Reputation Network Documentation

Release 0.1

Reputation Network

Nov 08, 2018

Contents

The Reputation Network, or REY, is a platform that enables safe, traceable personal data processing. It can be used by banks to use next-generation credit scores that safely use personal data from different data sources. It can also be used simply as a safer alternative to OAuth. Or it can be used to perform simple checks on personal data, like income checks based on credit card data, without letting the client know the subject's actual monthly income.

This is the documentation for building and deploying REY apps.

Note: There is a tech paper that discusses the Reputation Network design details in depth. For a conceptual description, the former paper is the reference to check.

CHAPTER 1

Contents

Keyword Index, Search Page

1.1 Introduction

The Reputation Network (or simply REY) enables safe, traceable personal data processing for a wide range of applications such as alternative risk scoring.

1.1.1 Interaction

As a user, making use of REY implies having control over your personal data when enabling third parties make use of it.

It differs from other approaches in that it respects the federated architecture of the Internet, where data are scattered across different, heterogeneous sources. Blockchain is used as decentralized ledger for keeping track of data usage.

In order to use a REY app, a user needs to follow the next steps:

- Download Metamask to create a decentralized identity, which simply consists of a pair of cryptographic keys.
- Submit write permissions to trusted websites where you have personal data. This will let them share data about you in REY to those readers you proactively accept. To achieve this, you just need to browse REY-enabled sites that would guide you and request your browser extension to build and sign the write permission using your private key.
- Then you're ready to use a REY app by providing the read permissions it could require. For example, the "Your entropy" sample app provides a measure of the amount of information you have in your Traity profile. To make it work, it requires your read permission to use the app, and your read permission for the app to read your Traity data.
- You can browse your exchanges of data at reputation.network. That's a blockchain explorer for REY that shows how, when, and by whom were your permissions used.

Note: There is a tech paper that discusses the Reputation Network design details in depth. For a conceptual description, the former paper is the reference to check.

1.1.2 Sample apps

In the *tutorial* section you can find the quick start guide to create your first Reputation Network app.

Some sample apps have been already built as demo for developers. They use Traity as data source and do a computation on its data. They are deployed and ready to use without cost:

- Bestreads app: share your reading score.
- Birthdoc app: share your verified birthdate.
- Federation app: bootstrap your reputation federation.
- Githubber app: share your programming languages.
- Magicnumber app: share your magic number.
- Spotistyle app: share your musical tastes.

1.2 Installation

You'll need a Linux or OS X system with Docker installed.

The different REY services can be run using REY's command-line interface rey-cli. To install `rey-cli`, simply run this command:

```
curl -o- https://raw.githubusercontent.com/reputation-network/rey-cli/master/bin/rey-
↪cli-install.sh | sh
```

You will have the `rey-cli` command available in your path. Type `rey-cli` to view the help.

1.3 Tutorial

Two REY apps are described step by step in this section. The hello world app is as simple as it could be, while the risk score app showcases using other REY apps as dependency.

1.3.1 Hello world app

Note: This tutorial has its source code on GitHub and it's available live at Heroku.

Our first REY app will simply return a magic number that is calculated out of the subject's public key. The steps that are involved are:

- Start a blockchain node.
- Implement the app's logic in a service.
- Start the gatekeeper to make the service available in a secure way.

- Publish the app in the registry.

Requirements

Make sure to have followed the *installation* instructions.

Start a blockchain node

A blockchain node is required to interact with REY's smart contract. For development purposes, we'll use a private blockchain node to avoid worrying about gas costs when interacting with the smart contract.

A docker image with REY's registry and smart contract already published is available, so simply run:

```
$ rey-cli dev node
```

This will launch the *development environment* blockchain node with a port open to RPC connections. So far, we just need to know that the account with address `0x88032398beab20017e61064af3c7c8bd38f4c968` has funds and is available to be used. As said, the smart contracts are already deployed and have the following addresses:

- Main smart contract address: `0x76C19376b275A5d77858c6F6d5322311eEb92cf5`
- Registry address: `0x556ED3bEaF6b3dDCb1562d3F30f79bF86fFC05B9`

Run a verifier

A *verifier* is an important element in REY's architecture that checks that apps run as expected. From the practical point of view, a verifier needs to be running to use REY apps.

To run a verifier, simply run:

```
$ rey-cli dev verifier -e VERIFIER_ADDRESS=0x44f1d336e4fdf189d2dadd963763883582c45312
```

The verifier needs to be published on blockchain so that clients can find out its endpoint. To do so, just run:

```
$ rey-cli dev cmd publish-manifest 0x44f1d336e4fdf189d2dadd963763883582c45312 http://
↪localhost:8082/manifest
```

Note: The development blockchain node has built-in accounts that have no password. When running REY commands, simply enter a blank password when prompted.

App logic

We'll build a simple Ruby service that computes the magic number given a subject's public key. Let's write a file called `helloworld.rb`:

```
require 'sinatra'
require 'sinatra/json'
require 'base64'

set :port, 8080

USERNAME, PASSWORD = (ENV['AUTHENTICATION'] || 'user:password').split(':')
```

(continues on next page)

(continued from previous page)

```
MANIFEST = {
  version: '1.0',
  name: 'Hello World',
  description: 'Returns a magic number',
  homepage_url: "http://localhost:8081",
  address: '0x88032398beab20017e61064af3c7c8bd38f4c968',
  app_url: 'http://localhost:8081/data',
  app_reward: 0,
  app_schema: { data: 30 },
  app_dependencies: []
}.freeze
APP_SEED = (MANIFEST[:address] + ENV['SECRET_SALT'].to_s).to_i(16).freeze

use Rack::Auth::Basic, "Protected Area" do |username, password|
  username == USERNAME && password == PASSWORD
end

def parse_subject_header(headers)
  Base64.decode64(headers['HTTP_X_PERMISSION_SUBJECT'] || 'null').gsub(/\A"|"\Z/, '')
end

get '/manifest' do
  json MANIFEST
end

get '/data' do
  subject_seed = parse_subject_header(request.env).to_i(16)
  json data: Random.new(APP_SEED + subject_seed).rand
end
```

The previous script requires the Ruby language and the Sinatra library (`gem install sinatra sinatra-contrib`) and can be run with:

```
$ ruby helloworld.rb
```

This will launch a server that listens on port 8080 and has two endpoints:

- `/manifest`: Returns the following manifest file that is used to provide basic information about the app:

```
{
  "version": "1.0",
  "name": "Hello World",
  "description": "Returns a magic number",
  "address": "0x88032398beab20017e61064af3c7c8bd38f4c968",
  "homepage_url": "http://localhost:8081",
  "app_url": "http://localhost:8081/data",
  "app_reward": 0,
  "app_schema": { "data": 30 },
  "app_dependencies": []
}
```

As you can see, we're using the address `0x88032398beab20017e61064af3c7c8bd38f4c968` to identify the app. This address was mentioned before, as it's one of the accounts that are funded and ready to use in the development blockchain node. A similar process would be required in a production environment (i.e., obtaining an account and funding it).

The schema shows the expected output of the app, which in this case will be an object with just a key called `data` and a value that can have a JSON-stringified length of up to 30 bytes. You can learn more about defining an app schema

in the *schema section*.

- `/data`: Returns the actual output of the app (a magic number).

As you can see, there's no kind of permission check in the service. This is left to REY's Gatekeeper.

Launch gatekeeper

REY's Gatekeeper is a proxy that implements most of REY's protocol to facilitate building REY apps. The Ruby service built previously does not have any kind of permission check, as this task is delegated to REY's Gatekeeper, which can fulfil the task with little configuration.

To run the gatekeeper, simply use:

```
$ rey-cli dev gatekeeper -e TARGET_URL=http://user:password@localhost:8080 -e␣
↪MANIFEST_URL=http://user:password@localhost:8080/manifest -e APP_
↪ADDRESS=0x88032398beab20017e61064af3c7c8bd38f4c968
```

It requires some parameters to specify where to find the manifest, the app's endpoint, and the app's address.

Notice that the Ruby service is not publicly accessible (it requires HTTP authentication). It does not make any kind of access check, so only the gatekeeper should be publicly accessible. This is why the app's manifest file has port number 8081 as `app_url`: app clients should query the gatekeeper, while the Ruby service requires HTTP authentication that only the gatekeeper should know.

Publishing the app

The app needs to be published in REY's registry so that others can find it just by its public key. The registry associates a public key with its manifest URL.

You can publish the app's manifest with:

```
$ rey-cli dev cmd publish-manifest 0x88032398beab20017e61064af3c7c8bd38f4c968 http://
↪localhost:8081/manifest
```

Remember that the manifest URL needs to be gatekeeper's one, as that's the one that does not require authentication. Gatekeeper will proxy the request to the manifest provided by the Ruby service.

Reading the app

You can now query your app for data, but first you need to have a blockchain identity. For simplicity we will use one of the already available idenitities (also known as accounts) on the development node, whose address is `0x60cb2204f342dd35bf5a328a03d86dd71d4372ec`.

To read what the app (with address `0x88032398beab20017e61064af3c7c8bd38f4c968`) returns about a subject (with address `0x60cb2204f342dd35bf5a328a03d86dd71d4372ec`), simply use with the following command:

```
$ rey-cli dev cmd read-app 0x88032398beab20017e61064af3c7c8bd38f4c968␣
↪0x60cb2204f342dd35bf5a328a03d86dd71d4372ec
```

1.3.2 Risk score app

Warning: This tutorial app is unfinished, but left here as a starting point. Stay tuned for the final version.

The sample risk score app is similar to the hello world one but has an app dependency. It gathers data from one app to build a risk score out of them.

In general, a risk score would involve some computation out several data sources. In this case, we'll simply extract the magic number generated in the *hello world example* and multiply it by 2. The steps that are involved are:

- Start a blockchain node.
- Run hello world app.
- Implement the app's logic in a service.
- Start the gatekeeper to make the service available in a secure way.
- Publish the app in the registry.

Requirements

Make sure to have followed the *installation* instructions.

Also, you should have already built the *hello world app*, as it is used as a dependency in the risk score app.

Start a blockchain node

A blockchain node is required to interact with REY's smart contract. For development purposes, we'll use a private blockchain node to avoid worrying about gas costs when interacting with the smart contract.

A docker image with REY's registry and smart contract already published is available, so simply run:

```
$ rey-cli dev node
```

This will launch the *development environment* blockchain node with a port open to RPC connections. So far, we just need to know that the accounts `0x88032398beab20017e61064af3c7c8bd38f4c968` and `0x6d644c57247de51da20797f14dceedfbc4ef6561` have funds and are available to be used. As said, the smart contracts are already deployed and have the following addresses:

- Main smart contract address: `0x76C19376b275A5d77858c6F6d5322311eEb92cf5`
- Registry address: `0x556ED3bEaF6b3dDCb1562d3F30f79bF86fFC05B9`

Run a verifier

A *verifier* is an important element in REY's architecture that checks that apps run as expected. From the practical point of view, a verifier needs to be running to use REY apps.

To run a verifier, simply run:

```
$ rey-cli dev verifier -e VERIFIER_ADDRESS=0x44f1d336e4fdf189d2dadd963763883582c45312
```

The verifier needs to be published on blockchain so that clients can find out its endpoint. To do so, just run:

```
$ rey-cli dev cmd publish-manifest 0x44f1d336e4fdf189d2dadd963763883582c45312 http://
↪localhost:8082/manifest
```

Note: The development blockchain node has built-in accounts that have no password. When running REY commands, simply enter a blank password when prompted.

Run hello world app

We need the hello world app up and running. In general, each app would be run by their own company, so each would use different blockchain nodes (hosted in each company's premises), with their own accounts.

However, for this example, we'll have both apps (hello world and risk score) share the same blockchain node. This means that, in order to start the hello world app, we'll just need to launch its own gatekeeper and its `server.rb` file.

To run hello world's main service:

```
$ ruby helloworld.rb
```

And to run hello world's gatekeeper:

```
$ rey-cli dev gatekeeper -e TARGET=http://user:password@localhost:8080/data␣
↪MANIFEST=http://localhost:8080/manifest APP_
↪ADDRESS=0x88032398beab20017e61064af3c7c8bd38f4c968
```

App logic

We'll build a service in Node.js that makes some computation out of the magic number that the Hello World app provides. Let's write a file called `server.js`:

```
const express = require('express');
const auth = require('http-auth');
const app = express();

const [username, password] = (process.env.AUTHENTICATION || 'user:password').split(':
↪');
const manifest = {
  "version": "1.0",
  "name": "Risk Score",
  "description": "Returns a risk score based on your data",
  "address": "0x6d644c57247de51da20797f14dceedfbc4ef6561",
  "homepage_url": "http://localhost:8083",
  "app_url": "http://localhost:8083/data",
  "app_reward": 0,
  "app_dependencies": ['0x88032398beab20017e61064af3c7c8bd38f4c968'],
};
const basic = auth.basic({ realm: "Web." }, (u, p, callback) => callback(u ===␣
↪username && p === password));
const wrap = fn => (...args) => fn(...args).catch(args[2]);

app.get('/manifest', auth.connect(basic), (req, res) => res.send(manifest));

app.get('/data', auth.connect(basic), wrap(async (req, res) => {
  const value = await parseInt(Math.random() * 10000);
```

(continues on next page)

(continued from previous page)

```
  res.send({ "data": value * 2 });
}));

app.listen(8082);
```

The previous script requires Node.JS and its dependencies can be installed with (`npm install express http-auth --save`). It can be run with:

```
$ node server.js
```

This will launch a server that listens on port 8082 and has two endpoints:

- `/manifest`: Returns the following manifest file that is used to provide basic information about the app:

```
{
  "version": "1.0",
  "name": "Risk Score",
  "description": "Returns a risk score based on your data",
  "address": "0x6d644c57247de51da20797f14dceedfbc4ef6561",
  "homepage_url": "http://localhost:8083",
  "app_url": "http://localhost:8083/data",
  "app_reward": 0,
  "app_dependencies": ['0x88032398beab20017e61064af3c7c8bd38f4c968']
}
```

As you can see, we're using the address `0x6d644c57247de51da20797f14dceedfbc4ef6561` to identify the app. This address was mentioned before, as it's one of the accounts that are funded and ready to use in the development blockchain node, and it's different from hello world's address. Also, note that the dependency of hello world app is stated by including risk score app's address in the `app_dependencies` list.

The schema shows the expected output of the app, which in this case will be an object with just a key called `data` and a value that can have a JSON-stringified length of up to 30 bytes. You can learn more about defining an app schema in the *schema section*.

- `/data`: Returns the actual output of the app.

Launch gatekeeper

REY's Gatekeeper is a proxy that implements most of REY's protocol to facilitate building REY apps. The Ruby service built previously does not have any kind of permission check, as this task is delegated to REY's Gatekeeper, which can fulfil the task with little configuration.

To run the gatekeeper, simply use:

```
$ rey-cli dev gatekeeper -p 8083 -e TARGET=http://user:password@localhost:8082 -e␣
↪MANIFEST=http://user:password@localhost:8082/manifest -e APP_
↪ADDRESS=0x6d644c57247de51da20797f14dceedfbc4ef6561
```

It requires some parameters to specify where to find the manifest, the app's endpoint, and the app's address. Note that this is risk score's gatekeeper, so we need to specify a port to prevent overlapping with hello world's gatekeeper port.

Publishing the app

The app needs to be published in REY's registry so that others can find it just by its public key. The registry associates a public key with its manifest URL.

You can publish the app's manifest with:

```
$ rey-cli dev cmd publish-manifest 0x6d644c57247de51da20797f14dceedfbc4ef6561 http://
↪localhost:8083/manifest
```

Reading the app

You can now query your app for data, but first you need to have a blockchain identity. For simplicity we will use one of the already available idenitities (also known as accounts) on the development node, whose address is `0x60cb2204f342dd35bf5a328a03d86dd71d4372ec`.

To read what the app (with address `0x6d644c57247de51da20797f14dceedfbc4ef6561`) returns about a subject (with address `0x60cb2204f342dd35bf5a328a03d86dd71d4372ec`), simply use with the following command:

```
$ rey-cli dev cmd read-app 0x6d644c57247de51da20797f14dceedfbc4ef6561␣
↪0x60cb2204f342dd35bf5a328a03d86dd71d4372ec
```

1.4 Environments

REY apps can be built locally in a development environment before being published on the testnet environment. This section describes the differences between both.

When using `rey-cli`, you can easily switch from one environment to another by changing the first argument (`dev` or `test`). You can also set a `custom` environment if you're running your own node, and specify your RPC URL and contract addresses.

1.4.1 Development

Before making our app available to everyone, we'll be building it inside our own development environment. To achieve this, the simplest way is to run your own private node with REY's smart contract and registry already deployed. There's a Docker image that provides a `geth` instance that has both smart contracts already deployed and ready to use at the following addresses:

- Main smart contract address: `0x76C19376b275A5d77858c6F6d5322311eEb92cf5`.
- Registry address: `0x556ED3bEaF6b3dDCb1562d3F30f79bF86fFC05B9`.

To start the node, make sure to have Docker properly installed, then run:

```
$ rey-cli dev node
```

This will launch the geth node with an open port to connect via RPC and interact with it. The following accounts have funds and are available to be used in the node:

- `0x31bb9d47bc8bf6422ff7dcd2ff53bc90f8f7b009`
- `0x88032398beab20017e61064af3c7c8bd38f4c968`
- `0xc25b4ff9eb6f52392eef1e103daacc7519795f01`
- `0x6d644c57247de51da20797f14dceedfbc4ef6561`
- `0xe370c47450427a2baa9bff3557bf574162f3ca54`
- `0xefdd1029b00e1add52c478f85c00c1011a347128`

- `0x60cb2204f342dd35bf5a328a03d86dd71d4372ec`
- `0x6224d471b8590de463d27b067174b566b4b0b041`
- `0x128ab682efe2a1ec3970d374d23a7f249fb9e8df`
- `0x44f1d336e4fdf189d2dadd963763883582c45312`

Note: The development blockchain node has built-in accounts that have no password. When running REY commands, simply enter a blank password when prompted.

The *tutorial* shows how to interact with the registry and smart contract to publish a hello world app on the development environment.

1.4.2 Testnet

The testnet enables your REY app to work outside your own development environment (i.e., your laptop or computer). It is available at Rinkeby Ethereum Testnet with the following addresses:

- Main smart contract address: `0xe410f8ff9ce89b2c2bd940967cac9dade139a0c7` (view on Etherescan).
- Registry address: `0xC05f9be01592902e133F398998E783b6cbD93813` (view on Etherscan).

To use the testnet, you'll need to set up your own Rinkeby node with your own, funded accounts. The contract addresses are already set up every time you pass `test` as environment to `rey-cli`.

You can use Traity's verifier (`0xd91f44fee5e3b81f61b4e7ab7fddf3f4caab1220`) to run your REY apps.

1.5 Reference

1.5.1 Permissions

Write permission set

A sample write permission set looks as follows:

```
{
  "subject": "0x6224d471b8590de463d27b067174b566b4b0b041",
  "writePermissions": [ ... ]
  "signature": {
    "r": "0xd442661375322973148b808f4bcb3fedffab5d5855acf2e7e2c37745676f5a23",
    "s": "0xee1026f612fbf69e0195e1cef5d29a3fe65f7a2575031bb1818150b090df509c",
    "v": "0x1d"
  }
}
```

The individual *write permissions* have been ommitted from the JSON object above. Each of the write permission should have a different subject address equal to each of the derived child keys from the write permission set's subject address (which is the subject's main public key).

Write permission

A sample write permission looks as follows:

```
{
  "writer": "0xc25b4ff9eb6f52392eef1e103daacc7519795f01",
  "subject": "0x6224d471b8590de463d27b067174b566b4b0b041",
  "signature": {
    "r": "0xd442661375322973148b808f4bcb3fedffab5d5855acf2e7e2c37745676f5a23",
    "s": "0xee1026f612fbf69e0195e1cef5d29a3fe65f7a2575031bb1818150b090df509c",
    "v": "0x1d"
  }
}
```

Read permission

A sample read permission looks as follows:

```
{
  "reader": "0xc25b4ff9eb6f52392eef1e103daacc7519795f01",
  "source": "0x88032398beab20017e61064af3c7c8bd38f4c968",
  "subject": "0x6224d471b8590de463d27b067174b566b4b0b041",
  "expiration": "2314320941",
  "signature": {
    "r": "0xd442661375322973148b808f4bcb3fedffab5d5855acf2e7e2c37745676f5a23",
    "s": "0xee1026f612fbf69e0195e1cef5d29a3fe65f7a2575031bb1818150b090df509c",
    "v": "0x1d"
  }
}
```

1.5.2 Transactions

A transaction is simply a *request* and a *proof*, signed by a verifier, and looks as follows:

```
{
  "writer": "0xc25b4ff9eb6f52392eef1e103daacc7519795f01",
  "subject": "0x6224d471b8590de463d27b067174b566b4b0b041",
  "signature": {
    "r": "0xd442661375322973148b808f4bcb3fedffab5d5855acf2e7e2c37745676f5a23",
    "s": "0xee1026f612fbf69e0195e1cef5d29a3fe65f7a2575031bb1818150b090df509c",
    "v": "0x1d"
  }
}
```

Request

A request consists of a *session* and a *read permission* and looks as follows:

```
{
  "readPermission": { ... },
  "session": { ... },
  "signature": {
    "r": "0xd442661375322973148b808f4bcb3fedffab5d5855acf2e7e2c37745676f5a23",
```

(continues on next page)

(continued from previous page)

```
    "s": "0xee1026f612fbf69e0195e1cef5d29a3fe65f7a2575031bb1818150b090df509c",
    "v": "0x1d"
  }
}
```

A request must be signed by the actor making the call to the REY app specified as source in the included read permission.

Proof

A proof consists of a *session* and a *write permission* and looks as follows:

```
{
  "writePermission": { ... },
  "session": "0xc25b4ff9eb6f52392eef1e103daacc7519795f01",
  "signature": {
    "r": "0xd442661375322973148b808f4bcb3fedffab5d5855acf2e7e2c37745676f5a23",
    "s": "0xee1026f612fbf69e0195e1cef5d29a3fe65f7a2575031bb1818150b090df509c",
    "v": "0x1d"
  }
}
```

A proof must be signed by the write permission's source.

Session

A sample session looks as follows:

```
{
  "subject": "0x6224d471b8590de463d27b067174b566b4b0b041",
  "verifier": "0xc25b4ff9eb6f52392eef1e103daacc7519795f01",
  "fee": "10", /* in parts per million */
  "nonce": "75482967549",
  "signature": {
    "r": "0xd442661375322973148b808f4bcb3fedffab5d5855acf2e7e2c37745676f5a23",
    "s": "0xee1026f612fbf69e0195e1cef5d29a3fe65f7a2575031bb1818150b090df509c",
    "v": "0x1d"
  }
}
```

1.5.3 Gatekeeper

REY's gatekeeper is a proxy to a target server that provides the actual functionality of a REY app. Using the gatekeeper is the easiest way to publish an app in REY.

The main idea is to abstract app developers from REY's protocol to just focus on the app's business logic.

Take, for example, an app that returns purchase data about people in an e-commerce. The gatekeeper would be the server that is available to the outer world. It'd process permissions and perform cashouts, and will forward valid requests to the target server that implements the actual business logic and returns the output.

Installation

To launch gatekeeper, simply use `rey-cli`:

```
$ rey-cli dev gatekeeper -e TARGET=http://127.0.0.1:9400/data -e MANIFEST=http://127.
↪0.0.1:9400/manifest -e APP_ADDRESS=0x88032398beab20017e61064af3c7c8bd38f4c968
```

Configuration

The gatekeeper is configured using the following environment variables:

- `APP_ADDRESS`: The public key of the app. It's expected that the private key is stored in the blockchain node.
- `TARGET`: Specifies the endpoint URL of the target server (i.e., the server that provides the actual implementation of the app). It must include any needed path or basic authentication credentials for the server.
- `MANIFEST`: Specifies where to find the manifest of the app. It must include any needed path or basic authentication credentials for the server.
- `DEPENDENCIES`: Comma-separated list of any other apps (as public keys) that might be used by this app. With this, the gatekeeper would reject those calls that do not include read permissions to use these apps.

1.5.4 Serializations

There are different ways of representing objects in REY.

Take the following read permission as example:

```
{
  "reader": "0xc25b4ff9eb6f52392eef1e103daacc7519795f01",
  "source": "0x88032398beab20017e61064af3c7c8bd38f4c968",
  "subject": "0x6224d471b8590de463d27b067174b566b4b0b041",
  "expiration": "2314320941",
  "signature": {
    "r": "0xd44266137532297 3148b808f4bcb3fedffab5d5855acf2e7e2c37745676f5a23",
    "s": "0xee1026f612fbf69e0195e1cef5d29a3fe65f7a2575031bb1818150b090df509c",
    "v": "0x1d"
  }
}
```

When submitting it to the smart contract, it's internally transformed to an array format instead of using objects:

```
[
  "0xc25b4ff9eb6f52392eef1e103daacc7519795f01",
  "0x88032398beab20017e61064af3c7c8bd38f4c968",
  "0x6224d471b8590de463d27b067174b566b4b0b041",
  "2314320941",
  [
    "0xd44266137532297 3148b808f4bcb3fedffab5d5855acf2e7e2c37745676f5a23",
    "0xee1026f612fbf69e0195e1cef5d29a3fe65f7a2575031bb1818150b090df509c",
    "0x1d"
  ]
]
```

Additionally, when submitting JSON objects, JWT is used as standard for HTTP services, while ABI is used when submitting data to the smart contract.

As a result, there are two possible serializations that are used within REY:

- JSON objects with key and values, serialized as a non-signed JWT. This is used in traditional HTTP services.
- JSON arrays with only values, serialized as tightly packed ABI data. This is used when submitting data to the smart contract.

Note that big numbers are always expressed as strings to prevent precision issues in JavaScript.

1.5.5 Schema

A schema is the expected structure of an app's output. REY apps return JSON output which, by definition, is a combination of arrays and key-value objects.

An app's schema is present in the app's *manifest file* so that the subject commits themselves to a specific output of the app when building a read permission.

Schemas are checked by verifiers when data transfers happen between an app and its client. The verifier validates that the output's structure matches the one the read permission was built for.

Examples

Defining a schema for an app is quite simple. Let's assume we're building an app that returns the favourite city of a person:

```
/* App sample output */
{ "city": "Madrid" }
```

The schema of this output will just be a JSON object with `city` as a key, and the maximum number of bytes of its value. Different cities have different lengths, so with this we're setting up a cap, so that, e.g., a JPG image can't be serialized and disguised as a city. The objective is to avoid unsolicited data leaks in an app's output. Thus, the schema will be:

```
/* App schema */
{ "city": 30 }
```

With this, we've set a limit of 30 bytes for the value of `city` key. As the string `"Madrid"` has a JSON-stringified length of 8 bytes, we're within the 30 bytes limit, which means the app's output is valid for this schema.

Let's consider a more complex example. Now the app returns more detailed information about the city, incluing the country to disambiguate between, e.g., Madrid, Spain, and Madrid, USA.

```
/* App sample output */
{ "city": { "name": "Madrid", "country": "Spain" } }
```

For this type of output, the same approach would apply. We'll assume countries will have a maximum JSON-stringified length of 30 bytes. The app schema would be:

```
/* App schema */
{ "city": { "name": 30, "country": 30 } }
```

Finally, let's assume an app returns a list of up to 10 favourite cities. One sample output would be:

```
/* App sample output */
{ "cities": [ { "name": "Madrid", "country": "Spain" },
              { "name": "Valencia", "country": "Spain" },
              { "name": "San Francisco", "country": "USA" } ] }
```

To build the schema, we'll replace values with the maximum expected JSON-stringified length again. For the array, two elements are used: the first one is the expected schema of each item, and the last one is the maximum length of the array. As said, we'll only support a maximum of 10 favourite cities, so the schema is as follows:

```
/* App schema */
{ "cities": [ { "name": 30, "country": 30 }, 10 ] }
```

It can be observed that there are no specific type checks. The objective of schema checks is not doing type checks, but to prevent data leaks. Therefore, the amount of allowed data is what only matters, as both numbers or strings can be disguised as different data types by using different encodings.

An advanced user would be able to observe that, given the app schema shown before, the app in question would be able to return a maximum of 10 x (30 + 30) = 600 bytes of data (including an overhead of double quotes). This measure is useful for the user to weight the amount of trust in the app and the amount of risk in the schema in order to decide whether to use the app or not.

Syntax

More formally, the syntax used to define a schema is shown here in Backus-Naur Form (BNF):

```
<schema>  ::= <object> | <array> | <number>;
<object>  ::= '{' '}' | '{' <members> '}';
<members> ::= <pair> | <pair> ',' <members>;
<pair>    ::= '"' <string> '"' ':' <number>;
<array>   ::= '[' <schema> ',' <number> ']';
```

Verification function

The function used by verifiers to check that an app's output fits its schema is the next one:

```
function checkSchema(data, schema) {
  if (typeof(schema) === 'number') {
    if (JSON.stringify(data).length > schema) {
      return false;
    }
  } else if (Array.isArray(schema)) {
    if (!Array.isArray(data)) {
      return false;
    }
    if (data.length > schema[schema.length - 1]) {
      return false;
    }
    for (let item of data) {
      if (!checkSchema(item, schema[0])) {
        return false;
      }
    }
  } else {
    if (typeof(data) !== 'object') {
      return false;
    }
    for (let key in data) {
      if (Object.keys(schema).indexOf(key) === -1) {
        return false;
      }
```

(continues on next page)

(continued from previous page)

```
      if (!checkSchema(data[key], schema[key])) {
        return false;
      }
    }
  }
  return true;
};
```

Encryption

Verifiers work on encrypted data. This means that they cannot just apply all of the above on raw data as is shown.

Let's consider again the following app output:

```
/* App sample output */
{ "city": "Madrid" }
```

To know what a verifier will be able to see, let's encrypt it using rey-sdk-js and some random key pair:

```
$ node
> const REY = require("rey-sdk");
> let key = new REY.Utils.EncryptionKey()
> key.createPair();
> key.encrypt({ "city": "Madrid" });
{ city: 'cFwfPaP3E/4tcryywWYEDN7go+pi1uTpA7jy7clI17KKO/nO0YuZ5vS3i7Ea9n/
↪y3LOF4cajYQOAQt/lBwDMsA==' }
```

As you can see, the string `"Madrid"` becomes `"cFwfPaP3E/4tcryywWYEDN7go+pi1uTpA7jy7clI17KKO/nO0YuZ5vS3i7Ea9n/y3LOF4cajYQOAQt/lBwDMsA=="`. This is an encrypted, Base64-encoded version of the string `"Madrid"`. Trying with a longer string produces a similar result:

```
> key.encrypt({ "city": "San Francisco" });
{ city: 'Dmr8kPZYYO9k3pQBhSRbp64bP2+fuOET7HcDONjXecFzvc9s77C6P2H/
↪xpLrww9ucjyDkH+YKljjepqor28ynQ==' }
```

The string `"San Francisco"` produces another Base64 string which has the same length as the previous one. As `"San Francisco"` is longer than `"Madrid"`, this means that the encryption algorithm hides the real length of the unencrypted data.

The `pkcs1` encryption algorithm being used provides an output whose length is a multiple of 64 bytes for every 22 bytes, which, after converting it to Base64, becomes even longer.

In order to estimate the length of the original, unencrypted data, the verifier reverse engineers the above formula. This lets verifiers estimate how many minimum bytes are actually being sent and detect those cases where there's a clear excess of information, compared to the schema.

As verifiers reverse engineer the unencrypted data length, encryption doesn't change the way schemas should be defined. Nevertheless, it's important to notice that a verifier cannot really make a difference between an unencrypted length of 14 and a length of 16 (as both would produce Base64 strings that are equally long). However, verifiers would spot a leak if the unencrypted length being transmitted is, e.g., 80, as it would produce a much longer encrypted string.

1.5.6 Registry

The registry enables apps to specify where to find their manifest file that shows what the app is about and how to run it. The registry is a smart contract whose data will typically need to be updated only once to specify the manifest URL.

Manifest file

A sample manifest file looks as follows:

```
{ "version": "1.0",
  "name": "Super Scoring App",
  "description": "This app provides great risk scores",
  "homepage_url": "http://example.com/super-scoring-app",
  "picture_url": "http://example.com/logo-square.png",
  "address": "0xc25b4ff9eb6f52392eef1e103daacc7519795f01",
  "app_url": "http://api.example.com/super-scoring-app/data",
  "app_reward": 10,
  "app_schema": { "score": 5 },
  "app_dependencies": [
    "0x31bb9d47bc8bf6422ff7dcd2ff53bc90f8f7b009",
    "0x88032398beab20017e61064af3c7c8bd38f4c968"
  ]
}
```

The manifest file must be serialized as JSON and has the following fields:

- Version: a version number of the app.
- Name: a human-readable name for the actor, which will be used in permission dialogs.
- Description: a human-readable description for the actor, which will be used in permission dialogs along with the name.
- Homepage URL: a URL where a user can get more information about the app, opt-in, and opt-out.
- Picture URL: a square picture that represents the actor, which will be used in permission dialogs along with the name and description.
- Address: the public key of the app.
- Verifier URL: a URL where to expect verifier functionality. If the actor is not a verifier, it can be omitted.
- Verifier fee: the expected fee for the verification service, in parts per million. If omitted, zero fee is assumed.
- App URL: a URL where to expect app functionality. If the actor is not an app, it can be omitted.
- App reward: the expected reward for the app's service. If omitted, zero reward is assumed.
- App schema: the *schema* to be used by the app when returning its output.
- App dependencies: a list of address for whom the app needs read permissions to properly run.

1.5.7 Verifier

REY's verifier is a proxy to REY apps that is an important part of REY's architecture.

Detailed information about it can be found at REY's tech paper. In essence, the verifier acts as a proxy to certify that the data transference between client and app conforms to a schema that has been accepted by the subject when signing the read permission. Thus, the verifier prevents data leaks in the system while being unable to view the actual data, which is transmitted in an encrypted way from REY apps to clients.

From the development point of view, a verifier needs to be running so that apps are queried through it by clients.

Running a verifier

To launch a verifier, simply use `rey-cli`:

```
$ rey-cli dev verifier -e VERIFIER_ADDRESS=0x44f1d336e4fdf189d2dadd963763883582c45312
```

This will launch a verifier at `localhost:8082` with the given blockchain address. However, this verifier needs to be published in the registry, so we'll need to run:

```
$ rey-cli dev cmd publish-manifest 0x44f1d336e4fdf189d2dadd963763883582c45312 http://
↪localhost:8082/manifest
```

Once that's done, apps can be queried through this verifier with:

```
$ rey-cli dev cmd read-app <APP_ADDRESS> <SUBJECT_ADDRESS>␣
↪0x44f1d336e4fdf189d2dadd963763883582c45312
```

Configuration

The verifier can be configured using the following environment variables:

- `VERIFIER_ADDRESS`: The address of this verifier instance.
- `LOG_LEVEL`: Minimum log level, defaults to `info`.
- `SECURED_PATH`: Path where the verifier's functionality is exposed, defaults to `/data`.
- `VERIFIER_FEE`: The fee (in parts per million) that the verifier will get out of each cashed out transaction.
- `MANIFEST_PATH`: The path where the manifest file will be published. Defaults to `/manifest`.
- `MANIFEST_VERSION`: The manifest's version.
- `MANIFEST_NAME`: The manifest's public name.
- `MANIFEST_DESCRIPTION`: The manifest's public description.
- `MANIFEST_HOMEPAGE_URL`: The manifest's home page URL.
- `MANIFEST_PICTURE_URL`: The manifest's public picture URL.
- `VERIFIER_URL`: The manifest's verifier URL. Defaults to `http://localhost:8082/data`.
- `VERIFIER_ACCOUNT_PASSWORD`: The verifier account's password that is used in the blockchain node. The account is used to sign transactions for the target apps.

1.5.8 Projects

REY's software is split across different projects that are available in several Github repositories. Here's a list of all of them.

rey-docs

This is REY's documentation (i.e., what you're reading now), which is deployed at Read the Docs.

- Github repository: reputation-network/rey-docs

rey-cli

This is the command-line interface for REY. It's the recommended way of using all the different projects and tools, as it facilitates managing different development environments and using the supplied docker images.

- Github repository: reputation-network/rey-cli

rey-cmd

This project is a set of scripts. They are the one-off commands which are executed from `rey-cli` to deploy a manifest file, fund a channel, run a REY app, etc. They are stored in a docker image to reduce the dependencies when running them from `rey-cli`.

- Github repository: reputation-network/rey-cmd
- Docker image: reputationnetwork/cmd

rey-sdk-js

This project is the JavaScript SDK for REY. It enables performing all of REY's actions, such as running apps or signing permissions, using a JavaScript interface.

- Github repository: reputation-network/rey-sdk-js

rey-gatekeeper

This project is the *gatekeeper*, a REY app proxy that deals with the REY protocol to simplify building REY apps. The gatekeeper is used in the *tutorial* section.

- Github repository: reputation-network/rey-gatekeeper
- Docker image: reputationnetwork/gatekeeper

rey-contracts

This project contains the smart contracts for the registry and the transaction tracking. They are deployed in a local blockchain node that is stored in a docker image. This simplifies development, as this blockchain node can be easily instantiated using `rey-cli dev node`.

- Github repository: reputation-network/rey-contracts
- Docker image: reputationnetwork/node

1.6 Links

- Website
- Source code
- Docker images
- Overview paper
- Tech paper
- Patent paper

- Discord chat

Index