USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

cc-pipe: Breaking Systemic Bottlenecks in RPKI Data Supply Chain with Concurrent and Conflict-Free Pipelines

Chenhui Yu, *Computer Network Information Center, Chinese Academy of Sciences; School of Computer Science and Technology, University of Chinese Academy of Sciences;* Yanbiao Li, *Computer Network Information Center, Chinese Academy of Sciences; School of Computer Science and Technology, University of Chinese Academy of Sciences; Hangzhou Institute for Advanced Study, University of Chinese Academy of Sciences;* Hui Zou, Yuxuan Chen, Shiyi Liu, and Gaogang Xie, *Computer Network Information Center, Chinese Academy of Sciences; School of Computer Science and Technology, University of Chinese Academy of Sciences*

https://www.usenix.org/conference/nsdi26/presentation/yu

This paper is included in the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation.

May 4–6, 2026 • Renton, WA, USA

ISBN 978-1-939133-54-0

Open access to the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation is sponsored by

جامعة الملك عبدالله للعلوم والتقنية

King Abdullah University of Science and Technology

cc-pipe: Breaking Systemic Bottlenecks in RPKI Data Supply Chain with Concurrent and Conflict-Free Pipelines

Chenhui Yu[1,2], Yanbiao Li[1,2,3,*], Hui Zou[1,2], Yuxuan Chen[1,2], Shiyi Liu[1,2], and Gaogang Xie[1,2,*]

[1]Computer Network Information Center, Chinese Academy of Sciences
[2]School of Computer Science and Technology, University of Chinese Academy of Sciences
[3]Hangzhou Institute for Advanced Study, University of Chinese Academy of Sciences

Abstract

While the Resource Public Key Infrastructure (RPKI) is essential for securing BGP, the high latency, limited scalability, and vulnerabilities in the data supply chain severely undermine its security guarantees and impede network operations. Within this supply chain, existing research identifies the Relying Party (RP) validation process as the primary performance bottleneck. This bottleneck originates from the standard monolithic architecture, which enforces strong consistency but incurs high latency. Previous work has pursued incremental optimizations within this architecture, yet achieving substantial gains remains difficult.

Based on extensive measurements, we identify inherent blocking within the paradigm as the root cause. To address this, we propose cc-pipe, a novel pipeline architecture that breaks the fundamental consistency–latency trade-off. By leveraging a predictive conflict graph, cc-pipe enables low-latency incremental data dissemination while preserving strong consistency guarantees. Evaluation with real-world deployment demonstrates that cc-pipe reduces average latency by up to 73.3% across all data with negligible router overhead. It also delivers significant scalability under projected future workloads, as well as robust resilience to misbehaving publication points.

1 Introduction

Border Gateway Protocol (BGP) [1] serves as the de facto inter-domain routing protocol of the Internet. However, it inherently lacks robust security mechanisms, leaving it vulnerable to malicious routing attacks such as prefix hijacking [2]. To address these critical attacks, the IETF has standardized the Resource Public Key Infrastructure (RPKI) [3].

As shown in Fig. 1, RPKI operates through its data supply chain to enhance BGP security. First, resource holders (Certification Authorities) create Route Origin Authorizations (ROAs) [4] that bind IP prefixes to authorized origin ASes, and publish them to Publication Points (PPs). Second, Relying Parties (RPs) periodically retrieve and validate RPKI data from PPs, producing a set of Validated ROA Payloads (VRPs). Third, routers use these VRPs to filter out unauthorized BGP announcements and prevent route hijacks. Currently, RPKI has achieved substantial adoption and success. Its growing adoption has extended its coverage to over 55% of BGP prefixes [5, 6], and it currently validates 71% of global Internet traffic.[1] Furthermore, it has exhibited its effectiveness in preventing major real-world attacks.[2]

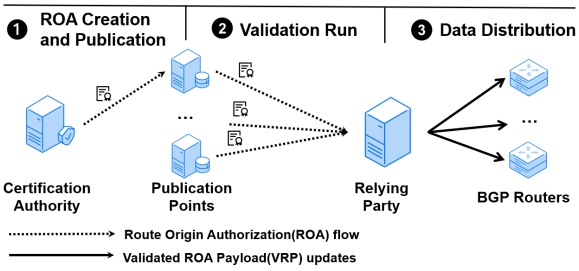


Figure 1: A simplified RPKI data supply chain.

However, this adoption has rendered the growing concerns of the RPKI data supply chain efficiency [7–12]. Currently, the data supply chain requires approximately 15 minutes to disseminate new RPKI data [7, 13]. In contrast, global BGP convergence can occur in as little as 1 minute [14, 15], which means RPKI introduces 15x operational latency. This substantial latency can cause routers operating with obsolete information to fail to identify malicious routes, ultimately leaving the network vulnerable to malicious activities such as traffic disruption [16], DoS attacks [17], and cryptocurrency theft [18, 19]. Furthermore, this latency also impedes legitimate network operations, including time-sensitive traffic engineering and DDoS mitigation [4, 20, 21]. Driven by continuous BGP route churn [22] and expanding RPKI deployment [23, 24], the necessity for RPKI data updates means that RPKI supply chain latency translates directly into BGP

* Corresponding authors: lybmath@cnic.cn and xie@cnic.cn.

[1]https://manrs.org/2024/05/rpki-rov-deployment-reaches-major-milestone/

[2]https://blog.apnic.net/2024/11/18/war-story-rpki-is-working-as-intended

operational risk, where every second counts.

Existing works have identified the overall RP validation process in the supply chain as the major bottleneck [7, 12, 13], a phase centered around the validation run. IETF mandates validation run in strict "wait-for-all" sequential order: an RP must first synchronize with *all* PPs and validate *all* ROAs before distributing any new data to routers by updating the VRP cache (termed as *cache update*) [25, 26]. This constraint guarantees strong consistency, which means it prevents intermediate changes to a route's validation state that lead to unsafe routing or traffic loss. However, strictly prioritizing consistency imposes a severe penalty in a distributed system: the overall latency becomes dictated by the slowest part (i.e., the straggler effect). Therefore, validation run suffers from poor scalability [8–11, 21, 27] and robustness [7, 12, 28, 29]. In future RPKI deployment scenarios, the latency may increase to over 41.5 minutes [30] or even 15 days [31], rendering the supply chain impractically slow for timely defense.

Prior efforts to reduce validation run latency have focused on workload reduction: ❶ Reducing the synchronization workload by compressing ROAs published in PPs [10, 11], designing new PP architectures [8], or replacing encoding schemes [9]. ❷ Reducing the validation workload by designing on-demand validation algorithms [32] and pruning RPKI data to reduce cryptographic operations [9]. However, these approaches are fundamentally limited by their acceptance of the validation run's monolithic paradigm; their performance remains capped by the inherent "wait-for-all" constraint. This constraint introduces significant blocking latency—defined as the period during which ready-to-use data is unnecessarily delayed from *cache update*. Our trace-driven analysis identifies this constraint as the primary performance bottleneck. On average, over a one-year trace collected from the most widely deployed RP software [33], blocking latency accounts for more than 84% of the total validation run latency.

To address this limitation, our work introduces an orthogonal approach. Rather than working on the existing paradigm, we re-architect the paradigm itself. Our design is to replace the monolithic RP validation run with a **C**oncurrent and **C**onflict-Free **Pipe**line (cc-pipe). This new paradigm aims to parallelize *cache update* with synchronization and validation to reduce blocking latency (Fig. 2), while preserving the strong consistency guarantees of the IETF standard. However, realizing this paradigm presents two primary challenges: ❶ Consistency: the system must preemptively resolve dependencies not just among already-obtained data, but against potential future conflicts that have not yet been fetched. ❷ Overhead: the high-frequency update stream must be regulated to prevent overwhelming the BGP router.

In this paper, we present cc-pipe to bridge this gap. It embodies a new paradigm of concurrent and conflict-free validation run. We summarize our contributions as follows:

- **Identifying and Modeling Bottleneck (§3).** We develop a formal model of the RPKI validation run and employ trace-driven analysis to quantify the substantial overhead inherent in the current paradigm. Our results reveal that *blocking latency* constitutes the dominant bottleneck. For the most widely deployed RP software, this latency accounts for more than 84% of the total validation time.
- **Designing a Novel Concurrent Paradigm (§4).** To address the bottleneck, we propose cc-pipe. Its core is a predictive conflict graph. This graph can forecast potential conflicts by using placeholders for unprocessed parts of the RPKI hierarchy. It acts as a dynamic scheduler that locks only the risky, dependent address spaces while allowing safe data to pass through instantly. Furthermore, to handle the high frequency of pipelined updates, cc-pipe incorporates a rate-limiting dispatcher and an extended cache maintenance strategy.
- **Designing and Implementing the System (§5).** To lower the barrier to practical deployment, we build cc-pipe as a reusable library and integrate it into two mainstream RP implementations [34, 35]. This implementation demonstrates the viability of our approach and offers a clear pathway toward real-world adoption and system evolution.
- **Evaluating Performance and Scalability (§6).** Our extensive evaluation demonstrates that cc-pipe reduces validation run latency by up to 73.3% with negligible router overhead. Furthermore, it exhibits superior scalability by reducing latency from over 2.4 hours to under 18 minutes under the heaviest projected workloads, and it demonstrates exceptional resilience by keeping latency stable at 8 minutes in failure scenarios where the stock RP exceeds 1 hour. Finally, a case study shows cc-pipe eliminates over 200s delay in hijack mitigation, demonstrating its immediate impact on operational security.

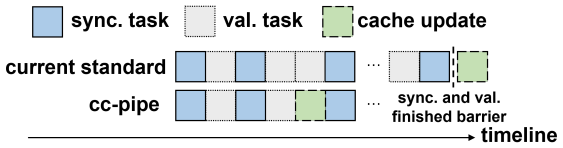


Figure 2: Different validation run paradigm.

2 Background

The RPKI uses a hierarchy of signed objects. A resource holder, acting as Certification Authority (CA), receives its own Internet resources in a Resource Certificate (RC) [36] from a superior CA. The CA can then use its RC to issue RCs to subordinate CAs or issue ROAs to ASes. In all cases, the resources in the issued objects must be a subset of the resources held by the issuing RC [37]. This creates a verifiable chain of ownership from a root Trust, that is, the self-signed RCs of five Regional Internet Registries (RIRs) around the world, down to individual route, origin AS bindings.

An RP periodically performs a validation run to check this entire object hierarchy. It starts from pre-configured Trust

Anchors [38] and recursively synchronizes with various PPs to get the latest RPKI objects and validate them. After synchronization and validation are finished [25, 26], the RP computes a new set of VRPs. A VRP is represented by (`prefix`, `maxLength`, `ASN`). It then compares this new set with the previous VRP snapshot stored in its cache to generate a series of VRP updates. Each VRP update is represented by (`prefix`, `maxLength`, `ASN`, `action`), where the `action` attribute indicates whether it is an announcement or a withdrawal. These VRP updates are committed as a single, monolithic batch. This batch refreshes the VRP cache. The cache uses these VRP updates for two key functions. First, it produces the latest VRP snapshot. Second, it packages the updates into versioned deltas for efficient, incremental synchronization with routers. After the VRP cache is updated, it notifies its connected routers that new data is available [25, 26].

The RP uses the RPKI-to-Router (RTR) protocol [25, 26] to distribute VRP updates to BGP routers. A router first attempts a serial query to request only the incremental deltas needed to update its VRP view to the latest version. However, a serial query can fail, most notably if the router's local version is too stale and the corresponding deltas have already been discarded by the RP. In such cases, the router falls back to a reset query to fetch the entire VRP snapshot.

Upon fetching the latest VRPs from an RP, routers perform Route Origin Validation (ROV) [39–41] on received BGP routes. This process assigns one of three RPKI validity states to each route, including *valid*, *invalid*, and *notFound*. A route can be represented by the tuple (`prefix`, `ASN`). A route is considered *valid* if at least one VRP matches it, meaning the route prefix is either the VRP prefix or its sub-prefix, the length of the route prefix does not exceed VRP maxLength, and the VRP AS equals the route AS. Conversely, a route is deemed *invalid* if a VRP covers it, but no VRP matches it, indicating that although the route prefix is the VRP prefix or its sub-prefix, the maxLength is not satisfied. Lastly, a route is categorized as *notFound* if no VRP covers it at all. Based on the validation results, routers can discard *invalid* routes and prefer *valid* routes to ensure inter-domain routing safety.

3 Modeling the RPKI Bottleneck

This section establishes a formal framework to deconstruct the current monolithic paradigm and lay the theoretical foundation for our concurrent architecture. To formally evaluate the inherent trade-off between latency and consistency, we first define and analyze three abstract models: the monolithic update model (representing the current IETF standard [25, 26]), the naive pipeline model (representing a theoretical ideal performance upper bound), and the conflict-aware pipeline model, which realizes a consistency-preserving pipeline. Furthermore, we analyze the practical overhead of a pipelined approach on BGP routers. This dual-faceted analysis of theoretical bounds and systemic costs allows us to derive the core principles and rigorous engineering requirements that govern our new paradigm.

3.1 Validation Run Models

A validation run validates RPKI objects to generate a set of VRP updates. Let $\mathcal{D} = \{d_1, d_2, \ldots, d_n\}$ be the set of validated RPKI objects, and let $t(d)$ be the validation completion time for an object $d \in \mathcal{D}$. The resulting set of VRP updates is denoted by $\mathcal{V} = \{v_1, v_2, \ldots, v_m\}$. The generation of a specific VRP update v depends on a prerequisite set of RPKI objects, $D_v \subseteq \mathcal{D}$ (detailed in Appendix B). Therefore, the earliest time at which v can be generated, denoted $T(v)$, is determined by the validation time of the last prerequisite object: $T(v) = \max_{d \in D_v} t(d)$. A VRP update v becomes available to BGP routers only after it is committed to the RP's VRP cache. As this timepoint is determined by the specific validation run model, we refer to it as $f(v)$.

Monolithic Update Model. We use the monolithic update model as the current standard, in which a "wait-for-all" barrier requires that all VRP updates wait for all RPKI objects to be validated. Consequently $f_{MU}(v) = \max(\{t(d) \mid d \in \mathcal{D}\})$.

While this model guarantees consistency, VRP updates are blocked from cache updates while waiting for this global synchronization. For a given VRP update v, we refer to this structural idle time as blocking latency ($L_B(v)$), which equals $f(v) - T(v)$. We use $L_{B,MU}(v)$ to represent the blocking latency of v in the monolithic update model, where $L_{B,MU}(v) = \max(\{t(d) \mid d \in \mathcal{D}\}) - T(v)$ quantifies the penalty of the straggler effect.

Naive Pipeline Model. A straightforward alternative is to update the VRP update v to the VRP cache as soon as it is obtained. We use the naive pipeline model to represent this paradigm, where $f_{NP}(v) = T(v)$ and $L_{B,NP}(v) = 0$ by definition. While this model minimizes the blocking latency, it is operationally impractical because it ignores dependencies between updates, leading to transient routing inconsistencies. For example, as shown in the middle panel of Fig. 3, applying interdependent VRP updates in an uncoordinated order can cause a route's validity to incorrectly flip from *notFound* to *invalid* before settling on *valid*. Such a transient *invalid* state can trigger BGP route withdrawals, leading to traffic loss and compromising routing security.

We term this harmful phenomenon an inconsistent ROV state. Formally, an inconsistent ROV state occurs when VRP updates cause a route's ROV state to experience a fluctuation from its initial state (S_{init}) to a final state (S_{final}) via an intermediate state (S_{inter}) during a validation run, where $S_{inter} \neq S_{init}$ and $S_{inter} \neq S_{final}$. This failure arises because it ignores the $f(v)$ order constraint between VRP updates that may change the ROV state of the same route. We formalize such dependencies as a *conflict relationship*: two VRP updates v_i and v_j are in conflict, written $\gamma(v_i, v_j) = 1$, *if and only if* the prefix of one update covers that of the other.

Conflict-Aware Pipeline Model. To prevent the inconsistent ROV state, a practical model must enforce the order

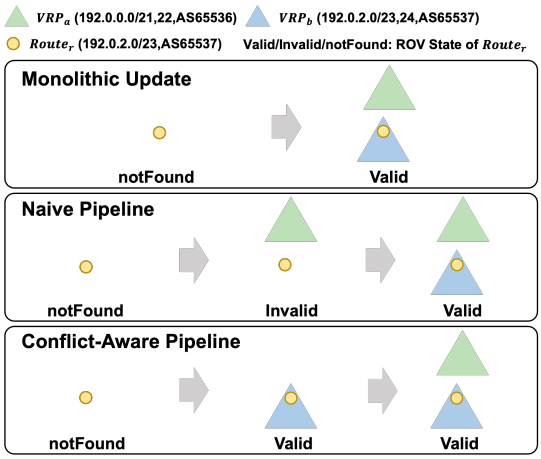


Figure 3: ROV state consistency: naive pipeline (middle) vs. consistent update order (others).

constraint among conflicting VRP updates. We use $o \preceq v$ to denote the temporal constraint that update o must be committed to the cache no later than v, i.e., $f(o) \leq f(v)$. To this end, we introduce the conflict-aware pipeline model, which strictly enforces these order constraints. Formally, $f_{CAP}(v) = \max\left(\{T(o) \mid o \in \mathcal{V}, \gamma(o,v) = 1, o \preceq v\}\right)$. By design, this model avoids the inconsistent ROV state. Furthermore, since $f_{CAP}(v) \leq f_{MU}(v)$ and by definition $L_B(v) = f(v) - T(v)$, it follows that $L_{B,CAP}(v) \leq L_{B,MU}(v)$. Consequently, this model reduces blocking latency compared to the monolithic model, while guaranteeing routing consistency.

In Appendix C, we formalize three ordering principles and prove they are sufficient to prevent all inconsistent ROV states. Practically, committing conflicting updates with an equal cache update time trivially satisfies these principles, an approach our architecture enforces in §4. [3]

3.2 Trace-Driven Latency Analysis

To quantify the real-world impact of these models, we analyzed their performance on historical data. Because no public dataset provides the fine-grained, per-VRP-update timestamps (e.g., $T(v)$) required for our analysis, we developed a custom trace-driven replay mechanism (detailed in Appendix D). With this mechanism, we replayed 16,300 validation runs using a full year of historical RPKI logs (June 1, 2023 – May 31, 2024) archived by `RPKIViews` [43]. Based on these traces, we measure performance using the average end-to-end latency ($\overline{L}$), which consists of computational latency ($\overline{L} - \overline{L}_B$) and blocking latency ($\overline{L}_B$). Crucially, $\overline{L}_B$ quantifies the systemic stall, that is, the duration a VRP update stalls after it is obtained ($f(v) - T(v)$).

Trace-driven analysis exposes the structural inefficiency of

[3] We assume routers apply deltas atomically, consistent with IETF RFC8210bis draft [42]. Our design leverages standard RTR resilience (e.g., TCP, serial checks) to handle transport anomalies without requiring router modifications.

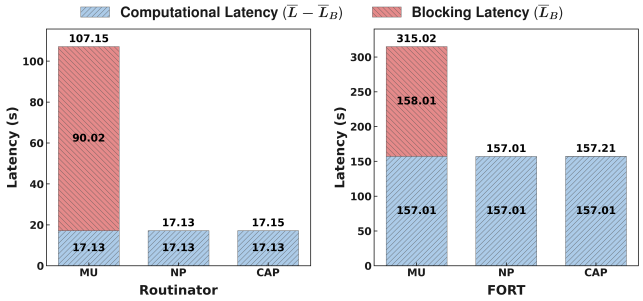


Figure 4: Latency of models.

the monolithic update (MU) model. As illustrated in Fig. 4, the overall average latency ($\overline{L}$) under the MU model is severely inflated by the "wait-for-all" barrier. Specifically, blocking latency ($\overline{L}_B$) accounts for 84.0% (90.02s) of the total time in Routinator [34] and 50.1% (158.01s) in FORT Validator [35] (hereafter referred to as FORT). This stark distribution quantitatively confirms that the temporal cost of the current standard is driven primarily by mandatory waiting, rather than productive computational activity ($\overline{L} - \overline{L}_B$).

The naive pipeline (NP) model represents the theoretical performance bound, eliminating all blocking latency by definition. However, its disregard for consistency makes it impractical for production environments.

The conflict-aware pipeline (CAP) model closely approaches the theoretical performance bound, reducing average latency by up to 86.9% relative to the monolithic standard by eliminating over 99.8% of blocking latency. Our trace-driven analysis reveals the underlying reason for this effectiveness: the vast majority of VRP updates (68.6%) are conflict-free. Among the remainder, 98.4% conflict only with VRP updates originating from the same RC, which are typically validated in close temporal proximity. These empirical findings provide strong evidence that a conflict-aware pipeline can practically resolve the latency-consistency trade-off.

3.3 Traversal Strategy in Pipelined Models

In the monolithic update model, the traversal strategy has a limited effect on latency. All VRP updates are committed to the cache together after all RPKI objects are validated ($\max_{d \in \mathcal{D}} t(d)$), which is determined by the last RPKI object to finish validation. The processing order of earlier objects, therefore, does not change the final cache update time for their corresponding VRP updates. The "wait-for-all" bottleneck effectively masks the impact of the traversal order.

The shift to a pipelined model fundamentally changes this dynamic by exposing the true composition of latency. In a pipeline, end-to-end latency is explicitly decomposed into two components: computational latency (the time from validation run start until the VRP update is obtained), and blocking latency (the time a ready update remains stalled awaiting conflict resolution). The traversal strategy directly influences both metrics; that is, prioritizing the right objects can produce VRP

updates faster and resolve dependencies earlier. Therefore, an intelligent traversal strategy becomes essential to unlock the performance potential of a pipelined architecture.

3.4 Cost Analysis of the Pipeline Model

While a pipeline model lowers latency by updating the VRP cache more frequently, it can increase the processing load on BGP routers. Quantifying this overhead is essential for architecting a practical design. The additional router workload stems from two main sources: CPU usage for processing frequent updates and the risk of incremental update failures.

3.4.1 Overhead from Frequent Notification

To quantify overhead, we assume routers employ *incremental validation*, an efficient strategy that, to the best of our knowledge, is adopted by all modern open-source routers [44, 45]. Under this strategy, when processing a delta of VRP updates from a serial query, a router only revalidates the subset of routes affected by each VRP update, not its entire BGP routing table. With this strategy, a pipeline can create two types of extra CPU load as described below: (1) notification overhead: a pipeline that splits one VRP delta into n VRP deltas requires the router to process $n-1$ more Notify messages. (2) redundant validation overhead: if VRP updates that affect the same BGP route appear in different deltas, the router may perform the same validation calculation multiple times within one validation run. Consequently, a well-designed pipeline architecture must carefully calibrate the notification granularity to avoid redundant validation work. The formal derivation of these overheads is detailed in Appendix E.

3.4.2 Overhead from Serial Query Failures

The primary operational risk of a pipelined model is the increased likelihood of serial query failures. A pipeline generates a higher volume of VRP deltas, which can lead to a serial number mismatch if an RP discards a delta that a router still needs. Such a failure forces the router to fall back to a *reset query* [25, 26], a computationally expensive process requiring it to download the entire VRP snapshot and re-validate all BGP routes. Our measurements show it consumes orders of magnitude more CPU resources than an incremental update, requiring 25.4s of CPU execution time on BIRD and 3.3s on FRRouting, compared to just 74.2ms and 12.1ms for a serial query with 1000 VRP updates. Therefore, preventing these failures is a critical requirement for any practical pipeline architecture.

3.5 Summary and Design Requirements

Our analysis underscores a fundamental tension: the monolithic update model ensures consistency only by imposing severe, systemic blocking latency. While our conflict-aware pipeline model establishes the theoretical feasibility of pipelining, transitioning to a production-grade system necessitates overcoming four primary technical barriers. First, the primary challenge is to create a concrete mechanism that can enforce the model's order constraint. This includes handling predictive constraints for parts of the RPKI hierarchy that have not yet been processed. Second, the design must replace end-of-run batching with incremental VRP update computation and cache update in real-time. Third, a complete solution must also address the router overhead problem. Finally, the pipeline's efficiency is inherently throttled by the underlying traversal strategy; an efficiency strategy is necessary. The following sections introduce cc-pipe, which synthesizes these requirements into a deployable architecture.

4 Core Mechanisms of cc-pipe

This section presents the core mechanisms of cc-pipe, developed to realize our concurrent and conflict-free paradigm formalized by the conflict-aware pipeline model in §3. To address the primary challenge of managing complex and predictive dependencies, we introduce the core component of our architecture: the conflict graph (§4.1). It provides a concrete mechanism to enforce the dependency rules of the conflict-aware pipeline model, ensuring ROV consistency. To bridge the abstract logic with practical implementation, we ground our design description in a concrete running example (§4.1.4). To provide the incremental data stream required by the pipeline, we designed a real-time VRP update and cache update mechanism (§4.2) to replace the standard monolithic process. Finally, to mitigate the router overhead identified in our cost analysis, cc-pipe integrates two specific solutions, including a rate-limiting dispatcher (§4.3.1) that manages the CPU load from frequent notifications, and an extended cache history policy (§4.3.2) that prevents serial query failures.

4.1 Guaranteeing Consistent ROV State

In this section, we present the core mechanism for enforcing the consistent ROV state defined in §3. We first articulate the design rationale (§4.1.1) for leveraging deterministic structural constraints over probabilistic heuristics. We then formally define the Conflict Graph (§4.1.2). Building on this structure, we detail the mechanism for Conflict-Free Clustering (§4.1.3), showing how the system partitions the validation data into conflict-free components for dissemination. Finally, we provide a running example (§4.1.4) to illustrate these concepts with a concrete example.

4.1.1 Design Rationale

To guarantee the consistent ROV state defined in §3, we explored the design space for enforcing the necessary ordering constraints but found three intuitive strategies structurally flawed. First, *Naive Pipelining* fails because it directly ignores the model's ordering requirements, applying conflicting updates in an arbitrary sequence that triggers transient invalid states. Second, *Sorting by Historical Prediction* attempts to satisfy ordering constraints by predicting future dependencies from past logs; however, it fails because future data updates (e.g., emerging sub-prefixes) are inherently unpredictable,

making correct sorting impossible. Third, *Per-RC Batching* fails because it operates on the false assumption that conflicts exist only within a single RC's issued objects; in reality, cross-issuer conflicts (e.g., parent vs. child) break this isolation.

These failures suggest that predicting volatile data and probabilistic heuristics are insufficient for guaranteeing strict consistency. Instead, we leverage the deterministic structural constraints inherent to the RPKI hierarchy. A fundamental invariant of RPKI is **resource containment**: the IP resources of any issued object are strictly a subset of its issuer's holdings. Regardless of how an RC changes its ROAs, the affected IPs are mathematically constrained within its IP resources.

This invariant motivates our *predictive conflict graph*. We use the verified RC as a predictive placeholder to lock this stable bounding box. By tracking these inclusion relationships, the system implements precise deferral. It defers the commit of each update only until the other update required for its ordering is obtained, strictly enforcing the Conflict-Aware Pipeline Model without a global wait.

4.1.2 Conflict Graph Design

Based on the rationale above, we formalize the conflict graph $G = (V, E)$. We treat the authority scope of an RC as a logical lock. It blocks the commitment of dependent data until the scope is fully resolved. We treat structural conflicts as gates. These ensure that no update is released if it conflicts with a lock or another update.

Node Definitions. We map these concepts to two types of nodes based on their *Impact Range* ($R(v)$)—the set of IP prefixes a node affects.

1. **VRP Update Node (v_{vrp}):** Represents a concrete update $(p, len, asn, action)$. Its impact range $R(v_{vrp})$ is strictly the singleton prefix p.

2. **RC Node (v_{rc}):** Represents a *pending* authority scope. It acts as a predictive placeholder for an RC and its unprocessed descendants. Its impact range $R(v_{rc})$ is the union of all resources listed in the RC's certificate (covering potential future announcements) and the resources held by this RC in the previous validation run (covering potential future withdrawals).

Edge Rules. An undirected edge (u, v) acts as the aforementioned "gate". It is added to E *if and only if* the impact ranges overlap ($R(u) \cap R(v) \neq \emptyset$).

- **RC-VRP Edge:** Connects a pending RC node to a VRP update node. This structural link enforces the "lock," preventing the VRP update's release until the RC node is removed.

- **VRP-VRP Edge:** Connects two concrete updates, ensuring they are grouped and committed atomically.

4.1.3 Conflict-Free Clustering

The conflict graph operates as a dynamic synchronization primitive designed to partition the continuous validation stream into *conflict-free clusters*. Its topology evolves throughout the run to strictly enforce ordering constraints, ensuring that updates are grouped and released only when they are provably isolated from potential conflicts. The formal proof of its correctness is presented in Appendix F. This dynamic process is driven by four core operations:

1. Initialization. At the start of a validation run, the graph is initialized with root RC nodes corresponding to the configured Trust Anchors, which are assigned an *impact range* encompassing the entire IPv4 and IPv6 address space, reflecting their role as root authorities.

2. Graph Expansion. The graph expands as the pipeline ingests processed data. Specifically, validating an RC triggers the insertion of an RC Node (establishing a lock). In contrast, a VRP Update Node is inserted only when a concrete VRP update (announce or withdraw) is computed by the method in §4.2. Crucially, this insertion is atomic: the system immediately computes intersections with existing nodes to establish all necessary edges defined in §4.1.2.

3. Graph Contraction. The graph contracts as the pipeline resolves locks. Specifically, an RC node v_{rc} is removed from the graph *if and only if* it is **fully explored**. This state is reached when all its direct descendant objects (ROAs and child Certificates) have been validated, and the generated nodes have been inserted into the graph. This removal serves as the architectural trigger: deleting v_{rc} removes its incident edges, thereby "unlocking" any VRP updates that were strictly waiting for this authority scope to be resolved.

4. Conflict-free Clustering. We define a VRP Component C as a connected subgraph of VRP nodes. A component C is safe to extract only when it is isolated from all potential future conflicts. This condition is met when no VRP update node within VRP Component C has an edge connecting to any RC node in the graph. When an RC node removal severs the last edge connecting C to the authority hierarchy, C satisfies this condition. The system then immediately extracts C from the graph and marks C as ready to commit to the VRP cache atomically. The resulting simultaneously available time strictly satisfies the ordering constraints of the Conflict-Aware Pipeline Model (§3.1), regardless of the underlying object arrival sequence.

4.1.4 Running Example

Fig. 5 illustrates the conflict graph's dynamics with a running example. The process begins with the graph initialized with a root `RIR` node (the Initial panel). As the RP traverses the RPKI topology, it generates `VRP update 1` (step ❶) and discovers the subordinate `RC1` certificate, adding its corresponding nodes to the graph (step ❷). The `RC1` node then acts as a predictive lock, blocking the conflicting `VRP update 1` (generated from newly issued `ROA1`) because the set of objects governed by the `RC1` certificate has not yet been fully explored (step ❸). This lock is released only when all objects directly issued by `RC1` are processed and added to the graph.

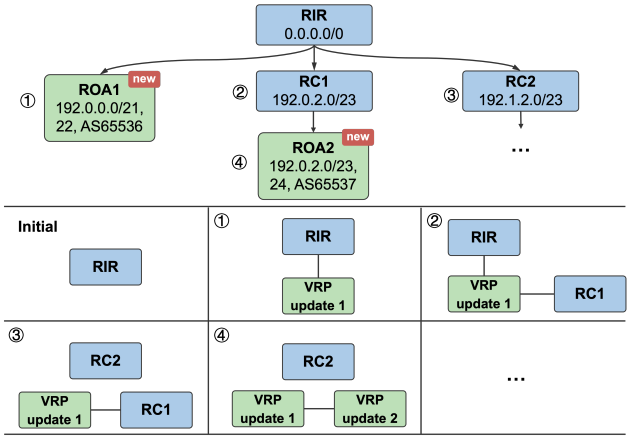


Figure 5: A running example showing the evolution of the conflict graph. The graph (bottom) is dynamically built as the Relying Party traverses an example RPKI topology (top).

After the `ROA2` is validated, `VRP update 2` (generated from the newly issued `ROA2`) is discovered, connected to the `VRP update 1` because of the overlapping impact range. Then the `RC1` placeholder node is removed, and the entire conflicting VRP component is unblocked and becomes ready for an atomic cache update (step ❹).

4.2 Maintaining VRP Update and VRP Cache

A pipeline architecture requires incrementally computing VRP updates and updating the VRP cache as objects are validated. To fulfill this requirement, our design introduces two mechanisms.

Compute VRP Updates. To generate updates in real-time, we manage the VRP state on a per-RC basis.[4] For each RC, we track its VRPs from the previous run (the "previous set") and the current run (the "current set"). By comparing these two sets, we can instantly generate a fine-grained VRP update for that specific RC. This process continuously produces the stream of updates that a pipeline needs.

Update the VRP Cache. Simply applying these incremental, per-RC updates to the global cache is unsafe. An update from one RC's isolated perspective can lead to incorrect cache operations. For example, a "withdraw" from RC_1 might wrongly remove a VRP that is still validly authorized by RC_2, because the global cache has deduplicated their identical authorizations.

To solve this, we introduce the *Global VRP Counter*. It tracks the number of RCs authorizing each unique VRP using a reference count. A local "announce" increments the VRP's count, while a "withdraw" decrements it. A VRP is only removed from the cache when its reference count transitions to zero, proving its *last* authorization is gone. It is only added when its count transitions from zero to one. The *Global VRP Counter* thus acts as a crucial safeguard, ensuring the global cache reflects the collective consensus of all CAs, not the misleading signals from a single, isolated perspective.

[4] We chose to calculate VRP updates at the RC level, as a finer granularity (e.g., per-ROA) would impose more maintenance overhead on RPs without yielding significant efficiency gains, which is due to the design of our conflict graph; and a coarser granularity (e.g., per-PP) would introduce undesirable latency.

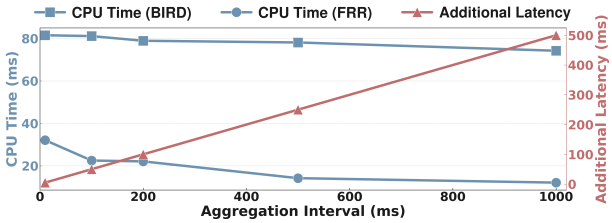


Figure 6: Impact of Aggregation Interval on Router Workload and Additional Latency.

4.3 Controlling Router Overhead

Our analysis in §3.4 showed that a pipelined model introduces three types of router overhead. The first type, redundant validation overhead, is inherently eliminated by our conflict graph. It guarantees that VRP updates with overlapping prefixes are grouped into the same atomic delta, ensuring a router processes any BGP route at most once per validation run. This section, therefore, focuses on the two remaining challenges: the CPU load from frequent notifications and the high cost of reset queries triggered by serial query failures. We present the two dedicated mechanisms cc-pipe uses to mitigate these specific overheads.

4.3.1 Mitigating Notification Overhead

To suppress notification bursts, cc-pipe employs a configurable rate-limiting strategy that aggregates VRP components finalized within a predefined *aggregation interval* into a single cache update and notification. Fig. 6 quantifies the resulting trade-off between propagation freshness and router overhead under a high-stress burst of 1,000 updates per second. Extending the interval from 10ms to 1,000ms reduces the notification load by 99%, cutting CPU execution time by up to 62% (from 32.1ms to 12.1ms on FRRouting). This empirical evidence confirms that notification overhead is an operator-tunable parameter rather than a systemic bottleneck. We adopt a default 500ms interval for our evaluation, striking a balance between sub-second VRP freshness and negligible routing plane load.

4.3.2 Avoiding Serial Query Failures

Standard RPs maintain a history of the latest n VRP deltas to support incremental router synchronization. However, a pipelined architecture generates multiple fine-grained deltas per validation run, which would rapidly flush a fixed-size buffer and shrink the effective history window. This forces routers to fall back to computationally expensive full reset queries. To address this, we shift the cache retention policy from counting deltas to tracking validation runs. We preserve all deltas generated across the most recent n validation

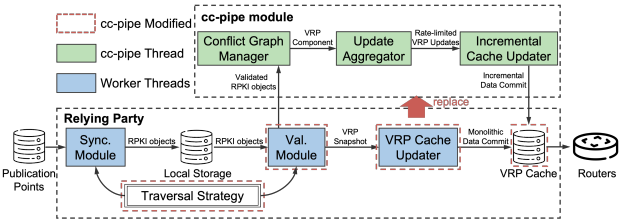


Figure 7: The overview of cc-pipe.

runs, ensuring a consistent history depth regardless of pipeline granularity and preventing serial query failures caused by pipelined architecture. The incremental overhead is strictly limited to the versioning metadata required to index these updates. Even with a history of 10,000 fine-grained deltas, this structural cost remains negligible (e.g., <2 MB), representing a highly efficient trade-off to prevent reset queries.

5 System Design and Implementation

5.1 System Architecture and Integration

The core logic of cc-pipe is engineered as a self-contained module that replaces the RP's monolithic, end-of-run cache updater with a concurrent, conflict-free pipeline, as illustrated in Fig. 7. However, its integration requires a fundamental re-architecture of the host RP's internal data path to support the new, pipelined data model. To avoid performance degradation, the cc-pipe module operates in a dedicated thread, which decouples the complex conflict analysis from the RP's primary synchronization and validation tasks to prevent our mechanism from introducing delays to the validation process.

This re-architecture involves four primary modifications, touching every stage of the VRP update generation process. First, we introduce the cc-pipe module itself, which fundamentally replaces the RP's monolithic, end-of-run cache updater. To feed this new pipeline, we modify the validation module's data flow, enabling a continuous delivery of validated objects. To manage the pipeline's high-frequency output, we also overhaul the VRP cache maintenance strategy with an enhanced retention policy, a critical change to prevent costly router reset queries (§4.3.2). Finally, as our model analysis in §3.3 established the traversal strategy's critical role in pipeline performance, we adapt this strategy where necessary (detailed in §5.2).

Data processing within the pipeline thread begins as the Conflict Graph Manager ingests validated objects to dynamically maintain the predictive conflict graph. This graph identifies transitively conflicting VRP updates, grouping them into atomic VRP components. Subsequently, the Update Aggregator aggregates these components via a configurable rate-limiting policy (§4.3.1) to balance latency against router overhead. Finally, the Incremental Cache Updater incrementally commits these clusters to the VRP cache, leveraging the Global VRP Counter (§4.2) to maintain global data correctness before notifying routers.

5.2 Pipeline-Aware Traversal Strategy

As established in §3.3, an effective traversal strategy is a prerequisite for a high-performance pipeline. This necessity is amplified by the predictive conflict graph (§4.1.2), where RC nodes act as coarse-grained locks that block VRP components until they are fully explored and removed from the graph.

While designing an optimal traversal strategy remains an open challenge, our analysis identifies a practical heuristic: prioritize low-latency data discovery and maximize the temporal locality of dependency resolution. Routinator's strategy effectively implements this. First, it prioritizes the validation of locally stored objects over network synchronization, ensuring fast data discovery to keep the pipeline active. Second, it processes all objects issued by a single RC in a batch. This batching accelerates dependency resolution by allowing the RC node to quickly reach a "fully explored" state, thereby promptly removing the lock from the conflict graph.

In contrast, FORT's Depth-First Search (DFS) is fundamentally misaligned on both fronts. It stalls data discovery on high-latency network I/O, and it fragments the validation of a single RC's descendants across deep recursive calls. This fragmentation delays the completion of the RC node, keeping the dependency lock active for an extended period and throttling the pipeline.

Given these observations, we adopt Routinator's traversal strategy for our implementation. This decision provides a sufficiently efficient baseline, allowing us to focus squarely on validating the core contribution of our conflict-free pipeline architecture.

5.3 Implementation

We implemented the cc-pipe module as a Rust library (∼3,500 lines of code) to provide a modular foundation for our new paradigm. To demonstrate its practicality and interoperability, we integrated this library into two major open-source RPs: Routinator (Rust) and, via a Foreign Function Interface, FORT (C). Although the module is self-contained, this integration required a targeted re-architecture of each host RP's data path, as detailed in §5.1. To handle high-concurrency workloads and reduce memory fragmentation, cc-pipe is explicitly configured to use `jemalloc` as its global memory allocator.

For Routinator, we developed a single enhanced version, cc-Routinator. For FORT, since its native DFS traversal is inefficient for pipelining, we developed four variants to isolate the pipeline's gains from traversal strategy improvements: (1) FORT: the unmodified validator; (2) cc-FORT (DFS): our pipeline with the original DFS; (3) FORT (Optimized Traversal): stock validator with our optimized traversal; and (4) cc-FORT: the fully optimized system.

6 Evaluation

This section presents a comprehensive evaluation designed to quantify the performance and overhead of our cc-pipe. Our goal is to demonstrate the new paradigm's benefits in perfor-

mance, scalability, and robustness, and measure its overhead. First, we assess cc-pipe's performance benefits and overhead under the real-world deployment (§6.2). Next, using an emulator, we evaluate its scalability for future growth scenarios (§6.3) and test its robustness against misbehaving PPs (§6.4). Finally, we present an end-to-end case study to verify how cc-pipe's reduced latency translates into rapid hijack mitigation at the router level (§6.5).

6.1 Methodology

6.1.1 Experimental Environment

Real-World Deployment. To evaluate cc-pipe under realistic conditions, our real-world deployment consists of a high-fidelity RPKI/BGP testbed that synchronizes directly with the live global RPKI ecosystem and distributes VRP updates to software routers processing real-world BGP traces.

Our testbed is deployed across two dedicated servers. This two-server architecture is a deliberate design choice to isolate the systems under test (the RPs) from the measurement environment (the routers). The first server hosts our six RP configurations. Each RP is deployed in a dedicated Virtual Machine with 8 CPU cores and 16GB of RAM, which provides strict resource isolation. The second server, equipped with 128 CPU cores and ∼630GB of RAM, hosts the entire BGP routing plane. This substantial hardware allocation is designed to host all BGP router instances (FRRouting and BIRD) alongside our custom BGP update sender, effectively eliminating resource contention. The BGP update sender injects BGP routes from the RIB in RRC00 [46] into our routers to provide a realistic trace.

Emulation. To evaluate scalability and robustness, which requires scenarios not yet present in the wild, our methodology relies on two key components: a synthetic topology generator and a container-based emulator.

To create high-fidelity test scenarios, we developed a Data-Driven RPKI Topology Generator (detailed in Appendix G). This generator is controlled by two key parameters: the RPKI deployment ratio and the total number of PPs. We chose these parameters because they represent the primary dimensions of the RPKI ecosystem's future growth and are the key drivers of the scalability challenges that cc-pipe aims to solve. Grounded in empirical data, the generator produces realistic RC hierarchies, ROA distributions, and PP layouts, allowing us to generate the precise datasets needed for our scalability and robustness tests. These generated topologies then serve as input to our Emulator, which provides a controlled environment to run the RP configurations in containers and collect their performance metrics.

6.1.2 Data Sources

Our evaluation uses three distinct data sources to assess performance, scalability, and robustness:

- **Performance:** A one-month live trial (Aug. 17 to Sep. 17, 2025) where our testbed RPs synchronized directly with the global RPKI ecosystem.
- **Scalability:** We generated synthetic data sets using our topology generator. To represent a fully deployed global RPKI ecosystem, we scaled the deployment rate from the current 0.57 to 1.0. We also scaled the number of PPs from the current 93 to 10,000.[5]
- **Robustness:** We evaluate resilience against slow PPs, treating availability failures (e.g., timeouts) as worst-case latency events that exacerbate the monolithic blocking bottleneck. We focus on timing anomalies, as content-related failures are handled by the host RP's validation engine. Our test targets two architectural boundary cases within a fixed baseline topology (0.8 deployment rate, 200 total PPs): **(1) RIR PP Failures**, which tests the "gating" effect of root RC nodes whose delayed removal stalls all VRP updates, and **(2) Non-RIR PP Failures**, representing more frequent compound failures in the wild. We simulate a fixed 130s synchronization latency for misbehaving PPs, reflecting real-world underperforming PPs [28], while maintaining a median of 0.815s for functional PPs.

6.1.3 Performance Metrics

Our analysis uses the metrics defined in §3.2. To deconstruct the Average Latency ($\overline{L}$), we analyze its two constituent components: Average Computational Latency ($\overline{L}_C$) and Average Blocking Latency ($\overline{L}_B$), where $\overline{L} = \overline{L}_C + \overline{L}_B$. This decomposition allows us to precisely identify the source of performance gains. For this evaluation, all metrics are aggregated across every VRP update generated throughout our experiments to ensure a comprehensive assessment.

We measure resource overhead on both RPs and BGP routers. For RPs, we record average and peak CPU and memory utilization. For BGP routers, we record average and peak CPU utilization. Consistent with our implementation (§5.3), we utilize `jemalloc` across all RPs to ensure a fair comparison of their inherent architectural memory footprints.

6.2 Evaluation with Real-World Deployment

6.2.1 Latency Reduction

cc-pipe achieves a performance breakthrough by eliminating the architectural root cause: blocking latency. This is clearly evident in Routinator, where cc-Routinator reduces average latency by 73.3% (from 235.3s to 62.8s). As illustrated in Fig. 8, this overall gain is directly attributable to the resolution of the "wait-for-all" bottleneck, reducing $\overline{L}_B$ by over 82%.

The FORT results empirically validate our analysis in §3.3: in a pipelined paradigm, the traversal strategy transitions from a peripheral implementation detail to a primary determinant of efficiency. When constrained by the stock DFS traversal strategy in FORT (DFS), the pipeline yields a marginal

[5] This upper bound for PPs is based on projections from prior work for a future, large-scale RPKI ecosystem [47].

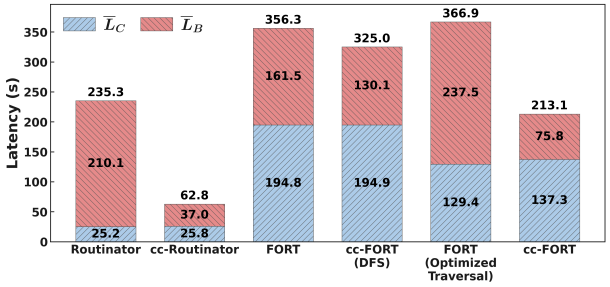


Figure 8: Average Latency of RPs.

7.29% gain. FORT (DFS) stalls in deep, slow subtrees that delay dependency resolution, effectively throttling the pipeline. While optimized traversal alone (FORT (Optimized Traversal)) provides no benefit due to the monolithic "wait-for-all" barrier, it is a necessary prerequisite that unlocks cc-FORT's 41.9% latency reduction. This synergistic effect proves that the pipeline's potential is only fully realized when coupled with an efficient traversal strategy. Consequently, the remainder of our evaluation focuses on the fully-optimized cc-FORT against the stock baseline.

6.2.2 Resource Overhead

cc-pipe reduces latency through a deliberate architectural trade-off: modest RP-side resource costs. As shown in Table 1, cc-pipe increases both memory and CPU usage at the RP. The most notable impact is on memory, with cc-Routinator's average and peak usage rising by 88% and 64%, respectively. CPU utilization increases marginally under average loads, though peak loads rise due to concurrent processing (e.g., from 449.5% to 499.5% for cc-Routinator). We contend these costs are highly justified; the additional resource usage is negligible on modern server hardware. In contrast to the 73.3% reduction in validation run latency, this incremental RP-side overhead represents a highly efficient utilization of available computational resources to enhance routing security.

Table 1: The CPU and memory consumption of RPs.

RP implementations	CPU (%)		Memory (MB)	
	avg.	peak	avg.	peak
Routinator	8.01	449.50	465	896
cc-Routinator	8.16	499.50	873.44	1,473.50
FORT	13.16	401.60	526.17	597.02
cc-FORT	14.24	458.50	775.20	1,075.73

Crucially, cc-pipe's benefits do not translate into increased routing plane load. We observed no statistical change in average or peak CPU utilization for BIRD (0.16%, 9.2%) or FRRouting (0.03%, 0.40%), regardless of the RP implementation. This confirms that modern routers can seamlessly handle the increased notification frequency from our pipelined architecture, proving its immediate deployability without requiring infrastructure modifications.

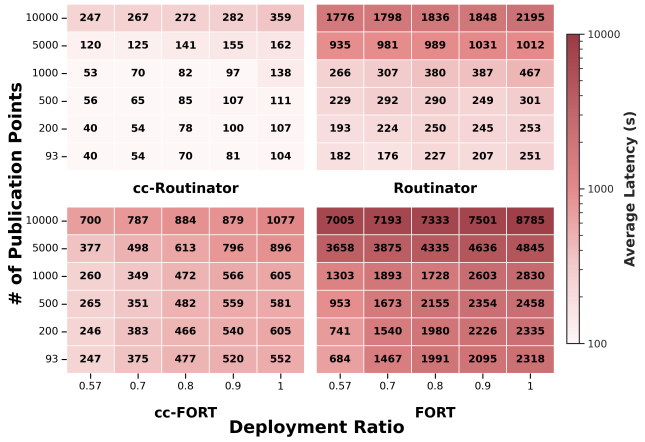


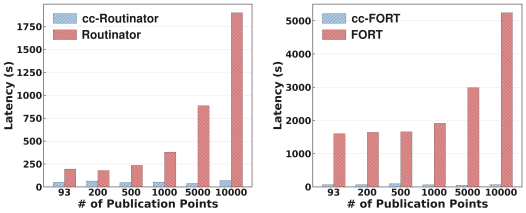
Figure 9: Average latency of RPs in Different Deployment Scenarios. The evaluation includes current RPKI ecosystem parameters, with a deployment ratio of 0.57 and 93 PPs.

6.3 Scalability towards Future Workloads

Scalability evaluation reveals a widening performance divergence between the monolithic baseline and cc-pipe as the RPKI ecosystem expands. As shown in Fig. 9, the stock Routinator's latency escalates from 380s at moderate scale to a prohibitive 2,195s under full-scale deployment (10,000 PPs, 100% deployment ratio). This severe degradation is caused by an explosion in *blocking latency*, a fundamental flaw of the monolithic model (Fig. 10a and Fig. 10b).

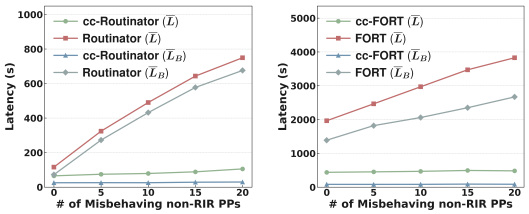
The scalability gap is even more pronounced in FORT, where the stock validator's inefficient DFS traversal gets stalled in slow subtrees. This stall inflates both the blocking latency and the time needed to calculate VRP updates. The VRP updates generation time ($\overline{L}_C$) alone can exceed 3,000s (Fig. 10c). In contrast, cc-FORT suffers from neither issue. Its blocking latency remains at a low level, and it handles the heaviest workloads in 1,077s, a stark contrast to the stock version's 8,785s.

Crucially, this experiment serves as a rigorous stress test for future Internet-scale operations. By simulating 10,000 PPs alongside a 100% deployment ratio (a scale where stock RPs are projected to stall for >2 hours), we confirm that cc-pipe remains robust under drastic topological expansion and the massive update volume inherent to such a scale. Importantly, this resilience does not come at the cost of prohibitive resource consumption. Even under this full-scale extreme scenario, the peak memory footprint remains highly practical for commodity hardware: recording only 3.56 GB for cc-Routinator and 1.75 GB for cc-FORT. Similarly, CPU demand stabilizes around 5–6 cores. This combination of stability and low overhead confirms cc-pipe's future-proof readiness to support a fully saturated global ecosystem, effectively shrinking the vulnerability window.



(a) Routinator($\overline{L}_B$) (b) FORT($\overline{L}_B$) (c) FORT($\overline{L}_C$)

Figure 10: Average latency for a deployment ratio of 1 across various PP counts, including the current count of 93.



(a) Routinator (b) FORT

Figure 11: Average latency across different numbers of slow non-RIR PPs.

6.4 Robustness Against Misbehaving PPs

6.4.1 Non-RIR Publication Points

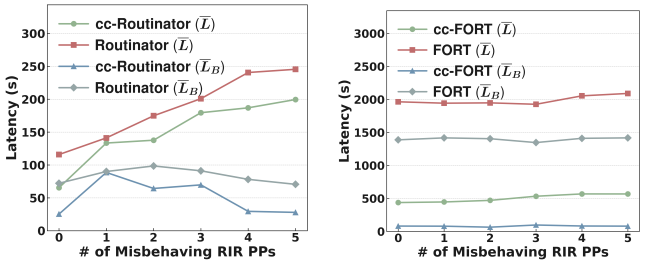
Fig. 11 reveals that both stock validators are vulnerable to slow PPs due to the monolithic "wait-for-all" bottleneck. With 20 slow PPs, Routinator's average latency increases by more than 6x, from 115.8s to 749.3s. Our data shows this is almost entirely due to blocking latency, which increases by over 9x from 72.1s to 675.7s. The stock FORT shows the same vulnerability. Its average latency nearly doubles, increasing from 1,963.6s to 3,827.2s under the same conditions.

In contrast, cc-pipe demonstrates high resilience in both implementations. The average latency of cc-Routinator remains low and stable, starting at 65.3s and only rising to 105.4s with 20 slow PPs. Similarly, the latency of cc-FORT stays flat and controlled, moving from 437.6s to just 481.6s. This resilience against 20 simultaneous failures confirms cc-pipe's immunity to compound failures. Furthermore, by maintaining stability under consistently slow conditions (a static worst-case), the system inherently covers variable latency scenarios.

6.4.2 RIR Publication Points

The experiment tests the impact of slow RIR PPs. Fig. 12a shows that the stock Routinator's performance degrades as slow RIRs are added. Its average latency increases from 115.8s with no slow RIRs to 245.3s with 5 slow RIRs. The stock FORT's latency is consistently high at around 2,000s and does not show a proportional increase with each added slow PP.

The cc-pipe-enhanced versions show different responses to this stress. The latency of cc-Routinator also increases, from 65.3s to 199.4s. This behavior is not a defect but an intended architectural constraint. A slow RIR correctly gates the pipeline to enforce the strict routing consistency. In contrast, the latency of cc-FORT increases more moderately (Fig. 12b), from 437.6s to 567.8s. This lower sensitivity is because FORT's overall validation run is dominated by its computationally intensive validation phase, not its data synchronization phase. Therefore, the fixed 130s synchronization latency from a slow PP has a smaller relative impact on its total latency. Overall, the results indicate that while cc-pipe cannot entirely remove the architectural dependency on timely RIR data, it provides a substantial latency reduction in all tested scenarios.



(a) Routinator (b) FORT

Figure 12: Average latency across different numbers of slow RIR PPs.

6.5 Case Study: Hijack Mitigation

To verify how cc-pipe's latency reduction enhances operational security, we conduct a controlled end-to-end hijack mitigation case study in a BGP testbed.[6] We configure a representative scenario with 200 PPs and a 0.8 deployment rate, reflecting a near-saturated RPKI ecosystem. We publish a remedial ROA at $T = 0$ and trigger an RP validation run at $T = 60s$. In the monolithic baseline, the RP sends the VRP update to the BGP router 253.2s after the validation run starts. Even though the RP has already fetched the necessary data, the update is blocked until the entire global validation process completes. In contrast, cc-pipe identifies the new ROA as conflict-free. It sends the update to the routing plane just 46.3s after the validation run starts. Crucially, our measurements reveal that the RTR transport for a VRP delta is marginal (typically < 1s) compared to the RPKI supply chain delay. This result confirms that cc-pipe effectively mitigates the internal processing bottleneck. By decoupling individual update propagation from the global sync barrier, cc-pipe reduces the routing vulnerability window by over 200 seconds in this scenario.

[6]This case study uses Routinator and cc-Routinator as the Relying Parties, and FRRouting as the BGP router.

7 Related Work

RPKI Performance Concerns. Foundational studies have identified major performance, scalability, and robustness issues in the RPKI data supply chain. Measurements of the RPKI data supply chain reveal a significant latency, with a median duration of 15 minutes [7]. Furthermore, existing work identifies the validation run as its most time-consuming step [7, 12]. This process lacks scalability; its latency is expected to worsen significantly with the increasing deployment ratio and number of PPs [8–11, 21, 27]. A survey showed that 35.3% of network administrators intend to deploy their own PPs in the future [8], indicating that the problem is likely to intensify. Another study projects that the number of PPs will reach 10,000 and predicts that future validation run times could grow to over 41.5 minutes [47], while another work predicted that it will take even 15 days [31]. Furthermore, the process lacks robustness, as its monolithic design means a few slow PPs can stall the entire validation [7, 12, 28, 29]. These critical and worsening issues with the RP validation process motivate the need for a fundamental paradigm shift.

RP Validation Run Optimization. Prior approaches to optimizing the RP validation run have primarily focused on reducing its workload. To decrease synchronization load, Su et al. introduced drr [8], which redesigns the PP architecture; Li et al. proposed hanging-roa [10] and hybrid-roa [11] to compress ROAs at PPs; and Schulmann et al. suggested a new RPKI design [9] that modifies the encoding scheme and protocol to reduce data volume. For validation load reduction, Ma et al. explored a partial validation algorithm [32], and the aforementioned design by Schulmann et al. [9] also minimizes cryptographic operations. While valuable, these approaches all treat the RP's internal, monolithic validation run paradigm as an unchangeable opaque box. They merely minimize inputs of validation run, so their performance remains capped by the inherent "wait-for-all" constraint, which blocks all results until completion of the validation run. Furthermore, many of these approaches require disruptive, ecosystem-wide changes, creating a high deployment barrier [28].

8 Discussion

8.1 Consistency Without Blocking

While prior research treats the RP workflow as an opaque box, focusing primarily on workload reduction, cc-pipe re-architects this core process, shifting the monolithic paradigm to conflict-aware concurrency. Crucially, this architecture is orthogonal to prior workload reduction strategies; combining them yields cumulative performance benefits. To maximize the performance gain, we further recommend that ISPs minimize RCs with excessive resource scopes to reduce dependency density and ensure high-quality service for repositories hosting such critical RCs.

8.2 Failure Modes and Safeguards

First, regarding dependency stalling (due to misconfiguration or attack), the conflict graph offers inherent topological isolation. Since dependencies arise strictly from IP prefix overlaps, the impact range of any RPKI data is structurally confined to the operator's own IP address hierarchy. Consequently, local misconfigurations or attacks cannot stall globally unrelated resource dissemination.

Second, regarding resource exhaustion (e.g., intentional graph bloating), the architecture admits a safeguard strategy. By monitoring the conflict graph and host system metrics (e.g., CPU and memory usage), the system can revert the specific affected subgraph to a coarse-grained batch mode. This ensures the system degrades gracefully to the monolithic baseline rather than crashing.

8.3 Deployment and Evolution

A key advantage of cc-pipe is its immediate deployability. Unlike many previous proposals that require significant, coordinated changes across the global RPKI ecosystem, cc-pipe is a self-contained software enhancement. Its benefits can be realized by operators immediately through a software upgrade.

The Conflict Graph provides a versatile foundation through two core mechanisms: predictive locking based on resource constraints and dependency management via dynamic edge creation. These mechanisms natively generalize to any RPKI objects. Regarding emerging standards like ASPA [48], which introduce a new class of topological conflicts dependent on lateral correlations, we identify the extension of our framework to manage these dependencies as a promising direction for future work.

9 Conclusion

We identified a fundamental flaw in the RP validation run: the inefficient monolithic paradigm. To address this, we introduce cc-pipe, a practical architecture that transitions from serial processing to a conflict-aware concurrent paradigm. By leveraging a predictive conflict graph, cc-pipe resolves the consistency–latency trade-off, achieving up to 73.3% average latency reduction. Our design offers a scalable, robust, and readily deployable solution that strengthens the global routing system without requiring router modifications. The source code of the cc-pipe is available at https://github.com/FIRLab-CNIC/cc-pipe.

10 Acknowledgements

We sincerely thank our shepherd, Arpit Gupta, and the anonymous reviewers for their valuable feedback on this paper. This work was supported by the National Key R&D Program of China (Grant No. 2022YFB3104800), the National Natural Science Foundation of China (Grant No. 62302476), and the Hangzhou Qianjiang Distinguished Expert program (Grant No. HZ20251217834).

References

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

[2] 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, 2009.

[3] Matt Lepinski and S Kent. Rfc 6480: an infrastructure to support secure internet routing, 2012.

[4] Job Snijders, Ben Maddison, Matt Lepinski, Derrick Kong, and Stephen Kent. A Profile for Route Origin Authorizations (ROAs). RFC 9582, May 2024.

[5] NIST RPKI monitor. https://rpki-monitor.antd.nist.gov, 2012.

[6] MANRS. Manrs observatory. https://observatory.manrs.org/, 2025.

[7] Romain Fontugne, Amreesh Phokeer, Cristel Pelsser, Kevin Vermeulen, and Randy Bush. Rpki time-of-flight: Tracking delays in the management, control, and data planes. In *International Conference on Passive and Active Network Measurement*, pages 429–457. Springer, 2023.

[8] Yingying Su, Dan Li, Li Chen, Qi Li, and Sitong Ling. drr: A decentralized, scalable, and auditable architecture for rpki repository.

[9] Haya Schulmann and Niklas Vogel. Pruning the tree: Rethinking rpki architecture from the ground up, 2025.

[10] Yanbiao Li, Hui Zou, Yuxuan Chen, Yinbo Xu, Zhuoran Ma, Di Ma, Ying Hu, and Gaogang Xie. The hanging roa: A secure and scalable encoding scheme for route origin authorization. In *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*, pages 21–30. IEEE, 2022.

[11] Yanbiao Li, Hui Zou, Yuxuan Chen, Yinbo Xu, Zhuoran Ma, Di Ma, Ying Hu, and Gaogang Xie. The hybrid roa: A flexible and scalable encoding scheme for route origin authorization, 2024.

[12] Khwaja Zubair Sediqi, Romain Fontugne, Amreesh Phokeer, Massimiliano Stucchi, Massimo Candela, and Anja Feldmann. Rpki syncing: Delay in relying party synchronization. In *2025 9th Network Traffic Measurement and Analysis Conference (TMA)*, pages 1–11. IEEE, 2025.

[13] Amreesh Phokeer. Tracking time delays in the rpki-based route origin validation supply chain. Accessed on 01 Sept. 2025.

[14] Z Morley Mao, Randy Bush, Timothy G Griffin, and Matthew Roughan. Bgp beacons. In *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*, pages 1–14, 2003.

[15] Alberto García-Martínez and Marcelo Bagnulo. Measuring bgp route propagation times. *IEEE Communications Letters*, 23(12):2432–2436, 2019.

[16] RIPE NCC. Youtube hijacking: A ripe ncc ris case study. https://www.ripe.net/publications/news/youtube-hijacking-a-ripe-ncc-ris-case-study/, 2008.

[17] Simone Ferlin and Michelle Alvarez. Bgp internet routing: What are the threats? https://www.ibm.com/think/x-force/bgp-internet-routing-what-are-the-threats, 2017.

[18] Maria Apostolaki, Aviv Zohar, and Laurent Vanbever. Hijacking bitcoin: Routing attacks on cryptocurrencies. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 375–392, 2017.

[19] Peter Loshin. Bgp routing security flaw caused amazon route 53 incident, 2018.

[20] Ioana Livadariu, Romain Fontugne, Amreesh Phokeer, Massimo Candela, and Massimiliano Stucchi. A tale of two synergies: Uncovering rpki practices for rtbh at ixps. In *International Conference on Passive and Active Network Measurement*, pages 88–103. Springer, 2024.

[21] Alain Durand. Resource public key infrastructure (rpki) technical analysis. *ICANN, Sep*, 2020.

[22] Geoff Huston. Bgp updates in 2025. https://blog.apnic.net/2026/01/09/bgp-updates-in-2025/, 2026.

[23] Daniele Iamartino. Study and measurements of the rpki deployment. 2015.

[24] Yossi Gilad, Avichai Cohen, Amir Herzberg, Michael Schapira, and Haya Schulmann. Are we there yet? on rpki's deployment and security. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017*. The Internet Society, 2017.

[25] R. Bush and R. Austein. The Resource Public Key Infrastructure (RPKI) to Router Protocol, Version 1. RFC 8210, September 2017.

[26] Randy Bush and Rob Austein. The Resource Public Key Infrastructure (RPKI) to Router Protocol. RFC 6810, January 2013.

[27] Haya Schulmann, Niklas Vogel, and Michael Waidner. Rpki: Not perfect but good enough. *arXiv preprint arXiv:2409.14518*, 2024.

[28] Donika Mirdita, Haya Schulmann, and Michael Waidner. {SoK}: An introspective analysis of {RPKI} security. In *34th USENIX Security Symposium (USENIX Security 25)*, pages 3649–3665, 2025.

[29] Koen van Hove, Jeroen van der Ham, and Roland van Rijswijk-Deij. Rpkiller: Threat analysis from an rpki relying party perspective, 2022.

[30] Tomas Hlavacek, Philipp Jeitner, Donika Mirdita, Haya Shulman, and Michael Waidner. Beyond limits: How to disable validators in secure networks. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 950–966, 2023.

[31] Eric Osterweil, Terry Manderson, Russ White, and Danny McPherson. Sizing estimates for a fully deployed rpki. *Verisign Labs, TR*, 1120005, 2012.

[32] Ma and Zhang. A reference implementation of ascertaining rpki signed objects to be validated in incremental updates, 2024.

[33] John Kristoff, Randy Bush, Chris Kanich, George Michaelson, Amreesh Phokeer, Thomas C Schmidt, and Matthias Wählisch. On measuring rpki relying parties. In *Proceedings of the ACM Internet Measurement Conference*, pages 484–491, 2020.

[34] Nlnetlabs/routinator. https://github.com/NLnetLabs/routinator, 2025.

[35] Nicmx/fort-validator. https://github.com/NICMx/FORT-validator, 2025.

[36] G. Michaelson G. Huston and R. Loomans. A Profile for X.509 PKIX Resource Certificates . RFC 6487, February 2012.

[37] S. Kent C. Lynn and K. Seo. X.509 Extensions for IP Addresses and AS Identifiers . RFC 3779, June 2004.

[38] Russ Housley, Sam Ashmore, and Carl Wallace. Trust anchor format. Technical report, 2010.

[39] Prodosh Mohapatra, John Scudder, David Ward, Randy Bush, and Rob Austein. BGP Prefix Origin Validation. RFC 6811, January 2013.

[40] Randy Bush. Origin Validation Operation Based on the Resource Public Key Infrastructure (RPKI). RFC 7115, January 2014.

[41] Geoff Huston and George G. Michaelson. Validation of Route Origination Using the Resource Certificate Public Key Infrastructure (PKI) and Route Origin Authorizations (ROAs). RFC 6483, February 2012.

[42] Randy Bush and Rob Austein. The Resource Public Key Infrastructure (RPKI) to Router Protocol, Version 2. Internet-Draft draft-ietf-sidrops-8210bis-22, Internet Engineering Task Force, September 2025. Work in Progress.

[43] Index of rpkiviews. https://www.rpkiviews.org/, 2024.

[44] Frrouting project. https://frrouting.org/, 2024.

[45] Bird Project. The bird internet routing daemon project. https://bird.network.cz/, 2024. Accessed: 2024-10-01.

[46] RIPE NCC. RIS Raw Data Archive - RRC00, 2025.

[47] Tomas Hlavacek, Philipp Jeitner, Donika Mirdita, Haya Shulman, and Michael Waidner. Beyond limits: How to disable validators in secure networks. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 950–966, 2023.

[48] Alexander Azimov, Eugene Uskov, Randy Bush, Keyur Patel, Job Snijders, and Russ Housley. A Profile for Autonomous System Provider Authorization. Internet-Draft draft-azimov-sidrops-aspa-profile-01, Internet Engineering Task Force, January 2019. Work in Progress.

APPENDIX

A List of Abbreviations

To assist readers, Table 2 provides a summary of the key abbreviations used throughout this paper.

B Defining the Object Set D_v

The earliest generation time $T(v)$ for any VRP update v is determined by the completion of validation for a set of prerequisite RPKI objects, denoted as D_v. This appendix provides a detailed description of the composition of D_v for both `announce` and `withdraw` VRP updates. In essence, D_v comprises the minimal set of objects that collectively form a verifiable proof for the VRP state change represented by v.

VRP announcement. An `announce` VRP update is generated when a Route Origin Authorization (ROA) is found to be valid in the current validation run, whereas it was not considered valid in the previous run. For an ROA to be deemed valid, it must not only be internally consistent but also be part of a valid chain of trust tracing back to a Trust Anchor. Therefore, the prerequisite object set D_v for an `announce` VRP update triggered by a ROA (denoted as ROA_v) includes:

Table 2: List of Abbreviations

Abbreviation	Full Form
General Network & Routing	
AS	Autonomous System
ASN	Autonomous System Number
BGP	Border Gateway Protocol
ISP	Internet Service Provider
RIR	Regional Internet Registry
NIR	National Internet Registry
RPKI Architecture	
RPKI	Resource Public Key Infrastructure
CA	Certification Authority
RP	Relying Party
PP	Publication Point (Repository)
RTR	RPKI-to-Router Protocol
TAL	Trust Anchor Locator
ROA	Route Origin Authorization
RC	Resource Certificate
VRP	Validated ROA Payload
CRL	Certificate Revocation List
MFT	Manifest
ROV	Route Origin Validation

- **The ROA Itself:** The ROA_v object must be successfully fetched and validated (e.g., its signature verified, expiry date checked).
- **The Certificate Chain:** The entire chain of RCs, from the one that issued ROA_v all the way up to the self-signed Trust Anchor certificate, must be valid. The validation of each certificate in this chain is a prerequisite.
- **Associated Manifests and CRLs:** At each traversed publication point in the chain, the corresponding Manifest and Certificate Revocation List (CRL) must be fetched and validated. Together, they ensure that all objects (including the RC, CRL, and ROA_v) are present, untampered, and have not been revoked.

The calculation of the `announce` VRP update can only proceed after the *last* of these dependent objects has been successfully validated. Thus, $T(v)$ is the maximum of all the $t(d)$ for every object d in this collective set D_v.

VRP withdrawal. A `withdraw` VRP update is generated when a ROA that was considered valid in the previous run is no longer valid in the current run. The invalidation can occur for several reasons, and the composition of D_v depends on the specific cause. The set D_v consists of the object(s) from the *current* run that prove the invalidity of the ROA from the *previous* run. Common causes include:

- **ROA Deletion:** The ROA object is no longer present at the publication point. The prerequisite object is the new **Manifest** file at that publication point, which no longer lists the ROA.
- **Certificate Revocation:** An ancestor RC in the ROA's chain of trust has been revoked. The prerequisite object is the new **CRL** issued by the parent of the revoked certificate, which now lists the certificate's serial number.
- **Object Expiration:** The ROA itself, or any ancestor RC, has expired. The prerequisite object d is the **expired object** itself.
- **Resource Contraction:** An ancestor RC has been re-issued with a smaller set of IP resources, such that it no longer covers the resources claimed in the ROA. The prerequisite object is this newly issued, more restrictive RC.

In each case, $T(v)$ is the timestamp when the validation of the minimal set of objects confirming the invalidation condition is complete.

C VRP update Ordering Principles

To systematically prevent the inconsistent ROV state defined in Section 3, this appendix proposes and formalizes a set of three ordering principles that govern the dispatch order for any conflicting VRP update pair. We then formally prove that adherence to these principles is sufficient to prevent all inconsistent ROV states during a validation run.

C.1 Formal Preliminaries

To ground our proof, we first formalize the ROV state determination process. For any given route r and a set of VRPs $\mathcal{V}_{vrp}$ held by a router, we define a set of attributes, $\mathcal{P}(r, \mathcal{V}_{vrp})$, which is a subset of $\{\text{MATCHED}, \text{COVERED}\}$.

- $\text{MATCHED} \in \mathcal{P}$ if $\exists v \in \mathcal{V}_{vrp}$ such that v matches route r.
- $\text{COVERED} \in \mathcal{P}$ if $\exists v \in \mathcal{V}_{vrp}$ such that v covers route r.

The ROV state is a deterministic function, $S(\mathcal{P})$, of this attribute set, where the MATCHED attribute has the highest precedence:

- If $\text{MATCHED} \in \mathcal{P}$, then $S(\mathcal{P}) = \textit{Valid}$.
- If $\text{MATCHED} \notin \mathcal{P}$ and $\text{COVERED} \in \mathcal{P}$, then $S(\mathcal{P}) = \textit{Invalid}$.
- If $\mathcal{P} = \emptyset$, then $S(\mathcal{P}) = \textit{NotFound}$.

An inconsistent ROV state occurs if, during a single validation run, the attribute set for a route r evolves in a way that causes its ROV state to transition through a harmful intermediate state.

C.2 Ordering Principles

To formalize our consistency model, let a VRP update be represented as a tuple $v = \langle p, l, a, op \rangle$, where p is the IP prefix, l is the `maxLength`, a is the origin ASN, and $op \in \{\texttt{ann}, \texttt{wit}\}$ represents the `announce` or `withdraw` operation.

Furthermore, we define the *authorization coverage* $C(v)$ of a VRP update as the topological "triangle" within the IP prefix tree. Formally, $C(v)$ is the set of all sub-prefixes subsumed by p whose prefix lengths are less than or equal to l. We use

$v_1 \preceq v_2$ to denote $f(v_1) \leq f(v_2)$, and $v_1 \simeq v_2$ to denote strict atomicity where $f(v_1) = f(v_2)$.

For any two conflicting VRP updates v_1 and v_2, their relative ordering is governed by three strict principles.

Principle 1: Authorization Precedence. When two conflicting updates perform different operations ($v_1.op \neq v_2.op$), the `announce` operation must strictly precede or be concurrent with the `withdraw` operation. Formally, without loss of generality, if $v_1.op = \texttt{ann}$ and $v_2.op = \texttt{wit}$, then: $v_1 \preceq v_2$.

Principle 2: Same-ASN Ordering. When two conflicting updates share the same operation and origin ASN ($v_1.op = v_2.op \wedge v_1.a = v_2.a$), their temporal order is resolved by their `maxLength` (l):

- **For `ann` operations:** The update with the larger `maxLength` takes precedence. If $v_1.l > v_2.l$, then $v_1 \preceq v_2$. This systematically prioritizes the propagation of more specific routing authorizations.
- **For `wit` operations:** The update with the smaller `maxLength` takes precedence. If $v_1.l < v_2.l$, then $v_1 \preceq v_2$. This prioritizes the revocation of less specific authorizations.

Principle 3: Different-ASN Ordering. When two conflicting updates share the same operation but originate from different ASNs ($v_1.op = v_2.op \wedge v_1.a \neq v_2.a$), their ordering is dictated by the intersection of their authorization coverages (the VRP "triangles"):

- **Intersecting Coverages:** If their authorization triangles overlap ($C(v_1) \cap C(v_2) \neq \emptyset$), they must be applied atomically: $v_1 \simeq v_2$.
- **Disjoint Coverages:** If their authorization triangles are strictly disjoint ($C(v_1) \cap C(v_2) = \emptyset$), the updates govern independent topological spaces. Their relative order defaults to the `maxLength` heuristic defined in Principle 2.

C.3 Sufficiency Proof

Proof Strategy. Let $\mathcal{P}_r$ be the ROV attribute set for a given BGP route r. A harmful, inconsistent state transition occurs if the ROV state experiences a transient fluctuation: $S_{init} \rightarrow S_{inter} \rightarrow S_{final}$, where $S_{inter} \neq S_{init}$ and $S_{inter} \neq S_{final}$.

First, a transient *Valid* state is impossible within a single validation run. Such a transition would require a VRP that provides a MATCHED attribute to be both created and withdrawn in the same validation run, which is impossible.

Therefore, harmful inconsistencies strictly reduce to two transient patterns over $\mathcal{P}_r$:

1. **Pattern 1: Transient *NotFound* State.** $\mathcal{P}_{init} \neq \emptyset \rightarrow \mathcal{P}_{inter} = \emptyset \rightarrow \mathcal{P}_{final} \neq \emptyset$.
2. **Pattern 2: Transient *Invalid* State.** $\mathcal{P}_r$ evolves from not solely {COVERED} to strictly {COVERED}, and back to not solely {COVERED}.

We use proof by contradiction to demonstrate that our three Ordering principles (§C.2) are strictly sufficient to preclude both patterns.

Preventing Pattern 1 (Transient *NotFound*). Assume Pattern 1 occurs. For $\mathcal{P}_r$ to become transiently empty, a `withdraw` operation v_w must take effect to remove the existing attributes. For $\mathcal{P}_r$ to subsequently become non-empty, an `announce` operation v_a must later take effect. This strictly dictates the temporal ordering: $f(v_w) < f(v_a)$, denoted as $v_w \prec v_a$. However, Principle 1 (Authorization Precedence) mandates that for any conflicting pair, $v_a \preceq v_w$. Thus, we have $(v_w \prec v_a) \wedge (v_a \preceq v_w) \implies \bot$. The pattern is impossible.

Preventing Pattern 2 (Transient *Invalid*). We systematically exhaust all transition paths yielding a transient *Invalid* state for route r:

- **Case: *Valid* → *Invalid* → *Valid*.** This path requires losing the MATCHED attribute and regaining it. Similar to Pattern 1, this necessitates a withdrawal v_w executed before an announcement v_a, yielding $v_w \prec v_a$. By Principle 1, $v_a \preceq v_w \implies \bot$.
- **Case: *NotFound* → *Invalid* → *Valid*.** This sequence requires first applying v_c ($v_c.op = \texttt{ann}$), which provides only a COVERED attribute, followed by v_m ($v_m.op = \texttt{ann}$), which provides a MATCHED attribute. This imposes the temporal requirement $v_c \prec v_m$. Because v_m strictly matches route r while v_c only covers it, v_m asserts a more specific authorization, implying $v_m.l > v_c.l$.
  - If $v_m.a = v_c.a$: Principle 2 mandates that the update with the larger `maxLength` precedes: $v_m \preceq v_c$. Contradiction ($v_c \prec v_m \wedge v_m \preceq v_c \implies \bot$).
  - If $v_m.a \neq v_c.a$: If $C(v_m) \cap C(v_c) \neq \emptyset$, Principle 3 strictly mandates atomicity: $v_m \simeq v_c$. Contradiction ($v_c \prec v_m \wedge v_m \simeq v_c \implies \bot$). Otherwise, by Principle 3, $v_m \preceq v_c$. Contradiction ($v_c \prec v_m \wedge v_m \preceq v_c \implies \bot$).
- **Case: *Valid* → *Invalid* → *NotFound*.** This path is the logical inversion of the previous case, driven by withdrawal operations. It requires withdrawing a matching VRP v_{mw} ($v_{mw}.op = \texttt{wit}$) before withdrawing a covering VRP v_{cw} ($v_{cw}.op = \texttt{wit}$). This imposes $v_{mw} \prec v_{cw}$. Again, $v_{mw}.l > v_{cw}.l$.
  - If $v_{mw}.a = v_{cw}.a$: For `withdraw` operations, Principle 2 mandates the smaller `maxLength` precedes: $v_{cw} \preceq v_{mw}$. Contradiction ($v_{mw} \prec v_{cw} \wedge v_{cw} \preceq v_{mw} \implies \bot$).
  - If $v_{mw}.a \neq v_{cw}.a$: If $C(v_{mw}) \cap C(v_{cw}) \neq \emptyset$, by Principle 3 (Intersecting Coverages), they must be atomic: $v_{mw} \simeq v_{cw}$. Contradiction ($\bot$). Otherwise, by Principle 3, $v_{cw} \preceq v_{mw}$. Contradiction ($v_{mw} \prec v_{cw} \wedge v_{cw} \preceq v_{mw} \implies \bot$).
- **Case: *NotFound* → *Invalid* → *NotFound*.** This transition requires an `announce` of a VRP update that provides only a COVERED attribute, followed by a corresponding `withdraw` of that same VRP update, all within a single validation run, which is impossible.

Conclusion. All transition paths leading to transient harmful states strictly violate the formulated topological and temporal

constraints. Therefore, the defined principles are strictly sufficient to guarantee absolute ROV consistency during pipelined execution.

D Replay Methodology

Data Source. Our methodology relies on the rich historical data archived by the `RPKIViews` project. This project deploys multiple Relying Parties (RPs) at vantage points across several continents, with each RP performing validation runs approximately every 30 minutes. For each run, `RPKIViews` archives two essential pieces of information: (1) the complete RP log file, which includes per-PP synchronization timings, and (2) a snapshot of all RPKI objects downloaded during that run.

Replay Process. Our replay methodology is designed to accurately reconstruct any historical validation run by precisely controlling an RP's inputs, namely, RPKI data content and synchronization latency. The general process, applicable to any RP software, proceeds as follows for a single historical run:

1. **Parse Historical Logs:** First, the historical log file from `RPKIViews` is parsed to extract the actual time the original RP took to synchronize with each PP.
2. **Simulate Synchronization Latency:** The RP's live network synchronization logic is bypassed. Instead, for each PP, the process sleeps for the duration extracted in the previous step, precisely simulating the network-induced latency of the original validation run.
3. **Inject Archived Data:** Immediately after the simulated synchronization latency for a given PP, the historical RPKI objects for that PP—retrieved from the corresponding `RPKIViews` archive—are injected directly into the RP's local cache.
4. **Execute Native Validation:** With its inputs (latency and data) controlled to mirror the historical state, the RP's own unmodified validation engine is then executed.

For this study, we implemented this replay methodology by creating customized versions of two mainstream RPs, Routinator and FORT Validator. Our customized implementations log the precise synchronization and validation timestamp for every RPKI object, as well as the overall start and end timestamps of the validation run.

E Router Overhead Analysis

To formally analyze the overhead of a pipelined model, we first define the key parameters of a validation run and the associated per-unit costs on a router:

- $c_{notify}, c_{vrp}, c_{rov}$: The per-unit CPU costs for processing a `Notify` PDU, a VRP update, and a single BGP route re-validation, respectively.
- N_v: The total number of VRP updates generated in the run.
- R_{total}: The set of all unique BGP routes affected by the N_v updates.
- n: The number of batches a pipeline uses to disseminate the updates.

In the standard **monolithic model** ($n = 1$), the router processes one notification and performs one set of re-validations on the $|R_{total}|$ affected routes. The total cost is:

$$Cost_{MU} = c_{notify} + N_v \cdot c_{vrp} + |R_{total}| \cdot c_{rov}$$

In a generic **pipelined model** ($n > 1$), overhead can increase from two sources. First, the router must process more notifications. Second, a single BGP route might be affected by VRP updates arriving in different batches, causing the router to perform redundant work. We define the number of these redundant re-validations as N_{rov_extra}. The total cost for a generic pipeline is therefore:

$$Cost_{NP/CAP} = n \cdot c_{notify} + N_v \cdot c_{vrp} + (|R_{total}| + N_{rov\_extra}) \cdot c_{rov}$$

By subtracting these two equations, we find the additional overhead (Δ_{cost}) has two components:

$$\Delta_{cost} = (n-1) \cdot c_{notify} + N_{rov\_extra} \cdot c_{rov}$$

This model reveals a critical insight for any pipelined design. The total overhead is driven by two distinct factors: the frequency of notifications and the potential for redundant re-validations (N_{rov_extra}).

F Proof of Conflict Graph Correctness

To prove that the conflict graph correctly groups all conflicting VRP updates, we must demonstrate that it is strictly impossible for any VRP update to be released before all its conflicting counterparts have been generated. We establish this safety guarantee using proof by contradiction.

Formal Axioms. Let p_v denote the IP prefix space of a VRP update v. Two updates v_1 and v_2 are in a conflict relationship if $p_{v_1} \cap p_{v_2} \neq \emptyset$. Recall that $f(v)$ is the cache update time and $T(v)$ is the pipeline timestamp when v is generated. Let $\mathcal{A}(t)$ be the set of active (unprocessed) RC nodes at time t, and P_C be the resource space of RC C.

Our framework enforces two core axioms based on RPKI semantics and graph logic:

A1. Hierarchical Allocation (RFC 3779): If C_{anc} is any ancestor RC of v, then $p_v \subseteq P_{C_{anc}}$.

A2. Predictive Locking Edge: An edge $e(C, v)$ exists at time t *if and only if* $C \in \mathcal{A}(t) \wedge P_C \cap p_v \neq \emptyset$. v can only be released at time $f(v)$ if it has no incoming edges from any active RC nodes.

Assumption. Assume the conflict graph fails to prevent a premature release. This implies there exist conflicting updates v_1, v_2 ($p_{v_1} \cap p_{v_2} \neq \emptyset$), such that v_1 is released before v_2 is even generated: $f(v_1) < T(v_2)$

Deriving the Contradiction. Consider the graph state at the release time $f(v_1)$. Because $f(v_1) < T(v_2)$, the object producing v_2 has not yet been processed, meaning its authorization

chain is still being traversed top-down. Thus, there must exist at least one ancestor RC of v_2, denoted C_{anc}, that remains active: $C_{anc} \in \mathcal{A}(f(v_1))$

By Axiom A1, the resource space of C_{anc} encompasses v_2: $p_{v_2} \subseteq P_{C_{anc}}$. Given the conflict $p_{v_1} \cap p_{v_2} \neq \emptyset$, it logically follows that C_{anc}'s space also overlaps with v_1: $p_{v_1} \cap P_{C_{anc}} \neq \emptyset$

By Axiom A2, since $C_{anc} \in \mathcal{A}(f(v_1))$ and $p_{v_1} \cap P_{C_{anc}} \neq \emptyset$, a predictive locking edge $e(C_{anc}, v_1)$ must exist at time $f(v_1)$.

However, Axiom A2 strictly prohibits the release of v_1 at $f(v_1)$ while it still has incoming RC edges. This yields a direct contradiction.

Conclusion. The assumption $f(v_1) < T(v_2)$ is false. The conflict graph structurally ensures that no VRP update is released until all its conflicting counterparts are generated and safely grouped within the same component.

G Synthetic Topology Generation

This appendix details our data-driven methodology for generating realistic RPKI topologies that mirror the structural properties of the global RPKI ecosystem.

G.1 Hierarchy Generation Process

The generator employs a multi-stage, stateful process to construct a certificate hierarchy based on real-world BGP and RPKI datasets.

1. **Route Selection:** To simulate a target *RPKI Deployment Rate* α, the generator first preserves the real-world ROA payloads and their corresponding RC issuance hierarchies. To reach the target rate α, it then randomly samples the required number of additional unprotected routes from a global BGP table snapshot (RRC00, Aug. 1, 2025).
2. **ROA Construction:** We generate ROAs for each selected route following two security-centric Best Current Practices (BCPs):
    - **Minimal-ROA (RFC 9319):** To mitigate forged-origin sub-prefix hijacks, the `maxLength` is set equal to the prefix length.
    - **Single-Prefix Objects (RFC 9455):** Each ROA contains exactly one prefix authorization. This granular mapping ensures that the revocation of one authorization does not inadvertently impact unrelated prefixes.
3. **RC-ASN Binding:** RCs are assigned to ROAs using a *reuse-or-create* policy. If an origin AS already possesses an associated RC, this RC is reused; otherwise, a new RC is instantiated and bound to that AS. This models the dominant real-world structure where 72.6% of ASes operate a single RC and 75% of RCs serve only one AS.
4. **Regional Delegation Modeling:** For each newly instantiated RC, the generator first maps the ASN to its specific country. It then checks for the existence of an NIR (e.g., JPNIC, CNNIC, or LACNIC-member NIRs like Brazil). If an NIR is present for that country, the new RC is attributed to this NIR. Otherwise, the attribution falls back to the corresponding RIR. Once the parent registry is identified, the generator emulates its specific structural model:
    - **Centralized (RIPE, LACNIC, APNIC):** New RCs are parented by a designated grandchild certificate of the RIR's Trust Anchor (TA) or the primary RC of an NIR.
    - **Direct (AFRINIC):** New RCs are directly parented by the primary child certificate of the RIR's TA.
    - **Structured (ARIN):** A two-level intermediate hierarchy is used, where each intermediate RC is capped at 500 grandchild certificates to model ARIN's specific scalability design. New RCs are parented by one of these intermediate RCs.

G.2 Repository Allocation Algorithm

To simulate the highly skewed load distribution of the RPKI data supply chain, we partition N fixed repositories into a two-tier architecture: **Tier-1** (5 RIR-managed PPs) and **Tier-2** ($N-5$ delegated PPs). The assignment follows a deterministic two-step process:

1. **Baseline Co-location:** Initially, every new RC is placed in the PP of its issuing registry (RIR or NIR). This reflects the empirical observation that over 98.8% of child RCs are co-located with their parents' publication points.
2. **Load Dispersal:** To populate the remaining Tier-2 PPs, the generator identifies empty delegated PPs and populates them by migrating clusters of 3 RCs from Tier-1 PPs. This cluster size matches the 75th percentile of RC counts observed in real-world delegated repositories, effectively modeling the "long tail" of small, independent publication points.