But can we conclude that the aggregate is spoofed? Unfortunately, as we will see below when we discuss mixtures of spoofed and closed-loop flows, for our purposes we cannot trust the

[3]we may exceed f_{dup} in some experiments because sources randomly decide if to duplicate a packet or not.



(a) Penny's impact when run on aggregates of 100 x 1MB flows. This is the only impact for aggregates classified as closed-loop.

(b) Penny's impact when run on individual flows – e.g., within aggregates with high volume of spoofed traffic.

Figure 9: Penny has limited impact on genuine, closed-loop traffic.

aggregate test when it concludes the aggregate is spoofed. In such cases Penny switches to test individual flows (see Section 5).

Where there are really no closed-loop flows, as in Figure 8c, in 99.9992% of the cases, Penny's individual flow test correctly finds no closed-loop flow. In the remaining 0.0008% cases, it incorrectly classifies *one* flow from the entire aggregate as closed-loop. This broadly agrees with the probabilities analytically computed in Figure 7, given that millions of flows are analyzed across our experiments. Even in these cases, Penny correctly classifies the aggregate as spoofed, since $n_{minGENUINE} = 2$.

In addition to synthetic traffic, we experiment with all the CAIDA traces [7] collected between 2014 and 2017. As these are recorded traces, there is no live sender to retransmit, so we replay them in an open-loop fashion. The goal is to check Penny's behavior in the presence of weird traffic patterns seen in the Internet. Penny always correctly concludes that the replayed traffic is not closed-loop.

The packet drop test does find long closed-loop flows even when spoofed traffic is prevalent. We finally run simulations with aggregates with both closed-loop flows and spoofed traffic. Once again, we experiment with Penny's *worst-case scenario*. In this case, spoofed traffic never includes any duplicate packet: this maximizes the number of dropped packets not being retransmitted, and hence attempts to bias Penny towards concluding that the entire aggregate is spoofed.

The outcome with mixed traffic aggregates depends on the fraction of *packets* in closed-loop flows, while per-flow tests depend on the fraction of closed-loop *flows*. We therefore evaluate scenarios where only 10%-20% of the flows per aggregate are closed-loop, and packets from closed-loop flows represent 50%, 20% and 10% of the total traffic in the tested aggregate. For each of these packet ratios, we run 10,000 experiments. Each closed-loop flow contains enough packets for the test to reach a conclusion on that flow.

Figure 8b shows that with 50% of the traffic spoofed, Penny concludes that the aggregate includes closed-loop traffic in many cases, but far from always. This effect is more pronounced when 80% or 90% of the traffic is spoofed (see Appendix B). Penny switches to testing individual flows when the aggregate is not classified as closed-loop. In each of our experiments, Penny finds at least $n_{minGENUINE}$ = 2 closed-loop flows, and hence correctly concludes that the mixed aggregates include closed-loop traffic.

6.2 Practicality

We now evaluate the practicality of Penny, in terms of both impact on user performance and implementability.

Flow performance degradation. We don't expect packet drop tests need to be run very often, but when they are run, we want the impact on end users to be minimal. We run experiments using the same topology, flow sizes, bottleneck and RTT, with and without Penny running a drop test, and compare the flow completion times. We first start background traffic that fills the bottleneck, then start a hundred flows from the aggregate to be tested with start times randomized over one second.

If most traffic in an aggregate is closed-loop, Penny's aggregate test directly reaches a correct conclusion with negligible impact on flow completion times, as shown in Figure 9a. The impact is so small because Penny drops only $n_{minDROP}$ = 12 packets in total, which typically affects only 12 flows, causing one packet drop per flow.

When Penny concludes that the aggregate traffic is spoofed, it switches to collecting $n_{singleCHECK}$ (100 by default) test results from individual flows. Given that the aggregate was originally classified as spoofed, only some of the individually tested flows are likely to be closed-loop. What is the impact of Penny on the few closed-loop flows Penny does test?

Figure 9b shows the impact of Penny testing 1MB-long Cubic flows. Results for NewReno are similar. 1MB flows tested with Penny take about 2-3x longer to complete. The impact on longer transfers (not shown) is less.

We see that irrespective of whether Penny drops 1% or 5% of the packets, the impact on flows is similar to a random loss rate of 1% and much less than a random loss rate of 5%. Why is this so? First,

Penny almost always reaches a conclusion after dropping 12 packets. When Penny drops 5% of packets, it comes to a conclusion quickly and drops no more packets allowing the flow to recover, whereas when it drops 1% of packets it drops the same number of packets but they are spread out through more time. In contrast, 5% random loss keeps dropping packets throughout the flow. Second, Penny specifically selects droppable packets to minimize performance degradation – e.g., it does not drop retransmissions.

Overall, Penny's impact on closed-loop flows looks more than acceptable, especially considering that Penny individually tests only a relatively small number of unspoofed flows received at unexpected routers, and only if they are mixed with a significant amount of spoofed traffic. Penny is also likely to have even less impact on hosts using modern congestion control algorithms, such as BBR [8] and COPA [2], which are less sensitive to packet loss.

Penny deployment. Finally we'd like to understand the hardware requirements and scalability of a Penny checker. We expect that network resources can be easily provided for the style of deployment envisaged in Figure 2. Packet processing is also expected to be not very challenging. Penny requires packet processing only slightly more complicated than forwarding in a software router, which can be performed at around 100Gb/s per core on today's hardware. Compared to a software router, Penny needs to look up flow state and update counters, but does not need to compute longest prefix match.

Penny does however hold per-packet state when waiting for retransmissions or for sequence gaps to fill. How much memory is needed for this state? We feed our Penny implementation with an increasing number of packets, artificially creating an increasing number of sequence gaps of various sizes, and measure how total memory usage grows with packet count. Results indicate that the memory consumption converges on 100-150 bytes per packet, irrespective of the number and size of sequence gaps. For example, holding state for 500 packets requires less than 100KB. Thus, so long as the total packet rate being redirected is managed, a Penny checker should easily be able to hold state to test thousands of aggregates simultaneously. The limit is in fact more likely to be the route-map state on the router performing the redirection.

7 Related Work

Today ISPs have very limited visibility on the traffic crossing their networks, mainly due to the combination of high traffic volume, large network size and lack of appropriate tools. For example, standardized protocols like SNMP [9], Netflow [13] or sflow [37] tend to provide aggregate information on the most popular destinations only [6].

A few recent contributions try to improve traffic visibility in large networks. Planck [39], Everflow [46] and Stroboscope [43] enable to control packets copied by routers to an external analyzer. Flowyager [41] eases the analysis of pre-collected measurements. Magnifier [6] combines sampled data and (negative) packet mirroring to track traffic ingress and egress points. TIPSY [30] describes a system to predict traffic ingress points depending on BGP announcements, within a large content provider and for traffic engineering purposes. Assuming that they could be deployed in ISP, more general techniques, such as in-band telemetry (e.g., [31, 33]), may enable per-flow or per-packet monitoring.

All the above approaches however focus on collecting traffic without discriminating between closed-loop flows and spoofed packets. Hence, they cannot be used to address use cases like the one depicted in Figure 1. To support such use cases, Penny assesses the nature of (unexpected) traffic entering at specific routers.

We note that our packet drop test can also be used to improve the accuracy and usefulness of existing monitoring systems, as it enables to discard spoofed traffic from the measurements that they report. Similarly, Penny can also improve the performance and robustness of in-router data-plane systems that analyze the forwarded traffic – e.g., to infer remote failures [22] or diagnose performance problems [17].

Our work is motivated by the prevalence of source spoofing in the Internet. Many efforts have been dedicated to prevent spoofing over the past decades, including the publication of Best Current Practices, such as BCP38 [16] and BCP84 [3], and proposals like SAVI [5]. Despite them, spoofing remains an unsolved problem. Recent studies [25, 26, 29] estimate a low adoption of spoofing mitigation techniques across the Internet. Reluctance to deploy anti-spoofing mechanisms is primarily driven by the perceived lack of economic benefits for ASes, as also highlighted in [28]. Given such a status quo, it is not surprising that source spoofing is often used within security attacks such as denial of service ones [24].

8 Discussion

We now discuss Penny's limitations and possible extensions.

Overcoming the limitations of the packet drop test. There are a few cases where Penny may not work well. Prominently, the individual flow drop test tends to be inconclusive when run on very short flows (less than about 250KB). If an aggregate consists primarily of short closed-loop flows, this will not be a problem—we drop a few packets from many different flows—but if a small number of closed-loop flows are mixed with a large amount of spoofed traffic, Penny will be unlikely to find them. In practice, this may not matter—the unspoofed traffic may be too little to care about—but if it does matter, there are several options available.

If the drop test is run periodically (e.g., daily) on traffic classified as spoofed, Penny will likely detect closed-loop traffic eventually. The rationale is that spoofed traffic tends to be part of a specific attack, so will only last for a limited time, whereas legitimate traffic tends to be more persistent.

A further option is to combine Penny drop tests with other techniques such as those discussed in Section 2, in spite of their drawbacks. Suppose for example that a aggregate test says "spoofed", but the individual flow tests are inconclusive because all the flows are too short. The main drawbacks of the NAT test are for longer flows - rerouting events are very unlikely during a very short flow. We can NAT the next flows from this aggregate and get a reliable conclusion rapidly. In such cases it would be advisable to avoid testing flows for protocols known to embed IP addresses, notably SIP and FTP.

Dealing with QUIC. Penny can directly test any flow aggregate that includes some TCP flows. While QUIC [23] is quickly growing in popularity, we don't expect that many traffic aggregates seen by *transit* ISPs will, anytime soon, include *only* QUIC packets. However, if this happens, we can also adapt Penny.

Dropping packets in the middle of a QUIC flow is not useful because packets are fully encrypted and we cannot recognize their retransmissions. We can however drop packets in the connection handshake (i.e., QUIC client hellos) as they can be distinguished from other packets [42]. We note that to open a connection, QUIC clients often simultaneously send both a 0-RTT hello packet (piggybacking on a previous connection) and a regular 1-RTT one. When this happens, both packets would need to be dropped. If we drop the first packet or packets of a QUIC flow, one of two things is likely to happen: either the client will retransmit the initial packet, or the client may switch to TCP [27]. In both cases, we can apply Penny's statistical approach. Dropping handshake packets causes higher performance degradation because retransmission timers for them tend to be set conservatively high. This means that our test might have a higher impact on QUIC applications than on TCP ones.

Preventing abuses of Penny. Since Penny is a traffic-based system, one of our design goals is robustness to worst-case traffic patterns, including those generated by current and future malicious users. Indeed, if Penny were to be employed in production networks to make traffic-related or business decisions, multiple actors may have incentives to bias, abuse or circumvent its tests. In Figure 1b, for example, AS3 may have economic incentives to prevent AS1 from detecting that the depicted traffic is closed-loop. To do so, AS3 may want to induce AS1's operators to ignore alerts from Penny.

As discussed in Section 5, what we really need to avoid is Penny misclassifying spoofed traffic as closed-loop, since this would trigger false alerts. Our design comes with statistical guarantees on its robustness against worst-case open-loop traffic, mimicking TCP flows and including a high number of unsolicited packet duplicates that can be confused with retransmissions. To our knowledge, such traffic patterns are *not* observed in the current Internet because current spoofers have *no* reason to generate them. However, providing worst-case guarantees can be instrumental both to incentivize Penny deployment and to make its design resilient against future malicious users who may target Penny.

On the other hand, we do not prevent attempts of malicious users to bypass Penny drop tests. For example, spoofers can send UDP traffic, or trains of out-of-order TCP packets: they are always ignored by Penny. This is aligned with Penny's objective, which is to spot closed-loop traffic, *not* to detect spoofing.

We can think of only one setting where spoofing can be actually detrimental to Penny: if a malicious user generates closed-loop traffic but spoofing the source address[4]. To do so, the attacker needs to have access to the machine with the spoofed source address, and to generate traffic from a different machine, in another location. For example, in Figure 1c, an attacker in AS5 can generate closed-loop spoofed traffic if they also have access to hosts in AS2 that report information about lost packets to the traffic sources in AS5.

We cannot see any motivation to generate closed-loop spoofed traffic, or any justification for its costs and complexity, other than to confuse Penny or similar systems. Assuming however that doing so might be worthwhile in the future Internet, Penny can trade test speed for further increased attack costs. For example, we can configure Penny to send alerts to the operators only when detecting closed-loop traffic from many sources with different IP addresses, or for stronger assurance, from multiple sources in different ASes. This would require the attacker to control multiple real hosts whose traffic to a specific destination enters the network running Penny at the same ingress port.

Detecting closed-loop flows as a network primitive. Penny is designed to reliably detect unexpected traffic paths such as the one in Figure 1 or in Appendix A, but we envision that its applicability can be broader in at least two ways.

On the one hand, identifying aggregates with closed-loop flows can be used to improve the accuracy and effectiveness of basically any traffic measurement and monitoring system deployed at an ISP. Penny drop tests allow measurements taken on misbehaving, likely spoofed traffic to be disregarded, as discussed in Section 7.

On the other hand, the packet drop test can be re-purposed to focus on spoofed traffic, with security applications such as de-prioritizing or discarding spoofed traffic destined to the ISP's customers. Refocusing on spoofed traffic, however, requires revisiting our design and implementation choices. For example, for this use we would need to avoid biasing the test towards finding closed-loop flows hidden among spoofed flows in an aggregate, as we do in Penny. Focusing on spoofed traffic would also come with tighter time constraints. We plan to further explore this perspective in future work.

9 Conclusions

Can an ISP know if traffic arriving at an unexpected ingress point is something to raise an alert over, or just background noise from spoofing? Before answering, consider that ISPs know very little about the traffic they provide transit for. For example, for each received TCP flow, they typically only see packets sent by one endpoint, don't know external network conditions such as the RTT, bottleneck bandwidth, or packet loss rate, and don't have any control over the end hosts.

In this paper, we present Penny, a system that identifies closed-loop, and hence not spoofed, TCP traffic. Penny has a statistical core that balances observations on retransmitted and non-retransmitted packets with unsolicited duplicates, and works under minimal assumptions about traffic senders. Its implementation includes mechanisms to deal with practical factors such as TCP quirks, lack of visibility on external network conditions, and presence of spoofers deliberately trying to bypass our system.

Our evaluation shows that Penny accurately and reliably identifies aggregates containing non-spoofed TCP flows, even when those flows are mixed with high rates of spoofed traffic from the same source IP prefix. Penny remains accurate under many different network conditions, and has a very limited impact on the performance of the tested flows.

This work does not raise any ethical issues.

10 Acknowledgements

We are grateful to the SIGCOMM anonymous reviewers and our shepherd, Venkat Arun, for their insightful comments. We also thank the members of the Network and Systems Group at UCL for their valuable feedback.

[4] as long as it is not spoofed, other closed-loop malicious traffic (e.g., involved in volumetric or reflection attacks) is not problematic for Penny, since it carries information about inter-domain traffic paths that Penny aims at exposing (see Figure 1).

References

[1] Martin Arlitt and Carey Williamson. 2005. An analysis of TCP reset behaviour on the Internet. *ACM SIGCOMM Computer Communication Review* 35, 1 (2005), 37–44.

[2] Venkat Arun and Hari Balakrishnan. 2018. Copa: Practical Delay-Based Congestion Control for the Internet. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 329–342.

[3] Fred Baker and Pekka Savola. 2004. BCP38 - Ingress Filtering for Multihomed Networks. https://tools.ietf.org/html/bcp84.

[4] Leandro M Bertholdo, Sandro LA Ferreira, João M Ceron, Lisandro Zambenedetti Granville, Ralph Holz, and Roland van Rijswijk-Deij. 2022. On the Asymmetry of Internet eXchange Points-Why Should IXPs and CDNs Care?. In *2022 18th International Conference on Network and Service Management (CNSM)*. IEEE, 73–81.

[5] Jun Bi, Jianping Wu, Guang Yao, and Fred Baker. 2015. RFC7513 - Source Address Validation Improvement (SAVI) Solution for DHCP. https://tools.ietf.org/html/rfc7513. https://tools.ietf.org/html/rfc7513

[6] Tobias Bühler, Romain Jacob, Ingmar Poese, and Laurent Vanbever. 2023. Enhancing Global Network Monitoring with Magnifier. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 1521–1539.

[7] CAIDA. 2019. Anonymized Internet Traces. https://catalog.caida.org/dataset/passive_merged_pcap.

[8] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. 2016. BBR: Congestion-Based Congestion Control. *ACM Queue* 14, September-October (2016), 20 – 53.

[9] Jeffrey D Case, Mark Fedor, Martin Lee Schoffstall, and James Davin. 1990. A Simple Network Management Protocol (SNMP). https://tools.ietf.org/html/rfc1157

[10] Balakrishnan Chandrasekaran, Georgios Smaragdakis, Arthur Berger, Matthew Luckie, and Keung-Chi Ng. 2015. A server-to-server view of the Internet. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*. 1–13.

[11] Cisco. 2023. Policy-Based Routing. https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/iproute_pi/configuration/xe-3e/iri-xe-3e-book/iri-pbr.pdf.

[12] Cisco. 2023. Route Maps. https://www.cisco.com/c/en/us/td/docs/security/asa/asa92/configuration/general/asa-general-cli/route-maps.pdf.

[13] Benoît Claise. 2004. Cisco Systems NetFlow Services Export Version 9. RFC 3954. https://tools.ietf.org/html/rfc3954

[14] HE Hurricane Electric. 2023. Hurricane Electric BGP Toolkit. https://bgp.he.net.

[15] Roni Even and Jonathan Lennox. 2021. Mapping RTP Streams to Controlling Multiple Streams for Telepresence (CLUE) Media Captures. RFC 8849. https://tools.ietf.org/html/rfc8849

[16] Paul Ferguson and Daniel Senie. 2000. BCP38 - Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing. https://tools.ietf.org/html/bcp38.

[17] Mojgan Ghasemi, Theophilus Benson, and Jennifer Rexford. 2017. Dapper: Data plane performance diagnosis of TCP. In *Proceedings of the Symposium on SDN Research*. 61–74.

[18] Petros Gigis. 2024. Bad Packets Come Back, Worse Ones Don't, Artifacts: Project Repository. https://github.com/pgigis/sigcomm2024-penny.

[19] Timothy G Griffin and Gordon Wilfong. 2002. On the correctness of IBGP configuration. *ACM SIGCOMM Computer Communication Review* 32, 4 (2002), 17–29.

[20] Sangtae Ha, Injong Rhee, and Lisong Xu. 2008. CUBIC: a new TCP-friendly high-speed TCP variant. *ACM SIGOPS operating systems review* 42, 5 (2008), 64–74.

[21] Tom Henderson and Sally Floyd. 1999. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 2582. https://tools.ietf.org/html/rfc2582

[22] Thomas Holterbach, Edgar Costa Molero, Maria Apostolaki, Alberto Dainotti, Stefano Vissicchio, and Laurent Vanbever. 2019. Blink: Fast Connectivity Recovery Entirely in the Data Plane. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 161–176.

[23] Jana Iyengar and Martin Thomson. 2021. QUIC: A UDP-Based Multiplexed and Secure Transport. RFC 9000. https://tools.ietf.org/html/rfc9000

[24] Mattijs Jonker, Alistair King, Johannes Krupp, Christian Rossow, Anna Sperotto, and Alberto Dainotti. 2017. Millions of Targets under Attack: A Macroscopic Characterization of the DoS Ecosystem. In *Proceedings of the 2017 Internet Measurement Conference* (London, United Kingdom) *(IMC '17)*. Association for Computing Machinery, New York, NY, USA, 100–113.

[25] Maciej Korczyński, Yevheniya Nosyk, Qasim Lone, Marcin Skwarek, Baptiste Jonglez, and Andrzej Duda. 2020. Don't forget to lock the front door! inferring the deployment of source address validation of inbound traffic. In *Passive and Active Measurement: 21st International Conference, PAM 2020, Eugene, Oregon, USA, March 30–31, 2020, Proceedings 21*. Springer, 107–121.

[26] Marc Kührer, Thomas Hupperich, Christian Rossow, and Thorsten Holz. 2014. Exit from Hell? Reducing the Impact of Amplification DDoS Attacks. In *23rd USENIX security symposium (USENIX security 14)*. 111–125.

[27] Mirja Kühlewind and Brian Trammell. 2022. Manageability of the QUIC Transport Protocol. RFC 9312. https://tools.ietf.org/html/rfc9312

[28] Franziska Lichtblau, Florian Streibelt, Thorben Krüger, Philipp Richter, and Anja Feldmann. 2017. Detection, classification, and analysis of inter-domain traffic with spoofed source IP addresses. In *Proceedings of the 2017 Internet Measurement Conference*. 86–99.

[29] Qasim Lone, Matthew Luckie, Maciej Korczyński, and Michel Van Eeten. 2017. Using loops observed in traceroute to infer the ability to spoof. In *Passive and Active Measurement: 18th International Conference, PAM 2017, Sydney, NSW, Australia, March 30-31, 2017, Proceedings 18*. Springer, 229–241.

[30] Michael Markovitch, Sharad Agarwal, Rodrigo Fonseca, Ryan Beckett, Chuanji Zhang, Irena Atov, and Somesh Chaturmohta. 2022. TIPSY: Predicting Where Traffic Will Ingress a WAN. In *Proceedings of the ACM SIGCOMM 2022 Conference* (Amsterdam, Netherlands) *(SIGCOMM '22)*. Association for Computing Machinery, New York, NY, USA, 233–249.

[31] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. 2014. DREAM: Dynamic Resource Allocation for Software-Defined Measurement. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (Chicago, Illinois, USA) *(SIGCOMM '14)*. Association for Computing Machinery, New York, NY, USA, 419–430.

[32] John Nagle. 1984. Congestion Control in IP/TCP Internetworks. RFC 896. https://tools.ietf.org/html/rfc896

[33] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. 2017. Language-Directed Hardware Design for Network Performance Monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (Los Angeles, CA, USA) *(SIGCOMM '17)*. Association for Computing Machinery, New York, NY, USA, 85–98.

[34] RIPE NCC. 2023. Routing Information Service (RIS). https://www.ripe.net/analyse/internet-measurements/routing-information-service-ris.

[35] University of Oregon. 2023. RouteViews Project. http://www.routeviews.org/.

[36] Jitendra Padhye, Victor Firoiu, Don Towsley, and Jim Kurose. 1998. Modeling TCP Throughput: A Simple Model and Its Empirical Validation. In *Proc. ACM SIGCOMM '98* (Vancouver, British Columbia, Canada) *(SIGCOMM '98)*. Association for Computing Machinery, New York, NY, USA, 303–314.

[37] Sonia Panchen, Neil McKee, and Peter Phaal. 2001. InMon Corporation's sFlow: A Method for Monitoring Traffic in Switched and Routed Networks. RFC 3176. https://tools.ietf.org/html/rfc3176

[38] Peter Phaal and Marc Lavine. 2004. sFlow Version 5. https://sflow.org/sflow_version_5.txt

[39] Jeff Rasley, Brent Stephens, Colin Dixon, Eric Rozner, Wes Felter, Kanak Agarwal, John Carter, and Rodrigo Fonseca. 2014. Planck: Millisecond-Scale Monitoring and Control for Commodity Networks. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (Chicago, Illinois, USA) *(SIGCOMM '14)*. Association for Computing Machinery, New York, NY, USA, 407–418.

[40] George F Riley and Thomas R Henderson. 2010. The ns-3 network simulator. In *Modeling and tools for network simulation*. Springer, 15–34.

[41] Said Jawad Saidi, Aniss Maghsoudlou, Damien Foucard, Georgios Smaragdakis, Ingmar Poese, and Anja Feldmann. 2020. Exploring Network-Wide Flow Data With Flowyager. *IEEE Trans. on Netw. and Serv. Manag.* 17, 4 (dec 2020), 1988–2006.

[42] The Chromium Projects. 2021. Parsing QUIC Client Hellos. https://www.chromium.org/quic/parse-client-hello/.

[43] Olivier Tilmans, Tobias Bühler, Ingmar Poese, Stefano Vissicchio, and Laurent Vanbever. 2018. Stroboscope: Declarative Network Monitoring on a Budget. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 467–482.

[44] Kevin Vermeulen, Ege Gurmericliler, Ítalo Cunha, David Choffnes, and Ethan Katz-Bassett. 2022. Internet scale reverse traceroute. In *Proceedings of the 22nd ACM Internet Measurement Conference*. 694–715.

[45] Kok-Kiong Yap, Murtaza Motiwala, Jeremy Rahe, Steve Padgett, Matthew Holliman, Gary Baldus, Marcus Hines, Taeeun Kim, Ashok Narayanan, Ankur Jain, et al. 2017. Taking the edge off with espresso: Scale, reliability and programmability for global internet peering. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. 432–445.

[46] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y. Zhao, and Haitao Zheng. 2015. Packet-Level Telemetry in Large Datacenter Networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (London, United Kingdom) *(SIGCOMM '15)*. Association for Computing Machinery, New York, NY, USA, 479–491.

APPENDIX
Appendices are supporting material that has not been peer-reviewed.

A Additional Use Cases

We now describe use cases for Penny additional to the one shown in Figure 1. We always take the perspective of AS1.

A.1 Unprofitable paths: another example

In Figure 10, an AS1 router located in the United States (US) receives traffic from source IPs geo-located in Europe (EU), within AS3, and destination IP owned by European-only ISP, AS2. The traffic is forwarded to AS1 by its provider AS5. The received traffic has a significant cost for AS1, since it has to be carried from US to EU within AS1 own network, over internal links (including expensive transatlantic ones).

Such a cost can be avoided if the traffic source is actually located in EU, as shown in Figure 10b. Indeed, AS5 could forward the traffic over the EU link instead carrying it internally from EU to US (which has a cost to AS5) and then forwarding it to AS1 over the US link. Note that sending traffic directly over the EU link would be economically attractive for AS5 as well; in fact, the current traffic path may be just the consequence of an AS5 router misconfiguration. In this case, AS1 may want to contact AS5 operators, and ask them information about the traffic displayed in the figure. Obviously, there is no need to further investigate the received traffic if it is spoofed, as shown Figure 10c: doing so would even be detrimental if network operators of multiple ASes get involved in the unnecessary troubleshooting.

The above discussion highlights a clear need to identify unexpected ingress traffic which is not source-spoofed. Penny can do so accurately, robustly and with minimal impact on the tested traffic.

A.2 Security: BGP hijack detection

In Figure 11, AS1 knows that its peer AS3 has economic incentives to send traffic for IPs owned by AS1, over the direct link AS3-AS1. Indeed, the BGP route announced by AS1 over such direct link is the shortest peer route towards AS1 IP prefixes. This route is also the best route that AS3 can learn, given that AS3 will only receive peer and provider routes for these prefixes – assuming that AS3 customers are correctly configured and do not provide transit to it.

Receiving non-spoofed traffic sourced at AS3 from its provider AS5 may be due to a BGP hijack such as the one shown in Figure 11b. Note that the displayed attack is invisible in the control plane if AS1 has no access to looking glasses or monitors in AS3, as also shown in the figure. Yet, alerting on the unexpected traffic may lead to detect and mitigate the attack if network operators of AS1 and AS3 talk to each other. Obviously, there is no need to further investigate the received traffic if it is spoofed, as shown Figure 11c: doing so would even be detrimental if network operators of multiple ASes get involved in the unnecessary troubleshooting.

The above discussion highlights a clear need to identify unexpected ingress traffic which is not source-spoofed. Penny can do so accurately, robustly and with minimal impact on the tested traffic.

A.3 New services: route-leak alert

Figure 12 shows how Penny enables ISPs to offer new services. In this example, AS1 offers a (paid) service alerting AS2 for routes that AS2 unwillingly leaks.

AS2 may in principle monitor traffic sourced at AS3 and destined for AS5. However, each AS typically has about one million destinations in its routing tables and hundreds of BGP peers, so monitoring each combination of destination IP, ingress point, and egress point is not cheap and not trivial. In fact, BGP misconfigurations and route leaks do continue to happen in the Internet.

Suppose now AS1 already deploys a monitoring infrastructure able to quickly detect unexpected inter-domain traffic paths. AS2 may be willing to give AS1 a list of networks that AS2 does not want to provide transit for (e.g., its providers). AS1 can then quickly flag cases where it receives from AS2 traffic sourced at any of the networks in the list (e.g., AS3 in the figure). This would also enable AS2 to simplify its monitoring system, provide defence in depth against possible misconfigurations, and potentially fix them faster. Obviously, raising alerts for spoofed traffic would only make the service less effective and much less valuable for AS2.

The above discussion highlights a clear need to identify unexpected ingress traffic which is not source-spoofed. Penny can do so accurately, robustly and with minimal impact on the tested traffic.

B Additional Experimental Results

We now report results of the sensitivity analyses we performed in addition to the ones discussed in Section 6.

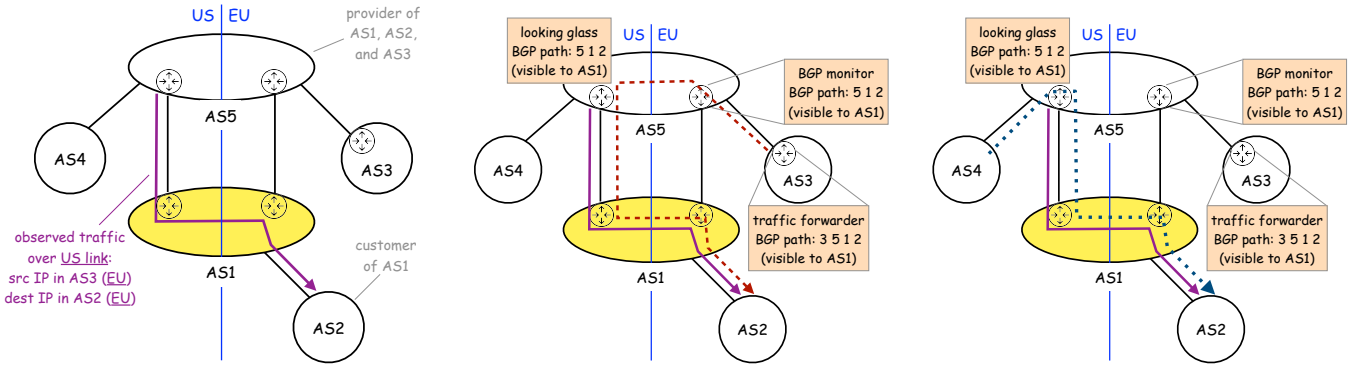
B.1 Dealing with worst-case traffic patterns

Figure 13 shows the worst-case probability for Penny to misclassify spoofed traffic for p_{drop} values different from the default 5%, which is displayed in Figure 7. The figures report the worst-case probabilities for any number of duplicates less or equal to 15% of the maximum x value in the plots, as more duplicates would exceed Penny duplicate threshold f_{dup}.

Note that the relationship between the expected number of dropped packets and Penny worst-case error probability is basically preserved across different values of f_{dup}. Notably, such an error probability is around 10^{-4} for 12 packet drops, consistently with Figure 7 and our choice of the default value for $n_{minDROP}$. Indeed, 12 is the expected number of packet drops for 400 droppable packets with p_{drop} = 3%, and for 1200 droppable packets with p_{drop} = 1%.

B.2 Penny's accuracy for mixed traffic aggregates

Figure 14 illustrates the results of Penny for traffic aggregates with a proportion of closed-loop flows different from Figure 8. As expected, higher rate of spoofed traffic increase the likelihood that Penny classify the aggregate as spoofed. When this happens, Penny switches to testing individual flows. In our experiments, Penny always finds the closed-loop flows in the aggregates even when they account for only 20% or 10% of the packets in the aggregate.

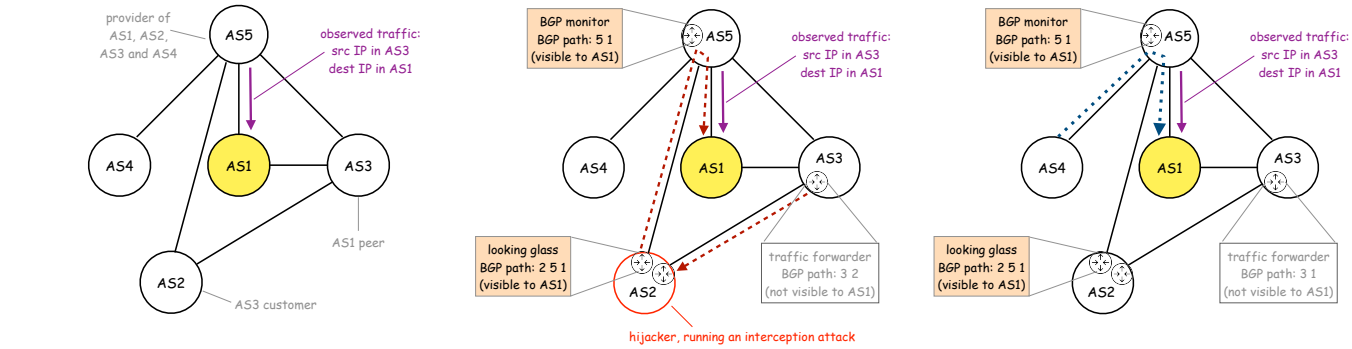


(a) AS1 receives EU-to-EU traffic at router in US, which has a cost.

(b) Option 1: actual unprofitable path, not detectable in the control plane.

(c) Option 2: spoofed traffic, not worth to raise an alert on.

Figure 10: By identifying closed-loop, non-spoofed ingress traffic, Penny enables ISPs (e.g., AS1) to detect unprofitable paths, such as the long, expensive internal path that AS5 forces AS1 to forward the traffic on in this figure.

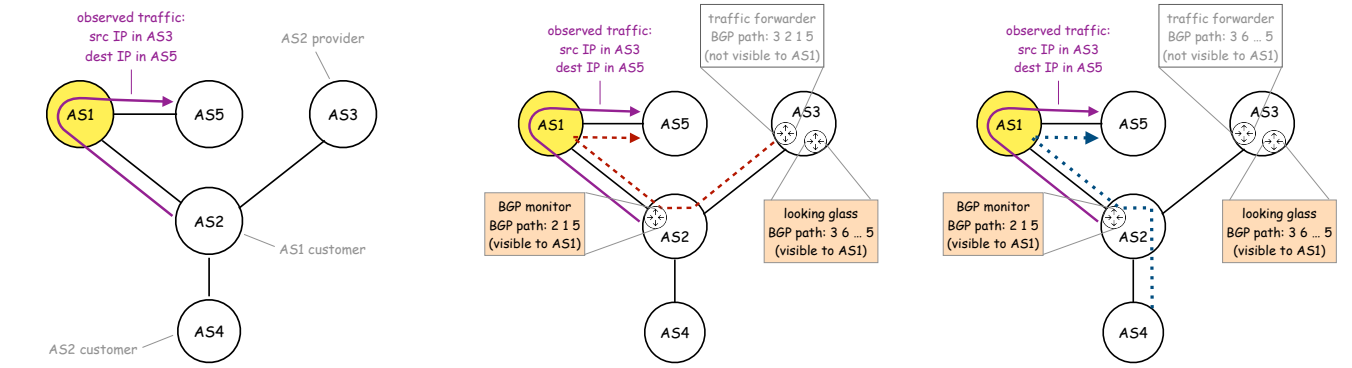


(a) AS1 receives from a provider (AS5) traffic sourced at a peer (AS3).

(b) Option 1: AS2 hijack an AS1's prefix, not detectable in the control plane.

(c) Option 2: spoofed traffic, not worth to raise an alert on.

Figure 11: By identifying closed-loop, non-spoofed ingress traffic, Penny enables ISPs (e.g., AS1) to detect hijacks not visible in the control plane.

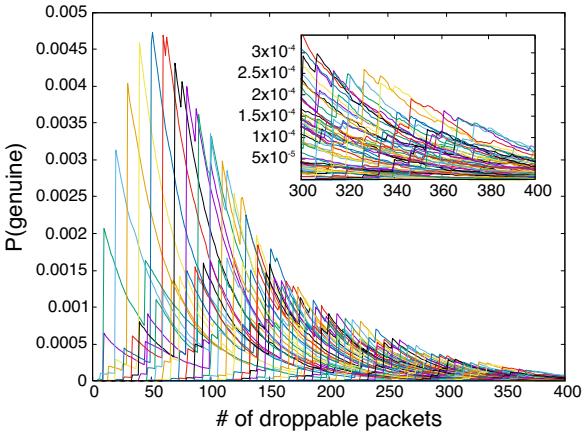


(a) AS1's customer, AS2, seems to provide transit to one of its providers (AS3).

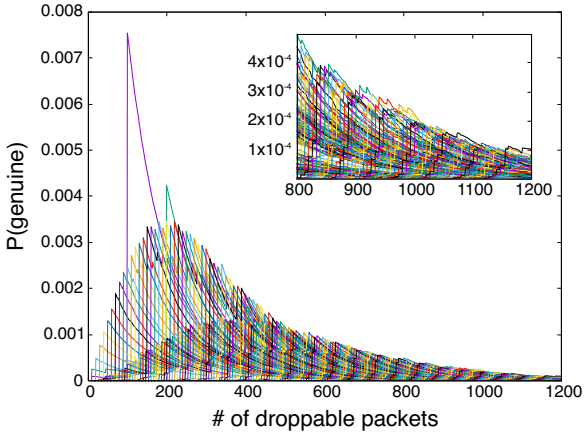
(b) Option 1: actual route leak, not detectable in the control plane.

(c) Option 2: spoofed traffic, not worth to raise an alert on.

Figure 12: By identifying closed-loop, non-spoofed ingress traffic, Penny enables ISPs (e.g., AS1) to offer additional services to their customers. In this example, AS1 can alert AS2 about route leaks occurring at AS2.



(a) Worst-case probability to misclassify spoofed traffic as closed-loop when p_{drop} is 3%. Each curve corresponds to a fixed number of duplicates between 2 and 60.



(b) Worst-case probability to misclassify spoofed traffic as closed-loop when p_{drop} is 1%. Each curve corresponds to a fixed number of duplicates between 2 and 180.

Figure 13: Worst-case probability for Penny to misclassify spoofed traffic as closed-loop with p_{drop} values different from the default 5%.

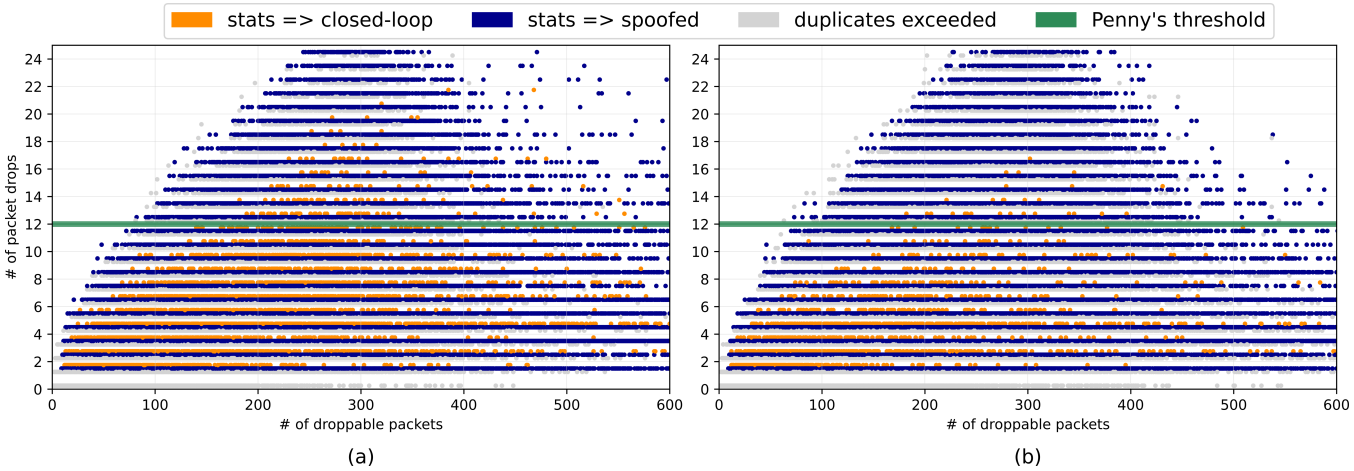


Figure 14: Accuracy of Penny's statistical model when (a) 20% of the traffic originates from closed-loop flows and (b) when 10% of the traffic comes from closed-loop flows.