From the Consent of the Routed: Improving the Transparency of the RPKI

Full version from June 9, 2014.

Ethan Heilman Danny Cooper Leonid Reyzin Sharon Goldberg
Boston University, Boston, MA 02139 USA

ABSTRACT

The Resource Public Key Infrastructure (RPKI) is a new infrastructure that prevents some of the most devastating attacks on interdomain routing. However, the security benefits provided by the RPKI are accomplished via an architecture that empowers centralized authorities to *unilaterally* revoke any IP prefixes under their control. We propose mechanisms to improve the transparency of the RPKI, in order to mitigate the risk that it will be used for IP address takedowns. First, we present tools that detect and visualize changes to the RPKI that can potentially take down an IP prefix. We use our tools to identify errors and revocations in the production RPKI. Next, we propose modifications to the RPKI's architecture to (1) require any revocation of IP address space to receive *consent* from all impacted parties, and (2) detect when misbehaving authorities fail to obtain consent. We present a security analysis of our architecture, and estimate its overhead using data-driven analysis.

1. INTRODUCTION

The RPKI [42] is a new infrastructure for securing interdomain routing with BGP. BGP has traditionally operated as a "default-accept" architecture: any autonomous system (AS) can originate a BGP routing announcement (*i.e.,* claim to be the destination for) for any IP prefix, and other ASes will accept the BGP announcement by default. This has made BGP vulnerable to a number of routing attacks, the most common [20, 22, 45, 54, 55, 60] and devastating [10, 14, 27, 33] of which are prefix- and subprefix hijacks. In a prefix hijack, a hijacking AS originates BGP routes for IP prefixes that were not allocated to it, causing the traffic for those prefixes to be intercepted by the hijacker's AS.

The RPKI prevents these attacks by providing a trusted mapping from allocated IP prefixes to ASes authorized to originate them in BGP. To do this, the RPKI establishes a top-down hierarchy of *authorities*, rooted at the Regional Internet Registries (RIRs), that allocate and suballocate IP address space, as well as authorize its use in BGP by individual origin ASes; routers use the RPKI to distinguish between hijacked BGP routes and routes originated by a legitimate AS. The RPKI also turns out to be surprisingly effective against attacks it was not designed to prevent [27]; there is evidence [43] that more advanced secure routing solutions [34, 41] provide limited benefits over what is already provided by the RPKI. The RPKI requires neither changes to BGP nor online cryptographic computations during routing. It is currently being rolled out by RIRs and adopted by individual network operators, and authorizes about 20,000 BGP routes as of January 2014 (Section 2.1).

From default-accept to default-deny. However, the security benefits of the RPKI are accompanied by a drastic shift from BGP's traditional "default accept" policies, to a new "default deny" mode: to prevent (sub)prefix hijacks, routers should only accept routes authorized by the RPKI, and discard all other routes by default (Section 3). Meanwhile, the RPKI's hierarchical architecture empowers centralized authorities to *unilaterally* revoke authorization (or *take down*) IP prefixes under their control. This shift has lead to concerns [7, 17, 47, 48, 65] that the RPKI creates powerful authorities with the technical means for taking down IP prefixes, and could be exploited by abusive authorities or governments to settle disputes or block undesirable content. This is a stark departure from the status quo, where these authorities (RIRs, National/Local Internet Registries, *etc.*) had the power to allocate IP address space, but not to impact routing to space that has been allocated [28, 58].

Transparency. In light of the risk of takedowns, it would be useful to have mechanisms that can detect when RPKI authorities misbehave; this could create social (and possibly legal) pressure to motivate misbehaving authorities to fall in line. However, the architecture of the RPKI also makes it difficult to distinguish between revocations due to disputes or censorship, and those due to business arrangements that were agreed upon by all impacted parties (Section 3).

Our contributions. As RPKI deployment continues to gain traction, we present an investigation of the risk of RPKI takedowns, and propose technical solutions that mitigate this risk by improving the transparency of the RPKI. Our contributions are:

1. Security audit (Section 3). We untangle the often-unintuitive interactions between the RPKI and BGP, and explain why it is *not* always the case that a revocation in the RPKI can take down an IP prefix in BGP.

2. Tools, measurement & modeling (Section 4). We start by working within the current RPKI specifications, and

SIGCOMM'14, August 17–22, 2014, Chicago, IL, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2836-4/14/08 ...$15.00. http://dx.doi.org/10.1145/2619239.2626293.

build two tools that can increase its transparency by detecting and reacting to RPKI problems. Our *detector*, that identifies changes to the RPKI that can takedown IP prefixes, and a *visualizer*, that visualizes the results. We test our tools on the production RPKI and identify real-life errors and revocations (Sections 3,4.1). Since RPKI deployment is still in its infancy, we also develop models of a future full-deployment of the RPKI, based on routing data and RIR information (Section 5.7).

3. Changes to the specifications (Section 5). The current RPKI specifications still place power squarely in the hands of authorities. To remedy this, we propose modest modifications to the architecture of the RPKI. Our design ensures that once a route is authorized by the RPKI, it switches from "default-deny" to "default-accept"; that is, authorities that revoke IP address space must first obtain *consent* from all entities holding allocations (and suballocations) of that space. Our design also allow parties viewing RPKI information from different vantage points to detect when (1) their views are not *consistent*, and (2) authorities fail to properly obtain consent. We prove the security properties of our design. Because our design sometimes requires many entities to provide consent, we also estimate its extra overhead using data-driven analysis (Section 5.7).

2. RPKI PRIMER

We overview the RPKI's certificate hierarchy, explain how RPKI information determines route validity, and explain how the RPKI can limit threats to BGP.

2.1 The hierarchical structure of the RPKI.

The RPKI arranges authorities in a strict hierarchy that mirrors the IP address allocation hierarchy.[1] An authority may issue cryptographic objects for IP addresses that are *covered* by its own IP addresses. (An IP prefix P *covers* prefix π if π is a proper subset of the address space in P (*e.g.,* 63.160.0.0/12 covers 63.160.1.0/24) or if $P = \pi$. Also, prefix 63.160.0.0/12 has *length* 12.) Each authority has a *resource certificate (RC)*, a certificate that contains its cryptographic public key and its set of allocated IP addresses [44]. An authority may issue signed objects for IP addresses covered by its allocation, specifically: (1) an RC that suballocates a subset of its addresses to another authority, or (2) a *route origin authorization (ROA)*, that authorizes a specified AS to originate a set of prefixes, and its subprefixes up to a specified length called maxLength, in BGP [42]. ROAs protect BGP from routing attacks (Section 2.2).

Model (Figure 1). We show how an RIR (ARIN) uses its RC to suballocate a prefix to another authority (Sprint), which then issues RCs suballocating this prefix to other authorities (ETB S.A. ESP., Continental Broadband). (This is an excerpt of one of our models of the fully-deployed RPKI; see Section 5.7.) We say Sprint is the *parent* of Continental Broadband, and extend this to child, ancestor, *etc.*, in the obvious way. Sprint issues two ROAs that authorize a specified prefix and its subprefixes up to maxLength 24; the remaining ROAs shown authorize only a single prefix.

Status of the RPKI (Table 2). As of January 13, 2014, the production RPKI contains ROAs for about 20K

[1]The roots of the RPKI are the five RIRs (Table 2); in the future, IANA could be a single root [42, Section 2.4].

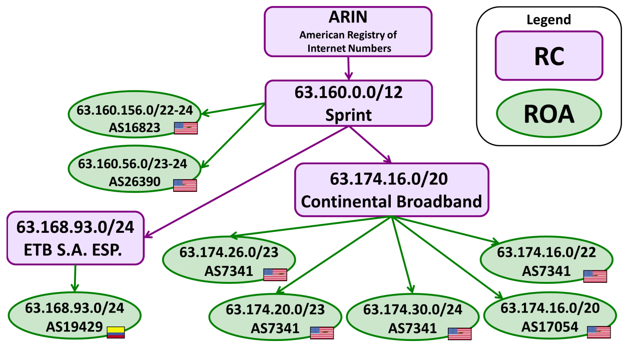


Figure 1: Excerpt of a model RPKI

prefix-to-origin-AS pairs. (About 488K prefixes were announced in BGP that day.) The RPKI's structure (Table 2) is slightly different than our models. At depth 0, there are trust anchors for each RIR (*e.g.,* ARIN). The trust anchors are long-lived certificates, who issue a handful of shorter-lived intermediate RCs to the RIRs (depth 1); our models and Figure 1 omit intermediate RCs. Intermediate RCs issue leaf RCs to organizations (*e.g.,* Sprint) who then issue ROAs. ARIN has an extra layer of intermediate RCs. RCs for suballocations like Continental Broadband and ETB in Figure 1 are absent; their ROAs could just be issued by Sprint. ROAs usually contain one AS and many prefixes (*e.g.,* all prefixes for AS 7341 could be issued in one ROA).

Publication points. The RPKI was designed to require minimal changes to BGP, and therefore operates entirely out-of-band. RPKI objects are stored in public repositories. Each RC has its own *publication point* (*i.e.,* directory in a file system) where it publishes every object it issued. Each RC also signs a *manifest* that logs the hash of every object present in its publication point. *Relying parties* download RPKI objects from publication points to their *local caches*, validate the objects, push information to their routers, and use it to inform routing decisions in BGP.

2.2 How the RPKI prevents threats to BGP.

BGP is especially vulnerable to subprefix hijacks because of longest-prefix-match routing: when a router learns BGP routes for a prefix and its subprefix, it always prefers the subprefix route. Subprefix hijackers exploit this by originating routes for subprefixes of a victim's prefix. This leads to a natural desideratum for the RPKI: a subprefix hijacker's route should be always be invalid when the legitimate route has a matching valid ROA.

Origin authentication. To achieve this desideratum, a relying party uses the RPKI for *origin authentication* as follows. For our purposes, a BGP *route* is an IP prefix π and an origin AS a. Once a relying party has "access to a local cache of the complete set of valid ROAs" [31, Sec. 2], these valid ROAs are used to classify each route (π, a) learned in BGP into one of three *route validation states* [31, 46]:

– *Valid:* There is a valid *matching ROA*. A matching ROA has (1) a matching origin AS a, and (2) a prefix P that covers prefix π, and (3) the specified maxLength no shorter than the length of π.
– *Unknown:* There is no valid *covering ROA*. A covering ROA is any ROA for a prefix that covers π.
– *Invalid:* The route is neither unknown or valid.

The rules above elegantly achieve the desideratum: if a le-

Depth	1	2	3	4
RIPE	4 RC	1909 RC	1512 ROA	
LACNIC	4 RC	282 RC	282 ROA	
ARIN	1 RC	1 RC	99 RC	151 ROA
APNIC from IANA	1 RC	450 RC	58 ROA	
AfriNIC	1 RC	27 RC	48 ROA	

Table 2: Valid ROAs and RCs at each depth of the production RPKI on January 13, 2014.

gitimate route in BGP has a matching valid ROA in the RPKI, a subprefix hijacker's route will always be "invalid" (because it is "covered" by the valid ROA). To stop subprefix hijacks, relying parties should *drop invalid routes*, *i.e.,* not select routes in BGP that are "invalid" per the RPKI.

3. AUDIT: THE RISK OF RPKI TAKEDOWNS

We explore RPKI threats that take down IP prefixes by causing legitimate BGP routes to be misclassified as "invalid".

A trace of the production RPKI. To aid our exposition, this section discusses real-life events that our tools detected in the production RPKI (described in Section 4). Events were observed in a trace we collected from 2013/10/23–2014/01/13. Each day, we used rcynic [2] to pull the state of the production RPKI to an empty local folder, and to cryptographically validate the result. The trace excludes a few days where our collector went down, and a few days where rcynic failed to properly sync to the repository (reporting `<rsync_transfer_failed kind="bad">`).

3.1 Tradeoff: BGP threats vs RPKI threats.

Now that we have understood how threats to BGP influenced the design of the RPKI, we look at how threats to the RPKI can impact routing with BGP. Specifically, to prevent subprefix hijacks on BGP, every route covered by ROA in the RPKI is classified as "invalid" *by default*, unless it has a matching valid ROA of its own. The RPKI is therefore a "default-deny" architecture,[2] and this comes with some risks; if a legitimate BGP route is wrongly classified as "invalid" by the RPKI, a relying party that *drops invalid routes* will lose connectivity to that route in BGP. As such, there have been concerns [7, 17, 47, 48, 65] that the RPKI creates a new technical means for IP-prefix takedowns.

Because moving to a default-deny regime is a drastic change for the routing infrastructure, relying parties are explicitly entitled to use their own "local policies" to decide what to do with "invalid" routes [31]. *Drop invalid routes* is the strictest local policy, and comes with the risk that threats to the RPKI can take down legitimate BGP routes. An more lenient policy suggested in RFC6483 [31] is to *depref "invalid" routes*: for a given prefix, a router should prefer "valid" routes over "invalid" routes. This policy implies that a router still selects an "invalid" route when there is no "valid" route for the *exact same* IP prefix. Thus, the router may still be able to reach routes that are wrongly classified as "invalid" as a result of problems with the RPKI. (However, availability of a route at one router can depend strongly on local policy used at other routers. For example, a router that uses the lenient *depref invalid* policy can lose connectivity to an "invalid" route if all its neighboring routers use the strict *drop invalid* policy.) On the other hand, this policy does *not* prevent subprefix hijacks [12, Section 5].

[2]The idea that the RPKI is a default-deny architecture is unintuitive, given the "unknown" route validity state. However, a BGP route without a matching valid ROA is "unknown" only when there is *no covering ROA*; see Section 2.2.

policy	routing attack	RPKI manipulation
drop invalid	stops (sub)prefix hijacks	prefix goes offline
depref invalid	subprefix hijacks possible	prefix may stay online

Table 3: Impact of different local policies.

Table 3 summarizes this difficult tradeoff; the local policy that is best at protecting against attacks on BGP is worst at protecting against problems with RPKI. While it is too early to tell what local policies will be adopted in the long run, better assurances about the trustworthiness of RPKI information are needed before relying parties can start enjoying the RPKI's full potential for protecting BGP from attack. Sections 4-5 focus on improving these assurances.

A note on granularity. While ROAs can have arbitrary prefix lengths, the longest IPv4 prefix length that is universally accepted by BGP routers is a /24. (Routers usually ignore longer prefixes to avoid bloating their routing tables.) Thus, our discussion on the RPKI's impact on IP prefix reachability should be thought of as having the granularity of a /24 (or shorter) IPv4 prefix, *i.e.,* no fewer than 256 IPv4 addresses; the RPKI, therefore, can be used for significantly coarser level of blocking than *e.g.,* the DNS [56, 57].

3.2 New threats created by the RPKI.

What threats to the RPKI can cause a legitimate BGP route to be classified as "invalid"? We consider two threat models: threats created by RPKI authorities themselves, and threats to the communication path between relying parties and the public RPKI repositories.

3.2.1 Threats created by RPKI authorities.

Because relying parties download RPKI objects from publication points that are *controlled by their issuer* [42, Section 8], the issuer can manipulate the contents of its publication points in any way it likes. [3] There are two ways an issuer can cause a legitimate BGP route to be classified as "invalid":

1. Adding a ROA. If a new ROA added to the RPKI covers a legitimate BGP route without its own matching ROA, the legitimate BGP route will be wrongly classified as "invalid". Consider, for example, this incident:

■ **Case Study 1: ROA misconfiguration.** On December 13, a new ROA was added to the production RPKI rooted at ARIN, authorizing prefix 173.251.0.0/17 with maxlength 24 to AS 6128. This caused a large portion of the address space to downgrade from "unknown" to "invalid", including several legitimate /24 routes announced in BGP that did not have matching ROAs (AS 53725 originating 173.251.91.0/24, AS 13599 originating 173.251.54.0/24).

2. Whacking a ROA. Alternatively, missing information can cause the RPKI to misclassify a legitimate BGP route as "invalid". To see how, we first distinguish between route validity and RPKI object validity. An RPKI object (*i.e.,* an RC, ROA, *etc.*) is valid if it has a valid signature, is not malformed or corrupted, *etc.* Meanwhile, the validity of a route depends *exclusively* on the set of valid ROAs in a

[3]This is in stark contrast to a traditional PKI, where the subject of the certificate delivers it to the verifier [37, p. 40]; a website sends its web certificate to a client in an SSL/TLS handshake. BGP lacks a handshake phase, and because the RPKI was designed to require minimal changes to BGP, it requires RPKI objects to be fetched out of band.

relying party's local cache. Thus, any action that prevents the delivery of a ROA object to a relying party has exactly the same effect — the corresponding BGP route becomes (a) "invalid" if the local cache contains some other valid covering ROA, or (b) "unknown" if there is no valid covering ROA. We therefore just say that a ROA is *whacked* whenever a relying party fails to receive the valid ROA object. There are many ways an authority can whack a descendant ROA:[4]

a. Revoking. An issuer can always revoke any object it issues using its certificate revocation list (CRL). A CRL is a list, signed by an RPKI authority A, of the objects issued by A that A has revoked [19].

b. Deleting or corrupting. An authority can also delete or corrupt any object in its publication point. If the authority logs appropriate change in its manifest, relying parties will accept the change without complaint. For example:

■ **Case Study 2: Deleted ROA.** On December 19, 2013, a ROA for (79.139.96.0/24, AS 51813), for a network in Russia, was deleted from the production RPKI. Meanwhile, since at least November 21, the RPKI also had a covering ROA mapping 79.139.96.0/19-20 to another Russian ISP, AS 43782. (This is a covering ROA because the /19 prefix covers the /24 prefix.) The covering ROA caused the route corresponding to the whacked ROA to downgrade from valid to invalid (per Section 2.2). Both the whacked ROA and covering ROA were issued by the same RC.

c. Overwriting. Each RPKI object is identified by a uniform resource identifier (URI), and an authority may overwrite any RC it issued, so that modified objects can have persistent URIs (thus simplifying operations like certificate renewal and key rollover [32]). An authority can thus overwrite an RPKI object with one for a different set of IP addresses (or a different key, *etc.*); children of the overwritten RC can be whacked as a result, because they are no longer covered by the RC (or signed by the wrong key, *etc.*):

■ **Case Study 3: Overwritten parent RC.** On January 5, 2014, a ROA for (196.6.174.0/23, AS 37688) for a backbone connectivity network in Nigeria was whacked because its parent RC was overwritten. The incident occurred as follows: On January 4, the ROA's parent RC was allocated prefix 196.6.174.0/23. On January 5, the RC was overwritten with an RC with the same key but for an IPv6 prefix 2c0f:f668::/32. The ROA in question (which remained in the publication point) became invalid, because it was no longer covered by its parent. Interestingly, the RC had no valid descendants until January 6, when it issued ROAs covered by the IPv6 prefix to a different AS (AS 37600, in Mauritius).

Moreover, if an RC gets whacked, all its descendent objects also get whacked. Our earlier work [18] showed how an authority can whack distant descendant ROAs in a targeted manner that causes no collateral damage to other ROAs.

3.2.2 Threats that disrupt RPKI object delivery.

A third party, who disrupts the communication path from an RPKI repository to a relying party, can whack a ROA just by corrupting a single bit in the ROA. (This is because the ROA would fail cryptographic validation.) Fortunately, however, the RPKI has a mechanism for detecting lost/corrupted information:

[4]To be sure that the BGP route corresponding to the whacked ROA becomes "invalid", an authority can just issue a covering ROA, as in Case Study 2.

Manifests. To provide assurance that no objects have been deleted from a publication point, a collision-resistant hash of the contents of every file issued by an RC is listed in a single file called a *manifest* [29, Section 2.1], which is digitally signed by the RC and stored at its publication point. The manifest must be updated whenever an RC issues, modifies, or revokes an object it issued. To prevent replay attacks and the propagation of stale information, manifests are short-lived; they usually expire and are renewed daily.

Of course, a third party can always corrupt or disrupt the delivery of the manifest itself. Consider the following:

■ **Case Study 4: Stale LACNIC manifests.** On December 20, 2013, an error at LACNIC whacked 4217 prefix-to-origin-AS pairs. The day before the incident, the RPKI rooted at LACNIC looked very similar to what is shown in Table 2. On December 20, the manifests and CRLs issued by all four of LACNIC's intermediate RCs (at depth 2) in our local cache all expired. The relying party software [2] raised an alarm and rejected all four of the intermediate RCs as invalid. All objects in subtree rooted at LACNIC became invalid; thus, there were no valid ROAs for any address space allocated by LACNIC, and BGP routes corresponding to LACNIC ROAs downgraded from "valid" to "unknown".

3.3 Transparency is (sometimes) difficult.

To mitigate risks created by the RPKI, a relying party should be able to detect when a problem with the RPKI has caused BGP route to be wrongly classified as "invalid". Fortunately, third-party attacks that disrupt object delivery are completely transparent – a relying party can always check that it received all the objects in a manifest, and that the manifest was valid and current. If not, relying parties should raise a *missing-information alarm*; RFC6486 [9, Sect 6.5] states that relying parties should react to such alarms at their own discretion.[5] Meanwhile, incidents where misconfigured or compromised RPKI authorities wrongly add or whack ROA are much less transparent; if the authority ensures that its manifest is consistent with the objects in its publication point, relying parties will not raise alarms.

Alarms. While the RPKI specifications do not focus on alarms, we think that they are crucial. While alarms alone cannot resolve problems, they do indicate that problems are present, and can trigger mitigation mechanisms. During a missing-information alarm, a relying party might try to reconnect the repository, search for missing objects in its local cache or at other relying parties, or search for unusual routing activity in the IP address space covered in the RC whose manifest triggered the alarm. Today, BGP misconfigurations are resolved by humans picking up the phone; RPKI alarms similarly provide guidance on when picking up the phone may be necessary. Indeed, in Section 5, alarms will be our primary tool for increasing transparency.

There is another important threat to transparency:

Mirror world attacks. In a mirror world attack, an (adversarial) RPKI authority presents one view of the RPKI to some relying parties, and a different view to others. (For example, the subject of an RC is shown a view containing the RC, but other relying parties are shown a view that omits the RC.) The current RPKI standard contains no mecha-

[5]The relying party software we used [2] in Case Study 4 rejected the stale manifest. Other policies are possible.

nisms to prevent this attack. Our proposal in Section 5 builds in protections against mirror-world attacks.

4. TOOLS TO IMPROVE TRANSPARENCY

We now develop tools that can improve the trustworthiness of RPKI data, without requiring any modifications to the RPKI itself. Our *detector* tool detects when a change to the RPKI causes a route to *downgrade* from "valid" to "invalid/unknown", (or from "unknown" to "invalid"), and our *visualizer* visualizes these downgrades. We released the source for all our tools,[6] so they can be incorporated into professionally-maintained systems like [38, 50, 61].

4.1 A tool for detecting downgrades.

One way to detect inconsistencies between BGP and the RPKI is to take a BGP feed [3, 5], run the RPKI validation algorithm in Section 2.2 on each BGP route, and identify BGP routes that the RPKI classifies as "invalid"; indeed, this is done in [38, 50, 61]. Our goal, however, is to detect when a *change* to the RPKI (*i.e.,* an added or whacked ROA) causes a change in the validity status of a route; it can therefore act as alert system for potentially-harmful changes to the RPKI. Moreover, we do not want our tool to be limited to a view of BGP provided by a specific BGP feed; instead, our detector considers the space of *all* possible routes (π, a), where π is a prefix and a is an origin AS, regardless of whether or not a particular route is visible to a particular BGP collector.

Challenges. The relationship between a single ROA and the validity of BGP routes is complex and dependant on the presence of other ROAs in the system. For example, a ROA giving an AS a prefix π of length 17 up to maxlength 22 actually makes $2^{(23-17)} - 1 = 63$ possible prefixes originated by AS a in BGP become "valid". If the ROA gets whacked, BGP routes for these prefixes do not necessarily become "invalid": some may become "unknown" (if there is no covering ROA), while others may remain valid (if there is another ROA for AS a and a super- or subprefix of π). Moreover, when such a ROA appears, it may downgrade "unknown" routes to "invalid" for any subprefix of π of any length, depending on the other ROAs already in the RPKI.

Data structures. Naturally, representing the space of all possible routes (π, a) requires an efficient data structure. Ours is based on the following observations. Consider the complete binary tree of all IP prefixes, with one IP address per leaf, and all possible prefixes as internal nodes. A ROA for prefix π, maxLength m, and origin AS a makes a subtree of this tree "valid" for AS a only; this subtree is rooted at π and goes down to depth m. This ROA also makes "known" (*i.e.,* the complement of "unknown"), or other ASes, the subtree rooted at π and going down to the bottom

We call these subtrees *triangles* and we represent them using *interval trees.* A triangle has one interval per prefix length. The granularity of each interval depends on the prefix length; intervals at length i have endpoints that are integer multiples of 2^{32-i}. For each prefix length, the total number of intervals kept is bounded by the number of prefixes in ROAs in the RPKI. We can use interval trees to efficiently perform unions, intersections, and complements of sets of triangles. This gives us a "prefix-validity" data structure that stores the validity states for *all* possible routes,

[6]`https://github.com/BUSEC/RPKI_Downgrade_Detector`

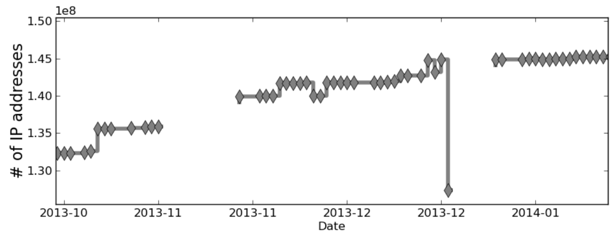


Figure 4: # of invalid IP addresses over time.

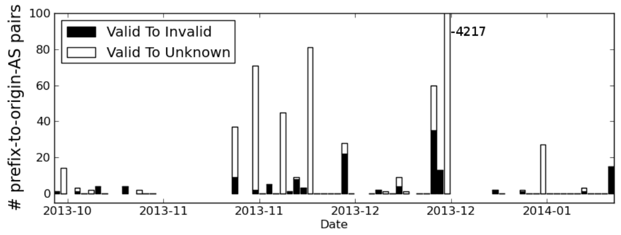


Figure 5: Downgrades due to whacked ROAs.

based on the ROAs in the RPKI; it can be created in time $O(n \log n)$ given ROAs for n (prefix, AS, maxlength)-tuples.

Our tool uses this data structure to compare a current RPKI state S_{cur} and a previous state S_{prev}. It processes each state once and creates its prefix-validity data structure. It then evaluates the impact of every single change from S_{prev} to S_{cur} in the context of other changes that occurred, by going through each (prefix, AS, maxlength)-tuple that appears or disappears from the RPKI (in the transition fromS_{prev} to S_{cur}), by extracting relevant information from the prefix-validity data structures.

Utilities for RPKI browsing. Our detector identified all the events in our case studies, and we analyzed each using custom-built python utilities (available upon request) that parse RPKI objects and search RPKI trees based on URI, public key, prefix, and AS number fields.

We also used our detector to check for downgrades between consecutive entries in our trace of the production RPKI:

Downgrades due to added ROAs (Figure 4). Two things happen whenever a ROA is added to the RPKI: (1) routes for the AS authorized in the ROA become "valid", and (2) the IP address space covered by the ROA transitions to the default-deny state, *i.e.,* all the "unknown" address space it covers downgrades to "invalid", as in Case Study 1. Figure 4 shows the latter transition over the course of our trace. We show the number of IP addresses that are "invalid" for at least one AS over time, marking each entry in our trace with a diamond. The drop in the number of "invalid" addresses on December 20 was due to the the stale LACNIC manifests described in Case Study 4.

Downgrades due to whacked ROAs (Figure 5). We show the number of (π, a) pairs that downgraded from "valid" to "invalid", and from "valid" to "unknown", for each consecutive entry in our trace that contained a successful pull of the RPKI. We show a value of zero if no downgrades occurred at some date, and a gap if our trace was missing an entry. To put these results in perspective, in January 2014, there were about 20,000 prefix-origin AS pairs authorized by ROAs in the RPKI. Most of the incidents in Figure 5 correspond to the whacking of a single ROA containing multiple prefixes; in many incidents, a new ROA appeared authorizing the prefix(es) in the whacked ROA to some other AS.

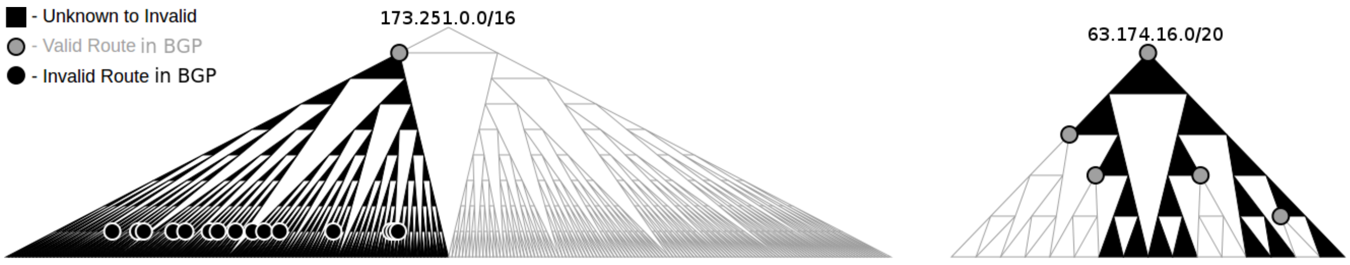


Figure 6: (l) Visualization of downgrades in Case Study 1. (r) Downgrades when the ROA (63.174.16.0/20, AS 17054) is added to the RPKI in Figure 1.

Case Study 4 explains the dramatic event on December 20.

4.2 A tool for visualizing downgrades.

We present a new visualization of the effects of a single ROA on the validity states of multiple BGP routes, based on a modification of the Sierpiński triangle [63]. Our visualization takes in information from our detector and from a specific BGP feed [3, 5], and presents a binary tree that visualizes the triangles described above, as well as the validity state of specific routes announced in BGP.

Figure 6(l) We visualize Case Study 1. The tool outputs the prefix tree rooted at 173.251.0.0/16, and shows how the triangle rooted at 173.251.0.0/17 transitions from unknown to invalid. The valid route for prefix 173.251.0.0/17 is shown with a grey circle. The added ROA also caused a number of /24 routes in the BGP feed to transition to the "invalid" state; we show these with black circles.

Figure 6(r). Consider the prefixes covered by Continental Broadband in Figure 1, and suppose S_{prev} has the all ROAs issued by Continental Broadband *except* the covering ROA for 63.174.16.0/20, and that the covering ROA is added in S_{cur}. We visualize downgrades from "unknown" to "invalid" in the transition from S_{prev} to S_{cur}. Routes uncovered by the four ROAs in S_{prev} are impacted during the transition; routes covered by the four ROAs in S_{prev} were *already* "invalid" in S_{prev}, and do not appear as downgrades.

5. CHANGES TO RPKI SPECIFICATIONS

While our detector can alarm when changes to the RPKI cause routes in BGP to downgrade to "invalid", the RPKI still lacks mechanisms that can detect when such changes are legitimate, or when they result from abuse. Because the purpose of the RPKI is to secure routing, not to provide a new technical means for RPKI authorities to takedown IP address space, we now propose modifications to the RPKI specifications to restore the balance of power between RPKI authorities and the subjects of the RCs they issue. We start with our design goals (Section 5.1), overview our design (Section 5.2), and then discuss its details (Sections 5.3-5.4). We then justify our design by presenting its exact security guarantees (Section 5.5), explain why alternative designs would not suffice to provide these guarantees (Section 5.6), and estimate its overhead via data-driven analysis (Section 5.7).

5.1 Design goals.

The following goals have guided our design.

Transparency through alarms. Our overarching goal is to detect RPKI abuse. We want *provable* guarantees that relying parties will raise alarms when something goes wrong; as discussed in Section 3.3, alarms act as a trigger for relying parties to investigate and react to problems with the RPKI. As in the current RPKI specifications, we suppose that relying parties use their own local policies to decide how to resolve alarms [9, Section 6]; alarms could indicate that certain IP prefixes should be monitored for routing anomalies, or that a relying party should revert to an older ("stale") set of RPKI objects that did not raise alarms.

The problem of how relying parties should propagate and react to alarms is outside the scope of this paper. First, technical means alone will not able to resolve a business dispute that results a unilateral revocation of IP address space in the RPKI; was the issuer of the space at fault, or was it the holder of the address space? Moreover, even requiring different relying parties to agree on a consistent (let alone, correct) course of action to resolve an alarm requires solving the famously-difficult Byzantine agreement problem [39]. We therefore assume that alarms will be resolved by out-of-band mechanisms (phone calls between network operators, social or even legal pressure), just like BGP misconfigurations are resolved today. Nevertheless, to discourage relying parties from raising false alarms, and to facilitate dispute resolution, our secondary goal is *accountability*; allowing relying parties to *provably* demonstrate, in as many situations as possible, (a) why they raised an alarm and (b) who is to blame.

Consent from subjects of RCs. Traditionally, routing has lacked a technical means for revoking IP address space; the RPKI changes this by allowing authorities to *unilaterally* revoke IP address space issued to their descendants. Our goal is to correct this power imbalance, so we require subjects of RCs to *consent* to RPKI modifications that can affect their address space. If consent is refused, this indicates that there is a dispute between the subject of the RC and its issuer, and relying parties should raise an alarm. During emergencies (disputes, lost/stolen keys) when it is impossible to obtain consent from the subject of the RC, the issuer can just unilaterally revoke the RC; these actions will be visible to relying parties, who will raise alarms and investigate the situation out-of-band. Theorem 5.1 formally states how we achieve this goal, and Section 5.7 analyzes the overhead of our consent mechanism.

Consistency for relying parties. We want different relying parties to see consistent views of the RPKI. Our goal is to prevent mirror-world attacks (Section 3.3) by allowing relying parties to compare their local views of the RPKI, and to raise an alarm whenever their views are inconsistent. Theorem 5.2 and 5.3 formally state this goal.

Practicality. Today, the RPKI specifications are still evolving [25, 35], since only a few authorities operate repositories (the five RIRs), and relying parties validate objects using a few evolving software packages like [2]. If our design is to be incorporated in the RPKI specifications, we need to achieve our goals with minimal changes to the current RPKI specifications. Thus, we cannot require synchronization between RPKI repositories or relying parties, and must retain most of the structure of the RPKI. Section 5.7 is a data-driven analysis of the practicality of our design.

5.2 Design overview.

To achieve our goals, we propose adding the following new elements to the RPKI. Some seem intuitively necessary; others may be less obvious, but we justify them in Section 5.6.

Consent through `.dead` objects (Section 5.3.1). Any action that *removes resources* (IP prefixes) or *invalidates* an RC requires the *consent* of the RC and all its impacted descendant RCs. To indicate its consent to having some/all of its resources taken away, an RC will sign a `.dead` object. The authority can then revoke (or remove resources from) the child by publishing the `.dead` object in its publication point and deleting (or overwriting) the child RC.[7] To simplify the exposition, we will not discuss consent from ROAs; instead we just suppose than any ROA that wishes to be entitled to consent is issued its own covering RC.[8]

Consistency through manifests (Section 5.3.2). In a default-deny architecture like the RPKI, a relying party must know that it has *all* the objects issued by an authority; otherwise, it might misclassify a legitimate BGP route as invalid because its ROA was missing. Fortunately, the RPKI's manifests contains the hash of every object the authority issues, and thus provide *positive attestation* to the authority's issued objects [9]. We therefore make manifests central to our design. Relying parties use them to check the consistency of their local caches, because if the manifests are equal, then (by the collision-resistance of the cryptographic hash) so are all the objects the authority issued. Because we cannot require relying parties to synchronize their interactions with the RPKI, we need to make sure they can still check consistency even if they are out of sync. To do this, we enable relying parties to be able to reconstruct all intermediate state of each publication point; to do this, we (1) maintain some extra state at each publication point, and (2) hash-chain manifests, so that each manifest contains the hash of the previous manifest.

Strict checks to raise alarms (Section 5.4). We design validation procedures that, in addition to validating RPKI objects, is also responsible for raising alarms. The procedure has two parts: a local consistency check (done by a single relying party to check consent) and a global consistency check (done by a pair of relying parties to check consistency). In a departure from current RPKI practice, where manifests play a secondary role, our local consistency check treats the manifest as the main cryptographic object, and validates one publication point and one manifest update at a time. Meanwhile, our global consistency check has relying parties compare hashes of the manifests in their local caches.

5.3 Procedures for RPKI authorities.

We now detail our new `.dead` objects, and our modifications to the RPKI's specification of manifests [9]. We also require a key rollover mechanism [32] for refreshing cryptographic keys; we present this mechanism in Appendix A.

Timing. In the procedures below, RPKI authorities assume that relying parties sync to every publication point within time t_s of the previous sync. In Appendix C, we discuss how the t_s effects the speed at which authorities can update their publication points.

5.3.1 Consent via `.dead` objects

Suppose an RC A wants to revoke or modify a valid child RC B. (RC B is "modified" if it is overwritten by a new RC B' at the same URI.) A needs the *consent* of B and all the RCs that descend from B that are whacked by the revocation/modification. Consent comes in the form of a `.dead` object, signed each consenting RC.

Revoking B requires a `.dead` object from all the descendants of B. Modifications that remove IP address space from B need `.dead` objects from each descendant of B that overlap with the removed space. No `.dead` objects are required when a modification has no impact on descendants of B (*e.g.,* modifying the parent pointer of B, or increasing the set of IP prefixes or AS numbers it certifies). All other modifications to B must be accomplished by issuing a fresh RC at a different URI. We estimate the number of entities that must be involved in issuing `.dead` objects in Section 5.7. We show how authorities can handle bulk modifications to the RPKI in Appendix C.

Constructing a `.dead`. `.dead` objects are constructed recursively as follows: Let D be a descendant of B that needs to consent. Before D can sign its own `.dead` object, D first collects `.dead` objects from each of its own descendants. D then signs its own `.dead` object, that includes (1) the hash of the `.dead` object issued by each *child* of D, (2) the hash of m_D, the manifest issued by D at the time D signs its own `.dead` object, and (3) the hash of the RC of D. D then provides its own `.dead` object and the `.dead` objects for all its descendants to its issuer C. At this point, D should no longer issue new objects or update its manifest. At the end of this process, A has received the `.dead` objects from B and all relevant descendants of B. (This recursive collection of `.dead` objects protects A, and its descendants, from being falsely accused of revoking a descendant without consent.)

RC A then simultaneously (a) deletes RC B, (b) puts all the `.dead` objects in the publication point of A, and (c) logs all the `.dead` objects in the updated manifest of A.

Each `.dead` object has the file extension `.dead.` and the filename as B, with the addition of disambiguating suffix indicating the serial number of B and the name of consenting

[7] Per Section 3.2, adding a ROA can also cause routes to downgrade from "unknown" to "invalid". However, we do not handle this because there is no way to obtain consent from entities originating "unknown" routes, since they do not participate in the RPKI. (If they did, they would already have ROAs, and their routes would not be "unknown".)Also, we expect very few routes to be classified as "unknown" once the RPKI reaches sufficient levels of deployment, especially since "unknown" routes can easily be eliminated if each RIR issues a ROA for AS 0 and all of its IP address space [46, Section 2.1].

[8] Actually, each ROA's parent issues an X.509 end-entity (EE) certificate for an ephemeral one-time-use key, which is used to sign the ROA message [42, Section 2.3]. The ROA and EE cert are stored in a single `.roa` file, so we have treated them as one object. But a ROA could instead consent via its EE cert, instead of asking for its own RC.

descendent RC. Future updates to the manifest of A need not log B or any `.dead` objects associated with B.

Make before break. Consent should be given and obtained in a "make-before-break" fashion. For B, this means (a) consenting to the modification only *after* it knows that a validly-issued replacement object (with a different URI) exists in the RPKI or (b) knowingly consenting to have its removed resources disappear completely from the RPKI. For A, it means obtaining the needed `.dead` objects ahead of time, so they can be published at the same time that A revokes/modifies B.

Because there is no guarantee that different directories are synchronized by relying parties simultaneously, consent needs to be obtained from any descendant object that might still be valid in the local caches of relying parties, *even* if the object has already been revoked. Thus, if B recently revoked C, and A is trying to revoke B, then consent from C is necessary unless time t_s has elapsed since C was revoked.

Key rollover. Any PKI needs a mechanism for refreshing cryptographic keys. Adapting the RPKI's key rollover mechanism to our design, while still preserving transparency and consistency, requires some care. Details are in Appendix A, where we discuss how an RC B signs a special "post-rollover" manifest and `.roll` object to signal that it has rolled over to a new RC B'; RC B' has a new cryptographic key, but has the same parent as B, and issues the same objects in the same publication point as B. Importantly, key rollover only requires consent from the RC whose key was rolled.

Inherit. The RPKI currently allows RC to be issued with their resources set to "inherit"; this means that the RC has the same set of resources (IP prefixes, AS numbers) as its issuer [30, Section 2]. This attribute is especially useful when an authority has a long-lived signing key, and uses this key to issue to issue a shorter-lived RC for itself, which is then used to issue ROAs and RC to other organizations; the long-lived key can therefore be stored securely and used infrequently. (Some intermediate RCs at depth 1 in the production RPKI have the "inherit" attribute (Table 2).) In our design, the inherit attribute allows a parent RC to *modify* its own resources without requiring consent from its child; the "inherit" attribute indicates the child's consent to have same set of resources as its parent.

5.3.2 Manifests as positive attestations

We modify manifests [9] by (a) adding information, and (b) changing the semantics by which they are interpreted.

Normative manifests. We make manifests normative: any objects not logged in the issuer's current manifest are treated as nonexistent by relying parties. Once manifests become normative, we can simplify other aspects of the RPKI:

1. Only manifests may expire. In the current RPKI, manifest are "updated" when their issuer overwrites them with a fresh manifest with a higher "manifestNumber" [9]; manifests are therefore short-lived objects — most expire and updated with fresh versions within 24 hours. In our design, if a manifest expires before it is updated, then all objects logged in the manifest become "stale", rather than "invalid"; "stale" indicates that up-to-date information is unavailable, but does *not* indicate that the objects logged in the expired manifest have explicitly been revoked. A stale manifest (or any object logged in a current manifest but not obtained by a relying party) raises a *missing-information alarm* at the relying party (Section 5.4). Thus, we no longer require expiration dates on ROAs and RCs: any RC/ROA not logged in the current manifest is automatically invalid. This also prevents an authority from issuing short-lived ROAs/RCs so it can circumvent the need to obtain consent.

2. Manifests must log only valid objects. Any issuer that logs an invalid object in its manifest (*e.g.,* an object pointing to the wrong parent, has a prefix that not covered by the issuer's RC, *etc.*) risks the ire of relying parties, who raise alarms per Section 5.4. (This is necessary for our consistency mechanisms to work properly; see Counterexample 2.) Issuers must also ensure that race conditions do not cause their manifests to log invalid objects; procedures that can ensure this are in Appendix C.)

By ensuring that manifests only attest to what is valid, we no longer need CRLs to attest to what is invalid. In fact, RCs/ROAs need not even be signed: signature on a manifest suffices, because the manifest contains the collision-resistant hash of every valid object (as in [26]).[9]

These changes may seem unorthodox, but we note that the RPKI is different from the usual PKI where relying parties obtain and validate objects one-by-one; instead, all objects in a publication point are downloaded *en masse*, so it suffices just to validate the manifest. All these changes can be implemented without any modification to the RPKI object formats by having relying parties ignore CRLs, expiration times/signatures on RCs and ROAs.

Hash chaining. Manifests are hash-chained: each new manifest includes the hash of the contents (excluding the signature) of the manifest it supersedes. A *horizontal chain* is a sequence of consecutively issued manifests, each superseding the next. For each manifest m, hash chaining defines successor and predecessor manifests of manifest m in the obvious way. We also require every manifest to include the hash of the manifest containing its issuer's RC ("parent manifest"), so that we can definitively determine the resources allocated to an RC at the time it signed its manifest. Thus, a *vertical chain* is a sequence of manifests, each containing the hash of the manifest above it and the hash of an RC that signed the manifest below it.

Reconstructing intermediate states. To prevent mirror world and other attacks (*e.g.,* Counterexample 1 in Section 5.6), relying parties must be able to reconstruct states that could have been seen by other relying parties. Thus, issuers must also provide relying parties with "hints" to enable reconstruction, between two syncs to a publication point, of every intermediate manifest along the horizontal hash chain along with its corresponding publication point state. The following information is maintained at each publication point:

1. First appearance. Manifests are numbered sequentially [9]. For each file whose URI and hash is logged in the

[9]An RPKI object should be considered invalid if its issuer's RC has a child pointer that does not point to the publication point holding the object; this is because a relying party could not find the object by traversing the hierarchy of RPKI publication points from the root downwards. This seems to be an omission from the RPKI validation procedure in the RFCs, which should be addressed regardless of our specific design. (This requirement is absent from the web PKI, since a relying party receives the entire certificate chain from during the SSL hankshake.)

manifest, the manifest logs the number of the first manifest in which this version of the file appeared.

2. Hints for disappearance. An issuer also provides (an unsigned, not in the manifest) "hints" file. For each object that was overwritten or deleted within time t_s, the "hints" file contains the URI of the object, the hash of the contents of the object, and the numbers of first and last manifest in which this version of the object appeared. Every object must be preserved in its publication point for time at least t_s, even if it is logged in a manifest for time less than t_s; this ensures that every relying party is able to obtain the short-lived object and reconstruct the appropriate publication point state. If a short-lived object (that was logged in a manifest for time less than t_s) is overwritten, and old version of the object should be given an appropriate suffix and preserved for time t_s after the object was first created.

When a certificate is revoked, its publication point, including the manifest, should be preserved for time t_s.

Preventing replays. RCs issued by an authority must have unique increasing serial numbers. Each manifest must log the highest serial number used; relying parties should check that this value is correct, and that new RCs have serial numbers that are higher than this value. Thus, an authority cannot put an old revoked/modified RC back in its publication point, and later reuse an old `.dead` object to revoke/modify it.

One manifest per publication point. To simplify the design, each publication point should have only a single manifest (except in the special case of key rollover, per Appendix A.). Also, an authority must publish its manifest before its issuer initially publishes its RC; this prevents relying parties from raising missing information alarms when an RC first comes online; see also Appendix C.

5.4 Validation procedures for relying parties.

The validation procedures for relying parties have three purposes: (1) to determine the set of valid RPKI objects, (2) to raise alarms when something goes wrong, and (3) whenever possible, to hold authorities accountable for causing alarms by proving that they violated procedures in Section 5.3. Since manifests are central to these procedures, we say that a manifest has been *obtained* if the relying party can (a) download a manifest that is validly signed by its issuer, OR (b) reconstruct the manifest as element in the horizontal hash chain that terminates in a manifest that is validly signed by its issuer, per Section 5.3.2.

Initial sync. The initial connection to the repository is as in the current RPKI: objects are downloaded and validated. If a relying party cannot obtain the valid current manifest of an RC R, or any object logged in the manifest, it raises a *missing information alarm.* Each RC is validated according to the "New RC Procedure" in Appendix B.2.4.

Local consistency check. In the current design of the RPKI, updates to a relying party's local cache are performed all at once. We, instead, require incremental processing of updates: a relying party updates its local cache one publication point and one consecutive manifest along the horizontal hash chain at a time. Updates are only performed for manifests issued by valid RCs. Incremental updates allow us to avoid race conditions that can occur when authorities update their manifests in parallel; see Counterexample 2. New RCs are an exception to this rule: the entire subtree rooted at a new RC should be downloaded and validated immediately to allow new RCs to quickly issue new objects.

alarm	description
missing-information	manifest is stale/missing OR object logged in manifest is missing
bad key rollover	RC issued "post-rollover manifest" but performed incorrect key rollover procedure
invalid syntax	RC issued malformed object
child too broad	RC issued object it does not cover
unilateral revocation	RC deleted/modified object without `.dead`
global inconsistency	manifest failed global consistency check

Table 7: Alarms

Relying parties can sync to publication points in any order and parallelize updates of publication points that are not in an ancestor-descendant relationship. (See Appendix C for more details.) When a relying party Alice obtains an updated state of a publication point, she reconstructs all the intermediate states of the publication point along the horizontal hash-chain. For each publication point, Alice must ensure that her local cache is no more than t_s behind the current state of the publication point. Otherwise, she may raise false alarms when authorities don't misbehave. Alice then compares pairs of consecutive states to check that (a) all new or modified objects are valid, and (b) all deletions and overwrites received proper consent. If not, she issues one of the middle alarms in Table 7. In some cases, the alarm is *accountable*, which means that Alice can specifically identify the RPKI authority that misbehaved, and can provide objects that will convince others of the authority's misbehavior (Section 5.5). Details are in Appendix B.

Global consistency check. To defeat mirror-world attacks, relying parties can confirm that other relying parties see the same RPKI objects. A trivial, but unwieldy, solution has relying parties check consistency by synchronizing and exchanging their entire local caches. Our solution dispenses with synchronization and requires the exchange of much less information. We ask only that one party (Alice) is no more than time t_g ("global consistency window") ahead of the other (Bob).

Bob sends Alice the hash h of the contents (excluding the signature) of latest manifest that he *obtained* for each publication point in his local cache. For every hash value h received from Bob, Alice checks that $h \in H_A$, where H_A is the set that contains the hash of each manifest that Alice obtained, going back for time t_g. The check fails if $h \notin H_A$, and Alice raises a *global-inconsistency alarm* implicating the manifest corresponding to h.

This alarm is *accountable* if the manifest corresponding to h has a matching serial number with one of the manifests in H_A from the same publication point, or if the manifest corresponding to h has an issue date and and time that is between two consecutive manifests in H_A from the same publication point. Alice and Bob can then demonstrate that the publication point contained improperly chained manifests. Otherwise, Alice and Bob need to investigate why information that one of them receives is so far ahead or behind information that the other one receives.

A pairwise interactive protocol is not actually required here; Bob can just post his hash values in a public location for any other relying party to use.

5.5 Security analysis.

We now discuss the security properties of our design. We only state theorems here; proofs are Appendix D.

Threat model. We suppose that the relying parties named in Theorems 5.1-5.3 are honest; everyone else can be arbitrarily malicious, but cannot break cryptography (*e.g.,* forge digital signatures, find hash collisions, steal keys, *etc.*). We therefore operate in the standard threat model used in *e.g.,* [21,40], where trusted auditors (relying parties) audit the behavior of a centralized authority (the RPKI).

We use the following definition: for an RC R and relying party Alice, let the *immediate successor* of R be an RC that (a) overwrites R, or (b) is named in the special post-rollover manifest of R at the time when Alice marks R as rolled-over (see Appendix A), or whichever happens first from Alice's point of view. Define the set of *successors* of R inductively as the set containing R and immediate successors for each of its elements.

Valid remains valid. The following theorem tells us that once an honest relying party Alice sees a valid object, the object will remain valid for Alice until the object consents to revocation (or Alice raises an alarm):

THEOREM 5.1. *Suppose an RC R was valid for a relying party Alice at time t_1. Consider some time $t_2 > t_1$. Then at time t_2, at least one of the following is true:*

1. *a successor of R is valid in the local cache of Alice and has all the resources of R;*
2. *a successor of R is valid in the local cache of Alice, is missing some resources that R had, and Alice observed `.dead` object(s) signed successors of R consenting to the revocation(s) of those resources;*
3. *at or before time t_2, Alice saw a `.dead` object signed by a successor of R, consenting to its revocation;*
4. *at or before time t_2, Alice raised a unilateral revocation alarm in response to a deleted or overwritten object; the alarm included a successor of R as a victim, and blamed an ancestor (or a successor of the ancestor) of the deleted or overwritten certificate as the perpetrator.*

No mirror worlds. We prove robustness to mirror world attacks. Relying parties can be sure that if they see a valid RC in a manifest, and use this manifest in the global consistency check, others who successfully check against them also saw the same valid object:

THEOREM 5.2. *Suppose the local cache of a relying party Bob contains a valid RC R and a manifest m that was issued by the parent of R. Suppose that R is not marked as stale; thus, the version of R in Bob's local cache is same as the one logged in m. Suppose another relying party Alice performs a global consistency check against Bob; Bob sends the hash of m to Alice, and Alice looks for it in her set H_A of hashed manifests. Suppose the global consistency check does not raise the global-inconsistency alarm for m. Then either*

1. *Alice raised an alarm with R as the victim when performing the local consistency check on a manifest in H_A, or*
2. *at any time t_2 after the global consistency check, at least one condition in Theorem 5.1 holds for R and Alice.*

No mirror worlds in the past. Because manifests are hash-chained, a successful global consistency check also implies that parties were consistent in the past.

THEOREM 5.3. *Consider relying parties Alice and Bob. Suppose that*

1. *Bob obtained k consecutive manifests $m_1, \dots m_k$ issued by one RC or by a sequence of successor RCs, and their verification did not raise "invalid syntax" alarms for Bob;*
2. *After Bob checked at least one of the manifests m_i for $1 \leq i \leq k$, Bob found that R was logged in m_i. Bob's local cache also has R marked as "valid" and not "stale";*
3. *Alice passes a global consistency check against Bob, where Bob sends h and Alice find $h \in H_A$. Let h be the hash of (a) manifest m_k in Bob's local cache, and (b) manifest m' in Alice's local cache.*
4. *Alice obtained $k-1$ most recent predecessors of m' and did not raise "invalid syntax" alarms for them.*

Then either (a) Alice raised an alarm with R as the victim when performing the local consistency check on m' or one of its $k-1$ predecessors, or (b) at any time t_2 after the global consistency check, at least one condition in Theorem 5.1 holds for R and Alice.

Alarms & accountability. Our theorems assume that relying parties Alice and/or Bob are *honest*, and ensure that honest relying parties raise alarms when their local caches are missing/contain objects that violate our specifications. But we cannot, in general, prevent *dishonest* relying parties from raising false alarms, because it is impossible to prevent someone from falsely claiming they are missing information. As such, our alarms come in two categories: *accountable* and *unaccountable.*

If Alice raises an accountable alarm, then she is certain that some RPKI authority misbehaved, she can identify this authority (the "perpetrator"), and she can prove her knowledge to others (even those who do not trust Alice) by publishing the relevant objects from her local cache. Assuming that cryptography is not broken, even a malicious Alice cannot create an accountable alarm that is false.

Alice raises an unaccountable alarm when she is missing information — when, from Alice's point of view, there is not enough information to distinguish between an attack by an RPKI authority, and a disruption to the delivery of RPKI objects (*e.g.,* by a third-party that tampers with the communication path). While Alice will still raise an alarm in these cases, the alarm will not be provable to others, and even Alice herself cannot tell who perpetrated the alarm. If the missing information is found, then the alarm will either be resolved or become accountable.

To ensure that as many alarms as possible are accountable, we require authorities to commit (by means of hash values) to the context that is necessary to determine the validity of objects. For example, the validity of a `.dead` depends on its signer's manifest; thus, a `.dead` includes the hash of its signer's manifest (Section 5.3.1). Similarly, the validity of manifest m depends on the validity of the manifest that logs the RC that issued m; thus, m includes the hash of its parent's manifest (Section 5.3.2). An alarm is accountable if these objects are present in Alice's local cache, and unaccountable if they are missing. A missing information alarm is always unaccountable. Section 5.4 and Appendix B. details when other alarms are accountable.

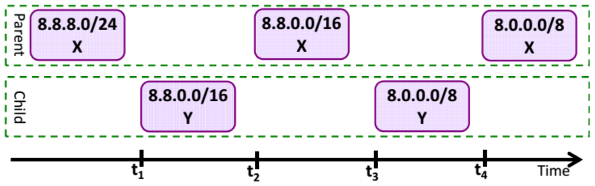
5.6 On the necessity of our modifications.

We give intuition for our design via two examples.

Counterexample 1: Not checking intermediate state. Suppose relying parties did not verify every consecutive manifest update. An authority X could then launch a mirror world attack violating Theorems 5.2 and 5.3:

At time t_1, X issues an RC Y; at time t_2, X replaces Y with a new valid RC Y' that contains an additional IP prefix; and continues swapping between Y (at odd-numbered times), and Y' (at even-number times). Alice syncs to X's publication point at odd-numbered times, and sees only Y; Bob syncs at even-numbered times and sees RC Y'. But, since Alice does not check full intermediate states (only manifest chains), she does not notice the transition from Y' to Y, and does not realize that X needs to get a `.dead` object from Y'. Thus, Alice and Bob live in mirror worlds.

Counterexample 2: Race conditions. A major challenge we faced was that relying parties can sync to different publication points at different times. This creates race conditions that complicate global consistency:



Suppose authority X has small address block. At time t_1, X issues a child RC Y that is invalid because Y contains more addresses than its parent X. At time t_2, parent X is overwritten by an RC for those addresses, and its child Y becomes valid. At time t_3, parent X overwrites Y with an RC that is invalid because it contains even more addresses. This process continues as above. Notice that from time t_1 to t_2, and from t_3 to t_4, the manifest of X logs an invalid object. Suppose Alice syncs just after t_1 and just after t_3; she decides that Y is invalid. Bob syncs just after t_2 and just after t_4; he decides that Y is valid. Hence, mirror worlds!

This problem cannot be caught by checking for `.dead` objects for Y, since in Alice's view Y is always invalid, and in Bob's view Y just keeps getting more resources. Nor can it be caught by the global consistency check, because Alice and Bob see the same manifests. Our design eliminates this problem by requiring relying parties to alarm if invalid objects are logged in the manifest; in this example, Alice would raise an alarm. Theorems 5.2 and 5.3 would be false without this requirement.

5.7 Data-driven analysis of our design.

We discuss the impact of our changes on the RPKI.

Less crypto. One immediate improvement is that a single digital signature on the manifest needs to be issued and verified, rather than individual signatures on each RC, ROA, CRL, and manifest (Section 5.3.2). To put this perspective, on January 13, 2014 there were $\approx 10,400$ validly-signed objects in the RPKI; our changes require the validation of only $\approx 2,800$ manifests.

No renewals. RCs/ROAs do not expire in our design (Section 5.3.2); hence recipients of resources are no longer dependent on their issuers for routine renewals.

Mandated interaction for obtaining consent. However, issuers are dependent on recipients to provide consent (via `.dead` objects) for revocations and certain modifications (Section 5.3.1). To find out how often those events happen, we use our trace of the production RPKI for 2013/10/23–2014/01/21 (Section 3). The largest event we observed was in mid-November 2013, when RIPE removed 3,336 RCs/ROAs and issued new ones with new parent/child pointers and

# ASes	1	2	3	4	5	6-10	10-30	98
RIPE	678	122	51	13	12	30	8	1
LACNIC	123	20	9	2	1	2	0	0
APNIC	26	8	2	0	2	0	0	0
ARIN	30	5	4	4	3	0	0	0
AfriNIC	9	2	1	1	0	0	0	0

Table 8: # of leaf RCs issuing ROAs for X ASes on January 13, 2014; X is in the top row.

# ASes	1-10	11-30	31-100	100-200	200 − 1073
	115,605	594	132	15	11

Table 9: Similar to the distribution in Table 8, except for direct-allocation RCs in our model.

public keys, as part of repository restructuring. Our design would require RIPE to obtain `.dead` objects from all of them, which seems to impose a large burden on RIPE. However, unless RIPE holds the secret keys of its descendants (in which case it can just issue `.dead` objects by itself), interaction is needed even if `.dead` objects were not required, because RIPE's descendant RCs would need to reissue their objects in new publication points.

Besides this event, we saw 4,443 instances of modified/revoked RCs/ROAs. Of these, 3,569 (80%) were renewals[10], that are not needed in our design. Meanwhile, at most 230 (5%) would need a `.dead` object. (Other changes merely added resources, changed the serial number, or did other things that do not require `.dead` objects)

How many parties need to consent? Next, we consider how many parties need to be involved in signing a `.dead` when an RC is revoked. Our estimates, made from the production RPKI and a model of the RPKI, suggest that we do not require many `.dead` objects *on average*:

1. Production RPKI. **Table 2** shows the structure of the production RPKI on January 13, 2014. Suppose that ROAs were able to sign off on `.dead` objects (*e.g.,* because they had requested their own covering RCs, or via the mechanism in footnote 8). How many entities would need to sign a `.dead` object if we wanted to revoke a leaf RC in Table 2? We use ASes as a proxy for entities, and estimate this by counting the number of ASes in ROAs issued by each leaf RC; the results for the production RPKI are in **Table 8**, which shows that on average, only 1.6 ASes need to consent to revoking a leaf RC, and that 93% of leaf RCs can be revoked with the consent of no more than 3 ASes.

2. Model. Because the RPKI is far from fully deployed, we also created a model for a full deployment of the RPKI using routing data for the week starting 2012/05/06. Our model has the RIRs sit at the highest layer of the hierarchy; they issues RCs to "direct allocations," *i.e.,* IP prefixes directly allocated by the RIRs (*e.g.,* Sprint in Figure 1). We obtained these top two layers of the hierarchy from files retrieved from the FTP site of each RIR. Our model omits intermediate RCs (since they are also held by RIRs). To model ROAs descended from each direct allocation, we extracted (prefix, origin AS)-tuples from BGP feeds [3,5] for the week starting 2012/05/06. (Any tuple not covered by a directly-allocated prefix was discarded as a bogon.)[11]We grouped tuples by AS and found that, on average, each direct allocation issues

[10]For RCs, a "renewal" was an RC overwritten with another RC that extended the validity date and did not modify the allocated resources. Because ROAs are supposed to be reissued with new filenames, a "renewal" for ROAs was another ROA issued in the same publication point with the exact same AS and IP prefixes.

ROAs for only 1.5 ASes; the distribution is in **Table 9**.

With great power comes great responsibility. Tables 8, 9 indicate that there are a small percentage of outlier RCs that issue ROAs for many (even hundreds!) of ASes. (In our model, out of 116,357 total direct-allocation RCs, 26 (0.02%) have more than 100 ASes and 221 (0.18%) have more than 25 ASes.) The biggest outlier in our model was an RC for 12.0.0.0/8 held by Sprint, that covers ROAs for 1073 distinct ASes; following this was an RC for 38.0.0.0/8 held by Cogent covering 721 ASes, and and RC for 63.64.0.0/10 held by Verizon covering 598 ASes. The biggest outlier in the production RPKI is an RC that seems to be held by Swisscom, that issues ROAs for 98 distinct RCs. Revoking these outliers requires a large number of `.dead` objects. However, we consider this to be a feature, not a bug: these RCs can impact routing to a large number of ASes, so revoking them should not be easy. Moveover, outright revocation may not be necessary if the goal is simply to change the resources given to them slightly, because we provide mechanisms for removing resources from RC that only require `.dead` objects from descendants that become invalid as a result (Section 5.3.1).

6. RELATED WORK

Routing security. The RPKI is a realization of a series of routing security proposals [6, 34, 52, 68] for origin authentication (Section 2.2). A number of works [10, 27, 43] argue that origin authentication can significantly improve routing security. The RPKI is also the first step towards a comprehensive solution for securing the current routing system with BGPSEC [41] or other proposals surveyed in [14, 33]. See also [69] for clean-slate architectures.

Censorship. Censorship is known to occur at all layers of the Internet's architecture; see *e.g.,* [49] for an overview. There is already evidence [8, 60] of routing-based censorship with BGP; we explored the risk that the RPKI could also be used for this purpose.

PKI design. Our modified RPKI architecture (Section 5) is related to an long line of work on public key infrastructure (PKI) design, culminating in recent efforts to harden the web PKI [51]. The idea of validating the state of the RPKI over time is related to certificate pinning [24], and the idea of hash-chaining manifests is related to append-only-logs [62] (or see [21] and references therein) and ideas related to certificate transparency [23, 36, 40]. While these works suppose that a single logger tracks a stream of ordered events, we have to deal with the race conditions (Counterexample 2) that result from allowing individual RPKI authorities to maintain their own logs (*i.e.,* manifests). The idea of correcting power imbalances in a hierarchical system also appeared in work on distributing certificate authorities [70] and centralized systems like the DNS [1, 15, 59]; these works distribute the *issuance* of objects, but we only distribute *revocation* (since revocation can harm IP prefix reachability).

[11]Figure 1 is derived from this model. We built a subtree of RCs below each direct allocation RC, with one RC for every prefix with (prefix, AS)-tuple in the BGP feeds. The ancestor relationship corresponds to the cover relationship for prefixes, and we collapsed parent-child pairs of RCs generated for the same AS, ASes in the same organization per [16] or when if child was a "stub AS" per [67].

The RPKI. Research on the RPKI covers measurement [53, 66] and policy questions [17, 47, 48, 65]. Our earlier HotNets paper [18] discusses the threat of misbehaving RPKI authorities, but does not provide solutions to any the threats it discusses; part of Section 3 overlap with [18], but the remainder of this paper presents new results.

Concurrently to our work, there have been efforts within the IETF to harden the RPKI against authorities that abuse their power [11, 13, 35]. Kent *et al.* [35] considers the threat of whacked ROAs and of "competing ROAs". A new ROA "competes" with an existing ROA if it contains prefixes covered by the older ROA; a competing ROA is a threat if BGP is attacked, since the AS in the competing ROA can perform a (sub)prefix hijack on the AS in the older ROA. Our architecture (Section 5) ignores this threat because we focus on the risk that the RPKI can take IP prefixes offline in the *absence* of an attack on BGP. Moreover, any authority that issues a competing ROA and then attacks BGP can be held accountable; the competing ROA itself is non-repudiable evidence of the attack. Both [35] and our design defend against whacked ROAs by comparing the state of the RPKI over time, but we also detect mirror world attacks and have a consent mechanism and exact security guarantees.

Systems have also been developed to monitor the RPKI [4, 38, 50, 61, 64]; most use a snapshot of ROAs from the RPKI to determine the validity state of routes in publicly-available BGP route collectors [3, 5]. Our detector (Section 4.1) is complementary because we detect when any *change* to the RPKI alters the validity state of *all* possible routes, not just the ones visible from a particular BGP vantage point at a specific time; it can therefore be used as an alert system (especially when RPKI deployment reaches steady state), even if a particular vantage point does not obtain a complete view of all routes announced in BGP. Our tools also provide a new way of visualizing downgrade events (Section 4.2).

7. CONCLUSION

We have explored a number of techniques to harden the RPKI against the risk of IP prefix takedowns. We have built tools for detecting takedowns within the existing RPKI specifications. We have also proposed changes to the specifications that (1) entitle parties to *consent* to revocations of their IP address space, and guarantee that relying parties can (2) detect misbehavior by RPKI authorities and (3) obtain a *consistent* view of information in the RPKI. Given the security improvements promised by the RPKI [10,27,43], we hope our work will catalyze further efforts to harden the RPKI against abusive authorities.

Acknowledgements. We thank Kyle Brogle for help with early stages of this work, Steve Kent for suggesting the term "ROA whacking", FCC CSRIC Working Group 6 - Secure BGP Deployment for inspiring this project, Tony Tauber, Ed Felten and various members of the IAB and the IETF SIDR working group for useful discussions, CAIDA for early access to their AS to organization mapping [16], and John Byers, Davide Proserpio, Ben Fuller, Phillipa Gill, Aanchal Malhotra, our shepherd Adrian Perrig, and the anonymous SIGCOMM reviewers for insightful comments on this draft. This work was supported by NSF awards 1017907, 1012798, and 1012910 and a gift from Cisco.

8. REFERENCES

[1] Github: namecoin repository.
`https://github.com/namecoin/namecoin`.

[2] rcynic software. http://trac.rpki.net.

[3] RIPE RIS raw data. http://www.ripe.net/data-tools/stats/ris/ris-raw-data.

[4] Rpki spider. http://rpkispider.verisignlabs.com/.

[5] University of oregon route views project. http://www.routeviews.org/.

[6] W. Aiello, J. Ioannidis, and P. McDaniel. Origin authentication in interdomain routing. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 165–178. ACM, 2003.

[7] S. Amante. Risks associated with resource certification systems for internet numbers, 2012.

[8] D. Anderson. Splinternet behind the great firewall of china. *Queue*, 10(11):40, 2012.

[9] R. Austein, G. Huston, S. Kent, and M. Lepinski. *RFC 6486: Manifests for the Resource Public Key Infrastructure (RPKI)*. Internet Engineering Task Force (IETF), 2012. http://tools.ietf.org/html/rfc6486.

[10] H. Ballani, P. Francis, and X. Zhang. A study of prefix hijacking and interception in the Internet. In *SIGCOMM'07*, 2007.

[11] R. Bush. *Responsible Grandparenting in the RPKI*. Internet Engineering Task Force Network Working Group, 2012. http://tools.ietf.org/html/draft-ymbk-rpki-grandparenting-02.

[12] R. Bush. *RPKI-Based Origin Validation Operation*. Internet Engineering Task Force Network Working Group, 2012. http://tools.ietf.org/html/draft-ietf-sidr-origin-ops-19.

[13] R. Bush. *RPKI Local Trust Anchor Use Cases*. Internet Engineering Task Force (IETF), 2013. http://www.ietf.org/id/draft-ymbk-lta-use-cases-00.txt.

[14] K. Butler, T. Farley, P. McDaniel, and J. Rexford. A survey of BGP security issues and solutions. *Proceedings of the IEEE*, 2010.

[15] C. Cachin and A. Samar. Secure distributed dns. In *Dependable Systems and Networks, 2004 International Conference on*, pages 423–432. IEEE, 2004.

[16] CAIDA. AS to organization mapping. http://as-rank.caida.org/?mode0=as-intro#as-org.

[17] Communications Security, Reliability and Interoperability Council III (CSRIC). Secure bgp deployment. *Communications and Strategies.*

[18] D. Cooper, E. Heilman, K. Brogle, L. Reyzin, and S. Goldberg. On the risk of misbehaving RPKI authorities. *HotNets XII*, 2013.

[19] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. *RFC 5280: Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. Internet Engineering Task Force (IETF), 2008. http://tools.ietf.org/html/rfc5280.

[20] J. Cowie. Rensys blog: China's 18-minute mystery. http://www.renesys.com/blog/2010/11/chinas-18-minute-mystery.shtml.

[21] S. A. Crosby and D. S. Wallach. Efficient data structures for tamper-evident logging. In *USENIX Security Symposium*, pages 317–334, 2009.

[22] A. de Beaupre. ISC Diary: Multiple Banking Addresses Hijacked, 2013. http://isc.sans.edu/diary/BGP+multiple+banking+addresses+hijacked/16249.

[23] P. Eckersley. Sovereign key cryptography for internet domains. Technical report, EFF, 2011.

[24] C. Evans, C. Palmer, and R. Sleevi, editors. *Public Key Pinning Extension for HTTP*. IETF Web Security, Internet-Draft, November 27 2013. http://tools.ietf.org/html/draft-ietf-websec-key-pinning-09.

[25] R. Gagliano, T. Manderson, and C. M. Cagnazzo. *Multiple Repository Publication Points support in the Resource Public Key Infrastructure (RPKI)*. Internet Engineering Task Force (IETF), 2013. http://tools.ietf.org/html/draft-ietf-sidr-multiple-publication-points-00.

[26] I. Gassko, P. Gemmell, and P. D. MacKenzie. Efficient and fresh cerification. In H. Imai and Y. Zheng, editors, *Public Key Cryptography*, volume 1751 of *Lecture Notes in Computer Science*, pages 342–353. Springer, 2000.

[27] S. Goldberg, M. Schapira, P. Hummon, and J. Rexford. How secure are secure interdomain routing protocols? In *SIGCOMM'10*, 2010.

[28] E. Goldman. Sex.com: An update. http://blog.ericgoldman.org/archives/2006/10/sexcom_an_updat.htm, 2006.

[29] G. Huston, R. Loomans, and G. Michaelson. *RFC 6481: A Profile for Resource Certificate Repository Structure*. Internet Engineering Task Force (IETF), 2012. http://tools.ietf.org/html/rfc6481.

[30] G. Huston, R. Loomans, and G. Michaelson. *RFC 6487: A Profile for X.509 PKIX Resource Certificates*. Internet Engineering Task Force (IETF), 2012. http://tools.ietf.org/html/rfc6487.

[31] G. Huston and G. Michaelson. *RFC 6483: Validation of Route Origination Using the Resource Certificate Public Key Infrastructure (PKI) and Route Origin Authorizations (ROAs)*. Internet Engineering Task Force (IETF), 2012. http://tools.ietf.org/html/rfc6483.

[32] G. Huston, G. Michaelson, and S. Kent. *RFC 6489: Certification Authority (CA) Key Rollover in the Resource Public Key Infrastructure (RPKI)*. Internet Engineering Task Force (IETF), 2012. http://tools.ietf.org/html/rfc6489.

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

[34] S. Kent, C. Lynn, and K. Seo. Secure border gateway protocol (S-BGP). *J. Selected Areas in Communications*, 18(4):582–592, April 2000.

[35] S. Kent and D. Mandelberg. *Suspenders: A Fail-safe Mechanism for the RPKI*. Internet Engineering Task Force (IETF), 2013. http://tools.ietf.org/html/draft-kent-sidr-suspenders-00.

[36] T. H.-J. Kim, L.-S. Huang, A. Perrig, C. Jackson, and V. Gligor. Accountable Key Infrastructure (AKI): A Proposal for a Public-Key Validation Infrastructure. In *Proceedings of the International World Wide Web Conference (WWW)*, May 2013.

[37] L. M. Kohnfelder. *Towards a Practical Public-key Cryposystem*. Massachusetts Institute of Technology, 1978. Bachelor's Thesis. http://groups.csail.mit.edu/cis/theses/kohnfelder-bs.pdf.

[38] LACNIC. RPKI looking glass. www.labs.lacnic.net/rpkitools/looking_glass/.

[39] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[40] B. Laurie, A. Langley, and E. Kasper. Certificate transparency. *Network Working Group Internet-Draft, v12, work in progress. http://tools. ietf. org/html/draft-laurie-pki-sunlight-12*, 2013.

[41] M. Lepinski, editor. *BGPSEC Protocol Specification*. IETF Network Working Group, Internet-Draft, July 2012. Available from http://tools.ietf.org/html/draft-ietf-sidr-bgpsec-protocol-04.

[42] M. Lepinski and S. Kent. *RFC 6480: An Infrastructure to Support Secure Internet Routing*. Internet Engineering Task Force (IETF), 2012. http://tools.ietf.org/html/rfc6480.

[43] R. Lychev, S. Goldberg, and M. Schapira. Is the juice worth the squeeze? BGP security in partial deployment. In *SIGCOMM'13*, 2013.

[44] T. Manderson, L. Vegoda, and S. Kent. *RFC 6491: Resource Public Key Infrastructure (RPKI) Objects Issued by IANA"*. Internet Engineering Task Force (IETF), 1973. http://tools.ietf.org/html/rfc6491.

[45] S. Misel. "Wow, AS7007!". Merit NANOG Archive, apr 1997. www.merit.edu/mail.archives/nanog/1997-04/msg00340.html.

[46] P. Mohapatra, J. Scudder, D. Ward, R. Bush, and R. Austein. *RFC 6811: BGP prefix origin validation*. Internet Engineering Task Force (IETF), 2013. http://tools.ietf.org/html/rfc6811.

[47] M. Mueller and B. Kuerbis. Negotiating a new governance hierarchy: An analysis of the conflicting incentives to secure internet routing. *Communications and Strategies*, (81):125–142, 2011.

[48] M. Mueller, A. Schmidt, and B. Kuerbis. Internet security and networked governance in international relations. *International Studies Review*, 15(1):86–104, 2013.

[49] S. J. Murdoch and R. Anderson.

[50] NIST. RPKI deployment monitor. http://www-x.antd.nist.gov/rpki-monitor/.

[51] NIST. Workshop on Improving Trust in the Online Marketplace, 2013. http://www.nist.gov/itl/csd/ct/ca-workshop-agenda2013.cfm.

[52] E. Osterweil, S. Amante, D. Massey, and D. McPherson. The great ipv4 land grab: resource certification for the ipv4 grey market. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, page 12. ACM, 2011.

[53] E. Osterweil, T. Manderson, R. White, and D. McPherson. Sizing estimates for a fully deployed rpki. Technical report, Verisign Labs Technical Report, 2012.
[54] A. Peterson. Researchers say u.s. internet traffic was re-routed through belarus. that's a problem. *Washington Post*, November 20 2013.
[55] A. Pilosov and T. Kapela. Stealing the internet, 2009.
[56] D. Piscitello. Guidance for preparing domain name orders, seizures & takedowns. Technical report, ICANN, March 2012.
[57] D. Piscitello. The value of assessing collateral damage before requesting a domain seizure. Technical report, ICANN, January 2013.
[58] I. G. Project. In important case, RIPE-NCC seeks legal clarity on how it responds to foreign court orders, 2011. http://www.internetgovernance.org/2011/11/23/in-important-case-ripe-ncc-seeks-legal-clarity-on-how-it-responds-to-foreign-court-orders/.
[59] V. Ramasubramanian and E. G. Sirer. The design and implementation of a next generation name service for the internet. *ACM SIGCOMM Computer Communication Review*, 34(4):331–342, 2004.
[60] Rensys Blog. Pakistan hijacks YouTube. `http://www.renesys.com/blog/2008/02/pakistan_hijacks_youtube_1.shtml`.
[61] RIPE. RPKI validator. `http://localcert.ripe.net:8088/trust-anchors`.
[62] B. Schneier and J. Kelsey. Automatic event-stream notarization using digital signatures. In *Security Protocols*, pages 155–169. Springer, 1997.
[63] W. Sierpiński. Sur une courbe dont tout point est un point de ramification. *Comptes Rendus de l'Acadamie des Sciences*, 160:302–305, 1915.
[64] Surfnet. RPKI dashboard. `http://rpki.surfnet.nl/validitytables.html`.
[65] The President's National Security Telecommunications Advisory Committee. Nstac report to the president on communications resiliency, 2011.
[66] M. Wählisch, O. Maennel, and T. Schmidt. Towards detecting BGP route hijacking using the RPKI. In *Poster: SIGCOMM'12*, pages 103–104. ACM, 2012.
[67] L. Wang, J. Park, R. Oliveira, and B. Zhang. Internet topology collection. `http://irl.cs.ucla.edu/topology/`.
[68] R. White. Deployment considerations for secure origin BGP (soBGP). draft-white-sobgp-bgp-deployment-01.txt, June 2003, expired.
[69] X. Zhang, H.-C. Hsiao, G. Hasker, H. Chan, A. Perrig, and D. G. Andersen. SCION: scalability, control, and isolation on next-generation networks. In *Security and Privacy (SP), 2011 IEEE Symposium on*, pages 212–227. IEEE, 2011.
[70] L. Zhou, F. B. Schneider, and R. Van Renesse. COCA: A secure distributed online certification authority. *ACM Transactions on Computer Systems (TOCS)*, 20(4):329–368, 2002.

APPENDIX

A. KEY ROLLOVER

Suppose RC A issued an RC B, and the subject of B needs to refresh its cryptographic keys. The RPKI's currently-specified key rollover mechanism operates as follows [32]:

1. A issues a new RC B' with the new key, but the same child pointer and resources;
2. after waiting for time at least t_s, B' *overwrites* all the RCs and ROAs issued by B with identical objects that are signed by the key of B' and have the parent pointer to B'; simultaneously, B' issues a new manifest including these objects, while B issues an empty manifest;
3. A revokes B.

Note that this mechanism only requires the participation of the entities holding RC A and RC B and B', and does not disturb the lower levels of the the RPKI hierarchy. For our design of Section 5, we have a key rollover mechanism that also involves only A and B:

1. A issues a new RC B' with the new key, but the same publication point and resources and B; this RC has a special empty manifest called "pre-rollover," and does not log any objects (except the usual vertical hash chain elements).
2. After waiting for at least time t_s, B issues an special empty manifest tagged "post-rollover," which includes a pointer (URI) to B' as well as the hash of B' and of a manifest m_A issued by A that includes B' (as proof that B' was issued). Simultaneously, B' overwrites all the RCs and ROAs issued by B with identical objects that point to B' (with their parent pointer), and issues a new manifest $m_{B'}$ for all the objects. Let m_B be the last manifest of B before the post-rollover manifest. The manifest $m_{B'}$ is a successor of m_B; it includes the hash of m_B and the hash of the post-rollover manifest of B.
3. B provides A with a signed `.roll` object, which, like a `.dead` object, demonstrates that B consents to be deleted. The `.roll` object is signed by B and includes the hash of the post-rollover manifest issued by B. After time t_s, A simultaneously (a) publishes the `.roll` object provided by B; (b) deletes B; and (c) updates the manifest of A to remove B and log the `.roll` object. Future updates to the manifest of A need not log the `.roll` object.

B. LOCAL CONSISTENCY CHECK

Relying parties use a local consistency check to validate their local caches (Section 5.4). The purpose of the local consistency check is (1) to determine the validity state of RPKI objects, (2) to raise alarms. Our notion of object validity remains almost identical to that in the current RPKI, but we change the algorithms for computing validity and precisely specify when and how to raise alarms.

Specifically, the local consistency check marks every RC in the local cache with the following designations. The first four designations are mutually exclusive.

1. *valid*;
2. *no-longer-valid*; RC R was valid in a previous state of the relying party's local cache. In the current local cache, R has not changed, but R is not valid because there is no valid chain of ancestor RCs that cover the resources in R.
3. *rolled-over*; RC R was valid in a previous state of the relying party's local cache. In the current local cache, R has undergone a proper key rollover procedure to some new RC R' (per Appendix A).
4. *never-was-valid*; RC R (at the URI of R and with the contents in R) was never valid in the local cache.
5. *stale*; The relying party could not correctly obtain a particular RC, and has thus reverted to an old version. Importantly, the "stale" designation should be used to inform routing; a relying party should look for unusual routing activity in the address space covered by a stale RPKI object, or by the RC whose manifest has become stale. As in Section 3.2.2, a relying party can try to find a valid version of a stale object by reconnecting to the publication point, asking other relying parties for the object, *etc.*

Alarms raised during the local consistency check always point to a specific victim objects O. The alarm is *accountable* if the relying party (1) can identify a specific perpetrator RC R that is responsible for causing the alarm, and (2) can conclusively *prove* (even to a third party that does not trust the relying party) that the perpetrator RC R was misbehaving. For an alarm to be accountable, the relying party must have the relevant objects signed by the perpetrator that, when taken together, show some violation of the rules. In all other cases, an alarm is *unaccountable*, and does not implicate a specific perpetrator. Importantly, a missing-information alarm is always unaccountable, since the relying party cannot prove who caused the information to be missing (the issuer, the repository, a bad connection, etc.). See also Section 5.5 for more discussion. The procedures below explicitly state when alarms are accountable or unaccountable.

B.1 Initial Sync.

The initial connection to the repository is as in the current RPKI: all objects are downloaded and validated. If a relying party cannot obtain the valid current manifest of an RC R, or any objects logged in the manifest (as in Section 3.2.2), she raises a *missing information alarm* (which is unaccountable) with the missing manifest, or object logged in the manifest, as a victim. Each RC is validated according to the "New RC Procedure" in Appendix B.2.4.

B.2 Subsequent syncs.

When a relying party obtains an updated state of a publication point, she runs the local consistency check by reconstructing all the intermediate states of the publication point. She then compares pairs of consecutive states (indicated by consecutive manifest numbers) to make sure that (a) all new or modified objects are valid, and (b) all deletions and overwrites received proper consent. If not, she issues a missing-information alarm, or one of the middle alarms in Table 7. The exact details of the local consistency check follow.

B.2.1 Checking the manifest & for missing objects.

We say that *a manifest has been obtained* if the relying party can (a) download a manifest that is validly signed by its issuer, OR (b) reconstruct the manifest as element in the horizontal hash chain that terminates in a manifest that is validly signed by its issuer, per Section 5.3.2.

If a manifest has not been obtained, then the relying party checks whether the RC issuing the manifest is valid in its local cache. If so, the relying party (a) raises a missing-information alarm, with the unobtained manifest as a victim, and (b) marks all the objects in the publication point as "stale." The local consistency check for this manifest then terminates.

Otherwise, the manifest has been obtained. The relying party raises an "invalid syntax" alarm if the manifest is syntactically incorrect (*e.g.,* if it logs an RC B along with the `.dead` or `.roll` object for that RC, or if the field showing the serial number of the last issued RC is smaller than in a previous manifest or in an RC logged in this manifest, or if the hash of the predecessor manifest is incorrect). The invalid-syntax alarm is always *accountable*, and has the manifest as a victim, and the RC issuing the manifest as a perpetrator.

Next, the relying party verifies that every object logged in the manifest has been correctly obtained (*i.e.,* the hash of the contents of the object is logged in the manifest). If not, the relying party raises a "missing information alarm" with the missing object's URI as a victim. If an object cannot be obtained, but a previous version of that object (with the same URI) is available in the local cache, then the previous version is marked as "stale." If the object can be obtained, but is syntactically incorrect (*e.g.,* malformed, wrong parent pointer, *etc.*), the relying party raises an "invalid syntax" alarm; the alarm is always *accountable*, and has the object as victim, and the RC issuing the object as perpetrator.

B.2.2 Extra objects.

Any objects obtained from a publication point that not logged in the manifest are ignored, except that they may be used, as needed, to reconstruct intermediate manifests, or to find valid versions of stale objects.

B.2.3 Checking for key rollovers.

The URI of a manifest persists from update to update (manifests at other URIs can be simply ignored), with one exception: if the manifest issued by B is updated to "post-rollover." Once that occurs, perform the following checks:

Check0 Check that the post-rollover manifest is correctly formatted, and includes the hash of an RC B',

Check1 Check that B' is present in the local cache.

Check2 Check that RC B' is valid,

Check3 Check that B' has the same parent and resources as B.

If the above checks pass, the relying party restarts the local consistency check specified in Section B.2.1, and tries to obtain the first manifest $m_{B'}$ issued by B' after the "pre-rollover" manifest, and checks it for the correct hashes (as described as Appendix A). Once such m'_B is obtained, B is marked as "rolled over". The "updated manifest" is then the manifest $m_{B'}$.

If any of the four checks fail, or if m'_B cannot be obtained, then B remains marked as "valid". Then the relying party acts as if the manifest for B has not been obtained and marks all the objects in the publication point as "stale." It also raises a "bad-rollover" alarm with the post-rollover manifest issued by B as a victim.

To determine whether the "bad-rollover" alarm is accountable, recall that the post-rollover manifest m_B is supposed to include the hash of manifest m_A of the parent A, and m_A is supposed include the hash of B'. The alarm is accountable and its perpetrator is B under the following conditions:

1. If Check0 fails. (This is because the relying party can present the malformed m_B as proof.)
2. If Check0 passes but Check1 fails, and the relying party possesses m_A that is logged in m_B, but m_A does not log B'. (This is because the relying party can present m_A and m_B that are inconsistent.)
3. If Check0, Check1, and Check2 pass but Check3 fails. (This is because the relying party can present B, m_B and B' as proof that Check3 failed.)

(Note also: if Check2 fails, or if m'_B cannot be obtained, then this raises a separate alarm, that may or may not be accountable according to the conditions of the alarm..)

B.2.4 Checking individual RCs.

Whenever a manifest is updated, a check is performed for every individual RC logged in the manifest.[12] We say that a manifest update *results from a key rollover*, if an RC B successfully rolled over to to RC B' (*i.e.,* without

[12]Recall that we simplified the exposition by not requiring consent from individual ROAs (since any ROA that wishes to be entitled to consent is issued its own covering RC, or see footnote 8); thus, we only check individual RCs.

raising a "bad-rollover alarm" per Section B.2.3), and we are checking the transition from m_B (the manifest issued by B before it issued its "post-rollover" manifest) to $m_{B'}$ (the first manifest issued by B' that is not a "pre-rollover" manifest). Otherwise, a manifest update results when an RC B overwrites its current manifest with a new manifest. This check for each individual RC therefore considers three things: (1) whether the manifest update resulted from a key rollover or not (see the the top and bottom halves of Table 10, respectively) (2) the status of the RC before the manifest update (valid, no-longer-valid, never-was-valid, or non-existent[13], (see the rows of Table 10) (3) whether the RC was unchanged, changed or deleted after the manifest updated (see the columns of Table 10). Based on these three things, the relying parties follow the specification in Table 10 below to either (a) do nothing, or (b) run one of four possible procedures: "New RC Procedure", "Deleted RC Procedure", "Overwritten RC Procedure", or "Rolled RC Procedure".

		After		
	Before	Unchanged	Changed	Deleted
No Key Roll	valid	■	Overwritten	Deleted
	rolled-over	■	░	Rolled
	no-longer-valid	■	▒	■
	never-was-valid	New RC	New RC	■
	non-existent	■	▒	■
Key Roll	valid	Overwritten		Deleted
	rolled-over	Overwritten		Rolled
	no-longer-valid	▒	▒	■
	never-was-valid	New RC	New RC	■
	non-existent	■	▒	■

Table 10: Procedures to run when checking an RC R, if the manifest logging R was updated due to (top) a key rollover, or (bottom) not. Black squares indicate "do nothing".

We now specify the four procedures.

New RC Procedure. Call the RC that is present after the manifest update C. Validate C.

If C is valid, download the entire subtree rooted at C, and validate its descendants as in an initial sync. (This is done to allow for speedy issuance of new objects; see Appendix C.)

If C is not valid, mark C as "never-was-valid". Then raise an alarm, with C as the victim, per the following:

- If C is invalid for syntactic reasons (malformed, wrong parent pointer, serial number not increasing, *etc.*), raise an "invalid-syntax alarm". The invalid-syntax alarm is accountable and has the issuer of C as its perpetrator.
- Else, let B be the RC that issued C according to the local cache of the relying party. Let m_B be the manifest logging C in the relying party's local cache. The only reason C is not valid is because its resources are not a subset of the resources of B. Thus, raise a "child too broad" alarm.

  The alarm is *accountable*, and blames B as its perpetrator, if the relying party local cache has an a view of B that is the same as the one that is logged in m_B. (B is blamed because it signed a manifest containing an new RC that is not valid.) More formally, let A be the parent of B. Recall that m_B contains the hash of m_A (*i.e.,* the manifest logging RC B that was issued by RC A). The alarm is accountable if m_B contains the hash of a manifest m_A, and m_A logs a version of B (with the same URI as B) that matches B in the relying party's local cache. Otherwise, the alarm is *unaccountable.*

[13]Non-existent implies the RC was not logged in the current manifest

Deleted RC Procedure. Let C be an RC that was valid prior to the manifest update, and was deleted after the manifest update. Mark all "valid" and "rolled-over" descendants of C in the relying party's local cache as "no-longer-valid".

If a proper `.dead` object (per Section 5.3.1) is not present for C, then raise a *unilateral revocation alarm* with C and all of its descendants as victims and B as the perpetrator. The alarm is accountable if C was not "stale" prior to the manifest update and the proper `.dead` object is not a victim of a missing information alarm. (The alarm can proven by presenting two consecutive manifests, one with C and one without C and without the `.dead`.) Otherwise, the alarm is unaccountable.

If a proper `.dead` object is present for C but not for some descendants that have been just marked as "no-longer-valid", then, similarly, raise *unilateral revocation alarm* with those descendants as victims. The perpetrator of the alarm is any descendant RC D of C such that (1) a proper `.dead` object is present for D but (2) a proper `.dead` object is absent for at least one child of D. The alarm is accountable for D as long as the `.dead` issued by D (1) logs a manifest m_D issued by D that is present in the relying party's local cache, and (2) does not log a `.dead` object for one of the children of D included in m_D. (Accountability follows because the relying party can prove, by publishing m_D and the `.dead` issued by D, that D should have obtained a `.dead` from its child but did not.)

Rolled RC Procedure. Let C be an RC that was "rolled-over" prior to the manifest update, and was deleted after the manifest update. Check that a `.roll` object is present for C.

Otherwise, raise a *unilateral revocation alarm* with C as its victim. Let B be the parent of C. Since the local consistency check proceeds incrementally, we know that B can be either "valid" or "rolled over" after this manifest update. If B is valid, blame B. If B is rolled over to an RC B', blame B'. The alarm is accountable if C was not "stale" prior to the manifest update and the proper `.roll` object is not a victim of a missing information alarm. (The unilateral-revocation alarm is accountable because the relying party can presenting two consecutive manifests, one with C, and one without C and without the `.roll`.)

Overwritten RC Procedure. Let C be the RC that was "valid" or "rolled over" prior to the manifest update, and let C' be the RC at that same URI after the manifest update. Let B be the parent of C. The following situations are permitted (and in all three of them, C' needs to be validated, the same way as in the "New RC Procedure"):

1. The manifest update resulted from a key rollover from B to B', and C' is identical to C, except that C' has a parent pointer pointing to B'. In this case, C' is marked with the same validity state as C (either "valid" or "rolled-over").
2. The manifest update did not result from a key rollover to B. C' is valid and has the same fields as C, except perhaps additional resources. In that case, re-evaluate the validity of every descendant of C in the local cache that is currently marked as "no-longer-valid" or "never-

was-valid".

3. The manifest update did not result from a key rollover to B. C' is valid and the same fields as C, except that some of its resources are removed. In that case, re-evaluate the validity of every descendant of C in the local cache. For each valid descendant RC of C that becomes "no-longer-valid" (because C is overwritten by C') check for `.dead` objects and raise "unilateral revocation" alarms as in the "Deleted RC Procedure".

Otherwise, run the "Deleted RC Procedure" on C and the "New RC Procedure" on C'.

C. TIMING UPDATES TO THE RPKI

Our design in Section 5 supposes that relying parties sync to each publication point within time t_s of the previous sync, but that relying parties can sync to publication points in any order. We therefore must consider procedures that RPKI authorities can use to update their publication points, without triggering undo alarms that result from race conditions.

More specifically, our local and global consistency checks in Section 5.4 and Appendix B are designed to make sure that no alarms are raised as long as (1) both the authority B and the relying party have local caches that are no more than t_s behind the current state of the RPKI, and (2) that the authority B follows the procedures outlined in Section 5.3. Importantly, these procedures require authority B to wait for at least time t_s between (a) noticing (in its local cache) that an update to the RPKI was performed by some other RPKI authority A, and (b) updating its publication point (of B), if the correctness of the update (per Section 5.3) depends on the update performed by A. This ensures that relying parties see the update performed by A before they try to verify the update preformed by B. Note, however, if the update does *not* depend on some other update made by another authority, that B can update its own publication point as frequently as it likes.

We now discuss how timing affects specific actions performed by authorities.

C.1 Small updates to the RPKI.

Upon being broadened. Suppose that RC B is *broadened*, *i.e.,* B receives new resources (IP prefixes, AS numbers) from its issuer A. Once A publishes the broadened version of B, the subject of B must wait at least t_s time (to ensure that relying parties have seen the broadened version B) before publishing any broadened child objects, *i.e.,* those objects whose validity depends on the new resources received by B.

Deleting or narrowing child objects. Suppose an RC A either deletes a child RC B or removes resources from B. Since relying parties verify this change with respect to the manifest of A, independently of its ancestors, no timing issues arise for A. However, B, which needs to collect `.dead` objects from its descendants, needs to be aware of timing issues, as follows.

Collecting `.dead` objects. Recall that in order to consent to its own revocation or narrowing (performed by its ancestor A), an authority B needs to obtain `.dead` objects from each of its affected child RCs and include their hashes into its own `.dead` object. Because relying parties verify these `.dead` objects during the update of the manifest of A, their local caches may not be up-to-date with respect to the publication point of B. Therefore, B needs to collect `.dead` objects from any child that is currently valid, as well any child that was valid up to time t_s in the past. Note also that if a relying party's cache of B's publication point up-to-date, the relying party may see `.dead` objects from recently deleted RCs that are no longer in the relying party's local cache; conversely, if a relying party is behind, it may see `.dead` objects for RCs it has not yet found.

Upon being narrowed. Suppose that an authority B signed a `.dead` object consenting to have its RC *narrowed*, *i.e.,* to have some resources removed. Then B should assume that the relying parties may see the newly narrowed version of B's RC even before B itself sees the narrowed version. Therefore, once B has signed and passed on its own `.dead` object, B should no longer issue objects that contain resources revoked with the `.dead`.

C.2 Bulk changes to the RPKI.

Finally, let us highlight how these issues affect large-scale changes to the RPKI.

A new subtree. Creation of a new subtree of RCs, no matter how deep, can be done quickly, because relying parties download new descendant objects immediately upon receiving a new RC (per the "New RC Procedure" in Appendix B.2.4). This means that entire *new* subtrees can be published at once (to avoid raising missing information alarms for manifests that are not found or are incomplete, the root of the new subtree should be published only once its manifest and descendant objects are available from the repository). The new subtree should propagate to all relying parties within time t_s of when its root is published.

Broadening an existing tree. On the other hand, if one wants to broaden every RC on a long vertical chain, the chain needs to be handled top-down, and time t_s needs to elapse for every step in the chain; this holds the exception of RCs with an "inherit" attribute in place of explicitly specified resources, which indicates that the RC has resources identical to its parent (see [30, Section 2] or Section 5.3.1). Thus, if an RC is broadened and wants to take advantage of the broader resources, it may do this more quickly by publishing completely new descendants, rather than by broadening existing ones (unless its existing descendants use the "inherit" attribute).

Deleting a subtree. Deletion of a new tree of RCs, no matter how deep, can be done quickly, by publishing all the `.dead` objects for the entire tree.

Narrowing a subtree. On the other hand, if one wants to narrow every certificate on a long vertical chain, the chain needs to be handled bottom-up, and time t_s needs to elapse for every step in the chain (with the exception of RCs that use the "inherit" attribute, which need not be narrowed explicitly, per Section 5.3.1).

D. PROOFS

We prove the theorems in Section 5.5.

D.1 Proof of Theorem 5.1.

This theorem holds because we require updates to manifests to be preformed one at a time, and because of the careful key rollover procedure, which ensures that successors are well-defined.

Our proof is by contradiction. Consider every update of the local cache that Alice performs between time t_1 and t_2, one pair of consecutive manifests at a time. Consider the first point t_B in time when none of the four conditions of the conclusion of theorem is true. Since R is valid at time t_1 (making the first condition in the theorem true at time t_1), there must have been at least one manifest update between time t_1 and t_B. Take the last manifest update the occurs immediately before t_B. Since one of the four conditions in the theorem was true before that manifest update, and since this condition stopped being true after the manifest update, the condition that stopped being true cannot be the third or fourth condition. (Because the third and fourth conditions, if true once, remain true forever.) Thus, by definition of t_B, it follows that right before the manifest update, there was some successor R' of R that satisfied either the first or the second condition in the theorem, and after the manifest update it stopped satisfying the condition. This means that, during the local consistency check by Alice, a valid R' was (1) deleted, (2) overwritten, (3) unchanged but marked as no-longer-valid, or (4) unchanged but marked as rolled-over.

For each of these four cases, we can determine which manifest must have been updated at time t_B, as follows. Cases (1) and (2) occur when the updated manifest was the one issued by the parent of R'; case (3) occurs when the updated manifest was issued by some ancestor of R'; and case (4) occurs when the updated manifest was issued by R' itself and the new manifest is post-rollover. We now consider how relying party Alice reacted in each of these four cases.

Case 1 is handled by the "Deleted RC Procedure" for checking individual RCs (Appendix B.2.4). Since R' was valid before time t_B, Alice checks for a `.dead` object signed by R'. If it is present, then the third condition becomes true. If not, then Alice raises a "unilateral revocation alarm" with R' as the victim, blaming the parent of R' (or its immediate successor in case the parent just performed a key rollover), and the fourth condition becomes true by Claim D.1.

Case 2 is handled by the "Overwritten RC Procedure" for checking individual RCs (Appendix B.2.4). Let R'' be the RC that overwrites R'. R'' is, by definition, a successor of R' and therefore also of R. If R'' is not valid, or is missing some resources in R' and lacking an appropriate `.dead` object, then Alice will raise an "unilateral revocation alarm" with R' as the victim, blaming the parent of R'' (which is the parent of R' or, in case of key rollover by the parent of R', its immediate successor). This makes the fourth condition true by Claim D.1.

The handle Case 3, observe from Appendix B.2.4 that RC R' can only be marked as 'no-longer-valid' by Alice when Alice executes the "Overwritten RC Procedure" or "Deleted RC Procedure" when some RC S that is an ancestor of R' is deleted or overwritten. These procedures require Alice to raise a "unilateral revocation" alarm blaming an ancestor of R' or its immediate successor (making the fourth condition true by Claim D.1) unless an appropriate `.dead` object signed by R' is present (making the third condition true).

Case 4 can happen only when Alice sees a valid successor R'' of R' with the same resources, as indicated by the post-rollover manifest signed by R'. This makes the first condition true.

Thus, in all four cases, one of the conditions in the theorem statement will be true. We have arrived at the desired contradiction, as long as we prove the following:

CLAIM D.1. *Before time t_B, all the ancestors of R' are successors of ancestors of R.*

PROOF. Let $R = R_1, R_2, \ldots, R_k = R'$ be the chain of immediate successors leading from R to R'. By the same reasoning as used for R' above, we can establish that before time t_B, each R_i satisfied either the first or the second condition of the theorem—that is, it was valid in Alice's local cache until Alice saw R_{i+1}. We can also establish that before time t_B, R_i did not satisfy the third of fourth condition of theorem.

The proof will proceed by nested induction: first through each change in the ancestors of an R_i, and then on a change from R_i to R_{i+1}.

Suppose that one or more ancestors of R_i change at some time t. Since manifest updates are processed one at a time (and R_i is not changing), this can happen only because some ancestor S of R_i was overwritten with another RC S'. When the overwrite occurred, either S' had the same parent pointer as S or not. In the former case, S' is the successor of S and all the other ancestors remain the same, and the inductive step holds. In the latter case, Alice treats the overwrite of S as a deletion of S, unless the overwrite is due to a key rollover by the parent of S. However, a deletion of S would make R_i no-longer-valid, while earlier we established that R_i is valid until R_{i+1} is seen. Thus, the overwrite of S must be due to a key rollover, which means parent of S' is a successor of the parent of S, and all the other ancestors remain the same. The inductive step holds.

Now consider the moment when R_{i+1} becomes a successor of R_i. This can happen if:

- Alice sees a proper key rollover by R_i. It follows that the ancestors of R_{i+1} and R_i are the same (otherwise, the key rollover was not preformed properly and R_{i+1} is not considered a successor of R_i). The inductive step holds.
- Alice sees R_{i+1} overwrite R_i. If R_{i+1} has the same parent pointer as R_i, then claim follows by nested induction. We are left to deal with the case when R_{i+1} has a new parent pointer. It follows that either:
  - The parent of R_{i+1} is a successor of the parent of R_i because of a proper rollover of the parent's key. In this case, all other ancestors of remain the same, and the inductive step holds.
  - Otherwise, Alice must treat the overwrite of R_i as a deletion of R_i (per the "Overwritten RC Procedure" in Appendix B.2.4). Thus, Alice must raise an alarm blaming the parent of R_i, who is a successor of the of the parent of R (making the fourth condition of the theorem true) unless Alice sees a `.dead` object signed by R_i (making the third condition of the theorem true). However, earlier we established that neither the third nor the fourth condition hold for R_i, so this impossible.

□

D.2 Proof of Theorem 5.2.

Since R is not marked as stale, the hash of R is included in m. Let h be the hash of m. Because the global consistency check succeeds for Alice, we know that $h \in H_A$; recall that H_A is the set that contains the hash of each manifest that Alice obtained, going back for time interval t_g. By the collision resistance of the hash function, Alice must have seen manifest m at some point in the past. Since Bob saw R logged in m, by collision-resistance of the hash, Alice also saw R logged in m. If Alice did not obtain R, then she must have raised the missing information alarm for R, and the theorem is true. So now consider the case that Alice obtained R when she updated to m.

We argue that either Alice will raise an alarm for R during the validation of m, or Alice must have seen R as valid at some time before t_2. Let $L_{<m}$ be the state of Alice's local cache just before she starts updating to manifest m. We consider the following cases.

- Suppose the same RC R with the same URI is present in $L_{<m}$. Then it is marked as either:
  - valid, in which case R remains valid after the update and local consistency check of m (per Table 10 in the individual RC check in Section B.2.4); or
  - no-longer-valid or rolled-over, in which case at some previous time it must have been valid; or
  - never-was-valid, in which case, as part of verifying m, Alice must have ensured that R is valid, or else raised an alarm with R as the victim (see "New RC Procedure" for checking individual RCs in Appendix B.2.4).
- Suppose a different RC R' with the same URI was present in $L_{<m}$, or no RC with the same URI was present in $L_{<m}$. Then R was validated by Alice as part of verifying m (in "Overwritten RC Procedure" or "New RC Procedure" in Appendix B.2.4). Either Alice raised an alarm with R as the victim, or she must have found R to be valid.

Thus, either Alice raised an alarm for R during the validation of m (making theorem true), or R was valid for Alice when she concluded the local consistency check on m or before. Call the time when R was found valid by Alice t_1; note that $t_1 < t_2$. We can now apply Theorem 5.1. □

D.3 Proof of Theorem 5.3.

Let $m'_1, m'_2, \ldots, m'_k = m'$ be the predecessors of m' according to Alice. Since Bob did not raise invalid syntax alarms for $m_1, \ldots, m_k$, we know that $h(m_{j-1})$ was included in m_j, for $1 < j \leq k$. Because the global consistency check succeeded, by collision resistance of the hash, we know $m' = m_k$. Since Alice raised no invalid syntax alarms for m' or its $k-1$ immediate predecessors, we know $h(m'_{j-1})$ was included in m'_j, for $1 < j \leq k$. Thus, by induction on i from k down to 1, and by collision resistance of the hash, we get that $m'_j = m_j$ for $1 \leq j \leq k$, and, in particular, $m_i = m'_i$. The rest of the proof proceeds the same way as the proof of Theorem 5.2. □