Telematics
Computer Systems

Freie Universität Berlin

Bachelor of Science Thesis

Monitoring and Inspection of RPKI repositories

Andreas Reuter

Matr. 4569130

Supervisor: Prof. Dr.-Ing. Jochen Schiller
Dipl.-Inf. Matthias Wählisch

Institute of Computer Science, Freie Universität Berlin, Germany

May 19, 2015

I hereby declare to have written this thesis on my own. I have used no other literature and resources than the ones referenced. All text passages that are literal or logical copies from other publications have been marked accordingly. All figures and pictures have been created by me or their sources are referenced accordingly. This thesis has not been submitted in the same or a similar version to any other examination board.

Berlin, May 19, 2015

(Andreas Reuter)

Abstract

Abstract

The Internet routing infrastructure is vulnerable to various BGP attacks. To secure inter-domain routing against these threats the IETF working group SIDR has designed the RPKI. The RPKI is a public key infrastructure that provides a trusted mapping from IP prefixes to Autonomous Systems authorized to originate them. This mapping is published in distributed repositories. Relying parties lack an easy way of monitoring a RPKI repository and inspecting its content in detail. This thesis presents the design and implementation of (i) a distributed monitoring system for RPKI repositories and (ii) a web application for inspection of RPKI objects. It also explains the purpose and workings of the RPKI and discusses the structure of RPKI repositories.

Contents

List of Figures

List of Tables

Listings

CHAPTER 1

Introduction

1.1 Motivation

The Internet connects billions of devices across the world. It has become the most important means to distribute and obtain information and has led to the rise of whole new industries. With its enormous growth, the Internet has gained great importance for almost all branches of society and has been classified as critical infrastructure in several countries. As society comes to rely more and more on the Internet, the potential damage from attacks on it increases.

The Internet is a network of networks. It is made up of thousands of interconnected *Autonomous Systems* (AS), identified by their *Autonomous System Number* (ASN). Each AS owns a set of IP addresses, usually aggregated as an *IP prefix*, that are assigned to the devices in its network. Autonomous systems announce their IP prefixes together with their ASN to other AS using the *Border Gateway Protocol* (BGP). An AS receiving an announcement can add its own ASN to the announcement and propagate it further to other AS. The BGP protocol is built on trust, since by the time it was designed the Internet consisted of only a few cooperative AS. This means an AS receiving an announcement must trust that the AS that sent it legitimately owns the IP prefixes contained in the announcement. This makes BGP, and with it the Internet, an easy target for malicious AS that use false routing information to disrupt or divert traffic flow.

Attacks on Internet routing have become more frequent over the last few years. While the majority of them only affected prefixes with relatively little incoming traffic, there were some incidents involving popular prefixes that affected large numbers of Internet users. To prevent certain routing attacks the *Resource Public Key Infrastructure* (RPKI) was defined by the IETF. Once fully deployed, RPKI prevents AS from announcing IP prefixes they do not legitimately own.

1.2 Problem

The RPKI is a Public Key Infrastructure using extended X.509 certificates and Certificate Revocation Lists. It also defines multiple non-X.509 cryptographic objects, such as Route Origin Authorizations (ROAs) and Manifests. ROAs are used to bind IP prefixes to an ASN and signify ownership of these prefixes by that AS. All objects part of the RPKI are published in repositories and can be downloaded by relying parties.

There exist some tools to inspect the content of certain RPKI objects [1], they are however extremely limited. The purpose of this bachelors thesis is to implement a tool that allow the inspection and monitoring of RPKI repositories and their contents. The software should allow network operators to:

1. Download RPKI repositories.
2. Cryptographically validate all RPKI objects in a repository and show validation results.
3. Find specific RPKI objects.
4. Inspect the contents of RPKI objects in detail.
5. Display RPKI objects in relation to each other.
6. Give basic statistics about the repositories.

The security benefits provided by the RPKI also empower centralized authorities to *unilaterally* take down any IP prefixes under their control [2], given wide enough deployment of the RPKI. Tools like the one proposed in this thesis are needed to increase transparency and give RPKI participants an option to detect misbehavior by RPKI authorities.

Analysis of the BGP attacks the RPKI is meant to prevent, has shown the need for appropriate tools that can help detect common mistakes by network operators. This could lead to improved training or automated detection of misconfigurations [3].

1.3 Thesis Structure

Section 2 of this thesis explains the technical background necessary to understand the purpose and design of the RPKI. Firstly we explain the fundamentals of a Public Key Infrastructure and present the widely used X.509 standard, which is also the basis for RPKI. Secondly we explain the RPKI itself, presenting the different kinds of objects and its deployment scheme in detail.

In section 3 we discuss the requirements for the system and present its architecture. Section 4 deals with the implementation of system. Afterwards we evaluate the performance of the system and analyze an important RPKI deployment issue. We summarize our work and discuss further development.

CHAPTER 2

Technical Background

2.1 Internet Routing

At the core of the technology powering the Internet is the *Internet Protocol* (IP) [4]. It enables the delivery of data packets across interconnected computer networks. Two versions of the Internet Protocol are in common use, version 4 and 6. Each participating host is identified by an IP address with a length of 32-bit for IPv4 and 128-bit for IPv6. For simplicity's sake we will exclusively use IPv4 addresses, written as 4 8-bit integers separated by dots.

2.1.1 IP Prefixes and Autonomous Systems

An *IP prefix* describes a set of IP addresses that share a common prefix. As an example, the IP prefix 192.168.178.0/24 describes all IP addresses that share the first 24 bit of the IP address 192.168.178.0. In this example that includes all IP addresses from 192.168.178.0 to 192.168.178.255.

The Internet is made up of thousands of computer networks called *Autonomous Systems* (AS), identified by their unique 32-bit Autonomous System Number (ASN). An AS is a connected group of one or more IP prefixes run by one or more network operators which has a *single* and *clearly defined* routing policy [5]. IP prefixes and ASNs are managed by the Internet Assigned Numbers Authority (IANA) which allocates them to the 5 Regional Internet Registries (RIRs) listed in Table 2.1. The RIRs then further allocate the prefixes and ASNs they were given by IANA to National Internet Registries (NIRs), Local Internet Registries (LIRs), and other organizations such as corporations, academic institutions, and ISPs [6].

Registry	Region
AFRINIC	African
APNIC	Asia/Pacific
ARIN	North America
LACNIC	Latin America and some Caribbean Islands
RIPE NCC	Europe, the Middle East, Central Asia

Table 2.1: RIRs and their covered regions

2.1.2 Border Gateway Protocol

The *Border Gateway Protocol* (BGP) was first published in June 1989 in RFC 1105 titled "A Border Gateway Protocol" [7]. Its most recent version (BGP-4) was published in 2006 [8]. Each AS connected to the Internet operates at least one BGP speaking router. Routers of different AS exchange prefix reachability information in the form of BGP Update messages [8]. An AS can use BGP Update messages to announce to other AS which IP prefixes it originates. It can also announce IP prefixes that it doesn't originate, but can reach via a sequence of other AS, referred to as an AS path. The AS path is an attribute of BGP Update messages. It is a list of AS that a data packet will pass through in order to reach a certain prefix. Which prefixes an AS will announce depends on its peers and its routing policy.

When an AS receives an update which contains a new route, it can send this information to its other neighbors after prepending itself to the AS path. This propagation of routing information is essential for reachability between AS that are not immediate neighbors.

The BGP route finding process is based on trust. This means that there is no inherent mechanism that prevents an AS from sending BGP Update messages containing wrong information. A malicious AS can exploit this trust by announcing fabricated or altered BGP Updates. This can cause the traffic flow to be diverted. For this thesis, we classify these attacks into two categories.

Prefix Hijacks

A prefix hijack is an attack wherein an AS announces to be the origin for a prefix it does not actually own. This invalid announcement can cause other AS to drop their previous route to the target prefix in favor of a more preferable route which leads to the attacking AS. Generally speaking BGP routers will prefer to send packets to the shortest available route, unless some routing policy is in place that overrides this behavior. Thus a prefix hijack attack will cause some parts of the Internet to use the bogus route, diverting traffic away from the legitimate owner AS. The degree of acceptance of the bogus route depends on multiple factors like the Internet topology hierarchy and routing policies [9, 10] as well as the type of prefix hijack [11]:

1. *Regular prefix hijack*, where the attacker claims to be the origin of an existing prefix. In this case only some parts of the Internet will use the bogus route and some AS will

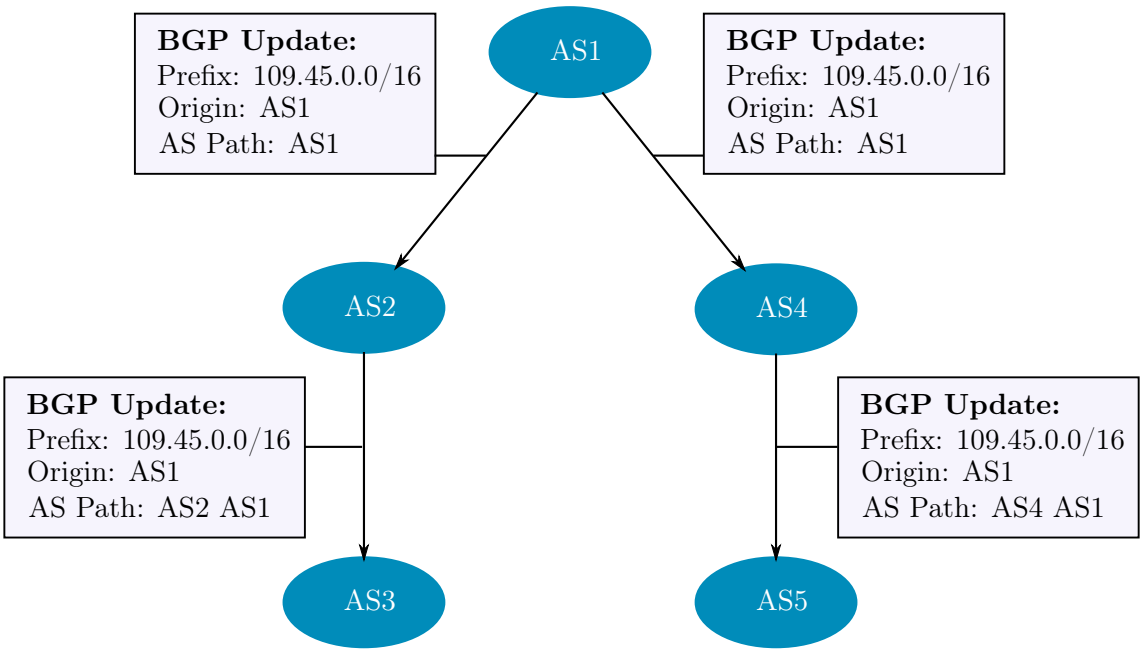


Figure 2.1: BGP Update message propagation: AS1 owns and originates 109.45.0.0/16. AS2 and AS4 prepend themselves to the AS Path before propagating the Update message.

still prefer the valid route.

2. *Subprefix hijack*, where the attacker claims to be the origin of a subprefix of an existing prefix. This will cause most of the Internet to accept the bogus route due to longest prefix matching.

In both cases of prefix hijacks once some or all of the traffic for the target prefix has been successfully diverted, the attacker now has multiple options. We classify these according to the convention in [12]:

Blackholing
The attacker simply drops the hijacked traffic.

Imposture
The attacker responds to the hijacked traffic, imitating the actual destinations responses. This can be used to obtain sensitive information.

Interception
The attacker becomes a man-in-the-middle by redirecting the traffic back to the victim AS. This allows recording and manipulating the hijacked traffic before it reaches its true destination. Note that in order to be able to forward the traffic to the legitimate destination, the attacker AS requires a valid route to the victim AS. This means that when performing the prefix hijack, the attacker has to make sure not to pollute the routing table of the AS needed for a route to the victim.

It is not clear how frequently these attacks occur. One of the most notable incidents occurred in 2008, when Pakistan Telekom (AS17557) started an invalid announcement of 208.65.153.0/24, a subprefix of YouTube's 208.65.152.0/22. This caused YouTube to be un-

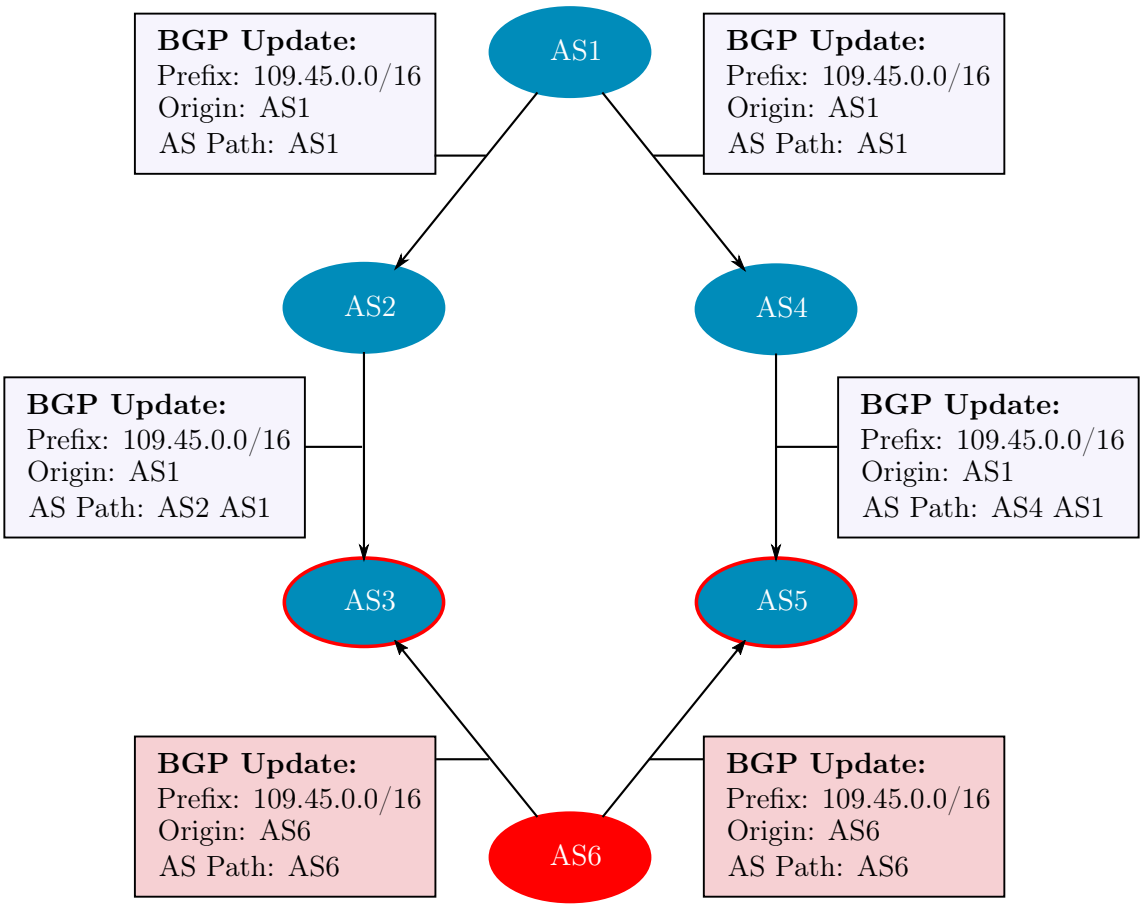


Figure 2.2: BGP Prefix Hijack attack: AS6 illegitimately announces 109.45.0.0/16 to AS3 and AS5. Since the AS Path to AS6 is shorter than to AS1, both AS prefer the bogus route to AS6.

available for large swaths of the Internet. A similar incident occurred in 2010 when China Telecom wrongly asserted ownership of a large number of prefixes. Both of these incidents have been attributed to misconfigurations of BGP routers.

AS path attacks

An AS path attack occurs when a BGP router announces a route with an unauthorized AS_PATH attribute in order to redirect traffic. The attacker can choose to divert the traffic away from its destined AS into its own network. Once the traffic has been successfully diverted to an AS of the attackers choosing, the attacker has the same options as in a prefix hijack.

2.2 Public Key Infrastructures

A public key infrastructure (PKI) is a set of technologies and procedures that, in its most basic form, enables entities to communicate with **confidentiality**, **integrity** and **authen-**

ticity. Public key infrastructures rely on *public key cryptography* to achieve these properties.

2.2.1 Public Key Cryptography

In Public Key Cryptography, also referred to as asymmetric cryptography, each communicating entity generates two keys referred to as *public key* and *private key*. These keys can be used to encrypt a message M into a ciphertext C. The mathematical relationship between the two keys ensures that the encrypt and decrypt operations are inverse functions of another and thus means that a message encrypted with the public key can be decrypted *only* with the private key and vice versa:

$$ENCRYPT(M, K_{pub}) = C \tag{2.1}$$
$$DECRYPT(C, K_{priv}) = M \tag{2.2}$$

and

$$ENCRYPT(M, K_{priv}) = C \tag{2.3}$$
$$DECRYPT(C, K_{pub}) = M \tag{2.4}$$

The private key must be kept safe and only accessible by its owner. The public key is published. For two communicating entities, here called Alice and Bob, this setup offers

Confidentiality
: Both Alice and Bob can be sure that only the other party can decrypt and read their messages. This is achieved by encrypting the message with the recipients public key. Once the message has arrived at the other end, they can use their private key to decrypt it. An adversary eavesdropping can not decrypt the message, providing they don't have access to the recipients private key.

Integrity
: Both Alice and Bob can be sure that the messages they receive from each other have not been altered in any way and have legitimately been sent by the other party. This is achieved by encrypting the message with ones own private key. Once the message has arrived at the other end, the recipient can use the senders public key to decrypt it. An adversary cannot fake or alter a message, providing they don't have access to the senders private key.

These two properties enable Alice and Bob to communicate securely. However they do not offer **authenticity**. In practice to achieve both confidentiality and integrity, the sender has to encrypt the plaintext message first with the recipients public key resulting in C_{conf}. As a second step the sender has to encrypt C_{conf} using their own private key, resulting in $C_{conf+integ}$. The sender can then safely send $C_{conf+integ}$ to the other party. Note that typically the second step involves not encrypting the entirety of C_{conf}. Instead a hash of the plaintext message is encrypted, resulting in a *digital signature* that is then sent along with C_{conf}. When communicating with Bob using this scheme, Alice can be sure that the conversation can neither be overheard nor manipulated by a third party. However, an adversary can still pose as Bob using their own public and private key. The message exchange would be perfectly secure, but Alice would have no way of knowing that she is not

actually talking to Bob. There is no inherent way of verifying the identity of the public key owner one is communicating with. This is what is meant with a lack of authenticity. As an example, an adversary could mount a man-in-the-middle attack and imitate a users bank website. From the victims point of view, the connection to the bank website is perfectly secure. However, entering their online banking credentials would be fatal. A public key infrastructure is one way of solving this authentication problem.

2.2.2 Roles in a Public Key Infrastructure

At its core, in order to achieve authenticity a public key infrastructure associates a public key with its owner. This gives users of public keys assurance that the corresponding private key is owned by the correct entity. This assurance is obtained by the use of *public key certificates*, also known as *digital certificates* [13]. Public key certificates are digitally signed data structures that contain some representation of an identity and a corresponding public key [14, Chapter 3]. In every PKI, there exist at least one *self-signed* certificate, also sometimes referred to as a *trust anchor*. Every entity participating in the PKI implicitly trust the owners of these self-signed certificates. The trust anchor can be used to issue more certificates, signing them with the corresponding private key. Using their private keys, the owners of these certificates can then in turn also issue certificates. Note however that not all certificates can be used to issue other certificates. An entity holding a certificate capable of signing other certificates is called a *Certificate Authority* (CA) and those certificates are sometimes referred to as *CA certificates*. CAs can have different methods of verifying an entities identity before issuing them a certificate. The content and the integrity of a certificate can be verified by using the CAs public key on the digital signature in the certificate. In a PKI there also needs to be a revocation mechanism in place that allows issued certificates to be invalidated. Another element of a PKI is a *repository* which gives communicating entities means to locate each others certificates. Some PKIs consist of more components. For instance, some choose to have a separate *Registration Authority* (RA) take care of verifying and accepting requests for certificates [15].

2.2.3 X.509

X.509 is a standard published by the ITU Telecommunication Standardization Sector (ITU-T). It defines a public key certificate framework to implement PKI-based public-private key security [16]. In 1995 the Public Key Infrastructure X.509 (PKIX) IETF working group was established with the goal of adapting the X.509 Standard to the Internet environment and defining a Internet Public Key Infrastructure (IPKI) [17]. The PKIX working group assumes a PKI model made up of the following components:

End Entity
: A certificate user and/or end user system that is subject of a certificate.

Certificate Authority (CA)

Registration Authority (RA), optional

CRL Issuer
: A system that issues Certificate Revocation Lists (CRLs). These are signed

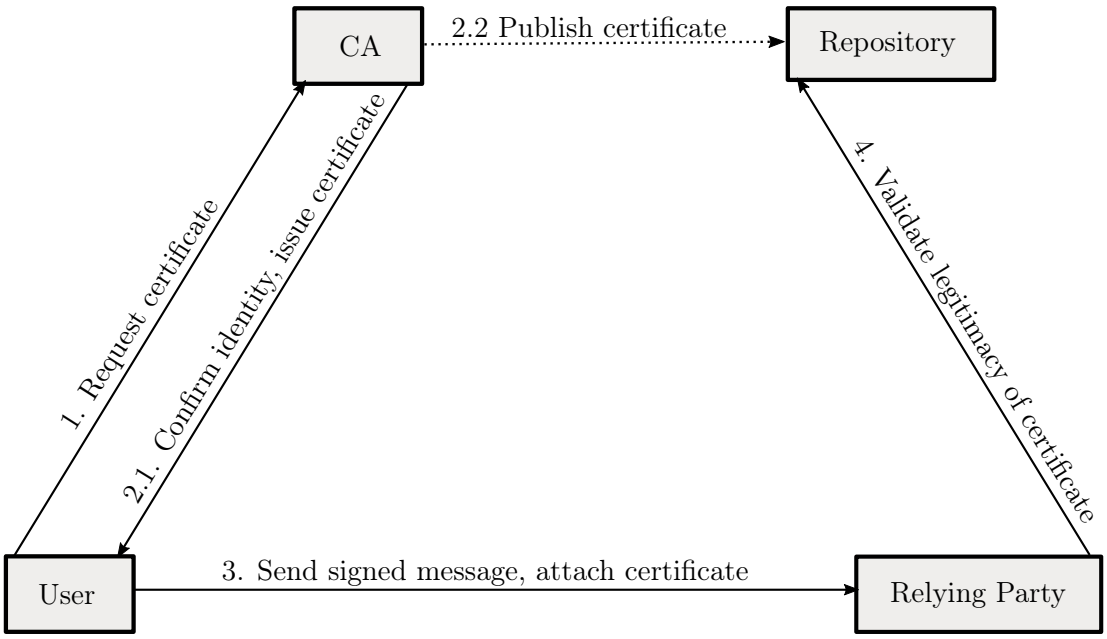


Figure 2.3: PKI user authentication example

objects used to revoke certificates. This can be a Certificate Authority.

Repository
A structure that holds certificates and CRLs and allows End Entities to access them.

X.509 certificates

The PKIX working group defines the profile for the use of X.509 certificates within Internet applications [13]. It is designed to grant interoperability and reusability. Certificates are defined using the ASN.1 syntax standard. The X.509 certificate format consists of these basic fields:

- **Version Number**
  Defines the X.509 version of the certificate. The current version is 3.
- **Serial Number**
  Number assigned by the issuer. The numbers are unique for each certificate issuer.
- **Signature Algorithm ID**
  Contains the identifier for the algorithm used to sign this certificate.
- **Issuer Name**
  Identifies the CA that issued the certificate. This is normally a hierarchical sequence of strings.
- **Validity Period**
  Defines the time period in which this certificate is valid. Consists of a not-before date and a not-after date.

- **Subject Name**
  Identifies the entity associated with the public key stored in the certificate. If this entity is a CA, the subject field must match the contents of the issuer field in all certificates issued by this CA.
- **Subject Public Key**
  Holds the certificates owner public key and identifies the algorithm with which the public key is used.

In addition to these basic fields, there are are some optional fields:

- **Issuer Unique Identifier**
  Allows two issuers to have the same issuer field value, as long as the issuer unique identifiers are different.
- **Subject Unique Identifier**
  Allows two subjects to have the same subject field value, as long as the subject unique identifiers are different.

With the introduction of version 3 of X.509 certificates, the option of adding arbitrary extension was added. An extension consists of an Object Identifier (OID) and a ASN.1 structure and can only occur once within a certificate. This allows communities to create extensions that are tailored to carry specialized information that may be unique to the communities domain. Extensions can be marked *critical*. This means that if a certificate-using system encounters a certificate with a critical extension that it does not recognize, it must reject the certificate. A non-critical unrecognized extension will be ignored.
The PKIX working group has defined some extensions that every conforming CA must support:

- **Basic Constraints**
  Defines whether the subject of this certificate is a CA or not. Only public keys contained in CA certificates can be used to verify certificate signatures.
- **Key Usage**
  Defines the purpose of the public key contained in this certificate. As an example, a key might only be used for key management and shall never be used to verify signatures.
- **Authority Key Identifier**
  This field is used to identify the public key associated with the private key that was used to sign this certificate. This is used when an issuer has multiple signing keys.
- **Subject Key Identifier**
  This field is used to identify certificates containing a particular public key. If the subject of this certificate is a CA, all certificates issued must have the CAs Subject Key Identifier value in their Authority Key Identifier field.
- **Certificate Policies**
  This field defines the set of policies which are adhered to during the certificate creation process. A certificate policy is defined by the X.509 Recommendation [X.509-00] as

  > *a named set of rules that indicates the applicability of a certificate to a particular community and/or class of application with common security requirements.*

In addition to those, there are some optional extensions that are necessary to explain in order to fully understand the RPKI:

- **CRL Distribution Points**
  This field defines how entities can obtain CRL information on this certificate.
- **Authority Information Access**
  This field defines the location and access methods for services and information about the issuer of this certificate.
- **Subject Information Access**
  This field defines the location and access methods for services and information about the subject of this certificate.

Certificate Revocation

In some circumstances a CA may wish to invalidate a certificate before the end of its validity period. This may be for example because the corresponding private key was compromised or the subject name changed. In the X.509 standard, certificates are revoked using a certificate revocation list (CRL). CRLs are signed objects that are periodically published by the CA. They contain a list of the serial numbers of unexpired revoked certificates and their revocation time. The CRL fields defined by the IETF are:

- **Signature Algorithm ID**
  Contains the identifier for the algorithm used to sign this CRL.
- **Issuer Name**
  This field identifies the entity that issued this CRL.
- **This Update**
  Indicates the time when this CRL was issued.
- **Next Update**
  Indicates the time when the next CRL will be issued at the latest.
- **Revoked Certificates**
  This field lists revoked certificates by their serial number and revocation date. It is optional and is not present if there are no revoked certificates.

Analogous to certificates, communities have the option the define private CRL extensions.

Certificate Validation

The goal of certificate validation is to verify the binding of the subject of the certificate to the public key contained in the certificate. A certificate C_n is considered to be issued by certificate C_{n-1} if the content of C_n's issuer field matches the content of C_{n-1}'s subject field. In order for a certificate C_n issued by a certificate C_{n-1} to be considered valid, these two conditions need to be met:

1. *Basic certificate processing* succeeds. This involves these steps:
   a) Verify the signature on C_n with the algorithm indicated in the signature algorithm

ID field of C_n using the public key found in C_{n-1}.

b) Verify that the validity period of C_n includes the current time.

c) Verify that C_n has not been revoked at the current time.

d) Verify that the contents of the issuer field in C_n match the subject field in C_{n-1}.

2. There is a path $\{C_1, C_2, ...C_{n-1}, C_n\}$ of certificates in which for all pairs (C_{i-1}, C_i) basic certificate processing succeeds. At the start of the path lies a trusted certificate, also called a *trust anchor*.

Note that communities can augment the basic algorithm and implement stricter checks that may be more appropriate for their domain.

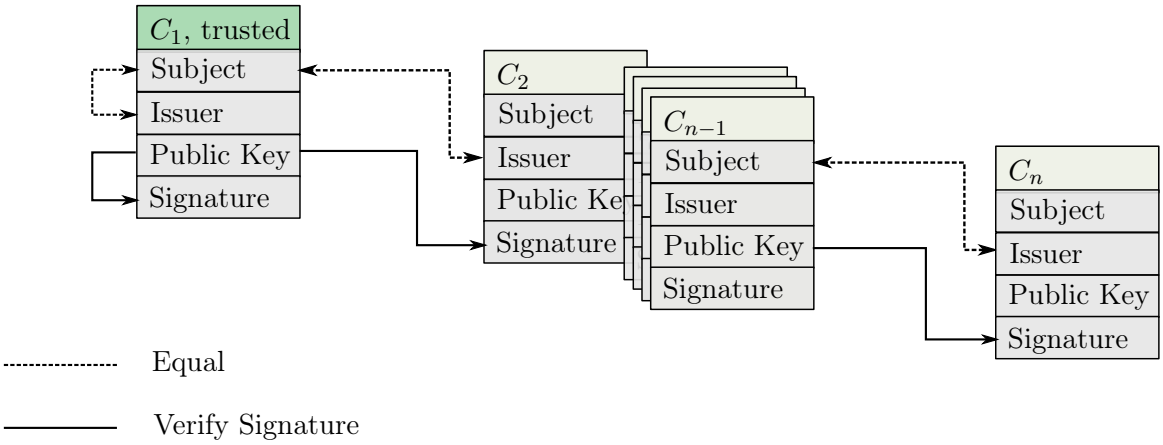


Figure 2.4: Simplified illustration of Certificate Path Validation

2.3 Resource Public Key Infrastructure

The RPKIs goal is to bind Internet Number Resources, defined as Autonomous System Numbers (ASNs) and IP addresses, to their legitimate owner. It is a X.509 PKI whose structure mirrors the allocation of Internet Number Resources by IANA. The RPKI also allows a prefix owner to authorize one or more AS to announce that prefix [18]. That authorization mechanism serves to detect and prevent prefix hijack attacks discussed in 2.1.2, however it offers no protection against AS path attacks. In order to secure the AS path attribute, work has begun on a BGP extension called BGPSEC [19]. The extension defines a new BGP Update message format, which contains the *BGPsec_path* attribute instead of the as_path attribute of regular update messages. The BGPsec_path attribute carries a digital signature for every AS that propagated the update message. This allows BGP speakers to validate an updates message AS path using the public keys of the AS seen in the path. BGPsec will use the RPKI structure to bind ASN to public keys. The corresponding private keys will then be used by BGP speakers to sign the path attribute.

2.3.1 Resource Certificates

The certificates used in the RPKI are called *Resource Certificates*. In this context a resource refers to an IP address or ASN. Resource certificates have, in addition to all the basic fields and extensions of X.509 certificates presented in 2.2.3, two critical extensions:

- **IP Resources**
  This field contains a set of IP addresses. This extension may also specify whether certain IPs are to be inherited from the issuing certificate.
- **AS Resources**
  This field contains a set of ASN. This extension may also specify whether certain ASN are to be inherited from the issuing certificate.

In a well formed resource certificate at least one of these extensions needs to be present. A certificates IP and AS resources must always be a subset of the resources held by the issuing certificate. This is because IP address space and ASN are allocated in a hierarchical manner.

In contrast to a normal PKI, the subject names in the certificates do not attest the identity of the public key owner. This is because the purpose of the RPKI is not to bind a public key to an identity, but to bind it to a set of resources.

For repository access the RPKI uses the open source file transfer protocol *rsync* [20]. There are three important fields containing rsync URIs in a resource certificate [21] [22], shown in Listing 2.1. These fields were presented as part of the X.509 standard already, but defined in a rather general and vague way.

- **CRL Distribution Points** (Line 44-47)
  The location of the CRL associated with the issuer of this certificate. In other words, the location of the CRL which could be used to revoke this certificate. This field must be omitted in self signed resource certificates.
- **Subject Information Access** (Line 38 - 41)
  The location of the directories which contains all signed objects published by this certificate authority. These directories are also referred to as *publishing points*.
- **Authority Information Access** (Line 54 - 56)
  The location of the directory in which the issuing certificate was published. This field must be omitted in self signed resource certificates.

```
1  Certificate:
2      Data:
3          Version: 3 (0x2)
4          Serial Number:
5              f2:cc:d6:8e:10:11:91:59:7d:c6:07:55:73:a6:1c:a0:03:cc:cc
6      Signature Algorithm: sha256WithRSAEncryption
7          Issuer: CN=2a246947-2d62-4a6c-ba05-87187f0099b2
8          Validity
9              Not Before: Dec 10 15:45:00 2013 GMT
10             Not After : Dec 10 15:45:00 2023 GMT
11         Subject: CN=f0421b26-017d-4996-83ed-00d627f7fd53
12         Subject Public Key Info:
13             Public Key Algorithm: rsaEncryption
14                 Public-Key: (2048 bit)
15                 Modulus:
16                     00:b5:fa:ee:1d:53:ae:65:14:88:03:07:fa:47:af:
17                     8c:02:64:47:64:17:29:8d:71:8b:93:a0:9d:4e:2b:
18                     ee:b7:2c:1a:c1:4d:51:df:0e:80:85:dc:01:d5:82:
19                     1d:65:4b:ab:cf:3e:20:cc:19:57:34:a7:91:88:25:
20                     dd:e9:1a:97:e1:57:90:b1:69:af:8e:e8:b9:8b:2f:
21                     8f:cf:60:d3:18:1c:c4:9c:5b:e1:f2:11:6c:f5:bf:
22                     2f:5d:6b:2f:75:da:7e:50:65:f8:4a:f1:b0:4b:ef:
23                     f9:02:4d:e7:e3:b6:8c:2d:fd:61:1c:3f:d1:f9:65:
```

```
24                      2d:0f:15:28:23:99:55:8c:80:8f:6f:31:87:8a:13:
25                      2e:b7:81:84:1b:43:9c:f1:c8:0f:a5:dd:88:4b:22:
26                      79:55:aa:bc:70:22:79:79:53:81:3e:2b:f7:89:9d:
27                      f0:92:60:ca:51:83:5d:92:be:9d:6e:1f:44:7d:d2:
28                      7d:f7:3e:41:f4:e1:21:e8:e0:d9:d3:15:cb:e8:2e:
29                      9f:b8:56:83:17:82:b6:ad:9a:75:88:5c:ee:af:23:
30                      3b:4d:7a:53:a1:2d:85:c7:6b:04:07:04:14:2f:7c:
31                      24:e0:8d:8c:b6:8d:b0:6e:d8:71:ca:36:f5:eb:59:
32                      91:08:a3:7d:d5:72:28:82:ce:a5:b7:09:cc:29:0c:
33                      1c:cd
34                  Exponent: 65537 (0x10001)
35          X509v3 extensions:
36              X509v3 Subject Key Identifier:
37                  FA:18:B4:F7:9A:06:E3:B0:11:99:6B:CC:23:19:7A:04:B0:39:36:A3
38              Subject Information Access:
39                  CA Repository - URI:rsync://rpki.arin.net/repository/arin-rpki-ta/5e4a23ea-
                        e80a-403e-b08c-2171da2157d3/2a246947-2d62-4a6c-ba05-87187f0099b2/f0421b26
                        -017d-4996-83ed-00d627f7fd53/
40                  1.3.6.1.5.5.7.48.10 - URI:rsync://rpki.arin.net/repository/arin-rpki-ta/5
                        e4a23ea-e80a-403e-b08c-2171da2157d3/2a246947-2d62-4a6c-ba05-87187f0099b2/
                        f0421b26-017d-4996-83ed-00d627f7fd53/f0421b26-017d-4996-83ed-00d627f7fd53.
                        mft
41
42              X509v3 Basic Constraints: critical
43                  CA:TRUE
44              X509v3 CRL Distribution Points:
45
46                  Full Name:
47                    URI:rsync://rpki.arin.net/repository/arin-rpki-ta/5e4a23ea-e80a-403e-b08c
                          -2171da2157d3/2a246947-2d62-4a6c-ba05-87187f0099b2/2a246947-2d62-4a6c-
                          ba05-87187f0099b2.crl
48
49              X509v3 Authority Key Identifier:
50                  keyid:C8:9D:5A:45:64:1A:6B:D2:23:FA:CA:96:82:30:8E:D6:D2:76:AD:7C
51
52              X509v3 Key Usage: critical
53                  Certificate Sign, CRL Sign
54              Authority Information Access:
55                  CA Issuers - URI:rsync://rpki.arin.net/repository/arin-rpki-ta/5e4a23ea-e80a
                        -403e-b08c-2171da2157d3/2a246947-2d62-4a6c-ba05-87187f0099b2.cer
56
57              sbgp-ipAddrBlock: critical
58                  IPv4:
59                    199.66.236.0/22
60
61              sbgp-autonomousSysNum: critical
62                  Autonomous System Numbers:
63                    29834
64
65              X509v3 Certificate Policies: critical
66                  Policy: 1.3.6.1.5.5.7.14.2
67                    CPS: https://www.arin.net/resources/rpki/cps.html
68
69      Signature Algorithm: sha256WithRSAEncryption
70           7b:a2:84:c7:1a:62:69:91:f1:5c:44:12:ec:12:f0:53:b1:0d:
71           f2:9a:62:97:bd:d3:80:e2:96:13:a3:df:68:8e:7f:3f:c4:6e:
72           b3:99:da:d6:83:89:e8:37:3e:21:1a:b9:0a:db:e5:7f:d0:7b:
73           42:c3:2b:0f:85:68:a0:4d:0e:22:b3:eb:85:2d:b5:2e:b2:55:
74           3c:f0:5e:41:00:6b:41:d8:eb:f1:91:97:6b:27:72:ba:05:59:
75           0c:db:00:ae:b7:9f:c8:e6:1d:2b:6d:6d:65:14:c0:37:aa:ff:
76           6a:a7:c1:16:4f:ce:93:9d:4b:d0:d8:ac:80:17:27:d4:85:42:
77           5e:3d:03:57:96:6c:38:a7:1e:b8:55:c1:6f:01:1c:7d:85:46:
78           10:d5:41:88:69:3a:a8:32:00:50:c6:ec:91:7b:36:ba:a1:d8:
79           00:87:42:66:06:42:d2:d4:95:b2:2c:1d:5b:34:90:f8:cc:c6:
80           a3:61:67:32:15:ba:06:b9:54:d4:cd:5c:88:18:df:4b:e3:f0:
81           ba:7a:c8:d0:28:89:b6:ba:85:eb:c4:cc:79:96:0c:1b:ff:31:
82           a8:2d:16:9d:a1:0e:00:95:ee:55:ed:1d:d9:3b:1b:56:4d:2f:
83           b4:55:de:0a:ef:f8:8d:a6:da:9f:4d:73:85:39:99:ec:ac:7a:
84           51:a0:2b:07
```

Listing 2.1: Example Resource Certificate

These fields can be used to easily build up a path of certificates $C_1, C_2, ..., C_n$ for validation and access CRL information, as described in section 2.2.3. Certificate validation in the RPKI involves one additional condition to be met during basic certificate processing:

e) Verify that the contents of the resource fields in C_n are fully encompassed by the resources held by C_{n-1}

This step enforces the hierarchical structure of resource ownership. Each of the five RIRs holds a self signed resource certificate used as a trust anchor in the global RPKI. The resource fields of a RIR trust anchor contain all IP prefixes and ASNs that were allocated

to that RIR by IANA, or have been transferred from another RIR. Hence a trust anchor can only be used to validate certificates which hold subsets of its resources. In practice this means in order to validate a certificate holding a resource allocated by RIPE, one needs to use the RIPE trust anchor certificate.

In order for relying parties to be able to access a trust anchor, RIRs publish a *Trust Anchor Locator* (TAL). A TAL is formatted data that can be used to retrieve and verify the authenticity of a trust anchor [23]. It contains the URI of the trust anchor as well as its public key. This setup gives RIRs operational flexibility. However, since a TAL does not offer any assurances about the identity of the referenced trust anchors owner, relying parties should have great confidence in the issuers of the trust anchor they are using [23].

Resource certificates establish a binding between a set of resources and a public key. However, this by itself is not sufficient information for a BGP router to recognize illegitimate announcements with. To allow prefix owners to authorize any AS of their choosing to announce their prefixes, additional functionality is needed. This comes in the form of RPKI signed objects called Route Origin Authorizations (ROAs).

2.3.2 RPKI Signed Objects

In this context, a signed object refers to a digitally signed, non-X.509 data structure that conforms to the Signed Object Template for RPKI [24] and uses the Cryptographic Message Syntax (CMS) as its standard encapsulation format. The signed object template is designed to be extendable.

One of the template fields, labeled *certificate*, is of particular interest because it pertains to the validation and revocation of RPKI signed objects. It contains the resource certificate that was used to sign the RPKI signed object. This certificate is called a *End-Entity (EE) certificate*. Unlike CA certificates, an EE resource certificate can not be used to issue other resource certificates. Its exclusive purpose is to sign and verify a single signed object, this means there is a one-to-one correspondence between end-entity certificates and signed objects [18]. This setup allows the revocation of signed objects by revoking the end-entity certificate that was used to sign it.

2.3.3 Route Origin Authorization

A ROA is a RPKI signed object that a prefix owner can issue in order to authorize an AS to originate a set of his prefixes [25]. In addition to all fields defined in the signed object template, a ROA also consists of these extension fields:

- **AS ID**
  Contains the ASN of the system which is given authorization.
- **IP address blocks**
  Contains the IP address blocks that this ASN is being authorized to originate. In this context an IP address block is an IP prefix coupled with a maximum prefix length. The maximum prefix length specifies how long the most specific prefix that the ASN is authorized to originate is allowed to be.

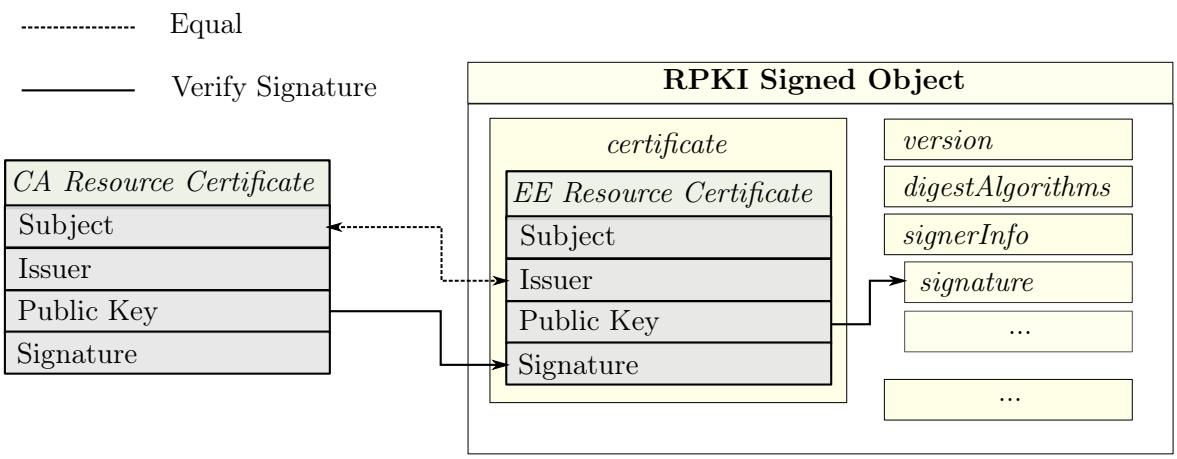


Figure 2.5: RPKI Signed Object structure: The RPKI Signed Object can be invalidated by revoking its EE resource certificate.

An example ROA might contain the following:

Prefix	167.120.0.0/22
Max. Length	24
ASN	1678

This means the AS with ASN 1678 is allowed to originate the prefix 167.120.0.0/22 and any of its /23 and /24 subprefixes, but not /25 or higher.

ROA validation

These conditions need to be met in order for a ROA to be considered valid:

1. The ROA needs to be a well formed RPKI signed object, i.e. conform to the RPKI signed object template.
2. The EE resource certificate contained within the ROA needs to be valid.
3. Each of the IP prefixes in the ROA needs to be contained within the IP resources extension of the EE certificate.

The information contained within ROAs is what BGP routers within the RPKI will base their routing decisions on. However, it is not feasible for a router to download all published ROAs periodically and store them as signed objects.

2.3.4 Deployment

The RPKI needs to offer routers a simple and reliable mechanism to access the information within ROAs. Since downloading, validating and storing the resource certificates and ROAs is not feasible for BGP routers, this function is delegated to *Local Caches*. Overall the RPKI deployment scheme is three-layered and consists of these components [26]:

The Global RPKI

Consists of the entirety of all RPKI repositories. Those can either be *hosted*, which means they are managed by a RIR, or organizations like LIRs or ISP can choose to manage their own RPKI. This is referred to as *delegated*. In this context a repository means the set of all objects, such as resource certificates, CRLs and ROAs, that are currently published by an organization.

Local Caches

A periodically refreshed cache of the *validated* global RPKI. Also referred to as *Relying Parties* (RPs)

Routers

BGP routers that fetch data from local caches. Note that a router or other relying parties fetching data from a local cache needs to have a trusted relationship with that cache.

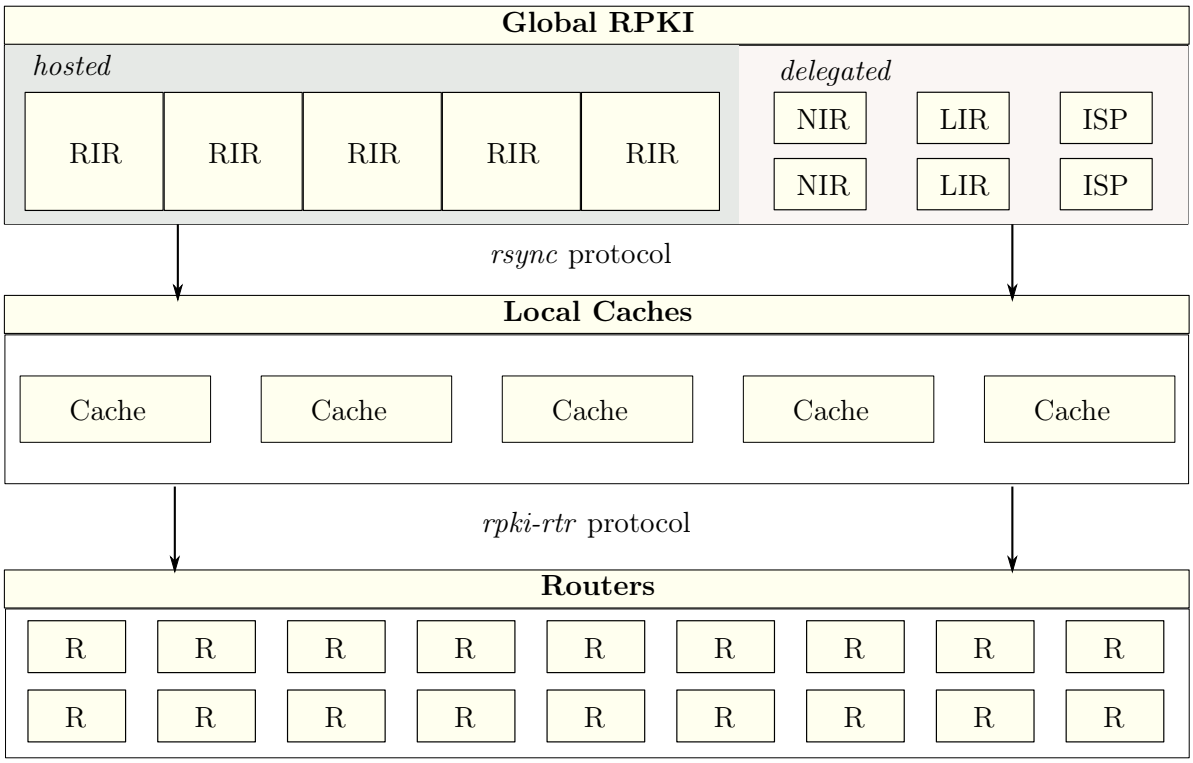


Figure 2.6: RPKI deployment structure

The 3-layered setup as presented here offers a simple and reliable mechanism for routers to access the RPKI information they need. However, an adversary can still attack the deployment structure itself in an attempt to manipulate BGP router behavior. Since all objects contained in the global RPKI are digitally signed, any modification would be easily detectable. But the attacker could still remove objects from a repository or replace them with a "stale" version. "Stale" in this context meaning an older but still valid version of the object. To defend against such attacks, publishers make use of signed objects called *manifests* [27].

Manifests

A manifest in the context of RPKI is a signed object which, like ROAs, conforms to the template for RPKI signed objects [24]. This means it is signed by the end-entity certificate that it contains and can thus be revoked by revoking that certificate. Similar to CRLs, manifests are issued periodically and contain the date of the next expected update. A manifest is issued by a CA and contains a list of all objects that CA has published at a particular publishing point (CAs may have multiple publishing points and consequently multiple manifests). The list contains file name and hash value for each published object such as resource certificates, ROAs, and CRLs. This allows a relying party to detect whether an object is stale (wrong hash) or missing (not in manifest).

CHAPTER 3

System Design

This chapter deals with the design of our software system: *Monitoring and Inspection of RPKI Objects* (MIRO). In the first section, we present the requirements which we have chosen to seperate into the *functional* and *non-functional* categories. We also discuss architectural decisions, derived from the requirements.

Sections 3.2 and 3.3 deal with the components of the system in more detail.

3.1 Requirements

The goal of the MIRO software system is to allow users to monitor and inspect RPKI objects. In this context, RPKI objects refer to Resource Certificates, Route Origin Authorizations (ROAs), Certificate Revocation Lists (CRLs), and Manifests (MFTs) found in RPKI repositories. The separate types of RPKI objects are logically linked to each other in the following way:

The resource certificates contain three fields that give information about the location of related RPKI objects:

1. Subject Information Access: This field contains the location of the resource certificate's publishing points. These publishing points are directories that must contain all objects issued by this resource certificate.
2. Authority Information Access: This field contains the location of the resource certificate that issued this resource certificate.
3. CRL Distribution Points: This field contains the location of the CRLs which can be used to revoke this resource certificate.

All locations are given in the URI format [28]. In the context of MIRO, we call the collection of RPKI objects that can be "reached" via the URIs contained in those fields starting at a given resource certificate a *logical repository*. A logical repository for a given resource certificate includes all RPKI objects issued by that certificate as well as all objects issued by those objects. The three fields presented above can be used to build up a complete logical repository from any given resource certificate. In the case of a self-signed certificate (trust

anchor), one can obtain all issued objects by following the URIs in the Subject Information Access field. This process can then be repeated recursively on any resource certificates found in the publishing points to obtain the complete logical repository. Given a non self-signed resource certificate, this process can simultaneously be used in the "reverse direction" by following the URI in the Authority Information Access field.

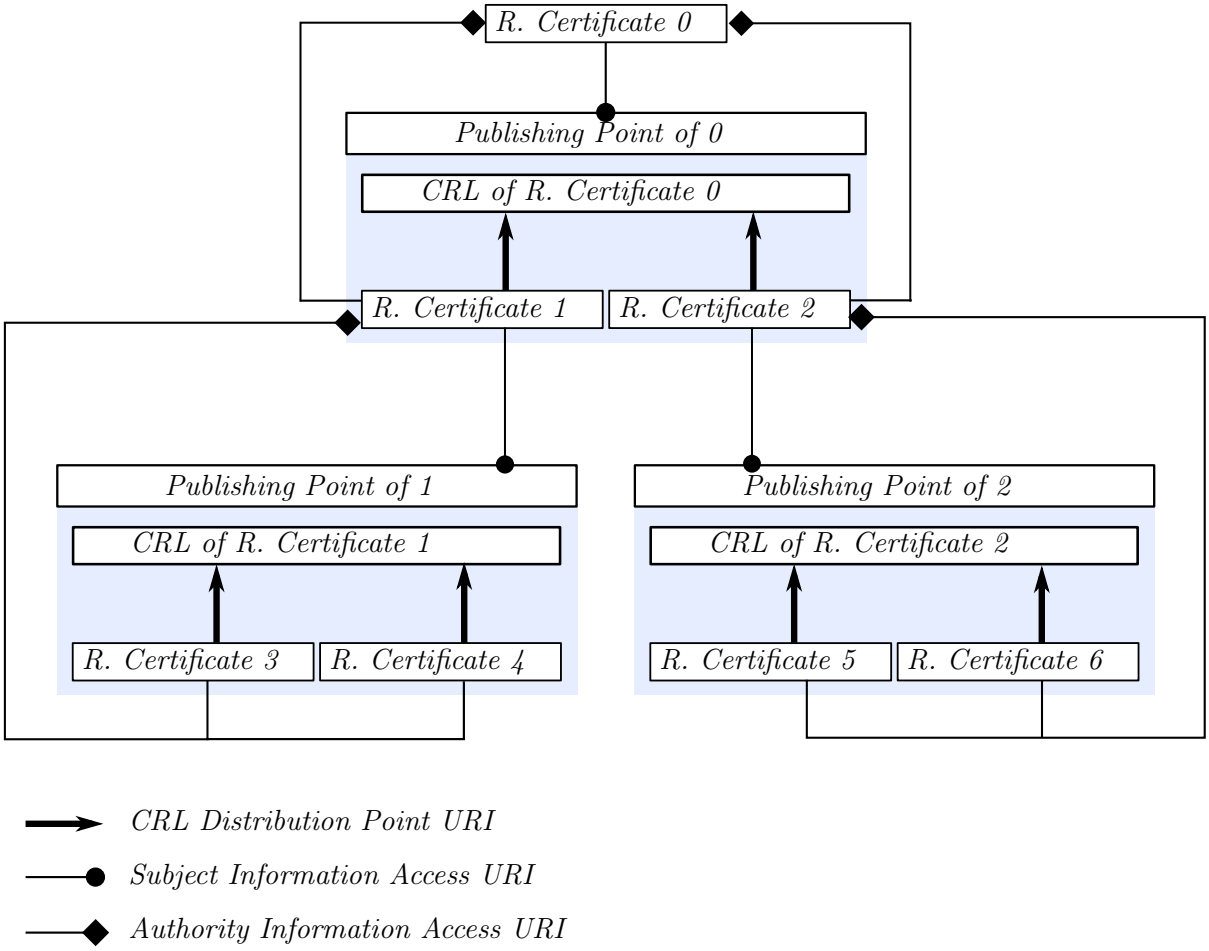


Figure 3.1: Logical structure of a RPKI repository: Resource certificate 0 is the trust anchor

3.1.1 High-Level Functional Requirements

The core functional requirement of MIRO is offering users an intuitive way to inspect RPKI objects , grouped by for example logical repositories, and browse through them using a graphical user interface (GUI). The GUI should mimic the tree structure of the logical repositories and display all information relevant to a given RPKI object including validation results. It should also offer filter functionality, which gives the user the opportunity to search for specific RPKI objects by field content.

The primary inputs for the system are self-signed trusted resource certificates, from here on called trust anchors. Starting from a trust anchor, its logical repository must be downloaded

and each RPKI object validated before it is passed to the GUI. The process of downloading, validating, and displaying a logical repository must be dynamically triggerable by other processes. This allows for periodic, but also on-demand, updating of repositories. The functionality to update repositories is very important since all RPKI objects validity is time dependent. The system should also offer a way to dynamically and seamlessly add or remove repositories during run-time. It should also allow for distributed monitoring to be able to detect inconsistencies in the global RPKI.

In addition to the above, basic statistics about validated repositories should be gathered, archived and displayed to the user.

3.1.2 Non-Functional Requirements

The system should be a web application to make it easily accessible to users. Installation, deployment, and configuration must be simple in order to allow owners of delegated and private RPKIs to use it to monitor their own setups. It should be robust and deal with invalid inputs and unforeseen failures while downloading, parsing, validating, and displaying RPKI objects.

In anticipation of the growth of the global RPKI, the system needs to be scalable and offer good performance in downloading, validating, and displaying substantial quantities of data.

It is possible that in the future more types of RPKI objects will be added such as BGPSEC router certificates used for AS path validation [29]. Thus the design of the system must be able to accommodate these additions using loosely coupled, modular components.

3.1.3 Architecture

From the requirements of the system emerges a clear distinction of tasks. One group of tasks deals with collecting, parsing, and validation of RPKI objects, another with displaying this data to the user. These task can be classified within a Model-View-Controller pattern:

Model: Logical repositories and their content. This also includes basic statistics about the repositories.

Controller: Downloading, parsing, validating, and updating the model. This also includes gathering and archiving basic repository statistics.

View: Displaying the model to the user.

We decided to split the system in two separate components. Firstly a *Validator* which corresponds to the controller in the MVC pattern. It also includes the definition of the model. Secondly a *Browser*, which represents the view part of the MVC pattern and displays the model passed to it by the Validator.

3.2 Validator

The Validator component needs to handle the tasks of downloading, parsing, validating, gathering statistics about, and exporting logical repositories. We call this sequence of task

our *processing chain*. As its input the Validator takes trust anchors, its output is the validated repository and basic statistics about its content. This section presents each step in the processing chain.

3.2.1 Downloading and Parsing

Downloading and parsing of RPKI objects must occur intermittently. This is because the publishing points for a certificate are stored in the subject information access field of the certificate. Because of this in order to download the objects issued by a given certificate, one needs to parse this field first.

The process for a repository starts with parsing the trust anchor and fetching all directories found in its subject information access field. Note that these directories may also be used as publishing points by other resource certificates. Hence, one cannot assume that every object in those directories was issued by the trust anchor.

For each CA resource certificate issued by the trust anchor found in the downloaded publishing points, the process is recursively repeated. The following pseudo code shows an algorithm that, when called with a trust anchor, downloads the complete logical repository:

```
1     def fetchIssued(resourceCert):
2         for pubPoint in resourceCert.subjectInformationAccess:
3             dir = downloadDir(pubPoint)
4             for obj in dir:
5                 if obj.issuer == resourceCert.subject:
6                     addToRepository(obj)
7                 if obj.type == CA_ResourceCertificate:
8                     fetchIssued(obj)
```

Listing 3.1: Fetch algorithm for a logical repository

Being able to fetch an entire repository only using the trust anchor as the starting point is an elegant and simple solution. However, the efficiency of this method depends on the structure of the logical repository. Consider a repository whose certificates publishing points are arranged hierarchically. This means the publishing points of a certificate would contain all of its issued certificates publishing points. As illustrated in Figure 3.2, this repository structure will be downloaded fully by the first call to fetchIssued(). Further calls could safely skip the downloading of publication points if we add a check that indicates if the publishing point was already downloaded previously.

It is also possible to structure a repository in a way that forces us to download more than just the publishing points of the trust anchor. In fact, the worst case would be a repository structure whose publishing points are arranged in a completely non-hierarchical manner with no publishing point containing any other publishing points, illustrated in Figure 3.3. This would cause every publishing point to be downloaded separately causing massive overhead as for each point a connection has to be built up and closed down. The structure of a repository can make an enormous difference. In October 2011, RIPE changed their repository structure to be completely hierarchical. The effects on the number of connections needed to fetch the repository can be seen in Figure 3.4.

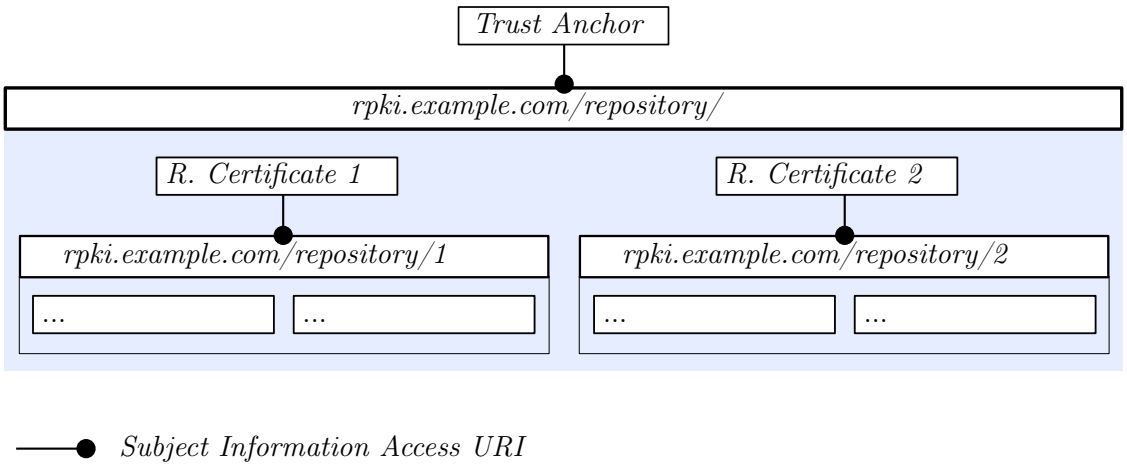


Figure 3.2: Hierarchical repository structure: This repository can be downloaded completely by fetching the trust anchors publishing point

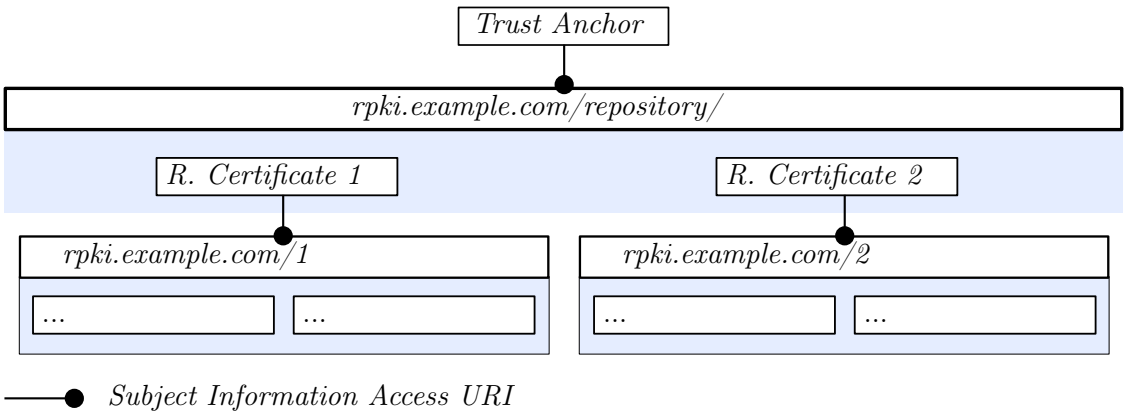


Figure 3.3: Flat repository structure: All three publishing points need to be downloaded separately

Prefetching

In order to minimize the amount of separate rsync calls, we make use of so called *Prefetching*. Prefetching means that before doing anything else we download as many publishing points as possible in bulk, which otherwise would have been downloaded using many individual calls. This saves us the overhead cost of opening and closing a connection for each call. As an example, consider that we are downloading a non-hierarchical repository and find these publishing point URIs:

Certificate	**Publishing point**
certificate1	rsync://rpki.example.com/repository/1/1/
certificate2	rsync://rpki.example.com/repository/1/2
certificate3	rsync://rpki.example.com/repository/1/3

All of these publishing points could be prefetched by downloading `rsync://rpki.example.com/repository/1`. This of course requires prior knowledge of the location of those pub-

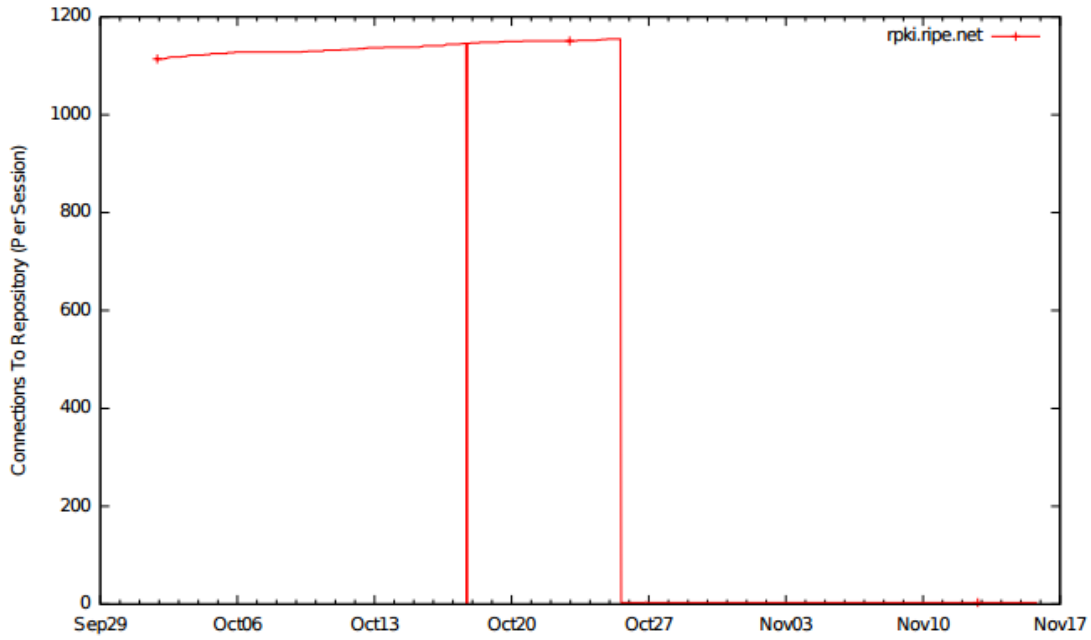


Figure 3.4: RIPE goes hierarchical: Connections needed to download the complete RIPE repository dropped dramatically once a hierarchical repository structure was adapted [30]

lishing points. To get this information we can run the download algorithm one time without prefetching and log the location of all downloaded publishing points. From that list of URIs we can derive a minimal set of Prefetch-URIs that contains the longest common prefixes of the URIs in the list. For table 3.1, the Prefetch-URI set would consist of only `rsync://rpki.example.com/repository/1`. Note that this method will download all data contained within that directory. This can be a problem if it contains data that is not part of the RPKI, leading to unnecessary downloads. This is a trade of between avoiding unnecessary rsync calls and possibly downloading unneeded data.

Prefetching is a crucial problem to solve since without it a huge, flat repository could cause thousands of individual rsync calls resulting in hours of overhead making updating a very tedious process.

3.2.2 Data Model

To represent the data we decided to adopt a model which mirrors the hierarchical relationships between the RPKI objects. As seen in Figure 3.5, in our model a CertificateObject holds references to all objects it issued: manifest, crl, and its ResourceHoldingObject children. It also holds a reference to its parent object. It is important to note that in our model, the CRL a CertificateObject is referencing is *not* the CRL pointed at by the certificates "CRL distribution points" field. Rather the referenced CRL is the one that was issued by this certificate and pertains to the revocation status of other objects issued by the

certificate.

At the top of the CertificateObject hierarchy is the trust anchor. In turn, it is contained by the ResourceCertificateTree class which also holds the name and download timestamp. Adapting the hierarchical nature of RPKI objects into our data model makes it easy for a front end to present it in a intuitive manner. It is also very convenient for validation, since a CertificateObject references all objects needed for the validation and the validation of its issued MFT, CRL and ROAs.

From here on we will call all objects issued by a certificate C the *children* of C. In the case of RPKI signed objects like ROAs and MFTs, which are technically issued by the EE certificate they contain, we will still refer to them as children of the certificate that issued the EE certificate.

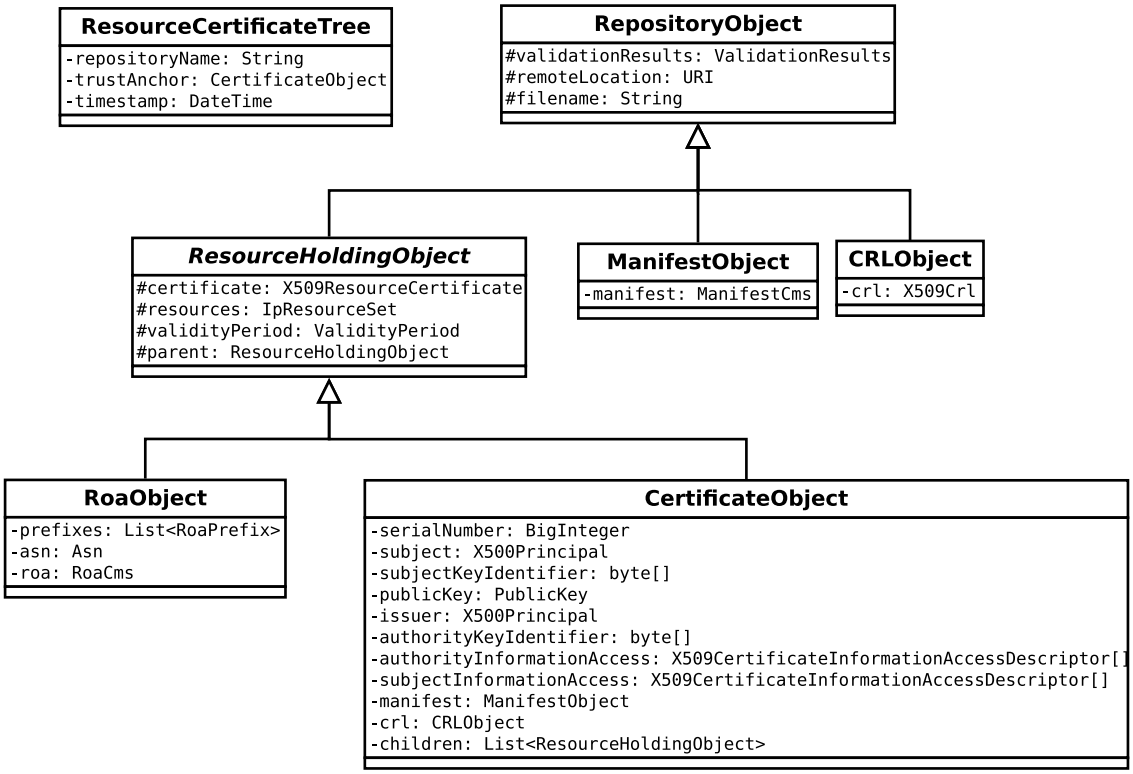


Figure 3.5: Data model in UML: Associations between classes have been omitted for clarity

3.2.3 Validation

Validation occurs once the complete repository has been downloaded and parsed. We iterate over the data in a top-down fashion, starting with the trust anchor. After the trust anchor itself is validated by making sure it conforms to [13] and [22], its MFT, CRL, and any children certificates are validated. For the validation of a non self-issued certificate it is necessary to obtain the CRL and MFT of the parent certificate, as mentioned in sections 2.2 and 2.3. Our data model makes this easy, since every non self-issued certificate contains

a reference to its parent certificate, which in turn contains a reference to its MFT and CRL.

For a given RPKI object, the validation can yield three different results. Note that these categories are not part of the official RPKI specification and are used by us in the context of the MIRO system:

1. *Valid*: The object conforms fully to the RFC specifications. In the case of resource certificates and CRLs, there must be a path of valid resource certificates starting from the issuing certificate to the trust anchor. In case of a RPKI signed object like a ROA or MFT, this path starts with the contained EE certificate and also ends at the trust anchor.
2. *Warning*: There exists a path of valid certificate as described above and this object is treated as valid. However, some inconsistencies with the specification exist such as a missing attribute, using wrong ASN.1 types, or the object is stale (still considered valid, but a newer version of it has been published). The warning status is used for objects that do not conform to the specifications completely, but whose diversions from it are minor and not problematic from a security point of view.
3. *Invalid*: There are multiple reasons why an object is considered to be invalid. If there exist no path of valid certificates to the trust anchor as described above, it is invalid. Likewise, if the signature on the object can not be verified using its supposed parent certificate. An expired object will also be considered invalid. In the case of resource certificates and ROAs, the resource extensions can be a cause for invalidity if the object does not contain a subset of the resources of its parent.

A top-down validation algorithm could look like this:

```
1    def validateRepository(trustAnchor):
2        workingQueue.add(trustAnchor)
3        while(workingQueue.notEmpty):
4            certificate = workingQueue.pop()
5            validateManifest(certificate.manifest)
6            validateCRL(certificate.CRL)
7            validateCertificates(certificate.issuedCertificates)
8            validateROAs(certificate.issuedROAs)
9            workingQueue.addAll(certificate.issuedCertificates)
```

Note that top-down validation is only one way of validating a repository. However, all validation algorithm must perform the same amount of validation operations. Therefore choosing an appropriate algorithm depends on the preferred way of iterating over the repository. In our case, we chose a top-down validation algorithm because it is simple, intuitive, and uses the hierarchical nature of our data model.

After a repository has been validated completely, we can gather statistics about it.

3.2.4 Statistics

Since MIROs primary purpose is inspection of RPKI objects and there are already a number of sources [31, 32, 33, 34] publishing detailed statistics about the global RPKI we decided to only gather some basic figures:

1. Total amount of objects
2. Total amount of objects with *Valid* validation status.
3. Total amount of objects with *Warning* validation status.

4. Total amount of objects with *Invalid* validation status.

For each of these numbers we also gather the amount of each object type, object type being certificate, ROA, MFT, or CRL. All causes for *Warning* and *Invalid* are logged and counted as well. In addition to those, we also break the objects down by host to see how distributed a repository is, which can lower the efficiency of prefetching. These numbers also allow us to verify that the Validator is working correctly, since we can compare them to statistics gathered by similar systems.

3.2.5 Export

Once we have the validated repository and statistics as Java objects, the last step in our processing chain is exporting. We have two distinct objects to export:

1. ResourceCertificateTree: Contains the validated trust anchor from which the rest of the validated repository is referenced. Also contains the timestamp from when the download of the repository started. It further holds some descriptive name for the repository.
2. RPKIRepositoryStats: Contains the statistics gathered about a repository. Also contains a descriptive name, download timestamp, and the filename of the trust anchor of the repository.

The information contained in both of these objects needs to be passed to the *Browser* component. We follow a modular approach, and define an interface *RepositoryExporter* with export methods for each object. This makes it easy to implement exporting using a specific format like JSON, CVS or exporting to a database.

3.3 Browser

The task of the Browser component is to display the data exported by the Validator and allow users to explore it. Since the Validator outputs two different types of data we chose to separate the front end into two parts that we present in this section. The RPKI Browser part displays the validated repository, the Statistics part displays visualizations of the stats. In addition to those two, we present the Model Updater as the third part of the Browser component. It offers the functionality to obtain current RPKI data from the Validator.

3.3.1 RPKI Browser

The RPKI Browser is the graphical user interface which enables users to inspect repositories and their content in detail. It is not feasible to display multiple repositories at once in a clean and pleasing fashion, hence only one is shown at a time. The user can choose which repository to display. A repository is shown in two parts:

1. *Repository View*: Shows the user the resource certificates and ROAs contained in a repository. The default repository view arranges them in a tree structure that mirrors the hierarchical relationships between the objects. Other views can be added by implementing the *RepositoryView* interface. Manifest are not shown in the repository

view since they exist to ensure secure transfer of files from the global RPKI and are of little interest on their own. CRLs are not shown either, since they only pertain to the validity of certificates.

2. *Detail View*: Users can select objects in the Repository View which then will be displayed in detail in the Detail View. For resource certificates shown in the repository view, the detail view includes their most recently issued Manifest and CRL. For ROAs it includes the EE certificate contained within the ROA.

   For every object, the detail view shows: *Validation Status, Filename*, and *Location URI*. Also a list of all (if any) validation warnings or errors.

   For resource certificates these fields are shown additionally: *Resources, Validity Period, Subject, Issuer, Subject Key Identifier, Authority Key Identifier, Serial Number*, and *Public Key*.

   For ROAs: *ASN, Signing Time*, and *IP address blocks*

   Manifest show the list of files and hashes they contain and CRLs a list of serial numbers and revocation times.

In order to allow users to find specific objects within the Repository View, the RPKI Browser includes filter functionality.

Filter

The filter enables users to find specific objects or specific sets of objects without having to browse through the repository manually. It is coupled to the Repository View and constraints it to only show the objects that match the currently set filter criteria. The available filter criteria are all attributes listed in the Detail View as well as the option of filtering for object type (i.e. CA resource certificates and ROAs). An example filter setting could be described as:

$$\text{issuer} == \text{"root trust anchor lacnic"} \wedge \text{validation\_status} == \text{VALID}$$

Applying this filter would show only valid objects that were issued by the LACNIC trust anchor.

3.3.2 Statistics

The Statistics part of the Browser component will display the information described in section 3.2.4 for each repository. The amounts of total, valid, invalid, and warned objects will be visually broken down by object type and, if historical data is available, plotted over time.

3.3.3 Model Updater

The purpose of the Model Updater is to offer an interface for outside processes to trigger the execution of the Validator processing chain described in section 3.2. Once the Validator has finished execution and has exported the validated repository data to the Browser component, it is the Model Updater's task to update the front end with the new data. It is important

to emphasize that the Model Updater is not responsible for scheduling the update, only for executing them. The decision *when* to update lies solely on the outside process triggering the Model Updater. The triggering process must not necessarily run on the same machine as MIRO.

CHAPTER 4

Implementation

This chapter deals with the implementation of the MIRO system. In the first section, we present the criteria we considered while choosing a programming language. We also present the development environment and platform. The second and third sections deal with the Validator and Browser components respectively. Here we show the implementation details of the requirements and problems determined in chapter 3.

4.1 Choice of Technology

4.1.1 Programming Language

For the implementation of the MIRO system we evaluated programming languages on these criteria:

Versatility
: In order to manage project complexity and future maintainability, we decided to use the same programming language for both the Validator and Browser component. Fulfilling this criteria requires a certain degree of versatility which not all languages can provide. This requirement excluded languages such as JavaScript, C, and C++. The former being unfit to handle the resource intensive processing chain of the Validator and the latter two being unfit to build a front-end with.

Performance and Scalability
: Anticipating the growth of the global RPKI, we needed a language that offers acceptable performance and scaling properties. The nature of RPKI data makes the processing chain of the Validator component a good candidate for parallelization: Logical repositories can be processed in parallel since they do not share any data.

Portability
: The MIRO system should be as independent of the underlying OS as possible, so it can be deployed on a multitude of different systems. The system should

come in a self-contained format that requires only minimal dependencies. This allows for easier distributed monitoring mentioned in section 3.1.1.

Existing software libraries

Taking into account proficiency and preference of the authors, we considered the languages Java and Python. Ultimately we have decided to use Java for these reasons:

The Global Interpreter Lock (GIL) of the widely used default python implementation *CPython* prevents true concurrent execution of python threads. This forces programmers to use separate CPython interpreter processes, which makes it difficult to effectively share data between them in a OS independent way. Java allows for true concurrent thread execution.

There already exist Java libraries that support the RPKI resource certificate extensions and validation of RPKI objects.

4.1.2 Framework

The MIRO system was developed using the Eclipse IDE Plugin Development Environment. Both the Validator and the Browser component are implemented as Eclipse Plugins. Eclipse Plugins are an abstraction layer that offers a format to explicitly define dependencies and export interfaces. This gives third parties that want to reuse the MIRO code a clean and modular approach. Note that the usage of Eclipse Plugins does not preclude using the code in other environments, it is merely an additional development convenience.

4.2 Validator

This section presents the implementation of the processing chain described in section 3.2.3.

4.2.1 Data Model

The Validator component makes heavy use of the open source Java library *rpki-commons* [35], developed by RIPE NCC and used in their *RPKI Validator* [36] system. The rpki-commons library offers an API to instantiate resource certificates, manifests, CRLs, and ROAs as Java objects:

RPKI Object	**Java Object in rpki-commons**
Resource Certificate	X509ResourceCertificate
Route Origin Authoriziation (ROA)	RoaCms
Manifest	ManifestCms
CRL	X509Crl

However, they contain no references to each other and therefore do not conform to the data model we specified in section 3.2.2. Rpki-commons offers static access functions for these java objects for all relevant fields described in section 3.3.1. These functions often involve

a relatively high amount of operations, a property that is undesirable for frequent use by, for instance, a front end. As an example of that consider Listing 4.1 showing the access method for the Authority Information Access field of a X509Certificate, the parent class of X509ResourceCertificate.

```
1  public static X509CertificateInformationAccessDescriptor[]
       getAuthorityInformationAccess(X509Certificate certificate) {
2          try {
3              byte[] extensionValue = certificate.getExtensionValue(org
                   .bouncycastle.asn1.x509.X509Extension.
                   authorityInfoAccess.getId());
4              if (extensionValue == null) {
5                  return null;
6              }
7              AccessDescription[] accessDescriptions =
                   AuthorityInformationAccess.getInstance(
                   X509ExtensionUtil.fromExtensionValue(extensionValue)).
                   getAccessDescriptions();
8              return X509CertificateInformationAccessDescriptor.
                   convertAccessDescriptors(accessDescriptions);
9          } catch (IOException e) {
10             throw new X509CertificateOperationException(e);
11         }
12 }
```

Listing 4.1: X509CertificateUtil.getAuthorityInformationAccess from rpki-commons

To avoid calling these intensive access functions unnecessarily, we implemented the data model described in section 3.3.1 and used memoization. This means we only call these expensive functions once on instantiation and store their result in an instance variable. As an example, Listing 4.2 shows relevant parts of our data models CertificateObject class instance variables, constructor, and access methods.

```
1  public class CertificateObject extends ResourceHoldingObject {
2      [...]
3      private X509CertificateInformationAccessDescriptor[] aias;
4
5      public CertificateObject(..., X509ResourceCertificate certificate
           ) {
6          [...]
7          aias = certificate.getAuthorityInformationAccess();
8          [...]
9      }
10
11     ...
12     public X509CertificateInformationAccessDescriptor[] getAias() {
13         return aias;
14     }
15     [...]
16 }
```

Listing 4.2: CertificateObject class

The instantiation of Certificate, Roa, Manifest, and CRLObjects is performed by the *RepositoryObjectFactory* class, using static factory methods instead of constructors as described in [37]. This gives the advantage of clearer names and the ability of returning subtypes of the return type. As an example, a static factory function called *createResourceHoldingObject()* can return instances of CertificateObject or RoaObject.

4.2.2 Processing Chain Overview

An instance of the *ResourceCertificateTreeValidator* (RCTValidator) class initiates all steps in the Validator processing chain with the exception of Statistics. The implementation of downloading, parsing, and validation is delegated to other classes.

The Validator input consists of a trust anchor certificate, as described in section 3.2. However, in section 2.3 we discussed *Trust Anchor Locators* (TALs) which offer a convenient mechanism to locate a trust anchor. Using TALs also ensures that the most recent version of the trust anchor is downloaded. Therefore, the Validator additionally offers a method that allows users to input a TAL instead of a trust anchor.

Once the trust anchor is downloaded, execution of the processing chain can begin. Shown here is the RTCValidator method that implements the processing chain on the highest abstraction level:

```
1  public ResourceCertificateTree createResourceCertificateTree(URI
       taLocation, String repoName, String timestamp)
2  {
3      [...]
4
5      /* Download the trust anchor */
6      String taPath = downloader.fetchObject(taLocation, BASE_DIR);
7
8      /* Instantiate the trust anchor */
9      ValidationResult result = ValidationResult.withLocation(taPath);
10     CertificateObject trustAnchor = createTrustAnchor(taPath, result);
11
12     [...]
13
14     /* Start the processing chain */
15     ResourceCertificateTree tree = new ResourceCertificateTree(this,
           repoName, trustAnchor, result, timestamp, BASE_DIR);
16     tree.populate();
17     tree.validate();
18     tree.extractValidationResults();
19     certTree = tree;
20     return certTree;
21 }
```

Listing 4.3: Implementation of the processing chain

Some notes on Listing 4.3

:

BASE_DIR, Line 6,15: This static string is the path to the local directory the downloaded

files will be written to. URIs such as the "taLocation" parameter can be converted as shown in Listing 4.4 to find the location of the downloaded file in the local filesystem:

```
1 public static String toPath(URI uri){
2         return BASE_DIR + uri.getHost()+uri.getPath();
3 }
```

Listing 4.4: Directory path conversion

ValidationResult, Line 9: The ValidatonResult class is part of the rpki-commons library. It stores validation results in a HashMap using Strings or URIs as keys. Line 9 instantiates the ValidationResult objects that will be used for this ResourceCertificateTree. It is given the local location of the trust anchor as a key and passed onto createTrustAnchor() in Line 10 so the result of syntax checks of the trust anchor file can be recorded.

The next sections explain the implementation of the processing chain in more detail, in particular lines 15-20 of Listing 4.3.

4.2.3 Downloading and Parsing

The downloading and parsing of the logical repository is done in the *populate()* method, called in line 16 of Listing 4.3. The method is part of the ResourceCertificateTree class. The populate method implements the algorithm presented in section 3.2.1:

```
1  public void populate() {
2      Queue<CertificateObject> workingQueue = new LinkedList<
           CertificateObject>();
3      workingQueue.add(trustAnchor);
4      CertificateObject cert;
5      while(!workingQueue.isEmpty()){
6          cert = workingQueue.poll();
7          getChildren(cert);
8          workingQueue.addAll(getCertificateObjectChildren(cert.
               getChildren()));
9      }
10     log.log(Level.INFO,"Reading done");
11 }
```

Listing 4.5: ResourceCertificateTree.populate

We iterate over the *workingQueue* queue, which initially only holds the trust anchor. The *getChildren()* function in line 7 populates the *children, manifest*, and *crl* instance variables defined in our data model in section 3.2.2. In line 8 we then add the next list of CertificateObjects to iterate over.

The implementation of the *getChildren()* method is shown in Listing 4.5:

```
1  public void getChildren(CertificateObject cw) {
2          for(X509CertificateInformationAccessDescriptor
               accessDescriptor :cw.getCertificate().
               getSubjectInformationAccess()){
3
4              if(!isPublishingPoint(accessDescriptor))
```

```
5                   continue;
6
7               int rtval = validator.fetchURI(accessDescriptor.
                    getLocation());
8               if(rtval != 0){
9                   log.log(Level.WARNING, "Could not download publishing
                        point: " + accessDescriptor.getLocation());
10                  continue;
11              }
12              findManifest(cw);
13              findCRL(accessDescriptor.getLocation(),cw);
14              findChildren(accessDescriptor.getLocation(),cw);
15          }
16  }
```

Listing 4.6: ResourceCertificateTree.getChildren

The getChildren function takes a CertificateObject cert as input. For each publishing point, listed in the Subject Information Access Field of the certificate, the manifest, CRL, and any children are found and references to them are added to cert. In line 7, the publishing point is downloaded within the RTCValidator method fetchURI:

```
1  public int fetchURI(URI desc) {
2          String result = toPath(desc);
3          if(wasPrefetched(desc))
4              return 0;
5          return downloader.downloadData(desc,result);
6  }
```

Listing 4.7: ResourceCertificateTreeValidator.fetchURI

In line 3 the fetchURI method checks whether the input URI has been prefetched to avoid redundant downloads. If the URI has not been prefetched, it is downloaded with a call to rsync within the downloadData method.

Prefetching

In section 3.2.1. we presented an option of how prefetching can be implemented. However, due to time constraints this approach has not been implemented yet. Instead we used the simpler, less elegant solution of manually determining the Prefetch URIs for the repositories of the five RIRs. The URIs are stored in a file and read by the Validator. The implementation of the solution we presented in section 3.2.1 is planned for the future.

4.2.4 Validation

The rpki-commons library includes functionality for the validation of X509ResourceCertificate, X509Crl, RoaCms, and ManifestCms objects. The results of the validation are recorded in a ValidationResult object, mentioned in section 4.2.2. Using this data structure and functionality, we implemented the top-down validation algorithm discussed in section 3.2.3. Rpki-commons also offers the CertificateRepositoryObjectValidationContext class, a data

structure that represents a validation context. The context keeps track of the state of the validation process. It contains the current "parent" certificate, i.e. the certificate whose public key is being used to verify the signature on the object that is being validated. The CertificateRepositoryObjectValidationContext also keeps track of the IP resources of the parent certificate and offers methods to verify a child's IP resources against them, as explained in section 2.3.1.

The logical repository is fully downloaded and parsed in the populate function (provided the rsync connections did not fail). Validation starts with the call to the validate method of the ResourceCertificateTree instance, found in line 17 of the createResourceCertificateTree function in Listing 4.3. This method further delegates the validation to an instance of the *TopDownValidator* class:

```
1  public class TopDownValidator {
2      [...]
3      private ValidationResult result;
4      private ResourceCertificateLocator locator;
5      private ValidationOptions options;
6      private Queue<CertificateObject> workQueue;
7      private CertificateRepositoryObjectValidationContext context;
8
9      public TopDownValidator(ValidationResult result,
           ResourceCertificateLocator locator, CertificateObject
           trustAnchor){
10         this.result = result;
11         this.locator = locator;
12         this.options = new ValidationOptions();
13         this.workQueue = new LinkedList<CertificateObject>();
14         this.context = new
               CertificateRepositoryObjectValidationContext(URI.create(
               trustAnchor.getFilename()), trustAnchor.getCertificate());
15         this.workQueue.add(trustAnchor);
16     }
17
18     public void validate() {
19         while(!workQueue.isEmpty()){
20
21             /* Get next Certificate */
22             CertificateObject parent = workQueue.remove();
23
24             /* If its not a trust anchor, set up a new context */
25             if(!parent.getIsRoot()){
26                 this.context = this.context.createChildContext(URI.
                       create(parent.getFilename()),parent.getCertificate
                       ());
27             }
28
29             /* Verify that mft and crl are not missing, rpki-commons
                   does not do this */
30             result.setLocation(new ValidationLocation(parent.
                   getFilename()));
31             result.warnIfNull(parent.getManifest(), "missing.manifest
                   ");
```

```
32                  result.warnIfNull(parent.getCrl(), "missing.crl");
33
34                  validateManifest(parent.getManifest());
35                  validateCrl(parent.getCrl());
36
37                  /* Validate children, e.g. CertificateObject, RoaObject
                        */
38                  /* This also adds the validated CertificateObjects to the
                         workQueue */
39                  validateChildren(parent);
40              }
41              log.log(Level.INFO,"Validating done");
42          }
43          [...]
44      }
```

Listing 4.8: TopDownValidator with validate method

The TopDownValidator uses a queue, stored in an instance variable, to iterate over the logical repository. The queue holds validated CertificateObjects, which initially is only the trust anchor, added in line 15. The validateChildren function is responsible for adding any validated CertificateObjects to the workQueue, ensuring that the entire repository is iterated over. The CertificateRepositoryObjectValidationContext mentioned previously is also stored as a instance variable. It is initialized with the trust anchors X509ResourceCertificate object, since the trust anchor is the first CertificateObject in the workQueue. For the CertificateObjects in further iterations, a new context is derived from the previous one in line 26. We decided to implement two additional validation checks that were missing from rpki-commons. In line 34 and 35 the existence of the CertificateObjects manifest and CRL are checked and, if missing, a warning is added to the validation result of the CertificateObject. The actual validation of the RPKI objects happens within the validateManifest, validateCrl and validateChildren methods, using rpki-commons functionality.

After the validate function returns, the validation results are copied from the rpki-commons ValidationResult data structure into fields of the RepositoryObjects. This is done for convenience, so that every RepositoryObject is self-contained and gives information about its validity without the necessity of looking it up in the ValidationResult HashMap. This is done in line 18, Listing 4.3. With the return of this function, the repository is now fully validated and ready to be used within the Browser Component.

Validation Correctness

To confirm the correctness of our validation we looked at the validation result from two other systems, RIPE NNCs *rpki-validator* [36] and Rob Austeins *rcynic* [38].

The rpki-validator also uses the rpki-commons library for parsing and validation, so as expected their validation results match ours. Rcynics validation results however showed some discrepancies to MIROs and rpki-validator results, namely a difference in RPKI object count and number of objects that passed validation with warnings. After further investigation the object count difference could be explained by rcynics susceptibility to connection problems, since it does not employ any prefetching and consequently more rsync calls are made. The

discrepancy in warning messages was explained by incorrect validation behavior of the rpki-commons library. We have implemented a fix and submitted a pull-request to RIPE NCC which was subsequently merged into the master branch. This eliminated major differences in validation results between rcynic and rpki-validator/MIRO. Note that some differences still exist, they are however caused by different validation strictness of the systems.

4.2.5 Statistics

Gathering the statistics about the logical repository was not implemented as part of the processing chain discussed in the previous section. We decided to offer a separate method to gather the statistics, so users can decide according to their own needs. The ResultExtractor class can be used to obtain the statistics:

```
1  public class ResultExtractor {
2
3      private ResourceCertificateTree currentTree;
4      private Result totalResult;
5      private List<Result> hostResults;
6
7      public ResultExtractor(ResourceCertificateTree tree) {
8          currentTree = tree;
9          totalResult = new Result("Total");
10         hostResults = new ArrayList<Result>();
11     }
12
13     public void gather() {
14         [...]
15     }
16     [...]
17 }
```

Listing 4.9: ResultExtractor

After being initialized with a ResourceCertificateTree, the ResultExtractor gathers the statistics described in chapter 3. Each Result object stores these statistics in HashMaps:

```
1  public class Result {
2
3      private HashMap<String,Integer> counter;
4      private HashMap<String,Integer> warning;
5      private HashMap<String,Integer> error;
6
7      private String descriptor;
8
9      public Result(String desc) {
10         descriptor = desc;
11
12         counter = new HashMap<String, Integer>();
13
14         warning = new HashMap<String, Integer>();
15
16         error = new HashMap<String, Integer>();
```

```
17          }
18
19          [...]
20  }
```

Listing 4.10: Result

The counter HashMap stores the statistics listed in section 3.2.4, the warning and error HashMaps store the frequency of validation warnings and errors. For each distinct host that is part of the repository, a separate Result object is stored as discussed in the requirements. Users can obtain these statistics in the form of a RPKIRepositoryStats object returned by a method of the ResultExtractor. The RPKIRepositoryStats object contains all results as well as the repository name, timestamp and trust anchor name.

4.2.6 Export

Both the Validator and Browser component were developed in Java. For the Browser this means obtaining the ResourceCertificateTree object is as simple as calling the createResourceCertificateTree method discussed in section 4.2.2. A simple *RepositoryExporter* interface can be implemented for export outside of the JVM. Using Google's *gson* library [39] an option for JSON export was added in order to offer users a way of persistently storing validated repositories.

4.3 Browser

The Browser component was developed using the Eclipse Remote Application Platform (RAP). RAP is a web framework with a pure Java API [40]. The RAP framework includes an implementation of the *Standard Widget Toolkit* (SWT). SWT is a mature, open-source widget toolkit that is used to develop the Eclipse IDE user interface. The RAP implementation of SWT is called the *RAP Widget Toolkit* (RWT) and compiles to JavaScript. The RAP framework was chosen for several reasons:

1. SWT is a mature and proven widget toolkit specifically designed to build complex graphical user interfaces. It is enhanced by JFace [41], a library built on top of SWT that offers additional widgets and functionality such as StructuredViewers and Databinding.
2. Cross browser support: RAP works with all relevant web browsers. This eases development significantly.
3. Ease of deployment: An RAP application can be exported as a *Web Application Archive* (WAR) file and can then be deployed easily with any servlet container.
4. Single sourcing: Since RAP implements SWT, the code can be reused for a possible Eclipse Rich Client Platform (RCP) application which runs on the desktop. This is an added perk and not something we plan to take advantage of in the immediate future.
5. It is integrated in the Eclipse IDE environment and is based on Eclipse plugins, like the Validator component.

4.3.1 Overview

The Browser component implementation can be separated into two parts:

User Sessions: This part is responsible for interaction with the user via RWT widgets. The user session code is run in a separate thread for every client that connects to the application server. The creation and termination of the separate user sessions is managed by the RAP framework. A user session includes the RPKIBrowser and Statistics parts presented in section 3.

Model Updater: This part is responsible for obtaining the ResourceCertificateTree object from the Validator component and distribute it to the user session threads. An update is triggered via an interface that the Model Updater exposes to other processes, as previously mentioned in section 3.3.3. The Model Updater code runs in its own thread, separated from user sessions.

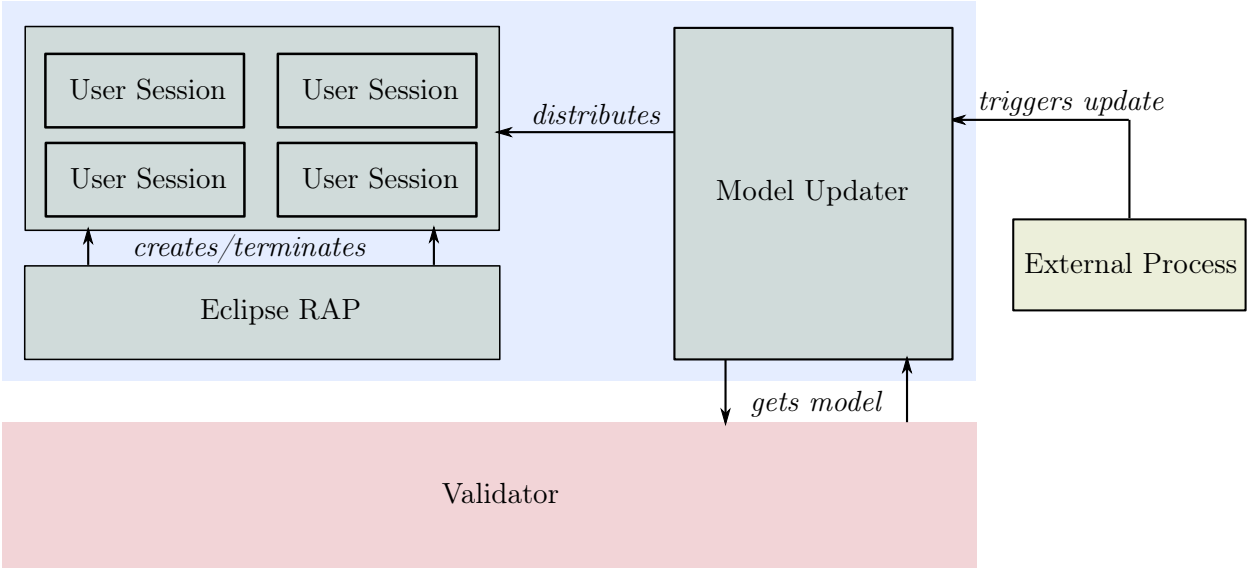


Figure 4.1: Browser component overview

4.3.2 ModelUpdater

In this section we present the different tasks that the ModelUpdater is responsible for and discuss their implementation. The ModelUpdater class implements the *Runnable* interface from the standard Java API. Its run method is called once the thread is started:

```
1  @Override
2  public void run() {
3      log.log(Level.INFO, "Thread started");
4      ServerSocket triggerSocket;
5      Socket clientSocket;
6      URI[] prefetchURIs;
7      try {
8          readConfig(CONFIG_FILE_LOCATION);
```

```
 9            prefetchURIs = readPrefetchURIs(PREFETCH_URI_FILE_LOCATION);
10            triggerSocket = new ServerSocket(UPDATE_PORT);
11        } catch (IOException e) {
12            log.log(Level.SEVERE, e.toString(),e);
13            throw new RuntimeException(e);
14        }
15
16        update(prefetchURIs);
17        while(run){
18            try {
19                clientSocket = triggerSocket.accept();
20                if(clientSocket.getInetAddress().isLoopbackAddress()){
21                    update(prefetchURIs);
22                }
23            } catch (IOException e) {
24                log.log(Level.SEVERE, e.toString(),e);
25                log.log(Level.SEVERE,"Update failed");
26                continue;
27            }
28        }
29
30        try {
31            triggerSocket.close();
32        } catch (IOException e) {
33            log.log(Level.SEVERE,e.toString(),e);
34        }
35        log.log(Level.INFO,"Quitting thread");
36    }
```

Listing 4.11: ModelUdpater.run

The config files are read in line 8 and 9. They contain the network port to listen on, the URIs that are to be prefetched, the input directory containing trust anchor locators, and the directory that the Validator component should use when downloading the repositories.

Update Trigger

For greater flexibility, we decided to allow triggering of updates via the network. Due to time constraints, we chose a very simple, temporary solution. The ModelUpdater listens on the hosts *loopback* network interface for incoming connections. Once a connection has been made, the ModelUpdater starts the update process. This allows processes running on the same host as MIRO to trigger updates, but not any processes running on other hosts since they cannot connect to the loopback interface. Using the loopback interface also precludes the need for authentication, since only known processes can connect to it. The implementation is simply a ServerSocket listening for incoming connections with a blocking accept call (Listing 4.11, line 19). Incoming connections trigger the update provided they were received over the loopback network interface (Listing 4.11, line 20 and 21).

A more sophisticated solution is planned for later stages of development, allowing triggering updates not just via the loopback interface. Allowing external processes (that are not running on the MIRO host) to trigger updates necessitates authentication.

Updating

The updating process starts with the update method of the ModelUpdater:

```
1  public void update(URI[] prefetchURIs) {
2       getModels(prefetchURIs);
3       notifyObservers();
4  }
5
6  public void getModels(URI[] uris){
7      log.log(Level.INFO, "Getting models");
8
9      RPKIRepositoryStats stats;
10     ResourceCertificateTree certTree;
11     int index = 0;
12     String name = "";
13     File[] talFiles = new File(TALDirectory).listFiles();
14     String[] statsKeys = new String[talFiles.length];
15     String[] modelKeys = new String[talFiles.length];
16     String key;
17
18     cleanInputPath();
19     ResourceCertificateTreeValidator treeValidator = new
           ResourceCertificateTreeValidator(inputPath);
20     treeValidator.preFetch(uris);
21     for(File talFile : talFiles){
22
23         /* Get repo name from TAL file, get ResourceCertificateTree,
               save it to application context and remember the key */
24         name = getRepositoryName(talFile.getName());
25         certTree = treeValidator.getTreeWithTAL(talFile.toString(),
               name);
26         key = certTree.getName();
27         context.setAttribute(key, certTree);
28         modelKeys[index] = key;
29
30         /* Get stats about the tree, save them to disk, save them to
               context and remember the key */
31         stats = getRPKIRepositoryStats(certTree);
32         ResultExtractor.archiveStats(stats, STATS_ARCHIVE_DIRECTORY +
               name);
33         key = STATS_NAME_PREFIX + name;
34         context.setAttribute(key, stats);
35         statsKeys[index] = key;
36
37         index++;
38     }
39
40     /* Get global RPKI stats over all processed repositories */
41     RPKIRepositoryStats totalStats = getTotalStats(statsKeys);
42     key = STATS_NAME_PREFIX + totalStats.getName();
43     context.setAttribute(key, totalStats);
44
```

```
45        String[] allStatsKeys = prependToStringArray(statsKeys,key);
46        Arrays.sort(statsKeys);
47
48        context.setAttribute(MODEL_NAMES_KEY, modelKeys);
49        context.setAttribute(STATS_NAMES_KEY, allStatsKeys);
50    }
```

Listing 4.12: ModelUpdater.update

After prefetching the URIs given as arguments, a loop iterates over trust anchor locators. For every trust anchor locator, the repository name is derived and the ResourceCertificateTree is acquired (line 24, 25). To distribute the ResourceCertificateTree objects to the user sessions, the *ApplicationContext* is used. An ApplicationContext represents an RAP application and contains a thread-safe, generic HashMap which can be accessed application-wide. In line 26 and 27, the ResourceCertificateTree is stored in the ApplicationContext and its key is written to the "modelKeys" array. This array is also placed in the ApplicationContext (line 48). It is distributed using the *Observer* design pattern [42], whereas the user sessions add themselves as observers to the ModelUpdater and get notified when an update is performed. Once they acquired the "modelKeys" array, the user sessions can get the ResourceCertificateTree objects from the ApplicationContext and display them. The distribution process for the

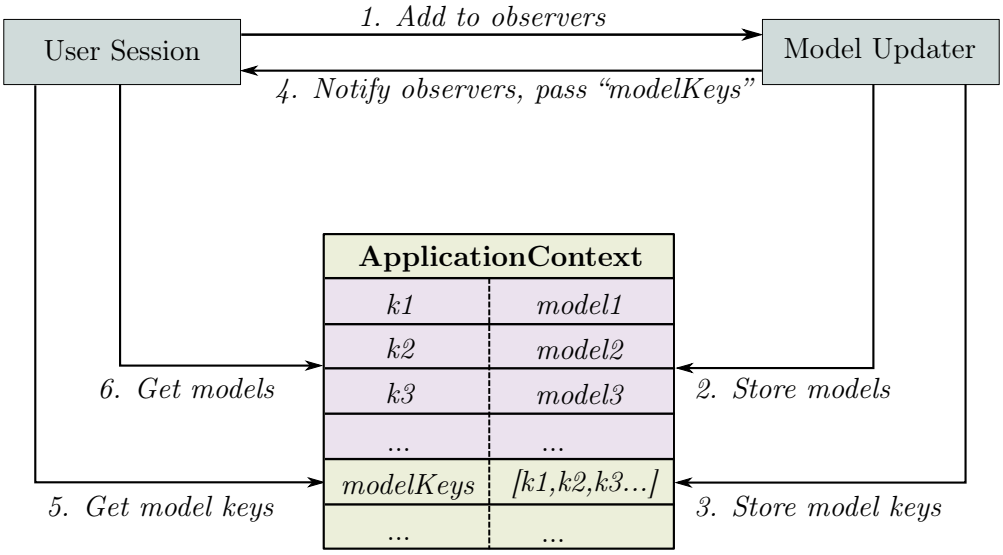


Figure 4.2: Model distribution: The ApplicationContext includes a generic, thread-safe HashMap that we use to store ResourceCertificateTree objects.

RPKIRepositoryStats objects is identical.

4.3.3 User Sessions

Once a user connects to the application, server side execution starts at a registered class that implements the *AbstractEntryPoint* interface defined in RAP. Client side execution is fully managed by the RAP framework.

The AbstractEntryPoint implementation is responsible for building up the user interface using SWT widgets. SWT comes with a number of useful widgets such as tables and trees for data representation, various buttons, text containers, and tab folders. Developers can build new widgets out of existing ones by using the *Composite* widget, which is essentially a widget container. A composite can have a layout, which determines how its containing widgets are displayed. An AbstractEntryPoint implementation is passed a reference to a composite which represents the web browser window of the client. In the following sections, all classes referred to as "widget" extend the Composite class.

We define a very simple UI layout for our application, consisting of a header bar containing navigation buttons and a content area that shows either the RPKIBrowser or the Statistics widget:

```
1 [...]
2 FormLayout layout = new FormLayout();
3 parent.setLayout(layout);
4
5 HeaderBar header = new HeaderBar(parent, SWT.NONE);
6 ContentContainer content = new ContentContainer(parent, SWT.NONE);
7 [...]
8 parent.layout();
```

Listing 4.13: Initialization of the GUI

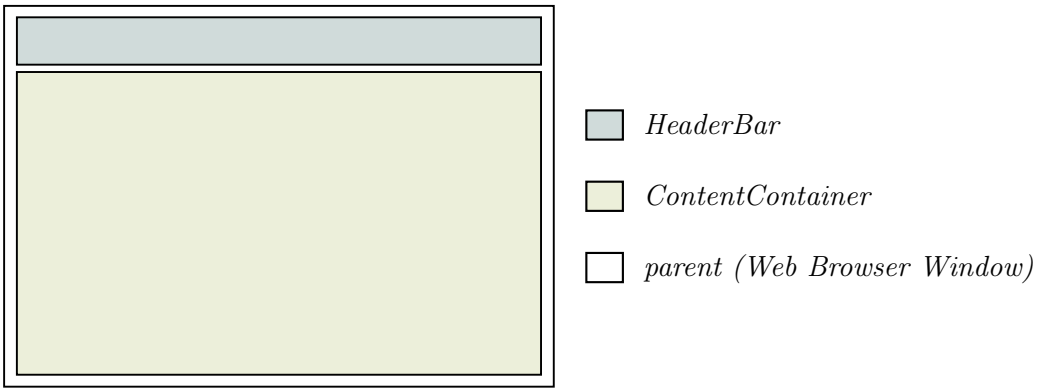


Figure 4.3: Visualization of Listing 4.13

RPKIBrowser

The RPKIBrowser widget contains the implementation of the *Repository View* and *Detail View* parts presented in section 3.3.1.

Repository View

The purpose of the Repository View is to give the user a structured way to browse RPKI objects within a logical repository. As mentioned previously in section 3.3.1, manifests and CRLs are not shown in the Repository View since on their own they contain no relevant

information to the user. They are however shown in the Detail View where they are linked to a resource certificate.

For the implementation of the Repository View part, we define this interface:

```
1  public interface RepositoryView {
2
3      public void setInput(ResourceCertificateTree tree);
4      public ResourceCertificateTree getInput();
5      public void setSelection(ResourceHoldingObject obj);
6      public ResourceHoldingObject getSelection();
7      public ViewerFilter[] getFilters();
8      public void setFilters(ViewerFilter[] filters);
9      public void resetFilters();
10     public RepositoryViewType getType();
11     public StructuredViewer getViewer();
12     public enum RepositoryViewType {
13         TREE, TABLE
14     }
15 }
```

Listing 4.14: RepositoryView interface

A RepositoryView object needs to display logical repositories (setInput, getInput), allow selection of objects (setSelection, getSelection), and allow filtering (setFilters, getFilters, resetFilters). A RepositoryView is identified by its RepositoryViewType, of which we define *Tree* and *Table*. These correspond to the JFace widgets *TreeViewer* and *TableViewer*, both subclasses of *StructuredViewer*.

Since multiple RepositoryView implementations exists, a mechanism to switch between them is needed. This is the task of the RepositoryViewContainer, a child-widget of the RPKI-Browser. It offers the functionality of switching between different RepositoryView objects by using a HashMap to store them by type:

```
1  public class RepositoryViewContainer extends Composite {
2
3      private HashMap<RepositoryViewType, RepositoryView> viewMap;
4      private StackLayout layout;
5      private RepositoryView currentView;
6
7      public RepositoryViewContainer(Composite parent, int style) {
8          super(parent, style);
9          viewMap = new HashMap<RepositoryView.RepositoryViewType,
               RepositoryView>();
10         [...]
11         initViews();
12         showView(RepositoryViewType.TREE);
13     }
14 [...]
15 }
```

Listing 4.15: RepositoryViewContainer

Filter

In section 3.3.1 we described filter functionality that is coupled to the Repository View and constraints it to only show objects that pass the filter criteria. To implement this, we used the JFace ViewerFilter API, which works in conjunction with the JFace StructuredViewers that were used for the RepositoryView implementations. We extend the *ViewerFilter* class and implement its select method:

```
1  public class ResourceCertificateTreeFilter extends ViewerFilter {
2
3      private List<ResourceHoldingObjectFilter> filters;
4
5      public ResourceCertificateTreeFilter() {
6          filters = new ArrayList<ResourceHoldingObjectFilter>();
7      }
8
9      @Override
10     public boolean select(Viewer viewer, Object parentElement, Object
           element) {
11         ResourceHoldingObject obj = (ResourceHoldingObject) element;
12         boolean selected = matchesAll(obj);
13         return getSelectResult(selected, viewer, obj);
14     }
15
16     public boolean matchesAll(ResourceHoldingObject obj){
17         boolean matches = true;
18         for(ResourceHoldingObjectFilter filter : filters){
19             matches &= filter.isMatch(obj);
20         }
21         return matches;
22     }
```

Listing 4.16: ResourceCertificateTreeFilter

For a given ResourceHoldingObject, the select method decides if this object passes the filter and is to be displayed or not. For each filter criteria described in section 3.3.1 exists an implementation of the ResourceHoldingObject interface that are stored in the *filters* list of the ResourceCertificateTreeFilter class. The matchesAll method (Listing 4.16, line 16 - 21) checks whether an object matches all filter criteria.

```
1  public interface ResourceHoldingObjectFilter {
2      public boolean isMatch(ResourceHoldingObject obj);
3  }
```

Listing 4.17: ResourceHoldingObjectFilter interface

Detail View

The Detail View shows detailed information about CertificateObjects and RoaObjects. We implemented a widget for each of the four supported RPKI objects, displaying all relevant information using SWT Text widgets and JFace StructuredViewers. Since there are two

kinds of objects that can be selected in the RepositoryView, CertificateObjects and RoaObjects, we implemented two different DetailViews: CertificateView and RoaView. Both Views are implemented as *TabFolders*, allowing child-widgets to be navigated using tabs. The CertificateView contains a CertificateWidget, a ManifestWidget, and a CRLWidget to display all information relevant to a resource certificate. The RoaView contains a RoaWidget and a CertificateWidget. The latter displays the EE resource certificate of the ROA, as presented in section 2.3.2. Once the user selects an objects in the RepositoryView, we want it displayed

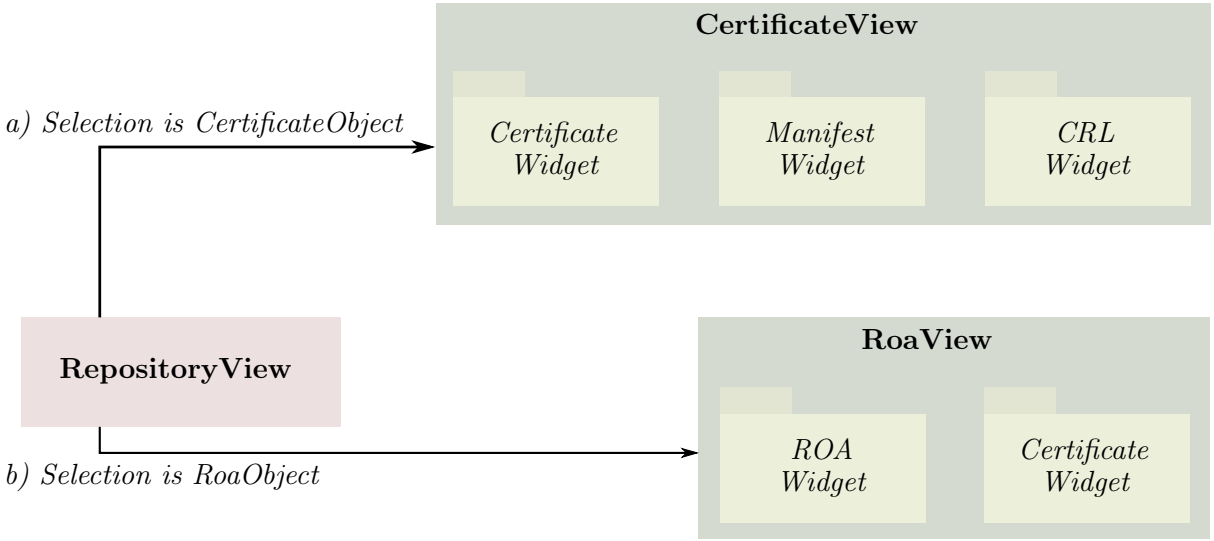


Figure 4.4: Detail View overview: The user selects a ResourceHoldingObject in the RepositoryView. The CertificateView or RoaView is shown to the user, depending on the subtype of the selection.

in the DetailView. To achieve this we use the JFace Databinding API. It allows us to bind the selection of a JFace StructuredViewer, which we used to implement the RepositoryView, to SWT Text widgets. The initDatabindings method of the RPKIBrowser class binds the abstract selection of each viewer to the CertificateView and RoaView:

```
1  private void initDatabindings() {
2          StructuredViewer viewer;
3          IViewerObservableValue selection;
4          DatabindingContext dbc;
5          for(RepositoryView view : repositoryViewContainer.getAllViews
               ()){
6              viewer = view.getViewer();
7              selection = ViewersObservables.observeSingleSelection(
                   viewer);
8              dbc = new DatabindingContext();
9              detailViewContainer.bindViews(selection,dbc);
10         }
11 }
```

Listing 4.18: JFace Databinding to StructuredViewer objects

DetailView and RepositoryView work together to allow the user to browse and inspect RPKI objects of a given logical repository. The functionality to filter a repository, switch to another RepositoryView, and switch to another logical repository is offered to the user via a toolbar. The finished RPKIBrowser is shown in Figure 4.5.

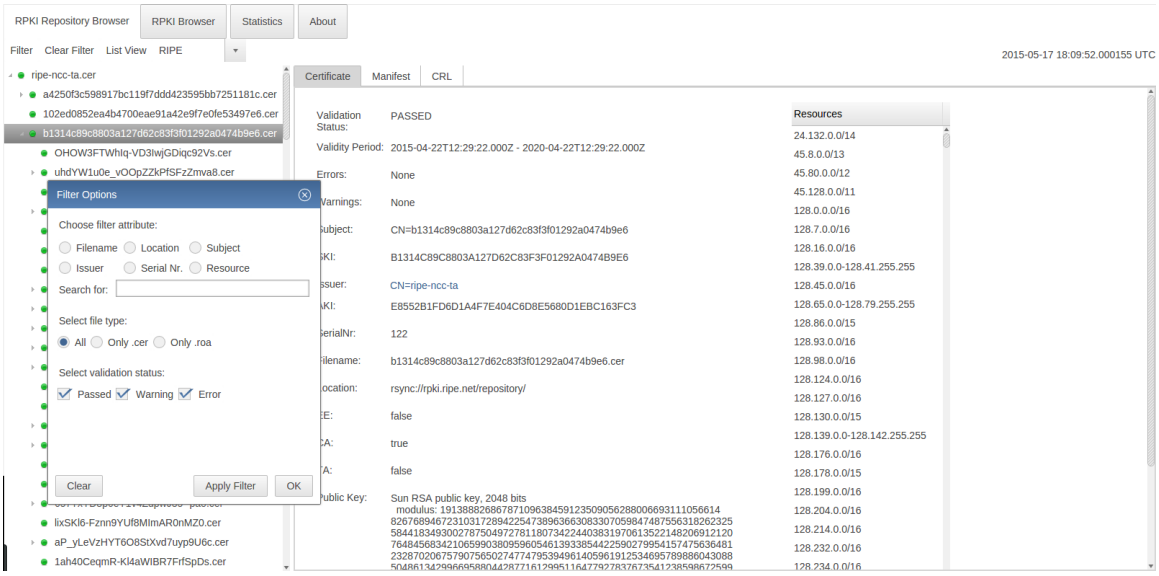


Figure 4.5: RPKIBrowser with default RAP theme.

Styling

The eclipse RAP framework offers an interface to a subset of CSS 2.1. This allows developers to style their widgets independent of their implementation. Figure 4.6 shows the finished RPKIBrowser with a custom CSS theme.

Stats

The Statistics part of the Browser component was implemented using the rap-d3charts library [43], an adaption of the d3 JavaScript library for eclipse RAP. However, due to performance issues we decided to reimplement the Statistics part using pure JavaScript.

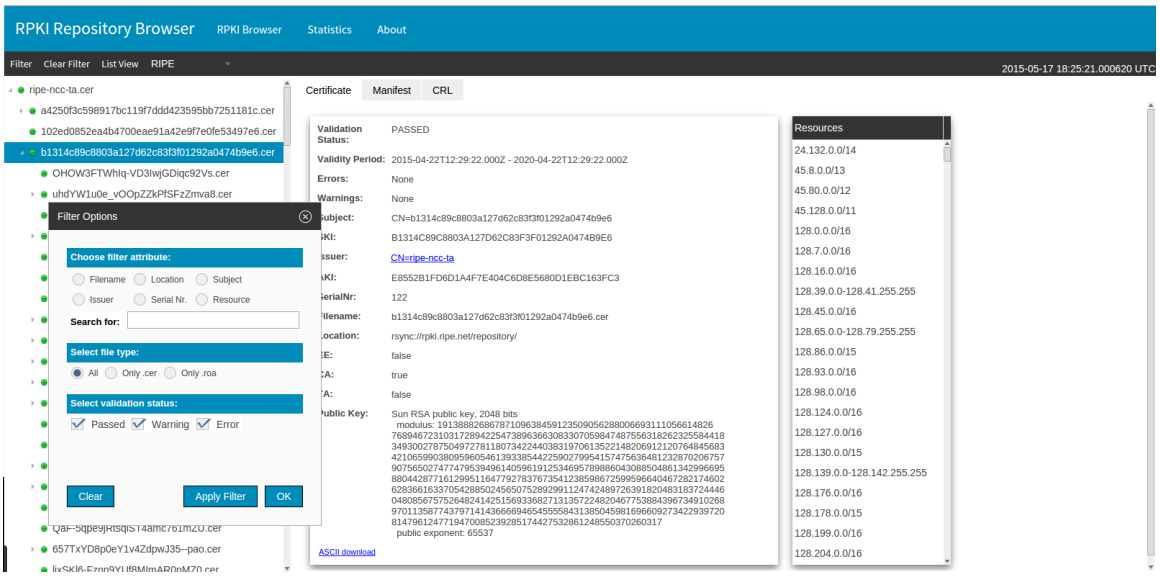


Figure 4.6: RPKIBrowser with custom CSS theme

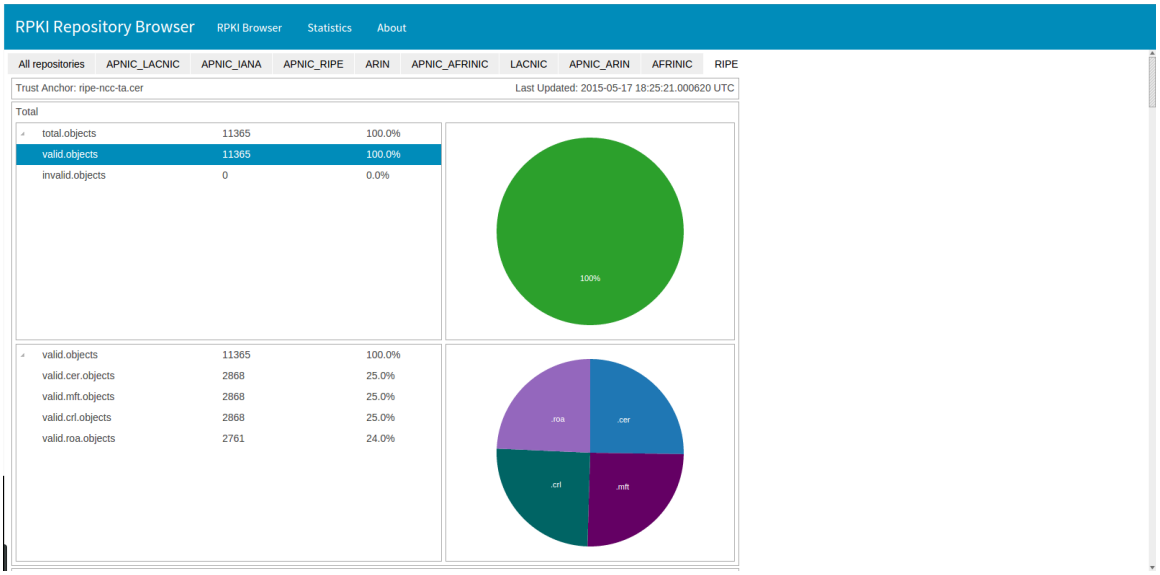


Figure 4.7: Statistics with custom CSS theme

CHAPTER 5

Evaluation

5.1 Performance

To test the performance of the MIRO software system we decided to use *Apache JMeter* [44]. JMeter allows load testing and measuring performance of a web application.

5.1.1 Test Setup

Using JMeter we recorded 3 test plans. A test plan in this context is essentially a sequence of HTTP requests to the MIRO software system. Two test plans focus on the RPKIBrowser part, one on the Statistics part:

1. Test plan 1 is aimed at the DetailView performance. It involves inspection of 4 random CertificateObjects and RoaObjects chosen from the RepositoryView. Inspection means using all DetailView functionality, such as viewing the manifest and CRL, or EE resource certificate associated with CertificateObjects or RoaObjects respectively. This was done for every repository available. Test plan 1 consists of 744 HTTP requests.
2. Test plan 2 is aimed at the RepositoryView performance. It involves switching between RepositoryView implementations (Tree and Table) to test responsiveness and synchronization, and using the filter functionality to find specific objects. Test plan 2 consists of 649 HTTP requests.
3. Test plan 3 is aimed at the Statistics part. It involves looking at all generated charts for all repositories. Since the Statistics implementation was quite simple and bareboned at the time of testing, this test plan only has 115 HTTP requests.

Using JMeter we replayed these test plans one by one with 50 threads simultaneously sending the recorded requests sequence. We modified the request timing with a random gaussian timer, so that not all 50 threads would send the same request at the same exact time. In addition to that, the threads were not started simultaneously, but over a ramp-up period of 2 seconds. The MIRO system was run using the Apache Tomcat [45] servlet container on a Arch Linux machine with a Intel i7-3630QM CPU and 8GB of RAM.

5.1.2 Results

Test plan 1 and 2

Test plan 1 showed a *median* response time of 5 milliseconds. The *average* response time however was 176 milliseconds. The minimum response time was 1 millisecond, while the maximum was **114,444 seconds**. For 90% of all requests, the response time was below 29 milliseconds.

Test plan 2 showed a *median* response time of 6 milliseconds and an *average* response time of 119 milliseconds. Similar to plan 1, there were some massive response time spikes with the maximum being **44,488 seconds**. As with plan 1, the minimum response time was 1 milliseconds. 90% of all requests had a response time below 50 milliseconds.

We investigated the massive response time spikes and found them to be caused exclusively by requests to the JFace TreeViewer widget. One response time spike was caused by the TreeViewer when expanding a CertificateObject with multiple thousands of child objects, causing an instantiation of an internal data structure for every child object within the TreeViewer. We solved this by using the TreeViewers SWT.VIRTUAL flag, allowing for lazy instantiation of objects displayed by the TreeViewer. This improved performance dramatically and stopped most response time spikes.

A second cause for the spikes was found in the implementation of the TreeViewers setSelection method. This was brought to the attention of the Eclipse RAP developers by issuing an enhancement request.

Test plan 3

Test plan 3 showed a *median* response time of 7 milliseconds and an *average* response time of 15 milliseconds. The minimum response time was 1 milliseconds and maximum 1175 milliseconds with 99% of requests having a response time of 62 milliseconds or lower. The maximum spike can be explained by the initial instantiation of the Statistics widgets.

The JMeter tests also revealed a memory leak. We analyzed this further using the Eclipse Memory Analyzer (MAT) [46] and were able to fix the leak.

5.2 Repository Structure Analysis

In section 3.2.1 we discussed the problem of repository structure and prefetching as a solution. In this section we present our findings on the performance of a simple prefetching solution in comparison to regular fetching. The prefetched URIs for all RIRs are shown in Table 5.1.

RIR	Prefetched URIs
AFRINIC	rsync://rpki.afrinic.net/repository
	rsync://rpki.afrinic.net/member_repository
	rsync://rpki.afrinic.net/rpki
APNIC	rsync://rpki.apnic.net/repository
	rsync://rpki.apnic.net/member_repository
ARIN	rsync://rpki.arin.net/repository
LACNIC	rsync://repository.lacnic.net/rpki
RIPE	rsync://rpki.ripe.net/repository

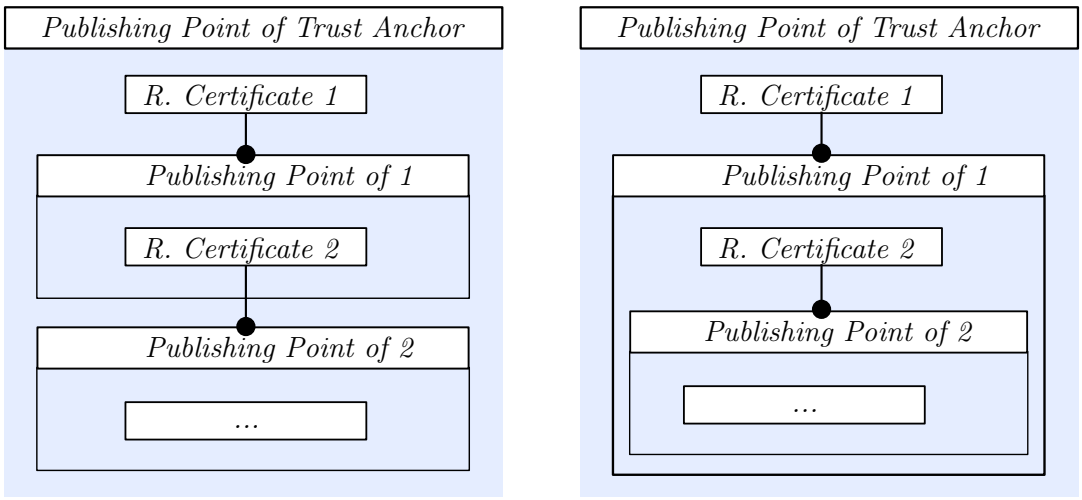
Table 5.1: Prefetched URIs for RIR repositories

Table 5.2 shows the performance comparison between regular fetching and prefetching for all 5 RIRs. Note that APNIC publishes 5 trust anchors, unlike the other RIRs. This was done in order to align APNIC's RPKI model with the overall administrative and associated registry structure of number resources in the Internet [47]. In our comparison we prefetched the complete physical APNIC repository which contains all 5 logical repositories referenced by the 5 APNIC trust anchors. In the cases of the ARIN, LACNIC, and RIPE trust anchors the results for regular fetching and prefetching are identical. This is because these three logical repositories are structured in a strictly hierarchical way: The ARIN, LACNIC, and RIPE trust anchors define rpki.arin.net/repository, repository.lacnic.net/rpki, and rpki.ripe.net/repository as their respective publishing points. These directories in turn contain all publishing points of all resource certificates that are part of their respective logical repositories.

	W/o Prefetching		W/Prefetching	
Repository	# rsync Calls	Time	# rsync Calls	Time
AfriNIC	35 calls	96 s	3 calls	11 s
APNIC	534 calls	1745 s	2 calls	35 s
APNIC AfriNIC	5 calls	16 s		
APNIC ARIN	25 calls	81 s		
APNIC IANA	492 calls	1610 s		
APNIC LACNIC	2 calls	6 s		
APNIC RIPE	10 calls	32 s		
ARIN	1 call	7 s	1 call	7 s
LACNIC	1 call	9 s	1 call	9 s
RIPE	1 call	23 s	1 call	23 s

Table 5.2: Performance comparison of collecting RPKI data with and without prefetching

In order to make a recommendation regarding repository structure, we define a logical repository to be **loosely hierarchical** if the following condition holds for all non self-signed



(a) A loosely hierarchical repository structure (b) A strictly hierarchical repository structure

Figure 5.1: Loosely and strictly hierarchical repository structures

CA resource certificates in the repository:

> The trust anchors publishing points contain the publishing points of all other resource certificates of the logical repository.

We define a logical repository to be **strictly hierarchical** if the following conditions holds for all non self-signed CA resource certificates in the repository:

> All publishing points defined by the resource certificate are included in the publishing points of its parent resource certificate.

A hierarchical repository (strict or loose) can be downloaded by relying parties with one rsync call for every publishing point of the trust anchor, thus minimizing overhead and making prefetching unnecessary. A strictly hierarchical repository further allows the downloading of subtrees with one rsync call. This property could prove useful with large repository where a relying party might only want to refresh a subset of the repository. A flat repository structure such as APNICs and AFRINICs forces relying parties to make multiple rsync calls to download the complete repository, potentially causing massive overhead seen in Table 5.2. The average percentage of repository files downloaded per request can be used as a metric to evaluate repository structure quality, as shown in Table 5.3.

Repository	Average % of Repository Files per rsync Call
AfriNIC	2.857 %
APNIC	
APNIC AfriNIC	20%
APNIC ARIN	4%
APNIC IANA	0.203%
APNIC LACNIC	50%
APNIC RIPE	10%
ARIN	100%
LACNIC	100%
RIPE	100%

Table 5.3: Average percentage of repository files downloaded per rsync call

We recommend a *strictly hierarchical* repository structure for repository maintainers. However, in the cases of APNIC and AFRINIC changing a flat repository to a strictly hierarchical one might require a change to the Subject Information Access field in a great number of resource certificates. Instead of switching to a strictly hierarchical structure, these RIRs could add new publishing points to their trust anchors that include the complete logical repository making loosely hierarchical. In the case of AFRINIC the trust anchor has only one publishing point: *rpki.afrinic.net/repository*. But due to the flat repository structure of AFRINIC, many resource certificates have publishing points in *rpki.afrinic.net/member_ repository* and *rpki.afrinic.net/rpki*. In order to avoid fetching these publishing points separately, one could add *rpki.afrinic.net/member_ repository* and *rpki.afrinic.net/rpki* as publishing points of the trust anchor. This would allow relying parties to fetch the complete repository in 3 calls without previous knowledge of the repository structure as prefetching would require.

CHAPTER 6

Outlook

6.1 Summary

The goal of the this bachelors thesis was the implementation of a system that allows monitoring and inspection of validated RPKI objects (MIRO) with a graphical user interface. This thesis presents the technical background necessary to understand the purpose of the MIRO system. The requirements for the software are discussed and a Model-View-Controller architecture of two loosely coupled components, the Validator and the Browser, is presented. The Validator component is responsible for downloading and validating RPKI repositories, while the Browser component implements a web application to inspect these repositories. Both components are implemented in Java, using the Eclipse Platform. The Browser component was implemented with the Eclipse Remote Application Framework (RAP) and was load tested with Apache JMeter to evaluate performance. This thesis also analyzes the significance of repository structure for relying parties and offers "Pre-Fetching" as a solution for the efficient download of non-hierarchical repositories. It also gives recommendations about repository structure to repository maintainers.

The source code for the MIRO system was published on GitHub [48] under the MIT license. A website with basic information about MIRO and references to the git repository is hosted at `http://rpki-miro.realmv6.org/`.

During the development of MIRO, two bugs were found in RIPE NCCs *rpki-commons* library pertaining validation of RPKI objects. A fix [49, 50] was implemented and were merged [51] into the master branch.

MIRO was presented at CeBIT 2015 as part of the Peeroskop project [52] funded by the Federal Ministry of Education and Research.

Following a call for input, the system has been used by researchers to identify that the cross-RIR resource space has increased and includes route origin authorization objects. It also helped to better understand how the five RIRs have implemented majority-minority address space, i.e., IP address space of which a super prefix is managed by a one RIR but some sub-prefixes of this prefix are managed by another RIR. This fragmentation has implications on creating resource certificates.

6.2 Future Work

Due to time constraints during the implementation of MIRO, some desired features were postponed. We plan to implement these features. In addition to new functionality for the users of MIRO, we also plan to make several non-functional changes that are aimed at improving performance and cover more use cases.

The planned new functionalities for users include:

1. Upload of external repository data: Give users of the web application the option of uploading, monitoring, and inspecting their own RPKI data through the web application GUI.
2. Monitoring alarms: Give users the option of configuring monitoring alarms that alert the user on repository changes.
3. Extend filter functionality: Extend the existing filter functionality to accept more complex filter criteria.
4. Statistic timeline: Reimplement the Statistics part in JavaScript and show historical data in addition to current snapshots.
5. Improved CLI for the Validator component to make distributed monitoring easier to set up.
6. Download of validated repositories: Allow users to download a complete, validated repository.

The non-functional changes include:

1. Improve overall *robustness* of the system. This includes improved logging, error handling, and additional memory analysis to find subtle memory leaks.
2. Implement Prefetching strategy that we presented in section 3.2.1.
3. Allow usage of X.509 Certificates with other or no extensions. This includes major changes to both the Validator and Browser component. For example to the validation step in the processing chain in order to allow third parties to easily implement extension specific validation algorithms and integrate them into the Validator.
4. Enable the Validator to parallelize the processing chain for multiple repositories.

Bibliography

[1] Rob Austein. Rpki utility programs. `http://rpki.net/wiki/doc/RPKI/Utils`, 2015. [Online; accessed 18-May-2015].

[2] Ethan Heilman, Danny Cooper, Leonid Reyzin, and Sharon Goldberg. From the consent of the routed: Improving the transparency of the rpki. In *Proc. of ACM SIGCOMM*, pages 51–62, New York, NY, USA, 2014. ACM.

[3] Matthias Wählisch, Olaf Maennel, and Thomas C. Schmidt. Towards Detecting BGP Route Hijacking using the RPKI. In *Proc. of ACM SIGCOMM, Poster Session*, pages 103–104, New York, August 2012. ACM.

[4] Jon Postel. Internet Protocol. RFC 791, IETF, September 1981.

[5] John Hawkinson and Tony Bates. Guidelines for creation, selection, and registration of an Autonomous System (AS). RFC 1930, IETF, March 1996.

[6] Internet Assigned Numbers Authority. Number resources. `http://www.iana.org/numbers`, 2015. [Online; accessed 17-March-2015].

[7] Kirk Lougheed and Jacob Rekhter. Border Gateway Protocol (BGP). RFC 1105, IETF, June 1989.

[8] Y. Rekhter, T. Li, and S. Hares. A Border Gateway Protocol 4 (BGP-4). RFC 4271, IETF, January 2006.

[9] M. Lad, R. Oliveira, B. Zhang, and L. Zhang. Understanding resiliency of internet topology against prefix hijack attacks. *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, 2007*, pages 368–377, 2007.

[10] X. Zhang H. Ballani, P. Francis. A study of prefix hijacking and interception in the internet. *SIGCOMM '07 Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 265–276, August 2007.

[11] Zheng Zhang, Ying Zhang, Y. Charlie Hu, and Z. Morley Mao. Practical defenses against bgp prefix hijacking. In *Proceedings of the 2007 ACM CoNEXT Conference*, CoNEXT '07, pages 3:1–3:12, New York, NY, USA, 2007. ACM.

[12] Changxi Zheng, Lusheng Ji, Dan Pei, Jia Wang, and Paul Francis. A Light-Weight Distributed Scheme for Detecting IP Prefix Hijacks in Real-Time. In *Proc. of SIGCOMM '07*, pages 277–288, New York, NY, USA, 2007. ACM.

[13] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. Internet

X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280, IETF, May 2008.

[14] Carlisle Adams and Steve Lloyd. *Understanding PKI.* Addison Wesley, 2 edition, November 2002.

[15] RSA Inc. Understanding public key infrastructure (pki), an rsa data security white paper. Technical report, RSA Data Security, 1999.

[16] ITU-T. Itu-t x.509 10/12. `http://www.itu.int/ITU-T/recommendations/rec.aspx?rec=X.509`, 2012. [Online; accessed 27-March-2015].

[17] PKIX WG. Pkix charter for working group. `http://datatracker.ietf.org/wg/pkix/charter/`, 1995. [Online; accessed 27-March-2015].

[18] M. Lepinski and S. Kent. An Infrastructure to Support Secure Internet Routing. RFC 6480, IETF, February 2012.

[19] Rob Austein, Steven Bellovin, Randy Bush, Russ Housley, Matt Lepinski, Stephen Kent, Warren Kumari, Doug Montgomery, Kotikalapudi Sriram, and Samuel Weiler. Bgpsec protocol specification draft-ietf-sidr-bgpsec-protocol-11. Internet-Draft draft-ietf-sidr-bgpsec-protocol-11, IETF Secretariat, January 2015. `https://tools.ietf.org/html/draft-ietf-sidr-bgpsec-protocol-11#page-34`.

[20] Wayne Davison. rsync web pages. `http://rsync.samba.org/`, 2015. [Online; accessed 02-April-2015].

[21] S. Weiler, D. Ward, and R. Housley. The rsync URI Scheme. RFC 5781, IETF, February 2010.

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

[23] G. Huston, S. Weiler, G. Michaelson, and S. Kent. Resource Public Key Infrastructure (RPKI) Trust Anchor Locator. RFC 6490, IETF, February 2012.

[24] M. Lepinski, A. Chi, and S. Kent. Signed Object Template for the Resource Public Key Infrastructure (RPKI). RFC 6488, IETF, February 2012.

[25] M. Lepinski, S. Kent, and D. Kong. A Profile for Route Origin Authorizations (ROAs). RFC 6482, IETF, February 2012.

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

[27] R. Austein, G. Huston, S. Kent, and M. Lepinski. Manifests for the Resource Public Key Infrastructure (RPKI). RFC 6486, IETF, February 2012.

[28] Tim Berners-Lee, Roy T. Fielding, and Larry Masinter. Uniform Resource Identifier (URI): Generic Syntax. RFC 3986, IETF, January 2005.

[29] Mark Reynolds and Stephen Kent. A profile for bgpsec router certificates, certificate revocation lists, and certification requests. Internet-Draft draft-ietf-sidr-bgpsec-pki-profiles-10, IETF Secretariat, January 2015. `http://www.ietf.org/internet-drafts/draft-ietf-sidr-bgpsec-pki-profiles-10.txt`.

[30] Rob A. Ripe goes hierarchical. January 2013. `http://www.hactrn.net/presentations/2013-01-15.ripe-goes-hierarchical.pdf`.

[31] Rob Austein. rcynic-summary. `http://www.hactrn.net/opaque/rcynic/`, 2015. [Online; accessed 1-May-2015].

[32] National Institute of Standards and Technology (NIST). rpki-monitor. `http://rpki-monitor.antd.nist.gov/`, 2015. [Online; accessed 1-May-2015].

[33] SURFnet / Jac Kloots. Rpki dashboard. `http://rpki.surfnet.nl/`, 2015. [Online; accessed 1-May-2015].

[34] RIPE NCC. Resource certification (rpki). `http://certification-stats.ripe.net/`, 2015. [Online; accessed 1-May-2015].

[35] M. Puzanov, O. Muravskiy, T. Bruijnzeels, E. Rozendaal, Y. Gonianakis, A. Band, A. Snare, and H. Westerbeek. rpki-commons. `https://github.com/RIPE-NCC/rpki-commons`, 2014.

[36] M. Puzanov, O. Muravskiy, T. Bruijnzeels, E. Rozendaal, Y. Gonianakis, A. Band, and A. Snare. rpki-validator. `https://github.com/RIPE-NCC/rpki-validator`, 2014.

[37] Joshua Bloch. *Effective Java (2nd Edition) (The Java Series)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2 edition, 2008.

[38] Rob Austein. rpki.net project. `http://www.rpki.net`, 2015. [Online; accessed 1-May-2015].

[39] J. Wilson et al. google-gson. `https://github.com/Google/gson`, 2014.

[40] Eclipse remote application platform. `http://eclipse.org/rap`, 2015. [Online; accessed 28-April-2015].

[41] Jface. `https://wiki.eclipse.org/JFace`, 2015. [Online; accessed 28-April-2015].

[42] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison Wesley, 2 edition, 1994.

[43] Ralf Sternberg. rap-d3charts. `https://github.com/ralfstx/rap-d3charts`, 2014.

[44] Apache jmeter. `http://jmeter.apache.org`, 2015. [Online; accessed 5-April-2015].

[45] Apache tomcat. `http://tomcat.apache.org/`, 2015. [Online; accessed 2-April-2015].

[46] Eclipse memory analyzer (mat). `https://eclipse.org/mat/`, 2015. [Online; accessed 2-May-2015].

[47] APNIC. Resource public key infrastructure (rpki). `https://www.apnic.net/services/services-apnic-provides/helpdesk/faqs/rpki#changes`, 2015. [Online; accessed 7-May-2015].

[48] Andreas Reuter and Matthias Wählisch. Rpki miro. `https://github.com/rpki-miro`, 2015.

[49] Andreas Reuter. rpki-commons. `https://github.com/RIPE-NCC/rpki-commons/commits/9afa034e72e9f2cf513c03c85db40a37de75c404`, 2014.

[50] Andreas Reuter. rpki-commons. `https://github.com/RIPE-NCC/rpki-commons/commits/400e884e3a6a45fcc5cb7ad66519fdc3ada0760e`, 2014.

[51] Thiago da Cruz. rpki-commons. `https://github.com/RIPE-NCC/rpki-commons/commits/dea01843365dbbd1de66b39e503c2d027775c0a1`, 2015.

[52] Peeroskop - peering monitor and microscopic analysis of the internet, 2015. [Online; accessed 7-May-2015].