
# Suppressing BGP Zombies with Route Status Transparency

Yosef Edery Anahory, *The Hebrew University of Jerusalem;* Jie Kong, Nicholas Scaglione, and Justin Furuness, *University of Connecticut;* Hemi Leibowitz, *The College of Management Academic Studies;* Amir Herzberg and Bing Wang, *University of Connecticut;* Yossi Gilad, *The Hebrew University of Jerusalem*

https://www.usenix.org/conference/nsdi25/presentation/anahory

This paper is included in the Proceedings of the 22nd USENIX Symposium on Networked Systems Design and Implementation.

April 28–30, 2025 • Philadelphia, PA, USA

978-1-939133-46-5

Open access to the Proceedings of the 22nd USENIX Symposium on Networked Systems Design and Implementation is sponsored by

# Suppressing BGP Zombies with Route Status Transparency

Yosef Edery Anahory[1], Jie Kong[2], Nicholas Scaglione[2], Justin Furuness[2], Hemi Leibowitz[3], Amir Herzberg[2], Bing Wang[2], Yossi Gilad[1]

[1]*School of Computer Science and Engineering, The Hebrew University of Jerusalem, Jerusalem, Israel*
[2]*School of Computing, University of Connecticut, Storrs, CT*
[3]*Faculty of Computer Science, The College of Management Academic Studies, Rishon LeZion, Israel*

## Abstract

Withdrawal suppression has been a known weakness of BGP for over a decade. It has a significant detrimental impact on both the reliability and security of inter-domain routing on the Internet. This paper presents *Route Status Transparency* (RoST), the first design that efficiently and securely thwarts withdrawal suppression misconfigurations and attacks. RoST allows ASes to efficiently verify whether a route has been withdrawn; it is compatible with BGP as well as with BGP security enhancements. We use simulations on the Internet's AS-level topology to evaluate the benefits from adopting RoST. We use an extensive real-world BGP announcements dataset to show that it is efficient in terms of storage, bandwidth, and computational requirements.

## 1 Introduction

The Border Gateway Protocol (BGP) determines how packets route between Autonomous Systems (ASes) on the Internet. Despite its crucial role in facilitating Internet communication, BGP is fragile, and when failures occur, they can cause significant disruptions. Some failures are due to benign errors and misconfigurations, while others are attributed to malicious actions [29, 41]. Many of these issues are addressed by the increasing adoption of the Resource Public Key Infrastructure (RPKI) [11, 25, 33], which allows an IP prefix owner to associate it with the ASes allowed to announce the prefix through BGP. Furthermore, the networking community has also devoted tremendous efforts to design and standardize protocols for route authentication (i.e., beyond the origin), most notably, BGPsec [6]. With route validation in place, announcing routes with invalid sources or bogus links becomes less of a problem for global Internet routing.

However, the authentication mechanisms do not solve the problem of *freshness*. That is, how can an AS know whether a BGP announcement has been withdrawn or not since it was announced? This key problem is the root cause of stuck routes, also known as "zombie" routes, where ASes choose to route traffic through withdrawn routes because these ASes are unaware that these routes have been withdrawn [15, 35]. This happens when some AS along the route withdraws an announcement, but a downstream AS does not propagate the withdrawal (intentionally or due to an error). The once valid route keeps appearing "alive" to the following ASes, who may opt to use it and export it further, accidentally misleading other ASes. Since the zombie route *was* legitimate, it continues to pass the route authentication checks even with RPKI and full-fledged BGPsec. Zombie routes can cause disconnections when an AS mistakenly sends traffic to a neighbor who no longer has a valid route to the destination IP address (since that route was withdrawn). Moreover, zombie routes can lead to inefficient routing where an AS would think it uses a better (e.g., cheaper or shorter) route to reach some destination, but a downstream AS will route the traffic differently than expected. In some instances, this misrouting has even led to the creation of routing loops [30]. In many cases, withdrawal suppression is attributed to benign errors rather than deliberate attacks [15], illustrating the fragility of Internet routing.

Currently, the only proposed approach to solve this problem is *key rollover* [39, 42], which requires route validation (i.e., BGPsec) to already be deployed. In key rollover, keys are rotated periodically; as a result, signed routes have a limited validity period and cannot be reused 'forever'. Thus, after a key rollover, the routes signed by the old key cannot be authenticated by the new key. This means that a withdrawal suppression can only be effective until the current keys are rotated. However, key rollover requires a hefty operational effort, since the AS's BGPsec key pair needs to be updated, a new certificate must be issued, and the old one revoked (if the rollover occurs when the previous key is still valid).

In this work, we present *Route Status Transparency (RoST)*, a novel mechanism for mitigating withdrawal suppression. RoST-adopting ASes publish cryptographically-signed information about the routes they export to protect their announcements against withdrawal suppression. Other-RoST adopting ASes download and use this information to verify whether any of the routes they receive were actually

withdrawn. Specifically, RoST uses the transitive attribute capability to identify and monitor routes. Namely, every AS in the path will generate a sequence of route IDs as the announcement is distributed along the path. By combining the transparent route status information with these route IDs, each AS that performs validation based on RoST can check if the announcements match the transparent information, and if not, identify withdrawn routes.

RoST detects and limits the impact of BGP withdrawal suppression to minutes, whether due to intentional attacks or misconfigurations, and adds only a negligible overhead to the BGP protocol. ASes that adopt RoST deploy a local machine (an "agent") that synchronizes with the route status transparency reports and identifies withdrawn routes. The agent machine performs the heavy lifting of validating these reports and then configures the local BGP routers to withdraw invalid routes. In contrast to key rollover, RoST can be adapted with vanilla BGP to mitigate the zombie routes phenomenon, thus circumventing the challenges involved in the deployment of BGPsec and the costs of key rollover. We evaluate RoST in terms of overhead using routing data from multiple vantage points on the Internet and compare it with key rollover.

In summary, this work addresses withdrawal suppression, a fundamental problem in today's Internet routing that will persist even when route authentication mechanisms are fully deployed. We design and evaluate the RoST protocol, the first protocol that efficiently detects and mitigates withdrawal suppression. RoST is compatible with the currently deployed BGP, and we show how it can be deployed without requiring changes to today's routers.

## 2 Background

**Route withdrawals and announcement suppression.** A BGP router can withdraw a route either explicitly by sending a withdrawal announcement or implicitly by sending an updated announcement. In Figure 1, we illustrate both explicit and implicit withdrawals. The figure illustrates a scenario in which AS 100 initially chooses AS 666 as its provider, and thus AS 100 sends an announcement with prefix 1.2/16 and AS-path 100 to its provider, AS 666. Then, AS 100 decides to switch its provider to AS 300, so it sends an *explicit* withdrawal announcement to AS 666. In an *implicit* withdrawal, AS 100 sends a new announcement to AS 666 with an artificially inflated AS-path, 100-100-100, known as 'path prepending'. This makes the path from AS 100 to AS 666 three-hop long, making the path from AS 100 to its current provider, AS 300, more favorable (e.g., AS 200 will route to prefix 1.2/16 through AS 300, instead of AS 666). A withdrawal suppression happens when some AS on the route fails to propagate a withdrawal message or an alternative route (explicit/implicit withdrawal, respectively). This can be unintentional due to misconfigurations or software bugs in

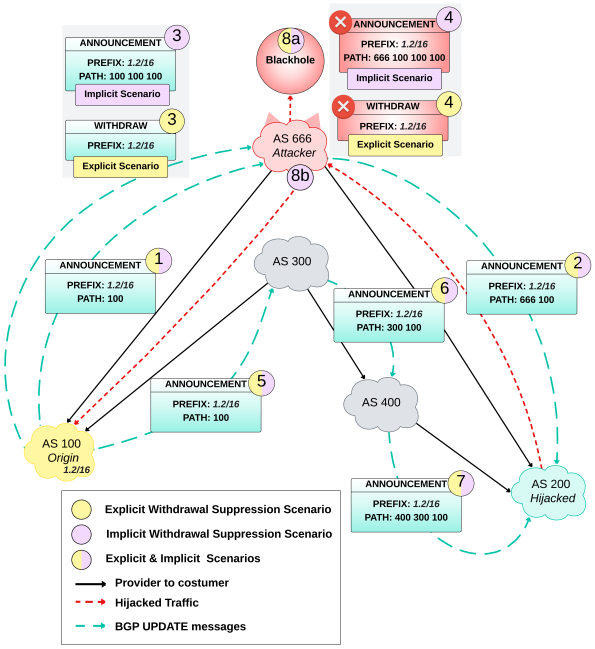



**Figure 1:** Illustration of explicit and implicit withdrawal suppression attacks. The diagram highlights the sequence of steps involved in each attack and the actions taken by AS 666. Steps unique to explicit withdrawal suppression are marked in yellow with a "b" label, and steps unique to implicit withdrawal suppression are marked in purple with an "a" label. Steps that are common to both scenarios are shown with a combination of yellow and purple.

BGP routers, or intentional, for malicious purposes such as to keep traffic flowing through a route with the attacker's AS (AS 666 in Figure 1).

**Stuck or zombie routes.** Measurement studies [15, 35] using route beacons confirm that zombie outbreak happens daily, which can cause traffic to follow undesirable and unavailable paths, leading to network congestion, degraded service quality, or even complete loss of connectivity. Specifically, the analysis in [15] reveals that the impact of zombie routes is influenced by the tier of the network where the zombie route was detected, which has significant implications. The authors demonstrate that zombie routes detected within a Tier-1 AS can affect the majority of ASes. A real-world example of this occurred on August 30, 2020, when CenturyLink, which had acquired Level 3 Communications (a Tier-1 AS), experienced a significant incident. According to a report by Cloudflare [36], CenturyLink inadvertently applied a firewall rule that blocked all internal BGP traffic, leading to extensive outages across their network. What made this incident particularly severe was the failure of affected routers to issue BGP withdrawal messages. Without these messages, external networks (including Cloudflare) continued to route

traffic through CenturyLink, unaware of the internal issues. The outage lasted several hours, and many networks were forced to manually reconfigure their systems to reroute traffic away from CenturyLink/Level 3.

## 3 RoST Design

In this section, we detail the design of RoST. We start with its goals (§3.1) and then present a high-level system overview (§3.2). We then detail how RoST agents maintain routes' status (§3.3), how the routes' status is made transparent (§3.4), how agents track the status of these routes (§3.5) and how routes advertisements are validated (§3.6). A detailed pseudocode for the main RoST operations is presented in Appendix A.

For simplicity, in this section we assume a single trusted repository and refer to it in a singular form ("the repository"). Later, in §7.2, we discuss how to extend it to support multiple repositories and to overcome repository faults. Finally, when we refer to AS-Paths, we write the path from left to right, where the rightmost AS is the origin of the IP prefix.

### 3.1 Design Goals

RoST aims to detect and mitigate withdrawal suppression in inter-domain routing on the Internet. It does not deal with path manipulation attackers that may alter or spoof route advertisements. Such attackers should be handled by path authentication mechanisms [6, 32]. We design it to achieve the following three desiderata:

- Efficiency, i.e., low computation, communication, and storage overhead.
- Compatibility with BGP and future security enhancement of BGP (e.g., BGPsec [6], BGP-iSec [32]). Namely, it should be possible to deploy RoST without extending or standardizing new routing protocols.
- Compatibility with BGP routers. That is, it should be possible to mitigate withdrawal suppression by having local RoST agent automatically issue commands to the router through existing command line or application interfaces rather than requiring specialized router software or hardware support.

### 3.2 High-Level Operation

RoST detects withdrawal suppression by providing route status transparency. We outline its operation following Algorithm 1. Each adopting AS deploys a RoST *agent*, which performs three main functions: (1) report the current routes of the AS to a public repository (Algorithm 1, lines 6-7), (2) retrieve the current routes of other ASes from the repository (Algorithm 1, lines 8–13), and (3) validate advertised BGP routes without relying on propagation of withdrawal announcements (Algorithm 1, lines 14–19).

**Algorithm 1** High-Level Operations of RoST Agent

```
1:  Input:
2:      AgentAS: AS number for which the agent is responsible
3:      BatchInterval: Time interval for periodic reporting
4:      R: Trusted repository
5:  procedure RoSTAgent(AgentAS, BatchInterval, R)
6:      // 1. Initialize asynchronous tasks
7:      Event 1 - Reports local routes to R every BatchInterval
8:      Event 2 - Subscribe to route updates from R
9:      Event 3 - Listen for BGP update messages from the router
10:     // 2. Handle incoming events
11:     while true do
12:         event ← Wait for the next available event
13:         if event is "Periodic Report Trigger" (Event 1) then
14:             Report local routes to R
15:         else if event is "Route Update from R" (Event 2) then
16:             Validate impacted routes
17:             if a route is invalid then
18:                 Propagate withdraw or alternative route
19:         else if event is "BGP Update Message" (Event 3) then
20:             Validate the new route using repository data
21:             if the route is invalid then
22:                 Propagate withdraw or alternative route
23:             Update route status before forwarding BGP message
```

Using a standalone agent that is separate from BGP routers promotes simpler deployment because it does not require replacing or updating existing BGP routers. Specifically, when the agent detects a route that was withdrawn but the withdrawal was suppressed, the agent instructs the BGP routers of the AS to remove the withdrawn route using the routers' existing API.

To illustrate how RoST detects withdrawal suppression, consider the scenario depicted in Figure 2. AS 4 has two available routes to a prefix that originates at AS 1, either via path 5-3-2-1 or 6-2-1. Assume that AS 4 chose to route through AS 6 as that AS path is shorter. Suppose that at some point, AS 2 decides to stop using AS 6 as a provider, and as a result, AS 2 sends a withdrawal update message to AS 6; however, AS 6 suppresses this withdrawal. This means that AS 4 would continue to route traffic through AS 6 since it is unaware of the withdrawal.

Using RoST, AS 4 *can* detect and mitigate this suppression. That is, once AS 2 issues the withdrawal, its agent updates the repository about the withdrawal. This allows AS 4's agent to learn about the withdrawal from the repository despite AS 6 suppressing it. To authenticate the information recorded at the repository, RoST assumes a standard RPKI deployment. This allows agents to sign the routes' status information they make transparent, and other agents to verify its authenticity. Given the increasing adoption of RPKI, with most unique prefix-

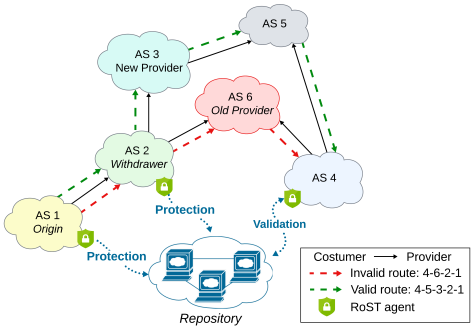



**Figure 2:** High-level overview of RoST operations. AS1, the origin AS, and AS2, the withdrawer, both perform the protection role by ensuring that their routing information is up-to-date and uploaded to the trusted repository when changes occur. When AS2 switches providers from AS6 to AS3, AS6, the old provider, continues to attract traffic using an outdated route. AS4, a customer of AS6, takes on the validation role by retrieving the updated route status from the trusted repository. This allows AS4 to detect any withdrawal suppression by AS6.

origin pairs already having valid RPKI records (54% [2]), we believe that it will not hinder RoST's adoption.

## 3.3 Tracking Route Status

An AS's agent maintains the status of the AS's routes using vectors called Route Status Vectors (RSVs). Each RSV represents the route updates to a specific IP prefix and whether or not this route is active. That is, an RSV is in the form of: $(Prefix, RouteID, Status)$.

The agent updates the repository with batches of route updates. These updates are identified using a *RouteID* that consists of two counters, $RouteID = (BatchID, PathID)$. The BatchID counter identifies the batch number where that update occurred, and the PathID counter indexes the specific path update for the RSV's prefix within that batch. This design decision is based on our analysis of real-world route announcements collected by public route collectors (see §4). Our analysis shows that while the average number of path changes per prefix remains under 20 per day, some prefixes can experience more than 1 million route changes per day (see Figure 5a). This can be caused by multiple reasons, such as path prepending, provider change, or path finding [15]. By periodically sending batched updates, RoST minimizes its space requirements: a burst of path changes would be 'absorbed' into one BatchID; PathID would reset (to 0) in the next batch, which avoids the rapid counter increments if RoST were to use a single counter. The PathID would increment within a batch only if the route has changed and still active. If the route is withdrawn, only the status field changes to invalid.

Every RoST agent maintains a set of RSVs per neighboring AS, reflecting the routes that were announced to the specific neighbor. We refer to this set as RSV-Out, and denote the set of RSVs that the agent of AS $x$ maintains for neighboring AS $y$ as RSV-Out$_{(x,y)}$. We refer to the pair $(x,y)$ as the *interface* between AS $x$ and AS $y$. The RSV-Out$_{(x,y)}$ structure also contains a *BatchCounter*, which is incremented periodically at each batch interval. To illustrate, refer to step ① in Figure 3, where we present how the entries of RSV-Out$_{(100,200)}$ are maintained at the end of batch 54 (i.e., BatchCounter is 54) in five different scenarios:

1. *Prefix 1.1/16, no changes during the current batch*: This prefix has an older BatchID (48) and PathID (1), meaning that the route to this prefix has not changed since batch 48. The status of this route remains active (T).
2. *Prefix 1.2/16, one route change*: The BatchID has been updated from 51 to 54, initialized with a new PathID (1). This reflects that the route to this prefix has been updated once in the current batch and remains active.
3. *Prefix 1.3/16, withdrawn route*: This prefix was previously active (T), but its status has been changed to withdrawn (F), i.e., the route has been withdrawn in the current batch (BatchID 54 and PathID 0).
4. *Prefix 1.4/16, multiple route changes*: Similarly to the case of prefix 1.2/16, but in this instance, there were three different route change announcements within the current batch, as indicated by the PathID of 3.
5. *Prefix 1.5/16, withdrawn route after multiple changes*: This prefix resembles the scenario with prefix 1.4/16, but the route has been withdrawn after two active route announcements (PathID 2) within the current batch, changing its status to withdrawn (F).

## 3.4 Making Routes' Status Transparent

At the end of each batch interval, the agent updates the repository about the routes that changed during that interval. It generates for every RSV-Out a corresponding delta, ΔRSV-Out, and reports it to the repository. To illustrate, refer to step ② in Figure 3, where ΔRSV-Out$_{(100,200)}$ contains all route status updates that were made to RSV-Out$_{(100,200)}$ during batch 54. Specifically, ΔRSV-Out$_{(100,200)}$ contains all the prefixes in RSV-Out$_{(100,200)}$ except prefix 1.1/16, since it is the only prefix that has not changed during batch 54.

In addition, each ΔRSV-Out contains three other values: (1) the batch counter, (2) a Merkle tree root hash committing to all route updates (the leaves are the entries of the (updated) RSV-Out), and (3) a signature over the batch counter, the interface pair, and the Merkle root, signed by the AS's private key (based on the RPKI deployment assumption). The Merkle tree commitment enables agents from other ASes to fetch an authenticated *subset* of the ΔRSV-Out that only contains updates relevant to the fetching agent, as we explain in §3.5.

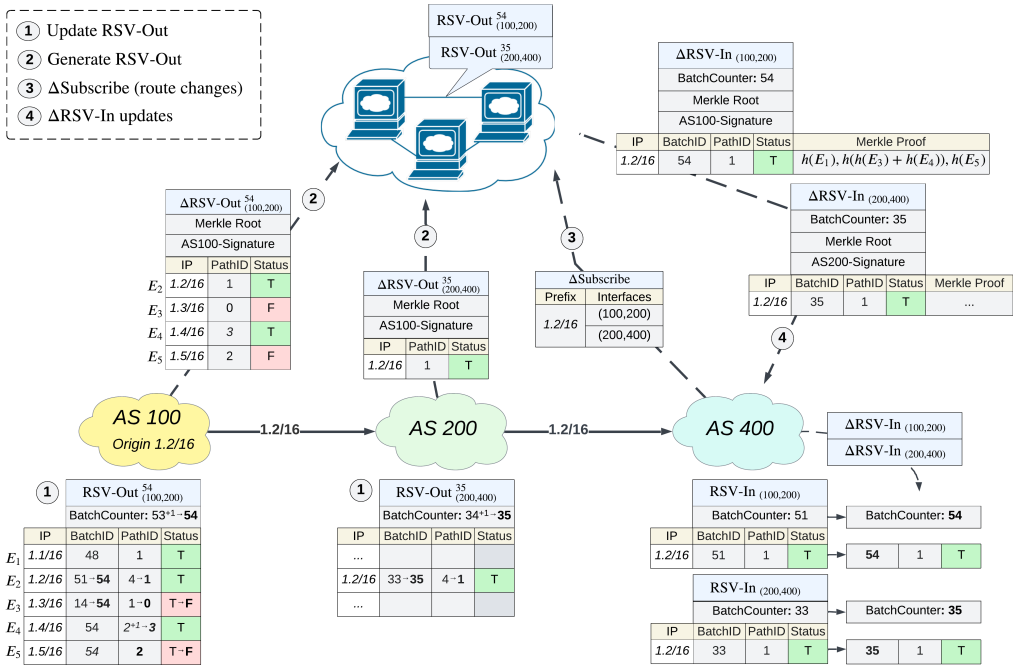



**Figure 3:** The diagram illustrates the steps involved in RSV processing and distribution between ASes and the trusted repository. AS 100 and AS 200 publish a ΔRSV-Out to the trusted repository, which contains updates for several prefixes. AS 400, which monitors the interfaces (100, 200) and (200, 400), subscribes to 1.2/16 for those interfaces to receive updates. The trusted repository sends ΔRSV-In when there are new updates, which AS 400 then merges with its existing RSV-In data. This ensures that AS 400 has the most current and validated route statuses, protecting the network against the use of outdated or invalid routes.

## 3.5 Retrieving Route Transparency Data

To learn about route changes, agents receive updates about routes from the repository. However, each agent only cares about some of the RSVs, pertaining to the routes used by their AS's BGP routers. Thus, when a router chooses to use a route to a specific prefix, the agent subscribes to updates about the current RSVs for that prefix from all the ASes on the route's AS-Path. For example, if AS $w$ uses a route with AS-Path $x-y-z$, then the agent of AS $w$ subscribes to RSVs of the routes that $z$ publishes to $y$, $y$ publishes to $x$, and $x$ publishes to $w$, because suppression can occur in each of these hops.

To track these RSVs, the agents store them per interface; we refer to them as RSV-Ins. To illustrate, observe AS 400 in Figure 3, which routes traffic to prefix $1.2/16$ using the AS-Path 400-200-100. As a result, AS 400 tracks the RSVs to this prefix from AS 100 to AS 200 (in RSV-In$_{(100,200)}$) and from AS 200 to AS 400 (in RSV-In$_{(200,400)}$).

To learn about RSVs updates, the agent sends a message called Δ*Subscribe* to the repository, indicating for each prefix which RSVs it is interested in. In response, the repository provides the agent with the latest RSV-Ins according to the subscription; it keeps the agent current using ΔRSV-In messages whenever the relevant RSVs update. For example, in Figure 3, AS 400 notifies the repository that it is interested in the RSV of prefix $1.2/16$ that AS 100 maintains for AS 200 and the RSV of prefix $1.2/16$ that AS 200 maintains for AS 400 (shown in step ③). Before batch number 54, the BatchID of the prefix $1.2/16$ in RSV-In$_{(100,200)}$ at AS 400 was 51. AS 100 and AS 200 changed the route to $1.2/16$ during batch 54 and 35 (respectively), and both ASes sent a ΔRSV-Out reflecting this change (shown as part of step ②). Thus, the repository sends two updates to the agent of AS 400 (based on its subscription): ΔRSV-In$_{(100,200)}$ and ΔRSV-In$_{(200,400)}$, with the updated RSVs (shown as part of step ④).

Using the Δ*Subscribe* request over querying for updates each time allows agents to receive updates sooner without repeatedly polling the repository. It is more efficient since the agent sends Δ*Subscribe* only once after the BGP router selects a new route.

Importantly, the repository attaches proofs to its updates using the information recorded in the latest ΔRSV-Out. The repository attaches the latest batch counter, the Merkle tree root hash and signature from the latest ΔRSV-Out and an inclusion proof for every RSV in the ΔRSV-In (i.e., the neighboring nodes on the tree in the path from the relevant leaf update to the root hash). The repository can compute such proofs since it receives the entire ΔRSV-Out and thus can recreate the entire Merkle tree. The agent, on the other hand, only retrieves a subset of the ΔRSV-Out; it checks the signature to verify the Merkle root and uses these inclusion proofs to verify that the RSVs were indeed issued

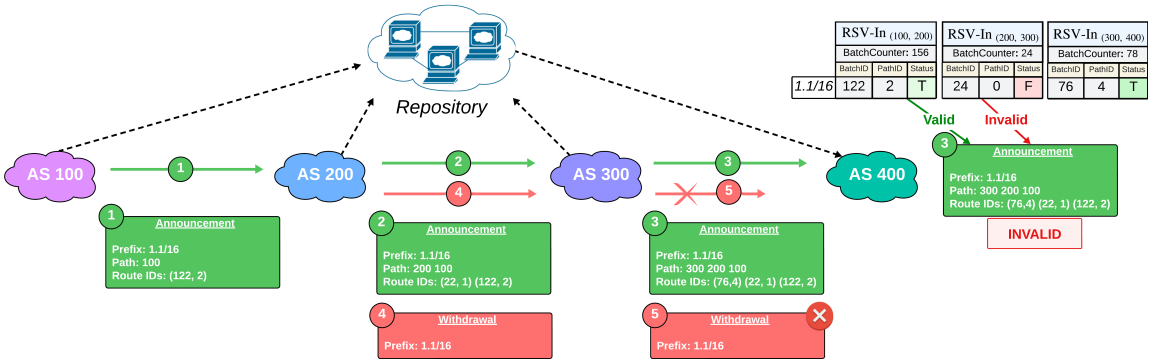



**Figure 4:** This diagram illustrates Route Status Validation for prefix 1.1/16 across different ASes. AS 100 originates the announcement and propagates it to AS 200, which in turn forwards it to ASes 300, 400. AS 200 later withdraws this prefix, but AS 300 suppresses this withdrawal. The corresponding RSV-In entries for each AS show the status and identifiers for the prefix. AS 400 detects the discrepancy between the announcement and the RSV-In received from AS 200 for the interface with AS 300. It marks the announcement as invalid.

by the claimed AS. After verification, the agent updates the corresponding RSV-In with the updated RSVs. Furthermore, the agent must check if actions need to be taken to withdraw a route that was suppressed; we discuss how an agent can do it by interacting with a standard router interface in §6.

## 3.6 Route Status Validation

The agent validates routes it receives through BGP against the information recorded in the repository. There are two triggers for running the validation procedure: either when an agent receives a route announced to its AS through BGP or when it receives an update from the repository.

To validate a route for prefix $p$, the agent must check the route against the relevant RSV-Ins records. To that end, RoST adds a new transitive extended community attribute [5, 26, 40] to BGP announcements. This new attribute contains a sequence of *RouteIDs*, where the BGP routers prepend their (current) *RouteID* to this sequence of *RouteIDs*, before forwarding an announcement. By doing so, agents that receive an announcement can validate whether the announced route is fresh, by comparing the *RouteIDs* in the transitive attribute against the corresponding *RouteIDs* that were retrieved from the repository in the relevant RSV-Ins.

For example, consider an origin AS $w$ that announces to neighboring AS $x$ the AS-Path for prefix $p$ for the first time during batch number $n$. AS $w$ would add to the announcement the RoST's transitive attribute containing a single *RouteID*, $(n,1)$. Assume that AS $x$ decides to announce this route to its neighbor, AS $y$, during batch number $n'$, and that AS $x$ has made two other changes to the route to prefix $p$ during batch $n'$. Thus, AS $x$ would prepend $RouteID = (n',3)$ to the proposed transitive attribute, i.e., the transitive attribute on the announcement sent from AS $x$ to AS $y$ would be: $(n',3),(n,1)$.

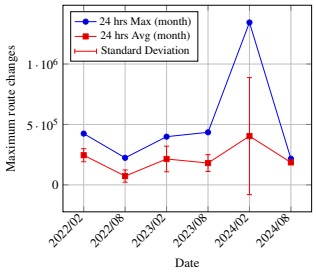
To validate the route for the prefix $p$ against its RSV, the agent checks the repository's data about the status of the *RouteIDs* in $p$'s BGP announcement's attributes. If at least one of the interfaces along the AS-Path of the route is not being tracked, the agent subscribes to updates from the repository to make sure it receives the records for all *RouteIDs*. If any of those routes' status is withdrawn, or if there was a later *RouteID* recorded for the same prefix (i.e., with a higher BatchID or the same BatchID but a higher PathID), then it considers that route withdrawn.

In Figure 4, we present route status validation for one route of AS 400, where the prefix is 1.1/16 and the AS-Path is 300-200-100. In the figure, the route withdrawal issued by AS 200 in batch 24 was dropped by AS 300 due to misconfiguration. AS 400 will be able to learn that AS 200 has withdrawn the routes since the status attribute in the RSV entry in RSV-In$_{(200,300)}$ has been set to false.

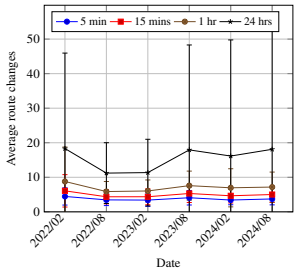
# 4 Evaluation Overhead

In this section, we evaluate RoST's overhead in terms of storage, communication, and computation. We use the Internet topology and traces from BGP routers to show that it is feasible to deploy RoST in today's Internet, considering the number of ASes, their announcements, and route update rate.

**Methodology.** To ensure a comprehensive and realistic evaluation of RoST, we use real-world BGP routing data collected across six distinct months: February and August of 2022, 2023, and 2024. We analyze data from Routing Information Base (RIB) snapshots and BGP update messages provided by RIPE NCC's Routing Information Service (RIPE RIS) [34]. RIPE RIS has multiple vantage points on the Internet that have peering links to multiple ASes and provide the full feed of their BGP traces and RIBs. We utilize RIB snapshots (generated every 8 hours) and traces of BGP




**(a)** Prefixes with the most daily route changes.




**(b)** Average over different time intervals.

**Figure 5:** Comparison of max and average route changes aggregated across the RIPE-RIS vantage points.

| | Vantage Point | #Prefixes | RSV-In #Entries |
|---|---|---|---|
| **IPv4** | AS 51185 | 991,499 | 3,578,688 |
| **IPv6** | AS 44085 | 220,074 | 1,078,736 |

**Table 1:** AS 51185 and AS 44085 are the RIPE VPs with the maximum number of prefixes connectivity for IPV4 and IPv6 respectively. We show the number of RSV-In entries required for each of them.

updates (batched every 5 minutes) from these vantage points over the data collection month. We collected data from 55 vantage points and processed around $10^{10}$ BGP updates.

## 4.1 Storage

Each agent has two storage-related functionalities: (1) storing information in RSV-Out vectors and (2) storing the RSV-In entries from other ASes. The repository stores the full RSV-Out vectors from all ASes.

| Storage Overhead | | | | | |
|---|---|---|---|---|---|
| | **Interfaces** | **#ASes** | **%** | **RSV-Out** | **RSV-In** |
| **Agent** | 1-10 (avg 2.14) | 68,888 | 89.11% | 22 MiB | 65 MiB |
| | 11-50 (avg 22.27) | 6,707 | 8.68% | 0.23 GiB | |
| | 51-200 (avg 76.23) | 1,607 | 2.08% | 0.80 GiB | |
| | 201-639 (avg 257.95) | 108 | 0.14% | 2.7 GiB | |
| **Repository** | 496,910 | 118,031 | 100% | 5.1 TiB | 3 TiB |

**Table 2:** Storage overhead of RoST with respect to the number of interfaces each AS has. RSV-Out/RSV-In are calculated for the average number of interfaces each group.

### 4.1.1 Agent Storage

In RSV-Out, every entry for an IPv4 IP prefix requires 89 bits[1] and every IPv6 entry requires 209 bits[2]. Based on RIPE RIS's traces, we observed that the ASes that announced

[1]Prefix: 40 bits, BatchID: 32 bits, PathID: 16 bits and status: 1 bit.
[2]Prefix: 136 bits, BatchID: 32 bits, PathID: 16 bits and status: 1 bit.

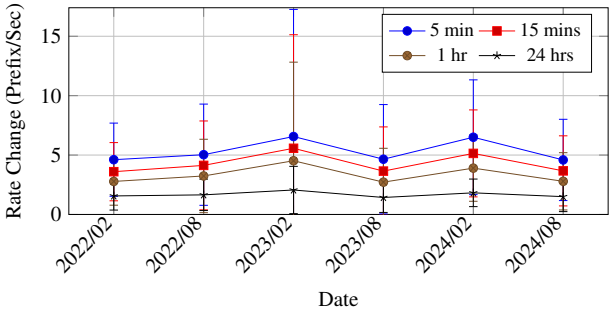



**Figure 6:** Average route updates per second for different batch sizes (dividing the average $\Delta RSVs$ size by the batch interval length). Across all sample dates, the average rates of changes per prefix were 5.32/s, 4.29/s, 3.33/s, and 1.67/s for the 5min, 15min, 1hr, and 24hrs intervals, respectively.

the maximum number of prefixes have announced 991,499 prefixes for IPv4 and 220,074 prefixes for IPv6, see Table 1. We follow a conservative approach and assume that RIPE RIS's traces do not contain all prefixes, since some prefixes announced to each neighbor might not always be visible to RIPE RIS's vantage points. Hence, we round these numbers to 1M IPv4 prefixes and 250K IPv6 prefixes. Furthermore, we assume a worst-case scenario, in which every RSV-Out has entries for *all* of these prefixes. Thus, by multiplying *entry size* $\times$ *#prefixes*, we estimate that the size of a single RSV-Out is 16.83MiB (10.6MiB for IPv4 plus 6.23 MiB for IPv6).

Each agent creates an RSV-Out for every neighboring AS. Based on the RIPE RIS traces, we found that the average number of interfaces per AS is 6.43 and that 89.11% of the ASes have at most 10 interfaces, see Table 2. Based on the aforementioned size of a single (worst-case scenario) RSV-Out, we get that for the average AS the storage size is 106.76MiB (16.84MiB $\times$ 6.43) and for ASes with 10 interfaces, the storage size is 168.39MiB. For ASes with the maximum number of interfaces observed, i.e., 639 interfaces, the storage size is 10.51GiB.

The agent is also responsible for retrieving and maintaining RSV-In associated with the interfaces of the other ASes appearing in the "best routes" of each of its prefixes. To quantify the expected overhead, we count for each of the 991,499 IPv4 prefixes and 220,074 IPv6 prefixes the number of interfaces based on their AS-paths. We observe that in such a worst-case scenario, any agent would need to process 3,578,688 entries for IPv4 and 1,078,736 entries for IPv6. By multiplying each of them with their corresponding entry size (see RSV-Out entry size for IPv4 and IPv6, 89 and 209 bits respectively, see above), we get 65MiB of storage, which can easily fit in a commodity agent machine's memory; see Table 2 (RSV-In column). Note that this number is identical to all agents because it does not depend on the number of interfaces.

#### 4.1.2 Repository Storage

There are two contributing components to the repository's storage requirement: (1) storing all RSVs from each AS in the network and (2) storing the prefixes that an AS validator is interested in receiving route updates from for each of the interfaces. We aggregate the cost for each AS given the number of interfaces there are on the Internet, and assuming the highest-cost scenario in which every AS announces a route to every prefix announced on the Internet to each of its neighbors. We conclude that the repository would need to store 8.1 TiB, which is feasible even under the demanding assumption above.

### 4.2 Communication

We quantify the increase in BGP update length due to adding RoST's identifying tags (encoded as transitive BGP attributes) and the bandwidth consumption when ASes exchange ΔRSVs with the repository.

**BGP update length overhead.** Each BGP announcement in our protocol must include a new transitive attribute, the *RouteIDs*. This attribute slightly increases the size of BGP messages. To quantify this increase, we analyzed the routing table from a RIPE-RIS route collector (rrc00) using the BGPdump tool on August 1st, 2024. The results show that the average AS-path length for IPv4 is 3.86 hops (when excluding AS path prepending). For each hop on the route, RoST encodes its identifier as a 7B attribute (1B for the length and 6B for BatchID and PathID). This 27B increase represents a modest increase compared to the average BGP update length, especially considering that a BGP update message is allowed up to a maximum size of 4096B [37].

**Bandwidth overhead.** Agents generate ΔRSVs at each batch containing route updates, and agents at other ASes periodically fetch and validate announcements they receive against these vectors. To evaluate RoST's bandwidth overhead, we measure the number of prefixes that experienced route changes at different batch intervals (5 minutes, 15 minutes, 1 hour, and 24 hours). The shorter the batch interval, the faster RoST would be able to react to withdrawal suppression, but the greater the bandwidth overhead. This is because a route update requires a single ΔRSV entry per batch, regardless of the number of updates within that batch and whether it was due to withdrawal or path modification.

We use the route updates observed from RIPE-RIS vantage points through six different months February and August 2022, 2023 and 2024. For each vantage point, we calculate the number of prefixes that experienced route changes within the different batch intervals we consider. Figure 6 illustrates the frequency of route changes; we use this data to calculate the bandwidth cost of sending and receiving updates from the repository.

#### 4.2.1 Agent Communication

**Bandwidth overhead for submitting ΔRSVs.** Each AS periodically publishes ΔRSV to the repository. As shown in Figure 6, ΔRSV should reflect average changes between 1.67 to 5.32 prefixes per second, depending on the interval length. The bandwidth overhead for a specific agent is also proportional to the number of interfaces its AS has, since any route update requires updating the ΔRSV for each interface. In Table 3 we can see the average bandwidth for each category of ASes and each different batch. Even for ASes in the category with the maximum number of interfaces (201-639), the average bandwidth consumption is very low, at only 122.13 Kbps even when batch interval is 5 minutes.

**Bandwidth overhead for fetching ΔRSVs from the repository.** A RoST agent synchronizes with the repository to obtain the latest route updates (ΔRSVs). Agents only request updates for active routes within their ASes to minimize bandwidth costs. The average cost per agent is negligible 0.21 Kbps for 5 minute intervals (see Table 3). To derive the repository's cost, we sum across all ASes and get 25.12 Mbps which are more than reasonable costs for a regular server with good Internet connection and negligible compared to the overall response.

The response bandwidth is more significant since it includes a Merkle tree inclusion proof for each updated entry. Each entry contains 89 bits for the RSV regular entry, plus an additional 640B for the Merkle proof, assuming SHA-256 as the hash function, and a tree with depth 20 to support the 1M IP prefixes announced on the Internet. To estimate the number of entries of a response, we refer again to the results in Figure 6, which represent the average number of prefix changes per second.

It is important to note that while these figures represent the prefixes that experienced route changes, the actual number of responses sent by the repository depends on how many interfaces have reported these changes. In the worst-case scenario, all interfaces along the path are impacted and report the changes. As mentioned earlier, on average, non-prepending paths have a length of 3.86 ASes. So the average bandwidth under these assumptions will be around 14KB/s (*route change/sec* $\times$ *path length* $\times$ *entry size*) for 5 minutes of batch interval, which, as explained for the ΔRSV-Out bandwidth, is acceptable for most machines. The overall response cost for the repository, considering a total of 100K ASes across the entire Internet, is between 7.9-12.63 Gbps (see Table 3 for exact results). We expect overhead to be much less than this, however, modern server and network hardware (e.g., Cisco [12]) can support such data transfer rates.

| Bandwidth Overhead | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Interf. | ASes | 5min | | | 15min | | | 1hr | | | |
| | | | ΔRSV-Out | ΔRequests | ΔRSV-In | ΔRSV-Out | ΔRequests | ΔRSV-In | ΔRSV-Out | ΔRequests | ΔRSV-In | |
| Agent | 1-10 | 89.11% | 1.01 Kbps | 0.21 Kbps | 106.97 Kbps | 0.82 Kbps | 0.17 Kbps | 86.26 Kbps | 0.63 Kbps | 0.13 Kbps | 66.96 Kbps | |
| | 11-50 | 8.68% | 10.54 Kbps | | | 8.50 Kbps | | | 6.60 Kbps | | | |
| | 51-200 | 2.08% | 36.09 Kbps | | | 29.11 Kbps | | | 22.59 Kbps | | | |
| | 201-639 | 0.14% | 122.13 Kbps | | | 98.49 Kbps | | | 76.45 Kbps | | | |
| Repository | 496,910 | 118,031 | 235.28 Mbps | 25.12 Mbps | 12.63 Gbps | 189.73 Mbps | 20.25 Mbps | 10.18 Gbps | 147.27 Mbps | 15.72 Mbps | 7.90 Gbps | |

**Table 3:** RoST bandwidth, showing 5-minutes, 15-minutes, and 1-hour intervals.

## 4.3 Computation

We next analyze the overhead associated with processing ΔRSVs at the agent and the repository. The steps that dominate the computational costs are: agents generating ΔRSV-Outs (§4.3.1), repository handling updates and requests (§4.3.2), and ASes handling ΔRSV-Ins from the repository (§4.3.3).

### 4.3.1 ASes Uploading Changes to the Repository

Generating a Δ vector involves three main operations:

1. Selecting entries updated in current batch: This can be performed in advance as entries are updated, incurring negligible computational overhead.

2. Calculating the Merkle root: A full RSV contains 1M entries per interface, which results in around 2M hashes per interface.

3. A signature that authenticates the Merkle root with the version and the interface.

The overall average hash rate for agents with the largest number of interfaces sending updates every 5 minutes, results in less than 1M hash/s (859,833-for all numbers, see Table 4; ΔRSV-Out columns). A MacBook Pro with a 3.2 GHz M1 processor can generate *3 million SHA-256 hashes per second* per core (the computation is also parallelizable with multiple cores). Thus, the computational overhead for the root calculation should be reasonable for agent machines. A signature authentication needs to be made for every ΔRSV-Out sent. Assuming the worst-case scenario in which all interfaces were involved in at least one route change at each batch, the number of signatures that need to be made equals the total number of interfaces. Majority of ASes have between 0-10 interfaces (with an average of 2). Assuming that they refresh the repository every 5 minutes, this would result in ≈ 2 signatures every 5 minutes. The same MacBoook used for hash/s benchmarking is capable of performing 100 RSA signatures per second, which would be enough even for AS with higher number of interfaces (639) refreshing every 5 minutes.

### 4.3.2 Repository Handling Updates and Requests

The repository first validates and stores each agent's updates. Since messages are sent using a TLS session, the validation process is minimal. Updates are stored by the repository which requires targeted reads and writes in the DB. We know from the bandwidth evaluation that if all 118,031 agents are sending updates every 5 minutes, this results in 235Mbps (i.e., 29MBps as noted in Table 3). The majority of common disks can read and write at a speed of 300 - 500 MBps, thus, this adds zero I/O latency. The repository also needs to generate ΔRSV-Ins. In this step, the repository needs to generate an inclusion proof for each of the entries that had a route change. If all agents are being updated every 5 minutes (21 entries will need to be generated per second per agent, *route change/sec* (Figure 6)×*3.86 avg path length*), then a total of ≈2.4M entries per second will need to be processed. Each entry requires a Merkle proof containing ≈20 entry hashes (Figure 6). In total, we get a total of 48M h/s. These costs can be improved by caching hashes of already computed entries, but even without such improvement, the overhead is feasible. (E.g, using 18 cores of CPUs, ≈ 54M h/s, at its full computational power.)

### 4.3.3 ASes Handling New Updates from the Repository

The repository pushes new entry updates in the form of ΔRSV-Ins to agents, and agents need to process them. The operations that dominate this process are the verification of authenticity and integrity of the data. The agents should first verify the signature (Merkle root+version+interface). On average, as explained above, if the agent gets updated every 5 minutes, then 21 entries are received per second. Assuming the worst case where each entry belongs to a different interface, this requires 22 verifications/s (Mac M1 Pro (1 core) can compute ≈ 25K verifications/s for RSA). The agent also needs to verify all entries (21 for the same scenario) against the set of hashes included in the Merkle proof, which requires ≈20 hashes per entry; this results in 410 h/s, which remains minimal. For detailed numbers for different intervals, see the agents' ΔRSV-In column Table 4.

| Computational Overhead | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | 5min | | 15min | | 1hr | |
| | Interf. | ASes | ΔRSV-Out | ΔRSV-In | ΔRSV-Out | ΔRSV-In | ΔRSV-Out | ΔRSV-In |
| Agent | 1-10 | 89.11% | 7,133 h/s + 2 sig/5min | 410 h/s + 21 vfy/s | 2,377 h/s + 2 sig/15min | 331 h/s + 17 vfy/s | 594 h/s + 2 sig/1hr | 257 h/s + 13 vfy/s |
| | 11-50 | 8.68% | 74,233 h/s + 22 sig/5min | | 24,744 h/s+ 22 sig/15min | | 6,186 h/s + 22 sig/1hr | |
| | 51-200 | 2.08% | 254,100 h/s + 76 sig/5min | | 84,700 h/s + 76 sig/15min | | 21,175 h/s + 76 sig/1hr | |
| | 201-639 | 0.14% | 859,833 h/s + 258 sig/5min | | 286,611 h/s + 258 sig/15min | | 71,653 h/s + 258 sig/1hr | |
| Repository | 496,910 | 118,031 | 0 h/s + 0 sig/5min | 48,475,803 h/s + 0 vfy/s | 0 h/s + 0 sig/15min | 39,090,450 h/s + 0 vfy/s | 0 h/s + 0 sig/1hr | 30,342,937 h/s + 0 vfy/s |

**Table 4:** RoST Computational Overhead, showing 5-minute, 15-minute, and 1-hour intervals. MacBook Pro with a 3.2 GHz M1 processor can generate *3 Million SHA-256 h/s, 25K RSA-vfy/s and 100 RSA-sig/s* per core. *Legend: h/s* stands for hashes per second, *sig* for signature, and *vfy* for verification.

## 4.4 Comparison to Key Rollover

Key Rollover operates in a different fashion than RoST. It relies on BGPsec and rotates its signing key often, making sure zombie routes become invalid. However, this procedure requires refreshing *all* routes an AS advertises since they were all signed by the BGPsec signing key. Thus, if key rollover is ubiquitously adopted, all routes on the Internet would need to be refreshed often . Each provider, for example, that announces routes for all prefixes to its customers, would need to refresh about 1.2M routes per customer, resulting in major computational and communication costs. RoST, in comparison, makes routing status transparent and available to agents in order to detect zombie routes (rather than proactively periodically revoking all routes). Thus, its overhead is much smaller. It only requires 258 signatures (on average) for the ASes with the higher number of interfaces when its agent submits a new RSV-Out (Table 2). Therefore, the agent can refresh its routing data at the repository at a much higher rate than key rollover may be invoked, which results in a shorter withdrawal suppression exposure interval.

# 5 Benefits Under Partial Adoption

Deploying BGP mechanisms takes effort and usually happens gradually over time. Thus, it is imperative to propose mechanisms that are simple and backward compatible, but also provide benefits even under partial adoption.

RoST is designed with this mindset, providing a direct incentive for adopting ASes. An AS that adopts RoST lets other adopters detect when its own announcements are suppressed, which helps the adopter ensure its routing preferences and business decisions are executed. Though without full adoption, RoST cannot guarantee detection of all withdrawal suppressions, its detection rate increases with the adoption rate.

To illustrate RoST's benefits, consider the scenario depicted in Figure 2, but assume that initially, only AS 4 adopts RoST. First, observe that AS 2 is now incentivized to adopt RoST because it would allow AS 4 to detect that AS 6 continues to attract traffic designated to AS 2 using an outdated route, which AS 2 wants to prevent. Second, once AS 2 adopts

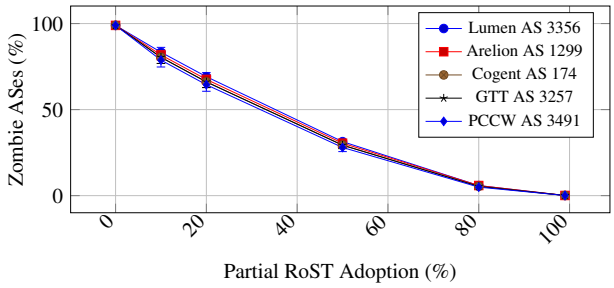



**Figure 7:** Impact of RoST in partial adoption on all Internet ASes when Tier-1 ASes suppress withdrawals. (Error bars mark standard deviation.)

RoST, AS 4 can detect this withdrawal suppression even though ASes 2 and 4 are the only adopters.

## 5.1 Evaluation

To evaluate the impact of partial adoption, we used the BGPy simulator [16], which allows simulating routes on the Internet's AS-level topology. We use a snapshot of the Internet topology from January 2025 from CAIDA [9]. We simulated the following scenario: a prefix is announced by a RoST-adopting AS, which then explicitly withdraws that announcement, while some Tier-1 AS suppresses the withdrawal. We then count the number of ASes that kept the withdrawn route active (i.e., use a zombie route). We make this measurement under different RoST-adoption rates across all Internet ASes, and assign different Tier-1 ASes the suppressor role. We use Tier-1s as suppressors in our simulations since they have the most customers, and therefore, intuitively, they should have the highest impact.

Figure 7 illustrate the impact of adopting RoST on other ASes on the Internet. We repeat the simulation for five different Tier 1 ASes as suppressors, where the adopting ASes are selected uniformly at random for each adoption rate and execution. Each data point in the graph is the average of 1,000 executions, each time with a different origin (error bars mark the 95% confidence interval). The graph shows the number of zombie ASes (i.e., ASes choosing the withdrawn route, y-axis)

monotonically decreases with the rate of adopters (x-axis). Moreover, this benefit shows as soon as ASes begin adopting (in contrast to some interdomain routing mechanisms that require near-ubiquitous adoption to provide benefits like BGPsec or key rollover that is built on top of it). Lastly, we notice that the number of non-zombie ASes is greater than the number of adopters, which illustrates the collateral benefit of adopting RoST. Namely, by adopting RoST and filtering zombie routes, the AS may also indirectly protect others since it would not announce the withdrawn route to them.

# 6 Compatibility with BGP Routers

We now describe how to support RoST on today's routers, without any modification to router software or hardware. Specifically, we describe how the RoST agent of an AS (external to the AS's routers, but with credentials to configure them) can: (1) attach the *RouteIDs* to announcements that the AS exports, and (2) filter withdrawn routes. We illustrate performing these operations on Cisco routers using their standard command interface, but this approach is compatible with devices from other common vendors (e.g., Juniper). Specifically, support for the BGP transitive extended community attribute is widespread [17] and includes the most common routers (such as Cisco [1] and Juniper [22] routers). A library such as Netmiko [8] allows automating interaction with routers for applying these commands. Of course, a complete implementation would need to provide fault tolerance, handling cases where an agent or the repositories fail, e.g., by defaulting to the standard BGP behavior.

## 6.1 Attaching *RouteIDs*

The agent needs to verify the *RouteIDs* of every BGP announcement, and if valid, the agent adds an additional *RouteID* to the community attribute before the announcements are propagated. To that end, we configure the BGP router to send BGP event logs to the agent:

```
logging host <agent-ip>
```

These logs are parsed by the agent to detect when new BGP routes are announced. To ensure that no routes are automatically advertised until the agent completes the validation and updating of the *RouteIDs*, we configure the router to suppress all routes by default in the outbound direction [14]:

```
# Block all routes from being advertised
ip prefix-list <prefix_list_name> permit 0.0.0.0/0 le 32
route-map <route_map_name> deny
 match ip address prefix-list <prefix_list_name>
neighbor <neighbor_ip> route-map out <route_map_name>
```

The above snippet shows how to suppress all prefixes (0.0.0.0/0 le 32) from being advertised to some specific AS (neighbor_ip). To attach a *RouteID* to an exported route, first we need to allow attaching extended community to advertisements with the neighbor (if not already enabled):

```
# Configure Neighbor to Send Extended Communities
router bgp <agent_AS_number>
    neighbor <neighbor_ip> send-community extended
```

Then, we can use Netmiko to dynamically attach a *RouteID* to an advertisement via the following router command:

```
set extcommunity rt <routeid>
```

## 6.2 Filtering Withdrawn Routes

The other functionality that RoST requires is to withdraw routes that the agent detects as withdrawn, even though the explicit/implicit withdrawal was suppressed. Thus, the agent needs to instruct the router to remove a specific (withdrawn) route from the Local RIB, which will cause the router to choose a new best route. Cisco supports this with the following command:

```
# Defines a prefix list to allow specific IP prefixes
ip prefix-list REMOVE permit <ip-prefix>
# Remove the routes matching the REMOVE prefix list
clear ip bgp <neighbor-ip> soft in prefix-list REMOVE
```

# 7 Deployment Considerations

We next discuss privacy and fault tolerance challenges in deploying RoST.

## 7.1 Privacy

Since RoST is a transparency system, it inventively reveals information about the status of routes ASes advertise. Some ASes might be less willing to share this information, at least for some of their neighbors, although it will help ensure that their announcements don't get suppressed. For example, some ASes might not want to expose (specific) paying customers. However, we argue that RoST's impact on privacy is limited in practice. The information made transparent by RoST is on par with what is available through existing or proposed BGP mechanisms. Moreover, a significant portion of this information is already inferred and made public today, for example, by services like RIPE RIS, which we used as part of our evaluation. Furthermore, many of the relationships between ASes can be deduced (explicitly or implicitly) from other BGP-related protocols. For example, RPKI reveals ASes' prefixes (via ROAs) and customer-provider relations (via resource certificates), and the latter can also be deduced from proposed mechanisms such as Only-To-Customer (OTC) [3] and Autonomous System Provider Authorization (ASPA) [4].

That said, some ASes may still decide to withhold some of their route status information, either by only sharing some of the prefixes they can reach through a neighbor or even

completely the fact that they peer with that neighbor. Luckily, this does not impact RoST beyond protecting the withheld routes. It only means that the withholding ASes deny other ASes the opportunity to detect suppression of those routes.

### 7.2 Repository Faults

So far, we described RoST with one repository, creating a single point of failure. In practice, however, we envision a deployment with multiple repositories operated by independent entities. This not only improves the scalability and robustness of the system but allows for mitigating the impact of a malicious repository that selectively hides status updates for some routes (note that since the agent updates are signed, the repository cannot generate fake updates). Such a deployment may also piggyback on the existing BGP repositories, such as the RPKI repositories maintained by RIRs, to maintain RoST' information.

A simple approach for running RoST with multiple repositories is to have agents send ΔRSV-Out updates to all repositories. Agents then receive updates from multiple repositories, ensuring that even when some repositories are down (e.g., due to a major outage) or censor the status for specific routes, the agent will retrieve this information from another repository. This approach ensures all (honest) repositories will have their data without the need to synchronize repositories (and it is the sender's interest to make sure their data is stored at the repository, §5).

Facilitating efficient synchronization across the repositories could save the agent communication costs from the previous solution Such synchronization may use Byzantine Fault Tolerance (BFT) protocols to ensure that all repositories hold the same route statuses despite malicious repositories. Further, the repositories can take turns being the leader in the BFT protocol to ensure a malicious repository cannot censor route status updates [10, 23]. Lastly, BFT protocols create a certificate for the data after repositories reach consensus. This allows agents to update from one repository and check the certificate to ensure other repositories have the same updates.

## 8 Related Work

Significant effort has been devoted to improving the security of interdomain routing [7, 18, 20, 31, 38]. Although not all proposed mechanisms can be used immidiately with currently deployed BGP, the increasing widespread adoption of RPKI [25] is promising since RPKI is a cornerstone for other cryptography-based mechanisms. Specifically, RPKI enables origin validation [11, 19, 24, 27, 33] against prefix hijack attacks, where an attacker advertises a false origin for a prefix to hijack traffic intended for the real origin. BGPsec [28] builds on RPKI to add signatures on route advertisements, providing route authentication that protects against bogus paths advertised with the correct origin. Recently, the IETF published other drafts for mechanisms that aim to increase routing reliability and security. For example, the Only-To-Customer (OTC) [3] transitive BGP path attribute defines a mechanism against route leaks, and the Autonomous System Provider Authorization (ASPA) [4] extends RPKI's origin validation to check ASes beyond the first hop [13].

However, none of the above security mechanisms addresses withdrawal suppression. Approaches that reduce router misconfigurations and software bugs, e.g., through better policy specification, router design, and troubleshooting tools [21, 29], can help reduce the chance of accidental withdrawal suppression, but they require adoption at the suppressing AS rather than the victim. The proposed solution to withdrawal suppression is key rollover [39, 42], which requires route authentication (e.g., via BGPsec). It incurs high operational costs, as well as significant computation and communication overheads from rotating keys often, as discussed earlier in §1. As a result, key rollover cannot operate at short timescales to quickly mitigate withdrawal suppression. In contrast, RoST addresses withdraw suppression by leveraging RPKI to have ASes sign route status data and publish it, requiring no change to RPKI and not relying on a route authentication mechanism. Furthermore, RoST is compatible with BGP and requires no change to BGP routers.

## 9 Conclusion

In this paper, we presented RoST, an efficient and practical protocol designed to defend against withdrawal suppression. RoST can integrate with current routing hardware by using standard APIs and is thus readily deployable. Further, we showed through simulations that RoST provides benefits even to early adopters. We evaluated RoST's overhead based on real-world routing data and showed RoST incurs reasonable storage, compute, and network costs and can thus mitigate withdrawal suppression at much shorter timescales compared to the key rollover alternative (which also relies on BGPsec).

## 10 Acknowledgments

We thank our shepherd, Oliver Hohlfeld, and the anonymous reviewers for their thoughtful comments, suggestions, and feedback. This work was supported by grant no. 2022701 from the United States-Israel Binational Science Foundation (BSF), by the Research Authority Fund of the College of Management Academic Studies, Rishon LeZion, Israel, by grants 2247810, 2410268, and 2149765 of the National Science Foundation, and by Dr. Herzberg's endowment from Comcast. The opinions expressed in the paper are those of the researchers and not of their institutions or funding sources.

## References


[1] Cisco ios bgp command reference. https://www.cisco.com/c/en/us/td/docs/ios/12_2s/feature/guide/fsnextcl.html.

[2] RPKI Monitor - Route Origin Validation. https://rpki-monitor.antd.nist.gov/ROV. Accessed: date.

[3] Alexander Azimov, Eugene Bogomazov, Randy Bush, Keyur Patel, and Kotikalapudi Sriram. Route Leak Prevention and Detection Using Roles in UPDATE and OPEN Messages. RFC 9234, May 2022.

[4] Alexander Azimov, Eugene Uskov, Randy Bush, Keyur Patel, Job Snijders, and Russ Housley. A Profile for Autonomous System Provider Authorization. Internet-Draft draft-ietf-sidrops-aspa-profile-07, Internet Engineering Task Force, January 2022. Work in Progress.

[5] Tony J. Bates and Enke Chen. An Application of the BGP Community Attribute in Multi-home Routing. RFC 1998, August 1996.

[6] Steven Bellovin, Randy Bush, and David Ward. Security Requirements for BGP Path Validation. RFC 7353, August 2014.

[7] Kevin Butler, Toni R. Farley, Patrick McDaniel, and Jennifer Rexford. A survey of BGP security issues and solutions. *Proceedings of the IEEE*, 98(1):100–122, 2010.

[8] Kirk Byers. Netmiko - multi-vendor library to simplify paramiko ssh connections to network devices, 2024. https://github.com/ktbyers/netmiko.

[9] CAIDA. The CAIDA AS Relationships Dataset, 2025-01-05. https://www.caida.org/catalog/datasets/as-relationships/.

[10] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[11] Taejoong Chung, Emile Aben, Tim Bruijnzeels, Balakrishnan Chandrasekaran, David Choffnes, Dave Levin, Bruce Maggs, Alan Mislove, Roland van Rijswijk-Deij, John Rula, and Nick Sullivan. RPKI is Coming of Age: A Longitudinal Study of RPKI Deployment and Invalid Route Origins. In *Proc. of IMC*. ACM, 2019.

[12] Cisco Systems. Unified computing system adapters white paper. https://www.cisco.com/c/en/us/products/collateral/interfaces-modules/unified-computing-system-adapters/whitepaper-c11-741550.html, 2024. Accessed: 2024-09-12.

[13] Avichai Cohen, Yossi Gilad, Amir Herzberg, and Michael Schapira. Jumpstarting BGP security with path-end validation. In *Proc. of ACM SIGCOMM*, pages 342–355. ACM, 2016.

[14] Extreme Networks. Extreme networks article detail. https://extreme-networks.my.site.com/ExtrArticleDetail?an=000082826, 2024. Accessed: 2024-09-12.

[15] Romain Fontugne, Esteban Bautista, Colin Petrie, Yutaro Nomura, Patrice Abry, Paulo Gonçalves, Kensuke Fukuda, and Emile Aben. BGP zombies: An analysis of beacons stuck routes. In *Passive and Active Measurement: 20th International Conference, PAM 2019, Puerto Varas, Chile, March 27–29, 2019, Proceedings 20*, pages 197–209. Springer, 2019.

[16] Justin Furuness, Cameron Morris, Reynaldo Morillo, Amir Herzberg, and Bing Wang. Bgpy: The bgp python security simulator. In *Proceedings of the 16th Cyber Security Experimentation and Test Workshop*, CSET '23, page 41–56, New York, NY, USA, 2023. Association for Computing Machinery.

[17] Yossi Gilad, Tomas Hlavacek, Amir Herzberg, Michael Schapira, and Haya Shulman. Perfect is the enemy of good: Setting realistic goals for BGP security. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks, HotNets 2018, Redmond, WA, USA, November 15-16, 2018*, pages 57–63, 2018.

[18] Amir Herzberg, Matthias Hollick, and Adrian Perrig. Secure Routing for Future Communication Networks (Dagstuhl Seminar 15102). *Dagstuhl Reports*, 5(3):28–40, 2015.

[19] Tomas Hlavacek, Haya Shulman, Niklas Vogel, and Michael Waidner. Keep your friends close, but your routeservers closer: Insights into rpki validation in the internet. In *Proceedings of the 32nd USENIX Conference on Security Symposium*, SEC '23, USA, 2023. USENIX Association.

[20] G. Huston, M. Rossi, and G. Armitage. Securing BGP: A literature survey. *IEEE Communications Surveys & Tutorials*, 13(2):199–222, 2011.

[21] Vinit Jain and Brad Edgeworth. *Troubleshooting BGP: A Practical Guide to Understanding and Troubleshooting BGP*. Cisco Press, 2016.

[22] Juniper Networks. BGP communities and extended communities match conditions overview. https://www.juniper.net/documentation/us/en/software/junos/routing-policy/bgp/topics/concept/policy-bgp-communities-extended-commu

nities-match-conditions-overview.html, 2024. Accessed: 2024-09-12.

[23] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. In *SOSP*, pages 45–58, 2007.

[24] APNIC Labs. Rpki, 2023.

[25] Matt Lepinski and Stephen Kent. An Infrastructure to Support Secure Internet Routing. RFC 6480, February 2012.

[26] Tony Li, Ravi Chandra, and Paul S. Traina. BGP Communities Attribute. RFC 1997, August 1996.

[27] Weitong Li, Zhexiao Lin, Mohammad Ishtiaq Ashiq Khan, Emile Aben, Romain Fontugne, Amreesh Phokeer, and Taejoong Chung. RoVista: Measuring and Understanding the Route Origin Validation (ROV) in RPKI. In *Proceedings of the ACM Internet Measurement Conference (IMC'23)*, pages 1–14, Montreal, Canada, October 2023. ACM.

[28] M. Lepinski (Ed.) and K. Sriram (Ed.). BGPsec Protocol Specification. RFC 8205, sep 2017.

[29] Ratul Mahajan, David Wetherall, and Tom Anderson. Understanding bgp misconfiguration. *ACM SIGCOMM Computer Communication Review*, 32(4):3–16, 2002.

[30] Paweł Małachowski. Bgp zombie routes, 2020. https://es.slideshare.net/slideshow/bgp-zombie-routes/238666533.

[31] Asya Mitseva, Andriy Panchenko, and Thomas Engel. The state of affairs in BGP security: A survey of attacks and defenses. *Computer Communications*, 124:45–60, June 2018.

[32] Cameron Morris, Amir Herzberg, and Samuel Secondo. Bgp-isec: Improved security of internet routing against post-rov attacks. In *Proceedings of the 2024 Network and Distributed System Security Symposium (NDSS 2024)*, 11 2023.

[33] National Institute of Standards and Technology (NIST). NIST RPKI Monitor, version 2.0. https://rpki-monitor.antd.nist.gov/. Accessed November 2023.

[34] RIPE NCC. Routing information service (ris), 2024. https://www.ripe.net/analyse/internet-measurements/routing-information-service-ris.

[35] Porapat Ongkanchana, Romain Fontugne, Hiroshi Esaki, Job Snijders, and Emile Aben. Hunting BGP zombies in the wild. In *Proceedings of the 2021 Applied Networking Research Workshop*, ANRW '21, page 1–7, New York, NY, USA, 2021. Association for Computing Machinery.

[36] Matthew Prince et al. August 30th 2020: Analysis of centurylink/level(3) outage, Jul 2022. https://blog.cloudflare.com/analysis-of-todays-centurylink-level-3-outage/.

[37] Yakov Rekhter, Susan Hares, and Tony Li. A Border Gateway Protocol 4 (BGP-4). RFC 4271, January 2006.

[38] Muhammad S. Siddiqui, Diego Montero, Rene Serral-Gracia, Xavi Masip-Bruin, and Marcelo Yannuzzi. A survey on the recent efforts of the Internet Standardization Body for securing inter-domain routing. *Computer Networks*, 80:1–26, April 2015.

[39] Kotikalapudi Sriram and Doug Montgomery. Design Discussion and Comparison of Protection Mechanisms for Replay Attack and Withdrawal Suppression in BGPsec. Internet-Draft draft-sriram-replay-protection-design-discussion-11, Internet Engineering Task Force, October 2018. Work in Progress.

[40] Dan Tappan, Srihari R. Sangli, and Yakov Rekhter. BGP Extended Communities Attribute. RFC 4360, February 2006.

[41] Pierre-Antoine Vervier, Olivier Thonnard, and Marc Dacier. Mind Your Blocks: On the Stealthiness of Malicious BGP Hijacks. In *NDSS*, 2015.

[42] Brian Weis, Roque Gagliano, and Keyur Patel. BGPsec Router Certificate Rollover. RFC 8634, August 2019.

## A RoST Pseudocode

In this section, we detail the operation of a RoST agent in pseudocode, given in Appendix A.

**Algorithm 2** RoST Agent

```
1: Input:
2:     AgentAS: AS for which this agent is responsible
3:     BatchInterval: Time interval for batching updates in ΔRSV-Out
4:     R: Repository
5: State:
6:     RSV-Out, RSV-In ← ∅                                  ▷ Tracks outgoing & incoming route status per interface
7:     SubscribedInterfaces ← ∅                             ▷ Tracks subscribed interfaces for each prefix
8:     PrivateKey ← LoadPrivateKey(AgentAS)                 ▷ Private key for signing RSV-Out data
9:     BatchInterval ← Input BatchInterval
10:    RepositoryConnection ← ConnectToRepository(R)        ▷ Manages communication with repository
11:    RouterConnection ← ConnectToRouter(AgentAS)          ▷ Connection to the router of 'AgentAS'
12: procedure RoSTAgent(AgentAS, BatchInterval, R)
13:    // Set up event-driven handlers for router and repository interactions
14:    Register ProcessBGPUpdate(U) as a handler for incoming BGP updates from router   ▷ See Line 18
15:    Register ProcessRSVInUpdates(ΔRSV-In) as a handler for RSV-In updates from repository R   ▷ See Line 24
16:    Schedule periodic tasks every BatchInterval to:
17:        Generate and upload ΔRSV-Out to repository R using GenerateAndUploadRSVOut( )   ▷ See Line 30

18: procedure ProcessBGPUpdate(U)
19:    for each Neighbor AS_y of AgentAS do
20:        if IsEligibleToPropagate(AS_y, U.Prefix) then   ▷ See Line 167
21:            Status ← ValidateRouteStatus(U)   ▷ See Line 34
22:            if Status = Valid or Pending then UpdateAndPropagateBGP(U, AS_y)   ▷ Updates RSV-Out state. See Line 108
23:            else if Status = Invalid then HandleInvalidRoute(U, AS_y)   ▷ See Line 90

24: procedure ProcessRSVInUpdates(ΔRSV-In)
25:    ValidatedPrefixes ← MergeRSVIn(ΔRSV-In)   ▷ Returns validated prefixes. See Line 54
26:    for each Prefix P in ValidatedPrefixes do
27:        for each Neighbor AS_y of AgentAS do
28:            if (RIBOutEntry ← QueryRIBOut(Prefix = P, Neighbor = AS_y)) ≠ ∅ then   ▷ See Line 158
29:                if ValidateRouteStatus(RIBOutEntry) = Invalid then HandleInvalidRoute(RIBOutEntry, AS_y) ▷ See
                   Lines 34 and 90

30: procedure GenerateAndUploadRSVOut
31:    for each interface (AS_x, AS_y) in RSVOut do
32:        if (DeltaInterface ← GenerateDeltaRSV(AS_x, AS_y)) ≠ ∅ then   ▷ See Line 130
33:            UploadRSVOut(DeltaInterface)   ▷ See Line 174

34: procedure ValidateRouteStatus(Route, AS_y)
35:    Extract Prefix, AS-Path, and RouteID sequence (BatchID, PathID) from Route
36:    Status ← Valid   ▷ Initial assumption that the route is valid
37:    MissingInterface ← False   ▷ Flag to check if subscription is needed
38:    for each (AS_prev, AS_current) in AS-Path do
39:        Retrieve RSVInInterface ← RSVIn(AS_prev, AS_current)   ▷ Interface-specific RSV-In
40:        Retrieve RSVInEntry ← RSVInInterface[Prefix]   ▷ Access RSV-In entry by prefix
41:        if (AS_prev, AS_current) ∉ SubscribedInterfaces[(Prefix, AS_y)] then
42:            MissingInterface ← True   ▷ Mark that the subscription needs to be updated
43:        if RSVInEntry does not exist or BatchID > RSVInEntry.BatchID then
44:            Status ← Pending   ▷ Mark status as pending due to missing data
45:        else if BatchID < RSVInEntry.BatchID or PathID ≠ RSVInEntry.PathID or RSVInEntry.Status = Inactive then
46:            return Invalid   ▷ Inconsistent or inactive route
47:    if MissingInterface = True then
48:        // Update the subscription state for this AS_y and Prefix
49:        Replace SubscribedInterfaces[(Prefix, AS_y)] with all interfaces in AS-Path
50:        // Prepare the full subscription set for all AS_y for the Prefix
51:        FullSubscriptionSet ← ⋃_{AS_y} SubscribedInterfaces[(Prefix, AS_y)]
52:        SubscribeToRSVInUpdates(Prefix, FullSubscriptionSet)   ▷ Send a single subscription for the prefix across all AS_y
53:    return Status   ▷ Return Valid or Pending status
```

```
54: procedure MERGERSVIN(ΔRSV-In)
55:     // Step 1: Extract batch-level metadata
56:     Extract BatchCounter, MerkleRoot, Signature, (AS_prev, AS_current) from ΔRSV-In
57:     // Step 2: Retrieve Interface-Specific BatchCounter
58:     RSVInInterface ← RSVIn(AS_prev, AS_current)
59:     if RSVInInterface does not exist then
60:         Initialize RSVIn(AS_prev, AS_current) ← ∅
61:         LocalBatchCounter ← 0                                   ▷ Default value for new interface
62:     else
63:         Retrieve LocalBatchCounter ← BatchCounter stored in RSVIn(AS_prev, AS_current)
64:     // Step 3: Check BatchCounter from ΔRSV-In
65:     if BatchCounter ≤ LocalBatchCounter then
66:         Log: "Stale ΔRSV-In batch from (AS_prev, AS_current) with BatchCounter BatchCounter"
67:         Reject and exit
68:     // Step 4: Verify Batch Signature
69:     Reconstruct SignedData ← HASH(BatchCounter, MerkleRoot, (AS_prev, AS_current))
70:     Retrieve PublicKey ← RPKI[AS_prev]
71:     if not VERIFYSIGNATURE(SignedData, Signature, PublicKey) then
72:         Log: "Invalid signature for ΔRSV-In batch from (AS_prev, AS_current)"
73:         Reject and exit
74:     // Step 5: Process Individual Entries with Merkle Proofs
75:     ValidatedPrefixes ← ∅                                       ▷ Initialize list of validated prefixes
76:     for each entry E in ΔRSV-In do
77:         Extract Prefix, BatchID, PathID, Status, MerkleProof
78:         // Validate Merkle Proof for Entry
79:         if not VERIFYINCLUSIONPROOF(Prefix, BatchID, PathID, Status, MerkleProof, MerkleRoot) then
80:             Log: "Invalid Merkle proof for Prefix from (AS_prev, AS_current)"
81:             Continue                                            ▷ Reject and skip this entry
82:         // Merge Valid Entry into Local RSV-In
83:         RSVIn(AS_prev, AS_current)[Prefix] ← (BatchID, PathID, Status)
84:         Add Prefix to ValidatedPrefixes
85:         Log: "Successfully merged Prefix from (AS_prev, AS_current)"
86:     // Step 6: Update BatchCounter for the Interface
87:     Update RSVIn(AS_prev, AS_current).BatchCounter ← BatchCounter
88:     Log: "Updated BatchCounter to BatchCounter for (AS_prev, AS_current)"
89:     return ValidatedPrefixes                                    ▷ Return the list of validated prefixes

90: procedure HANDLEINVALIDROUTE(Route, AS_y)
91:     Extract Prefix and specific route attributes from Route
92:     // Step 1: Remove the route from the router's RIB for the interface with AS_y
93:     REMOVEROUTEFROMRIB(Prefix, Route, AS_y)                     ▷ Remove route for the interface with AS_y. See Line 161
94:     Log: "Removed invalid route Route for Prefix to AS_y"
95:     // Step 2: Check for alternative routes in the router's RIB for the interface with AS_y
96:     AlternativeRoutes ← QUERYALTERNATIVEROUTES(Prefix, AS_y) ▷ Query remaining routes for the interface with AS_y. See Line 164
97:     if AlternativeRoutes is empty then
98:         // Step 3: Propagate withdrawal using UpdateAndPropagateBGP
99:         WithdrawalUpdate ← Generate withdrawal update message for Prefix
100:        UPDATEANDPROPAGATEBGP(WithdrawalUpdate, AS_y)           ▷ Attach RouteID and propagate withdrawal. See Line 108
101:        Log: "Withdrawal propagated for Prefix to AS_y"
102:    else
103:        // Step 4: Select the best alternative route and announce it using UpdateAndPropagateBGP
104:        BestRoute ← SELECTBESTROUTE(AlternativeRoutes)
105:        AnnouncementUpdate ← Generate announcement update message for BestRoute
106:        UPDATEANDPROPAGATEBGP(AnnouncementUpdate, AS_y)         ▷ Attach RouteID and propagate announcement. See Line 108
107:        Log: "Announced alternative route for Prefix to AS_y"
```

```
procedure UpdateAndPropagateBGP(U, AS_y)
    Extract Prefix, AS-Path, Action (e.g., Announce or Withdraw) from U
    // Step 1: Update RSV-Out for interface with AS_y
    Retrieve RSVOutInterface ← RSVOut(AgentAS, AS_y)
    // New Prefix Entry Handling
    if Prefix ∉ RSVOutInterface then Initialize empty entry for Prefix in RSVOutInterface
    // New Batch Handling for Prefix Entry
    if RSVOutInterface.BatchCounter > RSVOutInterface[Prefix].BatchID then
        Reset RSVOutInterface[Prefix].PathID ← 0
        Update RSVOutInterface[Prefix].BatchID ← RSVOutInterface.BatchCounter
    // Update Existing Entry Based on Action
    if Action = Announce then
        Increment RSVOutInterface[Prefix].PathID
        Update RSVOutInterface[Prefix].Status ← Active
    else if Action = Withdraw then
        Update RSVOutInterface[Prefix].Status ← Inactive
    // Step 2: Attach RouteID to the BGP Update Using API
    RouteID ← (RSVOutInterface[Prefix].BatchID, RSVOutInterface[Prefix].PathID)
    AttachRouteIDToUpdate(U, RouteID)                          ▷ Router API: Attach RouteID
    // Step 3: Forward Updated BGP Message to AS_y Using API
    ForwardUpdate(U, AS_y)                                     ▷ Router API: Forward message to neighbor
    Log: "Forwarded BGP update for Prefix to AS_y with RouteID (BatchID, PathID)"

procedure GenerateDeltaRSV(AS_x, AS_y)
    Retrieve RSVOutInterface ← RSVOut(AS_x, AS_y)
    // Step 1: Identify Delta Entries for the Current Batch
    Initialize DeltaEntries ← ∅
    for each Prefix in RSVOutInterface do
        if RSVOutInterface[Prefix].BatchID = RSVOutInterface.BatchCounter then
            Add RSVOutInterface[Prefix] to DeltaEntries
    // Step 2: Build Merkle Tree for All RSV-Out Entries
    Initialize AllEntries ← All entries in RSVOutInterface
    MerkleLeaves ← Hash(AllEntries)
    // Hash each entry in AllEntries
    MerkleRoot ← BuildMerkleTree(MerkleLeaves)
    // Step 3: Sign the Merkle Root, BatchCounter, and Interface
    ToSign ← Hash(MerkleRoot, RSVOutInterface.BatchCounter, (AS_x, AS_y))
    Signature ← SignWithPrivateKey(ToSign, PrivateKey)
    // Step 4: Increment BatchCounter and Update BatchStartTime
    Increment RSVOutInterface.BatchCounter
    RSVOutInterface.BatchStartTime ← CurrentTime()
    // Step 5: Prepare and Return Delta
    if DeltaEntries ≠ ∅ then
        return {(AS_x, AS_y), RSVOutInterface.BatchCounter − 1, DeltaEntries, MerkleRoot, Signature}
    return ∅
```

## Router API Module

The following functions interact directly with the router via the `RouterConnection` state. These functions handle operations such as querying or modifying the router's RIB and propagating updates to neighboring ASes. They are invoked by the `RoSTAgent` to enforce protocol logic.

**procedure** FORWARDUPDATE($U$, $AS_y$)
    Use *RouterConnection* to send $U$ to $AS_y$
    Log: "Forwarded BGP update for $U.Prefix$ to $AS_y$"

**procedure** ATTACHROUTEIDTOUPDATE($U$, *RouteID*)
    Use *RouterConnection* to append *RouteID* to $U$
    Log: "Attached RouteID ($RouteID.BatchID, RouteID.PathID$) to BGP update for $U.Prefix$"

**procedure** QUERYRIBOUT($Prefix, AS_y$)
    Query *RouterConnection* for RIB-Out entry matching $Prefix$ for neighbor $AS_y$
    **return** Matching entry or $\emptyset$ if not found

**procedure** REMOVEROUTEFROMRIB($Prefix$, $Route$, $AS_y$)
    Use *RouterConnection* to remove $Route$ for $Prefix$ on the interface with $AS_y$
    Log: "Route $Route$ for $Prefix$ removed from the interface with $AS_y$"

**procedure** QUERYALTERNATIVEROUTES($Prefix$, $AS_y$)
    Use *RouterConnection* to query all remaining routes for $Prefix$ on the interface with $AS_y$
    **return** List of alternative routes for $Prefix$ on the interface with $AS_y$ or $\emptyset$ if none found

**procedure** ISELIGIBLETOPROPAGATE($AS_y, Prefix$)
    Query *RouterConnection* for policy decision on propagating $Prefix$ to $AS_y$
    **return True** if the policy allows propagation, **False** otherwise

**Interaction with the repository** The following functions interact directly with the repository via the `RepositoryConnection` state. These functions handle operations such as subscribing to updates and uploading data to the repository. They are invoked by the `RoSTAgent` to implement repository-specific logic.

**procedure** SUBSCRIBETORSVINUPDATES($Prefix$, $InterfaceSet$)
    Use *RepositoryConnection* to send a subscription request for:
        $Prefix$ and all interfaces in $InterfaceSet$
    Log: "Subscribed to RSV-In updates for $Prefix$ with interfaces $InterfaceSet$"

**procedure** UPLOADRSVOUT($DeltaInterface$)
    Use *RepositoryConnection* to upload $DeltaInterface$