NuCypher Documentation

Release 0.1.0-alpha.20

NuCypher

Mar 26, 2019

Guides

A proxy re-encryption network to empower privacy in decentralized systems

---

The NuCypher network uses the Umbral threshold proxy re-encryption scheme to provide cryptographic access controls for distributed apps and protocols.

1. Alice, the data owner, grants access to her encrypted data to anyone she wants by creating a policy and uploading it to the NuCypher network.

2. Using her policy's public key, any entity can encrypt data on Alice's behalf. This entity could be an IoT device in her car, a collaborator assigned the task of writing data to her policy, or even a third-party creating data that belongs to her – for example, a lab analyzing medical tests. The resulting encrypted data can be uploaded to IPFS, Swarm, S3, or any other storage layer.

3. A group of Ursulas, which are nodes of the NuCypher network, receive the access policy and stand ready to re-encrypt data in exchange for payment in fees and token rewards. Thanks to the use of proxy re-encryption, Ursulas and the storage layer never have access to Alice's plaintext data.

4. Bob, a data recipient, sends an access request to the NuCypher network. If the policy is satisfied, the data is re-encrypted to his public key and he can decrypt it with his private key.

More detailed information:

- GitHub https://www.github.com/nucypher/nucypher
- Website https://www.nucypher.com/

CHAPTER 1

Whitepapers

Network

https://github.com/nucypher/whitepaper/blob/master/whitepaper.pdf

"NuCypher - A proxy re-encryption network to empower privacy in decentralized systems" by Michael Egorov, David Nuñez, and MacLane Wilkison - NuCypher

Economics

https://github.com/nucypher/mining-paper/blob/master/mining-paper.pdf

"NuCypher - Mining & Staking Economics" by Michael Egorov, MacLane Wilkison - NuCypher

Cryptography

https://github.com/nucypher/umbral-doc/blob/master/umbral-doc.pdf

"Umbral A Threshold Proxy Re-Encryption Scheme" by David Nuñez - NuCypher

Warning: NuCypher is currently in the *Alpha* development stage and is **not** intended for use in production.

1.1 Nucypher Quickstart

1.1.1 Ursula

Install Nucypher

```
$ pip3 install -U nucypher
```

Run a Federated-Only Development Ursula

```
$ nucypher ursula run --dev --federated-only
```

Configure a Persistent Ursula

```
$ nucypher ursula init --federated-only
```

Run a Persistent Ursula

```
$ nucypher ursula run --teacher-uri <SEEDNODE_URI> --federated-only
```

Replace `<SEEDNODE_URI>` with the URI of a node running on the network and domain you want to connect to (for example `0.0.0.0:9151` or `0xdeadbeef@0.0.0.0:9151`).

Run a Geth-Connected Development Ursula

Run a local geth node in development mode:

```
$ geth --dev
```

Run a local development Ursula connected to the geth node

```
$ nucypher ursula run --dev --provider-uri ipc:///tmp/geth.ipc --checksum-address
↪<GETH_DEV_ADDRESS>
```

Replace `<GETH_DEV_ADDRESS>` with the geth node's public checksum address.

1.2 NuCypher Federated Testnet (NuFT) Setup Guide

This guide is for individuals who intend to spin-up and maintain an Ursula node in the early stages of the NuFT while working with the NuCypher team to improve user experience, comfort, and code-quality of the NuCypher network. Following the steps will launch a functioning federated-only Ursula node operating from a machine you control. Before getting started, please note:

- We encourage you to launch your node on a machine that is able to remain online with as few interruptions as possible, for as long as possible. Although it is possible to restart a node, excessive deactivation/reactivation can contribute to a lack of usable feedback from the test network. Additionally, node uptime will be important for proper functioning of Mainnet. Thus, it's worthwhile to start working with a reliable machine now so that your nodes are compliant with the network's uptime requirements when the time comes.
- Setup requires knowledge of your machine's public-facing IPv4 address and local network configuration. It is advised to use a static IP address since changing IP addresses will require node reconfiguration. Configure port-forwarding rules on port 9151 if you are operating a node behind a firewall. Once successfully up and running, your node will be discoverable by all other nodes in the test network.
- NuFT transmits application errors and crash reports to NuCypher's sentry server. This functionality is enabled by default for NuFT only and will be deactivated by default for mainnet.

Warning: The “NuCypher Federated Testnet” (NuFT) is an experimental pre-release of nucypher. Expect bugs, downtime, and unannounced domain-wide restarts. NuFT nodes do not connect to any blockchain. **DO NOT** perform transactions using NuFT node addresses.

Important: Exiting the setup process prior to completion may lead to issues/bugs. If you encounter issues, report feedback by opening an Issue on our GitHub (https://github.com/nucypher/nucypher/issues)

1.2.1 Contents

1.2.2 Configure a NuFT Node

1.2.3 Stage A | Install The Nucypher Environment

1. Install Python and Git

If you don’t already have them, install Python and git. As of January 2019, we are working with Python 3.6, 3.7, and 3.8.

- Official Python Website: https://www.python.org/downloads/
- Git Install Guide: https://git-scm.com/book/en/v2/Getting-Started-Installing-Git

2. Create Virtual Environment

Create a system directory for the nucypher application code:

```
$ mkdir nucypher
```

Create a virtual environment for your node to run in using `virtualenv`:

```
$ virtualenv nucypher -p python3
...
```

Activate your virtual environment:

```
$ source nucypher/bin/activate
...
(nucypher)$
```

3. Install Nucypher

Install `nucypher` with `git` and `pip3` into your virtual environment.

```
(nucypher)$ pip3 install nucypher
```

Note: We recommend NuFT nodes install directly from master to help ensure your node is using pre-released features and hotfixes

Re-activate your environment after installing

```
$ source nucypher/bin/activate
...
(nucypher)$
```

1.2.4 Stage B | Configure Ursula

1. Verify that the installation was successful

Activate your virtual environment and run the `nucypher --help` command

```
$ source nucypher/bin/activate
...
(nucypher)$ nucypher --help
```

You will see a list of possible usage options (`--version`, `-v`, `--dev`, etc.) and commands (`status`, `ursula`). For example, you can use `nucypher ursula destroy` to delete all files associated with the node.

2. Configure a new Ursula node

```
(nucypher)$ nucypher ursula init --federated-only
...
```

3. Enter your public-facing IPv4 address when prompted

```
Enter Node's Public IPv4 Address: <YOUR NODE IP HERE>
```

4. Enter a password when prompted

```
Enter a passphrase to encrypt your keyring: <YOUR PASSWORD HERE>
```

Important: Save your password as you will need it to relaunch the node, and please note:

- Minimum password length is 16 characters
- There is no password recovery process for NuFT nodes
- Do not use a password that you use anywhere else
- Security audits are ongoing on this codebase. For now, treat it as un-audited.

1.2.5 Running a NuFT Node

1.2.6 Stage C | Run the Node (Interactive Method)

1. Connect to Testnet

NuCypher is maintaining a purpose-built endpoint to initially connect to the test network. To connect to the swarm run:

```
$(nucypher) nucypher ursula run --network <NETWORK_DOMAIN> --teacher-uri <SEEDNODE_
↪URI>
...
```

2. Verify Connection

This will drop your terminal session into the "Ursula Interactive Console" indicated by the `>>>`. Verify that the node setup was successful by running the `status` command.

```
Ursula >>> status
...
```

To view a list of known nodes, execute the `known_nodes` command

```
Ursula >>> known_nodes
...
```

You can also view your node's network status webpage by navigating your web browser to `https://<your-node-ip-address>:9151/status`.

Note: Since nodes self-sign TLS certificates, you may receive a warning from your web browser.

To stop your node from the interactive console and return to the terminal session

```
Ursula >>> stop
...
```

Subsequent node restarts do not need the teacher endpoint specified.

```
(nucypher)$ nucypher ursula run --network <NETWORK_DOMAIN>
...
```

Alternately you can run your node as a system service. See the *"System Service Method"* section below.

1.2.7 Stage C | Run the Node (System Service Method)

NOTE - This is an alternative to the "Interactive Method" and assumes you're using systemd.

1. Create Ursula System Service

Use this template to create a file named `ursula.service` and place it in `/etc/systemd/system/`.

```
[Unit]
Description="Run 'Ursula', a NuCypher Staking Node."

[Service]
User=<YOUR USER>
Type=simple
Environment="NUCYPHER_KEYRING_PASSWORD=<YOUR PASSWORD>"
ExecStart=<VIRTUALENV PATH>/bin/nucypher ursula run --network <NETWORK_DOMAIN> --
↪teacher-uri <SEEDNODE_URI>

[Install]
WantedBy=multi-user.target
```

2. Enable Ursula System Service

```
$ sudo systemctl enable ursula
```

3. Run Ursula System Service

To start Ursula services using systemd

```
$ sudo systemctl start ursula
```

Check Ursula service status

```
$ sudo systemctl status ursula
```

To restart your node service

```
$ sudo systemctl restart ursula
```

1.2.8 Updating a NuFT Node

Nucypher is under active development, you can expect frequent code changes to occur as bugs are discovered and code fixes are submitted. As a result, Ursula nodes will need to be frequently updated to use the most up-to-date version of the application code.

Important: The steps to update an Ursula running on NuFT are as follows and depends on the type of installation that was employed.

1. Stop the node

Interactive method

```
Ursula >>> stop
```

OR

Systemd method

```
$ sudo systemctl stop ursula
```

2. Update to the latest code version

Update your virtual environment

```
(nucypher)$ pip3 install -U nucypher
```

3. Restart Ursula Node

Re-activate your environment after updating

Interactive method:

```
$ source nucypher/bin/activate
...
(nucypher)$ nucypher ursula run --network <NETWORK_DOMAIN>
```

OR

Systemd Method:

```
$ sudo systemctl start ursula
```

1.3 Installation Guide

1.3.1 Contents

1.3.2 System Requirements

- At least 1 GB of RAM is required for key derivation functionality (SCrypt).
- We have tested `nucypher` with Windows, Mac OS, and GNU/Linux (GNU/Linux is recommended).

1.3.3 System Dependencies

If you don't already have it, install Python. As of January 2019, we are working with Python 3.6, 3.7, and 3.8.

- Official Python Website: https://www.python.org/downloads/

We also require the following system packages (Linux):

```
- libffi-dev
- python3-dev
- python3-virtualenv
```

1.3.4 Standard Installation

We recommend installing nucypher with either `pip` or `pipenv`

- Pip Documentation
- Pipenv Documentation

Standard Pip Installation

In order to isolate global system dependencies from nucypher-specific dependencies, we *highly* recommend using `python-virtualenv` to install `nucypher` inside a dedicated virtual environment.

For full documentation on virtualenv see: https://virtualenv.pypa.io/en/latest/

Here is the recommended procedure for setting up `nucypher` in this fashion:

1. Create a Virtual Environment

```
$ virtualenv /your/path/nucypher-venv
...
```

Activate the newly created virtual environment:

```
$ source /your/path/nucypher-venv
...
$(nucypher-venv)
```

Note: Successful virtualenv activation is indicated by '(nucypher-venv)$' prepending your console's prompt

2. Install Application Code with Pip

```
$(nucypher-venv) pip install -U nucypher
```

3. Verify Installation

   In the console:

```
nucypher --help
```

   In Python:

```
import nucypher
```

Standard Pipenv Installation

1. Install Application code with Pipenv

   Ensure you have `pipenv` installed (See full documentation for pipenv here: Pipenv Documentation). Then to install nucypher with pipenv, run:

```
$ pipenv install nucypher
```

2. Verify Installation

   In the console:

```
nucypher --help
```

   In Python:

```
import nucypher
```

1.3.5 Development Installation

Additional dependencies and setup steps are required to perform a "developer installation". Ensure you have `git` installed (Git Documentation).

Acquire NuCypher Codebase

Fork the nucypher repository on GitHub, as explained in the *Contribution Guide*, then clone your fork's repository to your local machine:

```
$ git clone https://github.com/<YOUR_GITHUB_USERNAME>/nucypher.git
```

After acquiring a local copy of the application code, you will need to install the project dependencies, we recommend using either `pip` or `pipenv`

Pipenv Development Installation

The most common development installation method is using pipenv:

```
$ pipenv install --dev --three --skip-lock --pre
```

Activate the pipenv shell

```
$ pipenv shell
```

If this is successful, your terminal command prompt will be prepended with `(nucypher)`

Install the Solidity compiler (solc):

```
$(nucypher) pipenv run install-solc
```

Pip Development Installation

Alternately, you can install the development dependencies with pip:

```
$ pip install -e .[development]
$ ./scripts/install_solc.sh
```

1.3.6 Systemd Service Installation

1. Use this template to create a file named *ursula.service* and place it in */etc/systemd/system/*.

   ```
   [Unit]
   Description="Run 'Ursula', a NuCypher Staking Node."

   [Service]
   User=<YOUR USER>
   Type=simple
   Environment="NUCYPHER_KEYRING_PASSWORD=<YOUR PASSWORD>"
   ExecStart=<VIRTUALENV PATH>/bin/nucypher ursula run --teacher-uri <SEEDNODE_URI>

   [Install]
   WantedBy=multi-user.target
   ```

   Replace the following values with your own:

   - `<YOUR_USER>` - The host system's username to run the process with
   - `<YOUR_PASSWORD>` - Ursula's keyring password
   - `<VIRTUALENV_PATH>` - The absolute path to the python virtual environment containing the `nucypher` executable
   - `<SEEDNODE_URI>` - A seednode URI of a node on the network you are connecting to

2. Enable Ursula System Service

```
$ sudo systemctl enable ursula
```

3. Run Ursula System Service

   To start Ursula services using systemd

```
$ sudo systemctl start ursula
```

4. Check Ursula service status

```
$ sudo systemctl status ursula
```

5. To restart your node service

```
$ sudo systemctl restart ursula
```

1.4 Ursula Configuration Guide

Before continuing, Verify your `nucypher` installation and entry points are functional:

Activate your virtual environment and run the `nucypher --help` command

```
$ source nucypher/bin/activate
...
(nucypher)$ nucypher --help
```

You will see a list of possible usage options (`--version`, `-v`, `--dev`, etc.) and commands (`status`, `ursula`). For example, you can use `nucypher ursula destroy` to delete all files associated with the node.

If your installation in non-functional, be sure you have the latest version installed, and see the Installation Guide

2(a). Configure a new Ursula node

Decentralized Ursula Configuration

```
(nucypher)$ nucypher ursula init --provider-uri <YOUR PROVIDER URI> --network
↪<NETWORK NAME>
```

Replace `<YOUR PROVIDER URI>` with a valid node web3 node provider string, for example:

- `ipc:///tmp/geth.ipc` - Geth Development Node (IPC; **Note**: Also use `--poa` flag)
- `http://localhost:7545` - Ganache TestRPC (HTTP-JSON-RPC)
- `ws://0.0.0.0:8080` - Websocket Provider

2(b). Configure a new Ursula node

Federated Ursula Initialization

```
(nucypher)$ nucypher ursula init --federated-only --network <NETWORK NAME>
```

3. Enter your public-facing IPv4 address when prompted

```
Enter Nodes Public IPv4 Address: <YOUR NODE IP HERE>
```

4. Enter a password when prompted

```
Enter a password to encrypt your keyring: <YOUR PASSWORD HERE>
```

5. Connect to a fleet

```
(nucypher)$ nucypher ursula run --teacher-uri <SEEDNODE_URI>
```

6. Verify Node Connection

This will drop your terminal session into the "Ursula Interactive Console" indicated by the `>>>`. Verify that the node setup was successful by running the `status` command.

```
Ursula >>> status
```

7. To view a list of known nodes, execute the `known_nodes` command

```
Ursula >>> known_nodes
```

You can also view your node's network status webpage by navigating your web browser to `https://<your-node-ip-address>:9151/status`.

Note: Since nodes self-sign TLS certificates, you may receive a warning from your web browser.

8. To stop your node from the interactive console and return to the terminal session:

```
Ursula >>> stop
```

9. Subsequent node restarts do not need the teacher endpoint specified:

```
(nucypher)$ nucypher ursula run
```

1.5 Contributing

1.5.1 Acquiring the Codebase

In order to contribute new code or documentation changes, you will need a local copy of the source code which is located on the NuCypher GitHub.

Note: NuCypher uses `git` for version control. Be sure you have it installed.

Here is the recommended procedure for acquiring the code in preparation for contributing proposed changes:

1. Use GitHub to Fork the *nucypher/nucypher* repository
2. Clone your fork's repository to your local machine

```
$ git clone https://github.com/<YOUR-GITHUB-USERNAME>/nucypher.git
```

3. Change Directories into `nucypher`

```
cd nucypher
```

3. Add *nucypher/nucypher* as an upstream remote

```
$ git remote add upstream https://github.com/nucypher/nucypher.git
```

4. Update your remote tracking branches

```
$ git remote update
```

5. Install the project dependencies: see the Developer Installation Guide

1.5.2 Running the Tests

Note: A development installation including the solidity compiler is required to run the tests

There are several test implementations in `nucypher`, however, the vast majority of test are written for execution with `pytest`. For more details see the Pytest Documentation.

To run the tests:

```
(nucypher)$ pytest -s
```

Optionally, to run the full, slow, verbose test suite run:

```
(nucypher)$ pytest --runslow -s
```

1.5.3 Making A Commit

NuCypher takes pride in its commit history.

When making a commit that you intend to contribute, keep your commit descriptive and succinct. Commit messages are best written in full sentences that make an attempt to accurately describe what effect the changeset represents in the simplest form. (It takes practice!)

Imagine you are the one reviewing the code, commit-by-commit as a means of understanding the thinking behind the PRs history. Does your commit history tell an honest and accurate story?

We understand that different code authors have different development preferences, and others are first-time contributors to open source, so feel free to join our Discord and let us know how we can best support the submission of your proposed changes.

1.5.4 Opening A Pull Request

When considering including commits as part of a pull request into *nucypher/nucypher*, we *highly* recommend opening the pull request early, before it is finished with the mark "[WIP]" prepended to the title. We understand PRs marked "WIP" to be subject to change, history rewrites, and CI failures. Generally we will not review a WIP PR until the "[WIP]" marker has been removed from the PR title, however, this does give other contributors an opportunity to provide early feedback and assists in facilitating an iterative contribution process.

1.5.5 Pull Request Conflicts

As an effort to preserve authorship and a cohesive commit history, we prefer if proposed contributions are rebased over master (or appropriate branch) when a merge conflict arises, instead of making a merge commit back into the contributors fork.

Generally speaking the preferred process of doing so is with an *interactive rebase*:

Important: Be certain you do not have uncommitted changes before continuing.

1. Update your remote tracking branches

```
$ git remote update
...  (some upstream changes are reported)
```

2. Initiate an interactive rebase over *nucypher/nucypher@master*

Note: This example specifies the remote name `upstream` for the NuCypher organizational repository as used in the *Acquiring the Codebase* section.

```
$ git rebase -i upstream/master
...  (edit & save rebase TODO list)
```

3. Resolve Conflicts

```
$ git status
... (resolve local conflict)
$ git add path/to/resolved/conflict/file.py
$ git rebase --continue
... ( repeat as needed )
```

4. Push Rebased History

After resolving all conflicts, you will need to force push to your fork's repository, since the commits are rewritten.

Warning: Force pushing will override any changes on the remote you push to, proceed with caution.

```
$ git push origin my-branch -f
```

1.5.6 Building Documentation

Note: `sphinx`, `recommonmark`, and `sphinx_rtd_theme` are non-standard dependencies that can be installed by running `pip install -e .[docs]` from the project directory.

Documentation for `nucypher` is hosted on Read The Docs, and is automatically built without intervention by following the release procedure. However, you may want to build the documentation html locally for development.

To build the project dependencies locally:

```
(nucypher)$ cd nucypher/docs/
(nucypher)$ make html
```

If the build is successful, the resulting html output can be found in `nucypher/docs/build/html`; Opening `nucypher/docs/build/html/index.html` in a web browser is a reasonable next step.

1.5.7 Building Docker

Docker builds are automated as part of the publication workflow on circleCI and pushed to docker cloud. However you may want to build a local version of docker for development.

We provide both a `docker-compose.yml` and a `Dockerfile` which can be used as follows:

Docker Compose:

```
(nucypher)$ docker-compose -f deploy/docker/docker-compose.yml build .
```

1.5.8 Issuing a New Release

Note: `bumpversion` is a non-standard dependency that can be installed by running `pip install -e .[deployment]` or `pip install bumpversion`.

Important: Ensure your local tree is based on `master` and has no uncommitted changes.

1. Increment the desired version part (options are `major`, `minor`, `patch`, `stage`, `devnum`), for example:

```
(nucypher)$ bumpversion devnum
```

3. Ensure you have the intended history and incremented version tag:

```
(nucypher)$ git log
```

4. Push the resulting tagged commit to the originating remote by tag and branch to ensure they remain synchronized.

```
(nucypher)$ git push origin master && git push origin <TAG>
```

5. Push the tag directly upstream by its name to trigger the publication webhooks on CircleCI:

```
(nucypher)$ git push upstream <TAG>
```

7. Monitor the triggered deployment build on CircleCI for manual approval.
8. Open a pull request with the resulting history in order to update `master`.

1.6 NuCypher Character Control Guide

The *Character Control* is a module that contains useful REST-like HTTP endpoints for working with NuCypher characters.

Important: Character control is currently a Work-In-Progress. Expect large, and even breaking, changes to this API often.

Warning: Character control is currently not intended for use over remote connections or on shared machines. The current API has not been secured and should not be used for production applications.

Table of Contents

1.6.1 API Request/Response Structure Overview

Status Codes

All documented API endpoints use JSON and are REST-like.

Some common returned status codes you may encounter are:

- `200 OK` – The request has succeeded.
- `400 BAD REQUEST` – The server cannot or will not process the request due to something that is perceived to be a client error (e.g., malformed request syntax, invalid request message framing, or deceptive request routing).
- `500 INTERNAL SERVER ERROR` – The server encountered an unexpected condition that prevented it from fulfilling the request.

Typically, you will want to ensure that any given response from a character control endpoint results in a 200 status code. This tells you that the server successfully completed the call.

If you are returned a 400, check the data that you're sending to the server. See below for what the character control API expects.

If you are ever given a 500 status code, we'd like to know about it! You can see our Contribution Guide for getting involved. Ideally, you can share some information with us about what you were doing when you encountered the 500 in the form of a GitHub issue, or just tell us in our Discord.

HTTP Methods

Currently, the character control API only uses the following HTTP methods:

- POST
- PUT

We don't exactly follow RESTful methodology precisely. Take careful note following the API endpoints to see what to send and what to expect as a response.

Request Format

The character control API uses JSON for all its endpoints. A request may look like:

```
{
    'bob_encrypting_key':
↪'0324df67664e6ea40f2eea8037c994debd4caa42117fe86cdb8cab6ac7728751ad',
    'label': 'dGVzdA==',
    'm': 2,
    'n': 3,
    'expiration_time': '2019-02-14T22:23:10.771093Z',
}
```

Take a look at `bob_encrypting_key`. Take note that it's a hex-encoded string. The character control API endpoints expect *all* keys to be encoded as hex.

Now, look at `label`. Notice that it's a base64-encoded string. Whenever the Python API expects the `bytes` type, the character control API will expect a base64 encoded string.

Integers, in our case `m` and `n` can be passed as is without encoding.

A datetime, like `expiration_time`, must be passed in as an ISO-8601 formatted datetime string.

If you are missing a required argument in your request, you will be returned a 400 status code.

Response Format

Like we determined above, the character control API uses JSON for all its endpoints. The same goes for our API's responses. One may look like:

```
{
    'result': {
        'treasure_map': 'Y8Wl+o...Jr4='
    }
}
```

The character control API will return the results of our Python API. If any data is returned, like a treasure map or a message kit, it will be serialized as base64 with the object name being a key inside `result`.

Be sure to also check the returned status code of the request. All successful calls will be 200. See the above "Status Codes" section on what to do in the event of a 400 or 500.

1.6.2 Character Control Endpoints

Alice

derive_policy_pubkey

This endpoint controls the `Alice.get_policy_pubkey_from_label` method.

- URL: `/derive_policy_pubkey`
- HTTP Method: `PUT`
- **Required arguments:**
  - `label` – encoded as base64
- Returns: a hex-encoded `policy_encrypting_pubkey`

grant

This endpoint controls the `Alice.grant` method.

- URL: `/grant`
- HTTP Method: `PUT`
- **Required arguments:**
  - `bob_encrypting_key` – encoded as hex
  - `label` – encoded as base64
  - `m` – an integer
  - `n` – an integer
  - `expiration_time` – an ISO-8601 formatted datetime string
- **Returns:**
  - `treasure_map` – encoded as base64
  - `policy_encrypting_pubkey` – encoded as hex
  - `alice_signing_pubkey` – encoded as hex

For more details on these arguments, see the nucypher documentation on the `Alice.grant` Python API method.

Bob

retrieve

This endpoint controls the `Bob.retrieve` method.

- URL: `/retrieve`
- HTTP Method: `POST`
- **Required arguments:**
  - `policy_encrypting_pubkey` – encoded as hex
  - `alice_signing_pubkey` – encoded as hex
  - `datasource_signing_pubkey` – encoded as hex
  - `label` – encoded as base64
  - `message_kit` – encoded as base64

- Returns: a JSON-array of base64-encoded decrypted plaintexts as `plaintext`

For more details on these arguments, see the nucypher documentation on the `Bob.retrieve` Python API method.

Enrico (DataSource)

encrypt_message

This endpoint controls the `Enrico.encrypt_message` method.

- URL: `/encrypt_message`
- HTTP Method: `POST`
- **Required arguments:**
  - `message` – encoded as base64
- Returns: `message_kit` and `signature` encoded as base64

For more details on these arguments, see the nucypher documentation on the `DataSource.encrypt_message` Python API method.

1.7 NuCypher Staking Guide

Ursula Staking Actions

Action	Description
`stake`	Initialize or view NuCypher stakes (used with `--value` and `--duration`)
`collect-reward`	Collect staking reward (Policy reward collection in future PR)

Ursula Staking Options

Option	Description
`--value`	Stake value
`--duration`	Stake duration of extension
`--index`	Stake index
`--list`	List stakes (used with `stake` action)
`--divide`	Divide stakes (used with `stake` action)

1.7.1 Interactive Method

```
(nucypher)$ nucypher ursula stake
```

Initialize a new stake

```
Stage a new stake? [y/N]: y

Current balance: 100000
Enter stake value in NU [15000]: 30000

Minimum duration: 30 | Maximum Duration: 365
```

(continues on next page)

(continued from previous page)

```
Enter stake duration in periods (1 Period = 24 Hours): 90

============================== STAGED STAKE ==============================

(Ursula)Sienna Scales GreenYellow Ferry (0x058D5F4cC9d52403c2F6944eC1c821a0e6E78657)
~ Value      -> 30000 NU (30000000000000000000000 NU-wei)
~ Duration   -> 90 Days (90 Periods)
~ Enactment  -> 2019-03-12 02:08:41.425755+00:00 (17967)
~ Expiration -> 2020-02-08 02:08:41.425912+00:00 (18300)

=========================================================================

* Ursula Node Operator Notice *
-------------------------------
...

Accept node operator obligation? [y/N]: y
Publish staged stake to the blockchain? [y/N]: y

Escrow Address ... 0x709166C66Ab0BC36126607BE823F11F64C2A9996
Approve .......... 0x7eef13ee7451adaa33d814ecd4953a46de0a10267f7b52dcb487031c900a8d80
Deposit .......... 0x75127e862e044309c9980fdb0e4d3120de7723dd8b70f62d5504e6851f672f4c

Successfully transmitted stake initialization transactions.
View your active stakes by running 'nucypher ursula stake --list'
or start your Ursula node by running 'nucypher ursula run'.
```

View existing stakes

```
(nucypher)$ nucypher ursula stake --list

| # | Duration     | Enact     | Expiration    | Value
| - | ------------ | --------- | -------------| -----
| 0 | 32 periods . | yesterday | 14 Apr ..... | 30000 NU
```

Divide an existing stake

```
(nucypher)$ nucypher ursula stake --divide


| # | Duration     | Enact     | Expiration | Value
| - | ------------ | --------- | -----------| -----
| 0 | 32 periods . | yesterday | 14 Apr ... | 30000 NU

Select a stake to divide: 0
Enter target value (must be less than 30000 NU): 15000
Enter number of periods to extend: 30

============================== ORIGINAL STAKE ============================

~ Original Stake: | 0 | 90 periods . | yesterday .. | 14 Apr ... | 30000 NU

============================== STAGED STAKE ==============================

(Ursula)Sienna Scales GreenYellow Ferry (0x058D5F4cC9d52403c2F6944eC1c821a0e6E78657)
~ Value      -> 15000 NU (15000000000000000000000 NU-wei)
~ Duration   -> 120 Days (120 Periods)
```

(continues on next page)

(continued from previous page)

```
~ Enactment  -> 2019-03-13 20:18:17.306398+00:00 (period #17968)
~ Expiration -> 2019-04-24 20:18:17.306801+00:00 (period #18010)

==========================================================================
```

1.7.2 Inline Method

Option	Flag	Description
stake value	–value	in NU
stake duration	–duration	in periods
stake index	–index	to divide

Stake 30000 NU for 90 Periods

```
(nucypher)$ nucypher ursula stake --value 30000 --duration 90
...
```

Divide stake at index 0, at 15000 NU for 30 additional Periods

```
(nucypher)$ nucypher ursula stake --divide --index 0 --value 15000 --duration 30
...
```

1.8 Deployment Guide

1.8.1 Geth Development Deployment

The fastest way to start a local private chain using an ethereum client is to deploy a single-host network with the geth CLI.

```
$ geth --dev
  ...
```

In another terminal

```
(nucypher)$ nucypher-deploy contracts --provider-uri ipc:///tmp/geth.ipc --poa
...
```

This will deploy the main NuCypher contracts, namely `NuCypherToken`, `MinerEscrow`, and `PolicyManager`, along with their proxies, as well as executing initialization transactions. You will need to enter the contract's upgrade secrets, which can be any alphanumeric string.

A summary of deployed contract addresses, transactions, and gas usage will be displayed on success, and a `contract_registry.json` file will be generated in your nucypher application directory.

```
Deployer Address is 0xB62CD782B3c73b1213fbba6ee72e6eaeEC2B327d - Continue? [y/N]:

Enter MinerEscrow Deployment Secret:
Repeat for confirmation:
Enter PolicyManager Deployment Secret:
Repeat for confirmation:
```

(continues on next page)

(continued from previous page)

```
Enter UserEscrowProxy Deployment Secret:
Repeat for confirmation:

Deployed!

Deployment Transaction Hashes for /home/kieran/.local/share/nucypher/contract_
↪registry.json

NuCypherToken (0x2d610671e756dbE547a9b9Dc9A46d6A90Ac9C08c)
**********************************************************
OK | txhash | 0xd1f4a35a9cf46456c38b4ab5f978ab4d70ce1979016e4c60d155f7112ca4538c␣
↪(793804 gas)
Block #2 | 0x610cf940169d8e92b6f363daef36719643a79cdfa63b961ef8147513aef8855f


MinersEscrow (0xca6b01013336065456f0ac0Dd98Ae6F3F786C0d2)
*********************************************************
OK | deploy | 0x9b1842bd08c680f8cfd3734722fea9118a582a55026d23fd8ad10af08996c27d␣
↪(5266629 gas)
Block #3 | 0x5d8c9daf95ad42ada2ab7a6645b59b7d778fc15fce2b19f73bcab78e8d41ffac

OK | dispatcher_deploy |␣
↪0x9e7d6420e58f9487923af6a92f34f379ea0b2333ba372324cb9ec73ba02b6193 (1184911 gas)
Block #4 | 0x337c2f77b4f5b18d4ecfdae171dbd9c53aed74d7ddf85f0d82f569e8a9544182

OK | reward_transfer |␣
↪0xc1a56af5e8f2961f6808a3d3de25ebc71f3aec10097ad6ac063c2416d6cf2a85 (51860 gas)
Block #5 | 0x1617cc77c2e9b159903c3d7a43f0f98a652d961168d48c23106a15860309c49f

OK | initialize | 0x78064a72b5e79caf957dbb24ab9277ffb269526d2ce4cb0fdb7390048b69e643␣
↪(95553 gas)
Block #6 | 0xc74846b426a44f2148d62207960da7e746c0de31be323368cd6afcc3a308a244


PolicyManager (0x124Bb5a44D2AcCB811Af8aab889F65DfCb2f9858)
**********************************************************
OK | deployment | 0x6afe2b645d6d9158ad79a0bc10c52e462eb33b18a3e9bbb1a99484888c3ecffb␣
↪(2756954 gas)
Block #7 | 0x9590d75afdfaa5ed30d77621bf265fdaee8e366ae9fce2c1a25d6903f27a3ed3

OK | dispatcher_deployment |␣
↪0x06d1399bd49e2f1afa72342866100b2281df110f36420b663df10b8e956c76de (1277565 gas)
Block #8 | 0xe6714a7cdb74e3025a64c3f5d077994664ee3c4e1ed6c71cca336ce2781edbd2

OK | set_policy_manager |␣
↪0x9981edb9d7d54db8c6735ee802c19a153b071e798e58bae38dcf1346ea28fa1e (50253 gas)
Block #9 | 0xe70a8252a4899fb4dd1f7a4c0f7d12933f7c321ae6c784b777c668a5d8494563

Cumulative Gas Consumption: 4084772 gas
```

1.9 Local Development Fleet Testing

1.9.1 Overview

Note: Currently only "Federated Only" mode is supported for local fleets

All Demo Ursulas:

- Run on `localhost`
- In `--federated-only` mode
- On the `TEMPORARY_DOMIAN` (implied by `--dev`)
- Using temporary resources (files, database, etc.)

1.9.2 Running A Local Fleet

1. Install Nucypher

Acquire the Nucypher application code and install the dependencies. For a full installation guide see the *Installation Guide*.

2. Run a Lonely Ursula

The first step is to launch the first Ursula on the network by running:

```
$ python run_lonely_demo_ursula.py
```

This will start an Ursula node:

- With seednode discovery disabled
- On port 11500

3. Run a Local Fleet of Ursulas

Next, launch subsequent Ursulas, informing them of the first Ursula:

```
$ python run_demo_ursula_fleet.py
```

This will run 5 temporary Ursulas that:

- All specify the lonely Ursula as a seednode
- Run on ports 11501 through 11506

4. Run an Entry-Point Ursula (Optional)

While the local fleet is running, you may want an entry-point to introspect the code in a debugger. For this we provide the optional script run_single_demo_ursula.py for your convenience.

```
$ python run_single_demo_ursula.py
```

This will run a single temporary Ursula:

- That specifies a random fleet node as a teacher
- On a random available port

1.9.3 Connecting to the Local Fleet

Alternately, you can connect any node run from the CLI by specifying one of the nodes in the local fleet as a teacher, the same network domain, and the same operating mode. By default, nodes started with the --dev flag run on a dedicated domain (TEMPORARY_DOMAIN) and on a different port than the production default port (9151). Local fleet Ursulas range from ports 11500 to 11506 by default.

Here is an example of connecting to a node in the local development fleet:

```
nucypher ursula run --dev --teacher-uri localhost:11501
```

Note: The local development fleet is an *example* meant to demonstrate how to design and use your own local fleet.

1.10 Finnegan's Wake Demo

1.10.1 Overview

Important: This demo requires connecting to a running network. By default the demo is hardcoded to connect to the *local demo fleet*.

This demo is an example of a NuCypher decentralized network allowing Alice to share data with Bob using proxy re-encryption. This enables the private sharing of data across public consensus networks, without revealing data keys to intermediary entities.

Step	Character	Operation
1	Alice	Alice sets a Policy on the NuCypher network (2-of-3) and grants access to Bob
2	Alice	Label and Alice's key public key provided to Bob
3	Bob	Bob joins the policy with Label and Alice's public key
4	Enrico	A data source created for the policy
5	Enrico	Each plaintext message gets encrypted by Enrico, which results in a messageKit
6	Bob	Bob receives and reconstructs Enrico from the Policy's public key and Enrico's public key
7	Bob	Bob retrieves the original message from Enrico and MessageKit

1.10.2 Install Nucypher

Acquire the `nucypher` application code and install the dependencies. For a full installation guide see the *Installation Guide*.

1.10.3 Download the Book Text

For your convenience we have provided a bash script to acquire the "Finnegan's Wake" text. However, feel free to use any text of your choice, as long you you edit the demo code accordingly.

To run the script:

```
(nucypher)$ ./download_finnegans_wake.sh
```

1.10.4 Run the Demo

After acquiring a text file to re-encrypt, execute the demo by running:

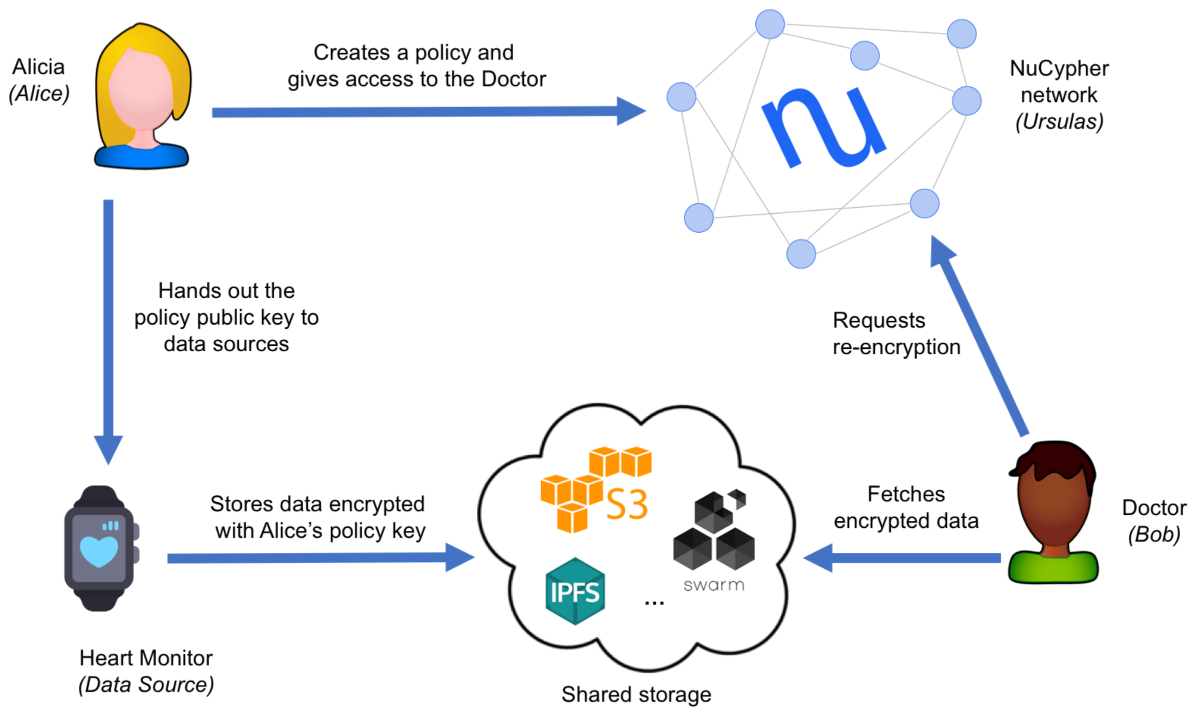
```
(nucypher)$ python3 finnegans-wake-demo.py
```

1.11 Heartbeat Demo

1.11.1 Overview

Important: This demo requires connecting to a running network. By default the demo is hardcoded to connect to the *local demo fleet*.

Alicia has a Heart Monitor device that measures her heart rate and outputs this data in encrypted form. Since Alicia knows that she may want to share this data in the future, she uses NuCypher to create a *policy public key* for her Heart Monitor to use, so she can read and delegate access to the encrypted data as she sees fit.



The Heart Monitor uses this public key to produce a file with some amount of encrypted heart rate measurements. This file is uploaded to a storage layer (e.g., IPFS, S3, or whatever you choose).

At some future point, she wants to share this information with other people, such as her Doctor. Once she obtains her Doctor's public keys, she can create a policy in the NuCypher network granting access to him. After this, her Doctor can read the file with encrypted data (which was uploaded by the Heart Monitor) and request a re-encrypted ciphertext for each measurement, which can be opened with the Doctor's private key.

This simple example showcases many interesting and distinctive aspects of NuCypher:

- Alicia can create policy public keys **before knowing** the potential consumers.
- Alicia, or anyone knowing the policy public key (e.g., the Heart Monitor), can produce encrypted data that belongs to the policy. Again, this can happen **before granting access** to any consumer.
- As a consequence of the previous point, the Heart Monitor is completely unaware of the recipients. In its mind, it's producing data **for Alicia**.
- Alicia never interacts with the Doctor: she only needs the Doctor's public keys.
- Alicia only interacts with the NuCypher network for granting access to the Doctor. After this, she can even disappear from the face of the Earth.
- The Doctor never interacts with Alicia or the Heart Monitor: he only needs the encrypted data and some policy metadata.

1.11.2 The NuCypher Characters

The actors in this example can be mapped naturally to *Characters* in the NuCypher narrative:

- Since Alicia is the only one capable of granting access, she retains full control over the data encrypted for her. As such, she can be considered as the **data owner** or the **policy authority**. This corresponds to the `Alice` character.

- The Heart Monitor, or any other data sources that **encrypt data** on Alicia's behalf, is portrayed by the `Enrico` character.
- Nodes in the NuCypher network are called `Ursula` in our terminology. They receive the access policy from Alice and stand ready to re-encrypt data in exchange for payment in fees and token rewards. In a way, they **enforce the access policy** created by Alicia.
- The Doctor acts as a **data recipient**, and only can decrypt Alicia's data if she grants access to him. This is modelled by the `Bob` character.

1.11.3 Install Nucypher

Acquire the `nucypher` application code and install the dependencies. For a full installation guide see the *Installation Guide*.

1.11.4 Run the Demo

Assuming you already have `nucypher` installed with the `demos` extra and a *local fleet of Ursulas* alive, running the Heartbeat demo only involves executing the `alicia.py` and `doctor.py` scripts, contained in the `examples/heartbeat_demo` directory.

First, run `alicia.py`:

```
(nucypher)$ python alicia.py
```

This will create a temporal directory called `alicia-files` that contains the data for making Alicia persistent (i.e., her private keys). Apart from that, it will also generate data and keys for the demo. What's left is running the `doctor.py` script:

```
(nucypher)$ python doctor.py
```

This script will read the data generated in the previous step and retrieve re-encrypted ciphertexts by means of the NuCypher network. The result is printed in the console:

```
Creating the Doctor ...
Doctor =  Maroon Snowman DarkSlateGray Bishop␣
↪(0xA36bcd5c5Cfa0C1119ea5E53621720a0C1a610F5)
The Doctor joins policy for label 'heart-data--e917d959'
---------------------- (82 BPM)                    Retrieval time:  3537.06 ms
--------------------- (81 BPM)                     Retrieval time:  2654.51 ms
------------------------- (85 BPM)                 Retrieval time:  1513.32 ms
---------------------------- (88 BPM)              Retrieval time:  1552.66 ms
----------------------- (83 BPM)                   Retrieval time:  1720.66 ms
--------------------- (81 BPM)                     Retrieval time:  1485.25 ms
--------------------- (81 BPM)                     Retrieval time:  1459.16 ms
--------------------- (81 BPM)                     Retrieval time:  1520.30 ms
---------------- (76 BPM)                          Retrieval time:  1479.54 ms
---------------- (76 BPM)                          Retrieval time:  1464.17 ms
--------------------- (81 BPM)                     Retrieval time:  1483.04 ms
---------------- (76 BPM)                          Retrieval time:  1687.72 ms
--------------- (75 BPM)                           Retrieval time:  1563.65 ms
```

1.12 Nucypher Ethereum Contracts

1.12.1 Contract Listing

- `NuCypherToken` ERC20 token contract
- `MinersEscrow` Holds Ursula's stake, stores information about Ursula's activity, and assigns a reward for participating in the NuCypher network. (The `Issuer` contract is part of the `MinersEscrow`)
- `PolicyManager` Holds a policy's fee and distributes fee by periods
- `Upgradeable` Base contract for upgrading
- `Dispatcher` Proxy to other contracts. This provides upgrading of the `MinersEscrow` and `PolicyManager` contracts
- `UserEscrow` Locks tokens for predetermined time. Tokens will be unlocked after specified time and all tokens can be used as stake in the `MinersEscrow` contract

1.12.2 Deployment Procedure

1. Deploy `NuCypherToken` with all future supply tokens
2. Deploy `MinersEscrow` with a dispatcher targeting it
3. Deploy `PolicyManager` with its own dispatcher, also targeting it
4. Transfer reward tokens to the `MinersEscrow` contract. These tokens are future mining rewards, and initial allocations
5. Run the `initialize()` method to initialize the `MinersEscrow` contract
6. Set the address of the `PolicyManager` contract in the `MinersEscrow` by using the `setPolicyManager(address)`
7. Pre-deposit tokens to the `MinersEscrow` if necessary:
    - Approve the transfer tokens for the `MinersEscrow` contract using the `approve(address, uint)` method. The parameters are the address of `MinersEscrow` and the amount of tokens for a miner or group of miners;
    - Deposit tokens to the `MinersEscrow` contract using the `preDeposit(address[], uint[], uint[])` method. The parameters are the addresses of the miners, the amount of tokens for each miner and the periods during which tokens will be locked for each miner
8. Deploy `UserEscrowProxy` with `UserEscrowLibraryLinker` targeting it
9. Pre-deposit tokens to the `UserEscrow`, and if necessary:

- Create new instance of the `UserEscrow` contract
- Transfer ownership of the instance of the `UserEscrow` contract to the user
- Approve the transfer of tokens for the `UserEscrow`
- Deposit tokens by the `initialDeposit(uint256, uint256)` method

1.12.3 Alice's Contract Interaction

Alice Authors a Blockchain Policy

Alice uses a network of Ursula miners to deploy policies. In order to take advantage of the network, Alice chooses miners and deploys policies with fees for those miners. Alice can choose miners by herself ("handpicked") or by using `MinersEscrow.sample(uint256[], uint16)` - This is known as ("sampling"). `sample` parameters are:

- The array of absolute values
- Minimum number of periods during which tokens are locked This method will return only active miners.

In order to place the fee for a policy, Alice calls the method `PolicyManager.createPolicy(bytes16, uint16, uint256, address[])`, specifying the miner's addresses, the policy ID (off-chain generation), the policy duration in periods, and the first period's reward. Payment should be added to the transaction in ETH and the amount is `firstReward * miners.length + rewardRate * periods * miners.length`. The reward rate must be greater than or equal to the minimum reward for each miner in the list. The first period's reward is not refundable, and can be zero.

Alice Revokes a Blockchain Policy

When Alice wants to revoke a policy, she calls the `PolicyManager.revokePolicy(bytes16)` or `PolicyManager.revokeArrangement(bytes16, address)`. Execution of these methods results in Alice recovering all fees for future periods, and also for periods when the miners were inactive. Alice can refund ETH for any inactive periods without revoking the policy by using the method `PolicyManager.refund(bytes16)` or `PolicyManager.refund(bytes16, address)`.

1.12.4 Ursula's Contract Interaction

Ursula Locks Tokens

In order to become a participant of the network, a miner stakes tokens in the `MinersEscrow` contract. The miner allows the (mining) contract to perform a transaction using the `NuCypherToken.approve(address, uint256)` method (ERC20 contracts allow access delegation to another address).

After that, the miner transfers some quantity of tokens (`MinersEscrow.deposit(uint256, uint16)`), locking them at the same time. Alternately the `NucypherToken.approveAndCall(address, uint256, bytes)` method can be used. The parameters are:

- The address of the `MinersEscrow` contract
- The amount of staked tokens
- The periods for locking (which are serialized into an array of bytes)

When staking tokens, the miner sets the number of periods the tokens will be locked, which must be no less than some minimal locking time (30 periods). In order to unlock tokens, the miner must be active during the time of locking (and confirm activity each period). Each stake is represented by the amount of tokens locked, and the stake's duration in periods. The miner can add a new stake using `MinersEscrow.deposit(uint256, uint16)` or `MinersEscrow.lock(uint256, uint16)` methods. The miner can split stake into two parts: one with the same duration and another with an extended duration. For this purpose, the `MinersEscrow.divideStake(uint256, uint256, uint16)` method is used. The first parameter is used to identify the stake to divide and the last two for the extended part of the stake. When calculating locked tokens using the `MinersEscrow.getLockedTokens(address, uint16)` method, all stakes that are active during the specified period are summed.

Ursula Confirms Activity

In order to confirm activity every period, miners call `MinersEscrow.confirmActivity()` wherein activities for the next period are registered. The method `MinersEscrow.confirmActivity` is called every time the methods `MinersEscrow.deposit(uint256, uint16)` or `MinersEscrow.lock(uint256, uint16)` is called. The miner gets a reward for every confirmed period.

Ursula Generates Staking Rewards

After the period of activity has passed, the miner may call `MinersEscrow.mint()` method which computes and transfers tokens to the miner's account. Also note that calls to `MinersEscrow.lock(uint256, uint16)` and `MinersEscrow.confirmActivity()` are included the `MinersEscrow.mint()` method.

The reward value depends on the fraction of locked tokens for the period (only those who confirmed activity are accounted for). Also, the reward depends on the number of periods during which the tokens will be locked: if the tokens will be locked for half a year, the coefficient is 1.5. The minimum coefficient is 1 (when tokens will get unlocked in the next period), and the maximum is 2 (when the time is 1 year or more). The reward is calculated separately for each stake that is active during the mining period and all rewards are summed up. The order of calling `mint` by miners (e.g. who calls first, second etc) doesn't matter. Miners can claim their rewards by using the `withdraw(uint256)` method. Only non-locked tokens can be withdrawn.

Ursula Generates Policy Rewards

Also the miner gets rewards for policies deployed. Computation of a policy reward happens every time `mint()` is called by the `updateReward(address, uint16)` method. In order to take the reward, the miner needs to call method `withdraw()` of the contract `PolicyManager`. The miner can set a minimum reward rate for a policy. For that, the miner should call the `setMinRewardRate(uint256)` method.

NuCypher Partner Ursula Staking

Some users will have locked but not staked tokens. In that case, an instance of the `UserEscrow` contract will hold their tokens (method `initialDeposit(uint256, uint256)`). All tokens will be unlocked after a specified time and the user can retrieve them using the `withdraw(uint256)` method. When the user wants to become a miner - he uses the `UserEscrow` contract as a proxy for the `MinersEscrow` and `PolicyManager` contracts.

1.13 Nucypher's Approaches to Upgradeable Contracts

Smart contracts in Ethereum are immutable… Even if a contract can be deleted, it still exists in the blockchain after `selfdestruct`, and only the storage is cleared. In order to fix bugs and provide upgrade logic it is possible to change the contract (address) and save the original contract's storage values.

1.13.1 Approach A

One simple way to achieve this is to create a new contract, copy the original storage values to the new contract, then self-destruct (mark as deleted) the old contract. When this happens, the client changes the address used for a requested contract.

Note: There will be two deployed versions of the contract during storage migration

1.13.2 Approach B

A more convenient way is to use a proxy contract with an interface where each method redirects to the *target* contract. This option is advantageous because the client uses one address most of the time but also has its own methods.

Important: If updates to the proxy contract's methods are made, then the client will need to change proxy address also.

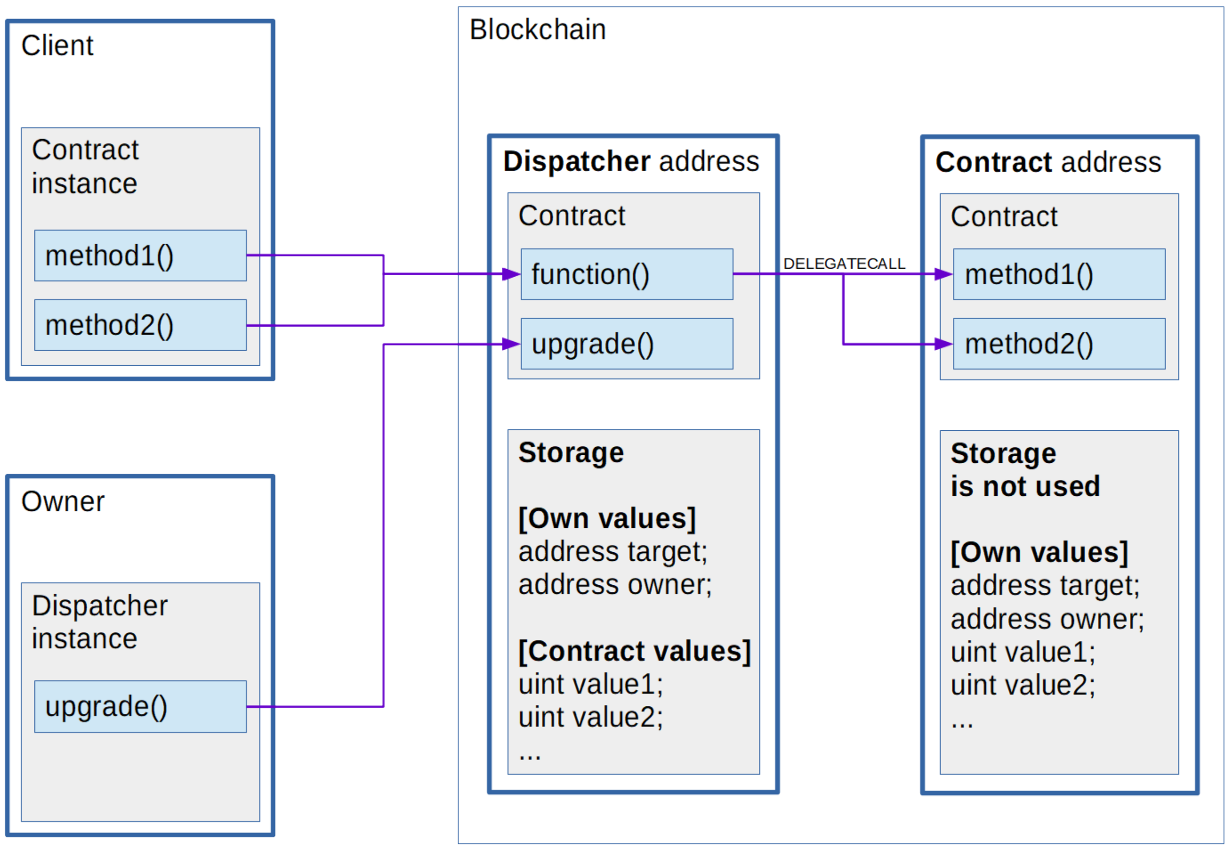
1.13.3 Approach C

Another way is using a fallback function in the proxy contract - this function will execute on any request, redirecting the request to the target and returning the resulting value (using opcodes). This is similar to the previous option, but this proxy doesn't have interface methods, only a fallback function, so there is no need to change the proxy address if contract methods are changed.

This approach is not ideal, and has some restrictions:

- Sending Ether from a client's account to the contract uses the fallback function and such transactions can only consume 2300 gas (http://solidity.readthedocs.io/en/develop/contracts.html#fallback-function).
- Proxy contracts (Dispatcher) hold storage (not in the contract itself). While upgrading storage, values must be the same or equivalent (see below).

1.13.4 Interaction scheme



`Dispatcher` - a proxy contract that redirects requests to the target address. It also holds its own values (owner and target address) and stores the values of the target contract, but not explicitly. The client should use the resulting contract or interface ABI while sending request to the `Dispatcher` address. The contract's owner can change the target address by using the `Dispatcher`'s ABI. The `Dispatcher` contract uses `delegatecall` for redirecting requests, so `msg.sender` remains as the client address and uses the dispatcher's storage when executing methods in the target contract.

Warning: If target address is not set, or the target contract does not exist, results may be unpredictable because `delegatecall` will return `true`.

`Contract` - upgradeable contract, each version must have the same ordering of storage values. New versions of the contract can expand values, but must contain all the old values (containing values from dispatcher **first**). This contract is like a library because its storage is not used. If a client sends a request to the contract directly to its deployed address without using the dispatcher, then the request may execute (without exception) using the wrong target address.

1.13.5 Development

- Use `Upgradeable` as base contract for all contracts that will be used with `Dispatcher`.
- Implement `verifyState(address)` method which checks that a new version has correct storage values.

- Implement `finishUpgrade(address)` method which copies initialization data from library storage to the dispatcher's storage.
- Each upgrade should include tests which check storage equivalence.

1.13.6 Sources

More examples:

- https://github.com/maraoz/solidity-proxy - Realization of using libraries (not contracts) but too complex and some ideas are obsolete after Byzantium hard fork.
- https://github.com/willjgriff/solidity-playground - Most of the upgradeable proxy contract code is taken from this repository.
- https://github.com/0v1se/contracts-upgradeable - Source code for verifying upgrade.

1.14 Characters

class `nucypher.characters.base.`**`Character`**(*domains: Set = (b'GLOBAL_DOMAIN',), is_me: bool = True, federated_only: bool = False, blockchain: nucypher.blockchain.eth.chains.Blockchain = None, checksum_public_address: str = NO_BLOCKCHAIN_CONNECTION, network_middleware: nucypher.network.middleware.RestMiddleware = None, keyring_dir: str = None, crypto_power: nucypher.crypto.powers.CryptoPower = None, crypto_power_ups: List[nucypher.crypto.powers.CryptoPowerUp] = None, *args, **kwargs*)

A base-class for any character in our cryptography protocol narrative.

exception InvalidSignature

Raised when a Signature is not valid.

exception SuspiciousActivity

raised when an action appears to amount to malicious conduct.

encrypt_for(*recipient: nucypher.characters.base.Character, plaintext: bytes, sign: bool = True, sign_plaintext=True*) → tuple

Encrypts plaintext for recipient actor. Optionally signs the message as well.

Parameters

- **recipient** – The character whose public key will be used to encrypt cleartext.
- **plaintext** – The secret to be encrypted.
- **sign** – Whether or not to sign the message.
- **sign_plaintext** – When signing, the cleartext is signed if this is True, Otherwise, the resulting ciphertext is signed.

Returns A tuple, (ciphertext, signature). If sign==False, then signature will be NOT_SIGNED.

classmethod from_public_keys(*powers_and_material: Dict*, *federated_only=True*, **args*, ***kwargs*) → nucypher.characters.base.Character

Sometimes we discover a Character and, at the same moment, learn the public parts of more of their powers. Here, we take a Dict (powers_and_key_bytes) in the following format: {CryptoPowerUp class: public_material_bytes}

Each item in the collection will have the CryptoPowerUp instantiated with the public_material_bytes, and the resulting CryptoPowerUp instance consumed by the Character.

# TODO: Need to be federated only until we figure out the best way to get the checksum_public_address in here.

public_keys(*power_up_class: ClassVar*)

Pass a power_up_class, get the public material for this Character which corresponds to that class - whatever type of object that may be.

If the Character doesn't have the power corresponding to that class, raises the appropriate PowerUpError (ie, NoSigningPower or NoDecryptingPower).

store_metadata(*filepath: str*) → str

Save this node to the disk. :param filepath: Output filepath to save node metadata. :return: Output filepath

verify_from(*stranger: nucypher.characters.base.Character, message_kit: Union[nucypher.crypto.kits.UmbralMessageKit, bytes], signature: umbral.signing.Signature = None, decrypt=False, label=None*) → bytes

Inverse of encrypt_for.

Parameters

- **stranger** – A Character instance representing the actor whom the sender claims to be. We check the public key owned by this Character instance to verify.
- **message_kit** – the message to be (perhaps decrypted and) verified.
- **signature** – The signature to check.
- **decrypt** – Whether or not to decrypt the messages.

Returns Whether or not the signature is valid, the decrypted plaintext or NO_DECRYPTION_PERFORMED

class nucypher.characters.lawful.**Alice**(*is_me=True*, *federated_only=False*, *network_middleware=None*, *controller=True*, **args*, ***kwargs*)

create_policy(*bob: nucypher.characters.lawful.Bob*, *label: bytes*, *m: int*, *n: int*, *federated: bool = False*, *expiration: maya.core.MayaDT = None*, *value: int = None*, *handpicked_ursulas: set = None*)

Create a Policy to share uri with bob. Generates KFrags and attaches them.

generate_kfrags(*bob: nucypher.characters.lawful.Bob*, *label: bytes*, *m: int*, *n: int*) → List

Generates re-encryption key frags ("KFrags") and returns them.

These KFrags can be used by Ursula to re-encrypt a Capsule for Bob so that he can activate the Capsule.

Parameters

- **bob** – Bob instance which will be able to decrypt messages re-encrypted with these kfrags.
- **m** – Minimum number of kfrags needed to activate a Capsule.
- **n** – Total number of kfrags to generate

revoke(*policy*) → Dict

Parses the treasure map and revokes arrangements in it. If any arrangements can't be revoked, then the node_id is added to a dict as a key, and the revocation and Ursula's response is added as a value.

class nucypher.characters.lawful.**Bob**(*controller=True*, **args*, ***kwargs*)

exception IncorrectCFragReceived(*evidence*)

Raised when Bob detects an incorrect CFrag returned by some Ursula

follow_treasure_map(*treasure_map=None*, *map_id=None*, *block=False*, *new_thread=False*, *timeout=10*, *allow_missing=0*)

Follows a known TreasureMap, looking it up by map_id.

Determines which Ursulas are known and which are unknown.

If block, will block until either unknown nodes are discovered or until timeout seconds have elapsed. After timeout seconds, if more than allow_missing nodes are still unknown, raises NotEnoughUrsulas.

If block and new_thread, does the same thing but on a different thread, returning a Deferred which fires after the blocking has concluded.

Otherwise, returns (unknown_nodes, known_nodes).

# TODO: Check if nodes are up, declare them phantom if not.

get_treasure_map_from_known_ursulas(*network_middleware*, *map_id*)

Iterate through the nodes we know, asking for the TreasureMap. Return the first one who has it.

peek_at_treasure_map(*treasure_map=None*, *map_id=None*)

Take a quick gander at the TreasureMap matching map_id to see which nodes are already known to us.

Don't do any learning, pinging, or anything other than just seeing whether we know or don't know the nodes.

Return two sets: nodes that are unknown to us, nodes that are known to us.

class nucypher.characters.lawful.**Enrico**(*policy_encrypting_key*, *controller: bool = True*, **args*, ***kwargs*)

A Character that represents a Data Source that encrypts data for some policy's public key

classmethod from_alice(*alice: nucypher.characters.lawful.Alice*, *label: bytes*)

Parameters

- **alice** – Not a stranger. This is your Alice who will derive the policy keypair, leaving Enrico with the public part.
- **label** – The label with which to derive the key.

Returns

class nucypher.characters.lawful.**Ursula**(*rest_host: str*, *rest_port: int*, *domains: Set = (b'GLOBAL_DOMAIN',)*, *certificate: cryptography.x509.base.Certificate = None*, *certificate_filepath: str = None*, *db_filepath: str = None*, *is_me: bool = True*, *interface_signature=None*, *timestamp=None*, *identity_evidence: bytes = NOT_SIGNED*, *checksum_public_address: str = None*, *password: str = None*, *abort_on_learning_error: bool = False*, *federated_only: bool = False*, *start_learning_now: bool = None*, *crypto_power=None*, *tls_curve: cryptography.hazmat.primitives.asymmetric.ec.EllipticCurve = None*, *known_nodes: Iterable = None*, ***character_kwargs*)

exception NotEnoughUrsulas

All Characters depend on knowing about enough Ursulas to perform their role. This exception is raised when a piece of logic can't proceed without more Ursulas.

exception NotFound

classmethod from_seednode_metadata(*seednode_metadata*, **args*, ***kwargs*)

Essentially another deserialization method, but this one doesn't reconstruct a complete node from bytes; instead it's just enough to connect to and verify a node.

work_orders(*bob=None*)

TODO: This is better written as a model method for Ursula's datastore.

1.15 Config

class `nucypher.config.node.`**`NodeConfiguration`**(*config_root: str = None, config_file_location: str = None, dev_mode: bool = False, federated_only: bool = False, is_me: bool = True, checksum_public_address: str = None, crypto_power: nucypher.crypto.powers.CryptoPower = None, keyring: nucypher.config.keyring.NucypherKeyring = None, keyring_dir: str = None, learn_on_same_thread: bool = False, abort_on_learning_error: bool = False, start_learning_now: bool = True, rest_host: str = None, rest_port: int = None, tls_curve: cryptography.hazmat.primitives.asymmetric.ec.EllipticCurve = None, certificate: cryptography.x509.base.Certificate = None, domains: Set[str] = None, interface_signature: umbral.signing.Signature = None, network_middleware: nucypher.network.middleware.RestMiddleware = None, known_nodes: set = None, node_storage: nucypher.config.storages.NodeStorage = None, reload_metadata: bool = True, save_metadata: bool = True, poa: bool = False, provider_uri: str = None, registry_source: str = None, registry_filepath: str = None, import_seed_registry: bool = False*)

‘Sideways Engagement’ of Character classes; a reflection of input parameters.

exception ConfigurationError

exception InvalidConfiguration

NODE_DESERIALIZER()

Binary data of hexadecimal representation.

hexstr must contain an even number of hex digits (upper or lower case).

NODE_SERIALIZER()

Hexadecimal representation of binary data.

The return value is a bytes object.

connect_to_contracts() → None

Initialize contract agency and set them on config

dynamic_payload

Exported dynamic configuration values for initializing Ursula

classmethod from_configuration_file(*filepath: str = None, **overrides*) → nucypher.config.node.NodeConfiguration

Initialize a NodeConfiguration from a JSON file.

classmethod generate(*password: str*, *no_registry: bool*, **args*, ***kwargs*) → UrsulaConfiguration
Shortcut: Hook-up a new initial installation and write configuration file to the disk

classmethod generate_runtime_filepaths(*config_root: str*) → dict
Dynamically generate paths based on configuration root directory

initialize(*password: str*, *import_registry: bool = True*) → str
Initialize a new configuration.

produce(***overrides*)
Initialize a new character instance and return it.

static_payload
Exported static configuration values for initializing Ursula

to_configuration_file(*filepath: str = None*) → str
Write the static_payload to a JSON file.

class nucypher.config.characters.**AliceConfiguration**(*config_root: str = None*, *config_file_location: str = None*, *dev_mode: bool = False*, *federated_only: bool = False*, *is_me: bool = True*, *checksum_public_address: str = None*, *crypto_power: nucypher.crypto.powers.CryptoPower = None*, *keyring: nucypher.config.keyring.NucypherKeyring = None*, *keyring_dir: str = None*, *learn_on_same_thread: bool = False*, *abort_on_learning_error: bool = False*, *start_learning_now: bool = True*, *rest_host: str = None*, *rest_port: int = None*, *tls_curve: cryptography.hazmat.primitives.asymmetric.ec.EllipticCurve = None*, *certificate: cryptography.x509.base.Certificate = None*, *domains: Set[str] = None*, *interface_signature: umbral.signing.Signature = None*, *network_middleware: nucypher.network.middleware.RestMiddleware = None*, *known_nodes: set = None*, *node_storage: nucypher.config.storages.NodeStorage = None*, *reload_metadata: bool = True*, *save_metadata: bool = True*, *poa: bool = False*, *provider_uri: str = None*, *registry_source: str = None*, *registry_filepath: str = None*, *import_seed_registry: bool = False*)

class Alice(*is_me=True*, *federated_only=False*, *network_middleware=None*, *controller=True*, **args*, ***kwargs*)

create_policy(*bob: nucypher.characters.lawful.Bob*, *label: bytes*, *m: int*, *n: int*, *federated: bool = False*, *expiration: maya.core.MayaDT = None*, *value: int = None*, *handpicked_ursulas: set = None*)

Create a Policy to share uri with bob. Generates KFrags and attaches them.

generate_kfrags(*bob: nucypher.characters.lawful.Bob*, *label: bytes*, *m: int*, *n: int*) → List

Generates re-encryption key frags ("KFrags") and returns them.

These KFrags can be used by Ursula to re-encrypt a Capsule for Bob so that he can activate the Capsule.

Parameters

- **bob** – Bob instance which will be able to decrypt messages re-encrypted with these kfrags.
- **m** – Minimum number of kfrags needed to activate a Capsule.
- **n** – Total number of kfrags to generate

revoke(*policy*) → Dict

Parses the treasure map and revokes arrangements in it. If any arrangements can't be revoked, then the node_id is added to a dict as a key, and the revocation and Ursula's response is added as a value.

class nucypher.config.characters.**BobConfiguration**(*config_root: str = None*, *config_file_location: str = None*, *dev_mode: bool = False*, *federated_only: bool = False*, *is_me: bool = True*, *checksum_public_address: str = None*, *crypto_power: nucypher.crypto.powers.CryptoPower = None*, *keyring: nucypher.config.keyring.NucypherKeyring = None*, *keyring_dir: str = None*, *learn_on_same_thread: bool = False*, *abort_on_learning_error: bool = False*, *start_learning_now: bool = True*, *rest_host: str = None*, *rest_port: int = None*, *tls_curve: cryptography.hazmat.primitives.asymmetric.ec.EllipticCurve = None*, *certificate: cryptography.x509.base.Certificate = None*, *domains: Set[str] = None*, *interface_signature: umbral.signing.Signature = None*, *network_middleware: nucypher.network.middleware.RestMiddleware = None*, *known_nodes: set = None*, *node_storage: nucypher.config.storages.NodeStorage = None*, *reload_metadata: bool = True*, *save_metadata: bool = True*, *poa: bool = False*, *provider_uri: str = None*, *registry_source: str = None*, *registry_filepath: str = None*, *import_seed_registry: bool = False*)

class Bob(*controller=True*, **args*, ***kwargs*)

exception IncorrectCFragReceived(*evidence*)

Raised when Bob detects an incorrect CFrag returned by some Ursula

follow_treasure_map(*treasure_map=None*, *map_id=None*, *block=False*, *new_thread=False*, *timeout=10*, *allow_missing=0*)

Follows a known TreasureMap, looking it up by map_id.

Determines which Ursulas are known and which are unknown.

If block, will block until either unknown nodes are discovered or until timeout seconds have elapsed. After timeout seconds, if more than allow_missing nodes are still unknown, raises NotEnoughUrsulas.

If block and new_thread, does the same thing but on a different thread, returning a Deferred which fires after the blocking has concluded.

Otherwise, returns (unknown_nodes, known_nodes).

# TODO: Check if nodes are up, declare them phantom if not.

get_treasure_map_from_known_ursulas(*network_middleware*, *map_id*)
Iterate through the nodes we know, asking for the TreasureMap. Return the first one who has it.

peek_at_treasure_map(*treasure_map=None*, *map_id=None*)
Take a quick gander at the TreasureMap matching map_id to see which nodes are already known to us.

Don't do any learning, pinging, or anything other than just seeing whether we know or don't know the nodes.

Return two sets: nodes that are unknown to us, nodes that are known to us.

class nucypher.config.characters.**UrsulaConfiguration**(*dev_mode: bool = False*, *db_filepath: str = None*, **args*, ***kwargs*)

class Ursula(*rest_host: str*, *rest_port: int*, *domains: Set = (b'GLOBAL_DOMAIN',)*, *certificate: cryptography.x509.base.Certificate = None*, *certificate_filepath: str = None*, *db_filepath: str = None*, *is_me: bool = True*, *interface_signature=None*, *timestamp=None*, *identity_evidence: bytes = NOT_SIGNED*, *checksum_public_address: str = None*, *password: str = None*, *abort_on_learning_error: bool = False*, *federated_only: bool = False*, *start_learning_now: bool = None*, *crypto_power=None*, *tls_curve: cryptography.hazmat.primitives.asymmetric.ec.EllipticCurve = None*, *known_nodes: Iterable = None*, ***character_kwargs*)

exception NotEnoughUrsulas
All Characters depend on knowing about enough Ursulas to perform their role. This exception is raised when a piece of logic can't proceed without more Ursulas.

exception NotFound

classmethod from_seednode_metadata(*seednode_metadata*, **args*, ***kwargs*)
Essentially another deserialization method, but this one doesn't reconstruct a complete node from bytes; instead it's just enough to connect to and verify a node.

work_orders(*bob=None*)
TODO: This is better written as a model method for Ursula's datastore.

dynamic_payload
Exported dynamic configuration values for initializing Ursula

generate_runtime_filepaths(*config_root: str*) → dict
Dynamically generate paths based on configuration root directory

produce(***overrides*)
Produce a new Ursula from configuration

static_payload
Exported static configuration values for initializing Ursula

class nucypher.config.storages.**ForgetfulNodeStorage**(**args*, ***kwargs*)

all(*federated_only: bool*, *certificates_only: bool = False*) → set
Return s set of all stored nodes

clear(*metadata: bool = True*, *certificates: bool = True*) → None
Forget all stored nodes and certificates

classmethod from_payload(*payload: dict*, **args*, ***kwargs*) → nucypher.config.storages.ForgetfulNodeStorage
Alternate constructor to create a storage instance from JSON-like configuration

get (*federated_only: bool*, *host: str = None*, *checksum_address: str = None*, *certificate_only: bool = False*)
Retrieve a single stored node

initialize() → bool
Returns True if initialization was successful

remove (*checksum_address: str*, *metadata: bool = True*, *certificate: bool = True*) → Tuple[bool, str]
Remove a single stored node

store_node_metadata (*node*, *filepath: str = None*)
Save a single node's metadata and tls certificate

class nucypher.config.storages.**LocalFileBasedNodeStorage** (*config_root: str = None*, *storage_root: str = None*, *metadata_dir: str = None*, *certificates_dir: str = None*, **args*, ***kwargs*)

exception NoNodeMetadataFileFound

all (*federated_only: bool*, *certificates_only: bool = False*) → Set[Union[Any, cryptography.x509.base.Certificate]]
Return s set of all stored nodes

clear (*metadata: bool = True*, *certificates: bool = True*) → None
Forget all stored nodes and certificates

classmethod from_payload (*payload: dict*, **args*, ***kwargs*) → nucypher.config.storages.LocalFileBasedNodeStorage
Instantiate a storage object from a dictionary

get (*checksum_address: str*, *federated_only: bool*, *certificate_only: bool = False*)
Retrieve a single stored node

initialize() → bool
One-time initialization steps to establish a node storage backend

remove (*checksum_address: str*, *metadata: bool = True*, *certificate: bool = True*) → None
Remove a single stored node

store_node_metadata (*node*, *filepath: str = None*) → str
Save a single node's metadata and tls certificate

class nucypher.config.storages.**NodeStorage** (*federated_only: bool*, *character_class=None*, *serializer: Callable = <built-in function hexlify>*, *deserializer: Callable = <built-in function unhexlify>*)

NODE_DESERIALIZER()
Binary data of hexadecimal representation.

hexstr must contain an even number of hex digits (upper or lower case).

NODE_SERIALIZER()
Hexadecimal representation of binary data.

The return value is a bytes object.

exception NodeStorageError

exception UnknownNode

all (*federated_only: bool*, *certificates_only: bool = False*) → set
Return s set of all stored nodes

clear () → bool
Remove all stored nodes

classmethod from_payload (*data: dict*, **args*, ***kwargs*) → nucypher.config.storages.NodeStorage
Instantiate a storage object from a dictionary

get (*checksum_address: str*, *federated_only: bool*)
Retrieve a single stored node

initialize ()
One-time initialization steps to establish a node storage backend

remove (*checksum_address: str*) → bool
Remove a single stored node

store_node_metadata (*node*, *filepath: str = None*) → str
Save a single node's metadata and tls certificate

class nucypher.config.storages.**TemporaryFileBasedNodeStorage** (**args*, ***kwargs*)

initialize () → bool
One-time initialization steps to establish a node storage backend

1.16 Crypto

nucypher.crypto.api.**ecdsa_sign** (*message: bytes*, *privkey: umbral.keys.UmbralPrivateKey*) → bytes
Accepts a hashed message and signs it with the private key given.

Parameters

- **message** – Message to hash and sign
- **privkey** – Private key to sign with

Returns signature

nucypher.crypto.api.**ecdsa_verify** (*message: bytes*, *signature: bytes*, *pubkey: umbral.keys.UmbralPublicKey*) → bool
Accepts a message and signature and verifies it with the provided public key.

Parameters

- **message** – Message to verify
- **signature** – Signature to verify
- **pubkey** – UmbralPublicKey to verify signature with

Returns True if valid, False if invalid.

nucypher.crypto.api.**keccak_digest** (**messages*) → bytes
Accepts an iterable containing bytes and digests it returning a Keccak digest of 32 bytes (keccak_256).

Although we use SHA256 in many cases, we keep keccak handy in order to provide compatibility with the Ethereum blockchain.

Parameters **bytes** – Data to hash

Return type bytes

Returns bytestring of digested data

`nucypher.crypto.api.`**`secure_random`**(*num_bytes: int*) → bytes

Returns an amount *num_bytes* of data from the OS's random device. If a randomness source isn't found, returns a *NotImplementedError*. In this case, a secure random source most likely doesn't exist and randomness will have to found elsewhere.

Parameters **`num_bytes`** – Number of bytes to return.

Returns bytes

`nucypher.crypto.api.`**`secure_random_range`**(*min: int*, *max: int*) → int

Returns a number from a secure random source betwee the range of *min* and *max* - 1.

Parameters

- **`min`** – Minimum number in the range
- **`max`** – Maximum number in the range

Returns int

`class` `nucypher.crypto.powers.`**`BlockchainPower`**(*blockchain: Blockchain*, *account: str*)

Allows for transacting on a Blockchain via web3 backend.

`not_found_error`

alias of *`NoBlockchainPower`*

`sign_message`(*message: bytes*)

Signs the message with the private key of the BlockchainPower.

`unlock_account`(*password: str*, *duration: int = None*)

Unlocks the account for the specified duration. If no duration is provided, it will remain unlocked indefinitely.

`verify_message`(*address: str*, *pubkey: bytes*, *message: bytes*, *signature_bytes: bytes*)

Verifies that the message was signed by the keypair.

`class` `nucypher.crypto.powers.`**`CryptoPowerUp`**

Gives you MORE CryptoPower!

`class` `nucypher.crypto.powers.`**`DecryptingPower`**(*pubkey: umbral.keys.UmbralPublicKey = None*, *keypair: nucypher.keystore.keypairs.Keypair = None*)

`not_found_error`

alias of *`NoDecryptingPower`*

`class` `nucypher.crypto.powers.`**`DelegatingPower`**(*keying_material: Optional[bytes] = None*, *password: Optional[bytes] = None*)

`generate_kfrags`(*bob_pubkey_enc*, *signer*, *label*, *m*, *n*) → Tuple[umbral.keys.UmbralPublicKey, List]

Generates re-encryption key frags ("KFrags") and returns them.

These KFrags can be used by Ursula to re-encrypt a Capsule for Bob so that he can activate the Capsule. :param bob_pubkey_enc: Bob's public key :param m: Minimum number of KFrags needed to rebuild ciphertext :param n: Total number of KFrags to generate

`class` `nucypher.crypto.powers.`**`DerivedKeyBasedPower`**

Rather than rely on an established KeyPair, this type of power derives a key at moments defined by the user.

class `nucypher.crypto.powers.`**`KeyPairBasedPower`**(*pubkey: umbral.keys.UmbralPublicKey = None*, *keypair: nucypher.keystore.keypairs.Keypair = None*)

exception `nucypher.crypto.powers.`**`NoBlockchainPower`**

exception `nucypher.crypto.powers.`**`NoDecryptingPower`**

exception `nucypher.crypto.powers.`**`NoSigningPower`**

exception `nucypher.crypto.powers.`**`PowerUpError`**

class `nucypher.crypto.powers.`**`SigningPower`**(*pubkey: umbral.keys.UmbralPublicKey = None*, *keypair: nucypher.keystore.keypairs.Keypair = None*)

`not_found_error`
alias of `NoSigningPower`

exception `nucypher.crypto.signing.`**`InvalidSignature`**
Raised when a Signature is not valid.

class `nucypher.crypto.signing.`**`SignatureStamp`**(*verifying_key*, *signer: umbral.signing.Signer = None*)
Can be called to sign something or used to express the signing public key as bytes.

`fingerprint`()
Hashes the key using keccak-256 and returns the hexdigest in bytes.

Returns Hexdigest fingerprint of key (keccak-256) in bytes

class `nucypher.crypto.signing.`**`StrangerStamp`**(*verifying_key*, *signer: umbral.signing.Signer = None*)
SignatureStamp of a stranger (ie, can only be used to glean public key, not to sign)

`nucypher.crypto.utils.`**`fingerprint_from_key`**(*public_key: Any*)
Hashes a key using keccak-256 and returns the hexdigest in bytes. :return: Hexdigest fingerprint of key (keccak-256) in bytes

`nucypher.crypto.utils.`**`recover_pubkey_from_signature`**(*prehashed_message*, *signature*, *v_value_to_try=None*) → bytes
Recovers a serialized, compressed public key from a signature. It allows to specify a potential v value, in which case it assumes the signature has the traditional (r,s) raw format. If a v value is not present, it assumes the signature has the recoverable format (r, s, v).

Parameters

- **`prehashed_message`** – Prehashed message
- **`signature`** – The signature from which the pubkey is recovered
- **`v_value_to_try`** – A potential v value to try

Returns The compressed byte-serialized representation of the recovered public key

1.17 Keyring

class nucypher.config.keyring.**NucypherKeyring**(*account: str, keyring_root: str = None, root_key_path: str = None, pub_root_key_path: str = None, signing_key_path: str = None, pub_signing_key_path: str = None, delegating_key_path: str = None, tls_key_path: str = None, tls_certificate_path: str = None*)

Handles keys for a single identity, recognized by account. Warning: This class handles private keys!

- **keyring**
  - **.private**
    - key.priv
    - key.priv.pem
  - **public**
    - key.pub
    - cert.pem

exception AuthenticationFailed

exception KeyringError

exception KeyringLocked

classmethod generate(*password: str, encrypting: bool, rest: bool, host: str = None, curve: cryptography.hazmat.primitives.asymmetric.ec.EllipticCurve = None, keyring_root: str = None, checksum_address: str = None*) → nucypher.config.keyring.NucypherKeyring

Generates new encrypting, signing, and wallet keys encrypted with the password, respectively saving keyfiles on the local filesystem from *default* paths, returning the corresponding Keyring instance.

lock() → bool

Make efforts to remove references to the cached key data

static validate_password(*password: str*) → List

Validate a password and return True or raise an error with a failure reason.

NOTICE: Do not raise inside this function.

nucypher.config.keyring.**unlock_required**(*func*)

Method decorator

1.18 Keystore

class nucypher.keystore.keypairs.**DecryptingKeypair**(**args, **kwargs*)

A keypair for Umbral

decrypt(*message_kit: nucypher.crypto.kits.MessageKit*) → bytes

Decrypt data encrypted with Umbral.

Returns bytes

class `nucypher.keystore.keypairs.`**`HostingKeypair`**(*host: str*, *checksum_public_address: str = None*, *private_key: Union[umbral.keys.UmbralPrivateKey, umbral.keys.UmbralPublicKey] = None*, *curve=None*, *certificate=None*, *certificate_filepath: str = None*, *generate_certificate=True*)

A keypair for TLS'ing.

class `nucypher.keystore.keypairs.`**`Keypair`**(*private_key=None*, *public_key=None*, *generate_keys_if_needed=True*)

A parent Keypair class for all types of Keypairs.

`fingerprint`()

Hashes the key using keccak-256 and returns the hexdigest in bytes.

Returns Hexdigest fingerprint of key (keccak-256) in bytes

`serialize_pubkey`(*as_b64=False*) → bytes

Serializes the pubkey for storage/transport in either urlsafe base64 or as a bytestring.

Parameters **`as_b64`** – Return the pubkey as urlsafe base64 byte string

Returns The serialized pubkey in bytes

class `nucypher.keystore.keypairs.`**`SigningKeypair`**(**args*, ***kwargs*)

A SigningKeypair that uses ECDSA.

`sign`(*message: bytes*) → bytes

Signs a hashed message and returns a signature.

Parameters **`message`** – The message to sign

Returns Signature in bytes

class `nucypher.keystore.keystore.`**`KeyStore`**(*sqlalchemy_engine=None*)

A storage class of cryptographic keys.

`add_key`(*key*, *is_signing=True*, *session=None*) → nucypher.keystore.db.models.Key

Parameters **`key`** – Keypair object to store in the keystore.

Returns The newly added key object.

`add_policy_arrangement`(*expiration*, *id*, *kfrag=None*, *alice_pubkey_sig=None*, *alice_signature=None*, *session=None*) → nucypher.keystore.db.models.PolicyArrangement

Creates a PolicyArrangement to the Keystore.

Returns The newly added PolicyArrangement object

`add_workorder`(*bob_pubkey_sig*, *bob_signature*, *arrangement_id*, *session=None*) → nucypher.keystore.db.models.Workorder

Adds a Workorder to the keystore.

`del_key`(*fingerprint: bytes*, *session=None*)

Deletes a key from the KeyStore.

Parameters **`fingerprint`** – Fingerprint of key to delete

`del_policy_arrangement`(*arrangement_id: bytes*, *session=None*)

Deletes a PolicyArrangement from the Keystore.

`del_workorders`(*arrangement_id: bytes*, *session=None*)

Deletes a Workorder from the Keystore.

`get_key`(*fingerprint: bytes*, *session=None*) → Union[nucypher.keystore.keypairs.DecryptingKeypair, nucypher.keystore.keypairs.SigningKeypair]
Returns a key from the KeyStore.

Parameters **`fingerprint`** – Fingerprint, in bytes, of key to return

Returns Keypair of the returned key.

`get_policy_arrangement`(*arrangement_id: bytes*, *session=None*) → nucypher.keystore.db.models.PolicyArrangement
Returns the PolicyArrangement by its HRAC.

Returns The PolicyArrangement object

`get_workorders`(*arrangement_id: bytes*, *session=None*) → nucypher.keystore.db.models.Workorder
Returns a list of Workorders by HRAC.

`exception` `nucypher.keystore.keystore.`**`NotFound`**
Exception class for KeyStore calls for objects that don't exist.

1.19 Network

`exception` `nucypher.network.middleware.`**`NotFound`**

`exception` `nucypher.network.middleware.`**`UnexpectedResponse`**

`class` `nucypher.network.nodes.`**`FleetStateTracker`**
A representation of a fleet of NuCypher nodes.

`state_template`
alias of `FleetState`

`class` `nucypher.network.nodes.`**`Learner`**(*domains: Set*, *network_middleware: nucypher.network.middleware.RestMiddleware = <nucypher.network.middleware.RestMiddleware object>*, *start_learning_now: bool = False*, *learn_on_same_thread: bool = False*, *known_nodes: tuple = None*, *seed_nodes: Tuple[tuple] = None*, *node_storage=None*, *save_metadata: bool = False*, *abort_on_learning_error: bool = False*, *lonely: bool = False*)
Any participant in the "learning loop" - a class inheriting from this one has the ability, synchronously or asynchronously, to learn about nodes in the network, verify some essential details about them, and store information about them for later use.

`exception NotATeacher`
Raised when a character cannot be properly utilized because it does not have the proper attributes for learning or verification.

`exception NotEnoughNodes`

`exception NotEnoughTeachers`

`exception UnresponsiveTeacher`

`keep_learning_about_nodes`()
Continually learn about new nodes.

`learn_from_teacher_node`(*eager=True*)
Sends a request to node_url to find out about known nodes.

load_seednodes(*read_storages: bool = True*, *retry_attempts: int = 3*)
Engage known nodes from storages and pre-fetch hardcoded seednode certificates for node learning.

stop_learning_loop(*reason=None*)
Only for tests at this point. Maybe some day for graceful shutdowns.

tracker_class
alias of *FleetStateTracker*

exception nucypher.network.protocols.**SuspiciousActivity**
raised when an action appears to amount to malicious conduct.

class nucypher.network.server.**TLSHostingPower**(*host: str*, *public_certificate=None*, *public_certificate_filepath=None*, **args*, ***kwargs*)

exception NoHostingPower

not_found_error
alias of *TLSHostingPower.NoHostingPower*

1.20 Policy

class nucypher.policy.models.**Arrangement**(*alice*, *expiration*, *ursula=None*, *arrangement_id=None*, *kfrag=UNKNOWN_KFRAG*, *value=None*, *alices_signature=None*)
A Policy must be implemented by arrangements with n Ursulas. This class tracks the status of that implementation.

encrypt_payload_for_ursula()
Craft an offer to send to Ursula.

revoke()
Revoke arrangement.

value = None
These will normally not be set if Alice is drawing up this arrangement - she hasn't assigned a kfrag yet (because she doesn't know if this Arrangement will be accepted). She doesn't have an Ursula, for the same reason.

class nucypher.policy.models.**FederatedPolicy**(*ursulas: Set[nucypher.characters.lawful.Ursula]*, **args*, ***kwargs*)

make_arrangements(*network_middleware: nucypher.network.middleware.RestMiddleware*, *value: int*, *expiration: maya.core.MayaDT*, *handpicked_ursulas: Set[nucypher.characters.lawful.Ursula] = None*) → None
Create and consider n Arrangement objects.

class nucypher.policy.models.**Policy**(*alice*, *label*, *bob=None*, *kfrags=(UNKNOWN_KFRAG,)*, *public_key=None*, *m: int = None*, *alices_signature=NOT_SIGNED*)
An edict by Alice, arranged with n Ursulas, to perform re-encryption for a specific Bob for a specific path.

Once Alice is ready to enact a Policy, she generates KFrags, which become part of the Policy.

Each Ursula is offered a Arrangement (see above) for a given Policy by Alice.

Once Alice has secured agreement with n Ursulas to enact a Policy, she sends each a KFrag, and generates a TreasureMap for the Policy, recording which Ursulas got a KFrag.

exception MoreKFragsThanArrangements
Raised when a Policy has been used to generate Arrangements with Ursulas insufficient number such that we don't have enough KFrags to give to each Ursula.

enact(*network_middleware*, *publish=True*) → dict
Assign kfrags to ursulas_on_network, and distribute them via REST, populating enacted_arrangements

hrac() → bytes
This function is hanging on for dear life. After 180 is closed, it can be completely deprecated.

The "hashed resource authentication code".

A hash of: * Alice's public key * Bob's public key * the label

Alice and Bob have all the information they need to construct this. Ursula does not, so we share it with her.

make_arrangements(*network_middleware: nucypher.network.middleware.RestMiddleware*, *deposit: int*, *expiration: maya.core.MayaDT*, *ursulas: Set[nucypher.characters.lawful.Ursula] = None*) → None
Create and consider n Arrangement objects.

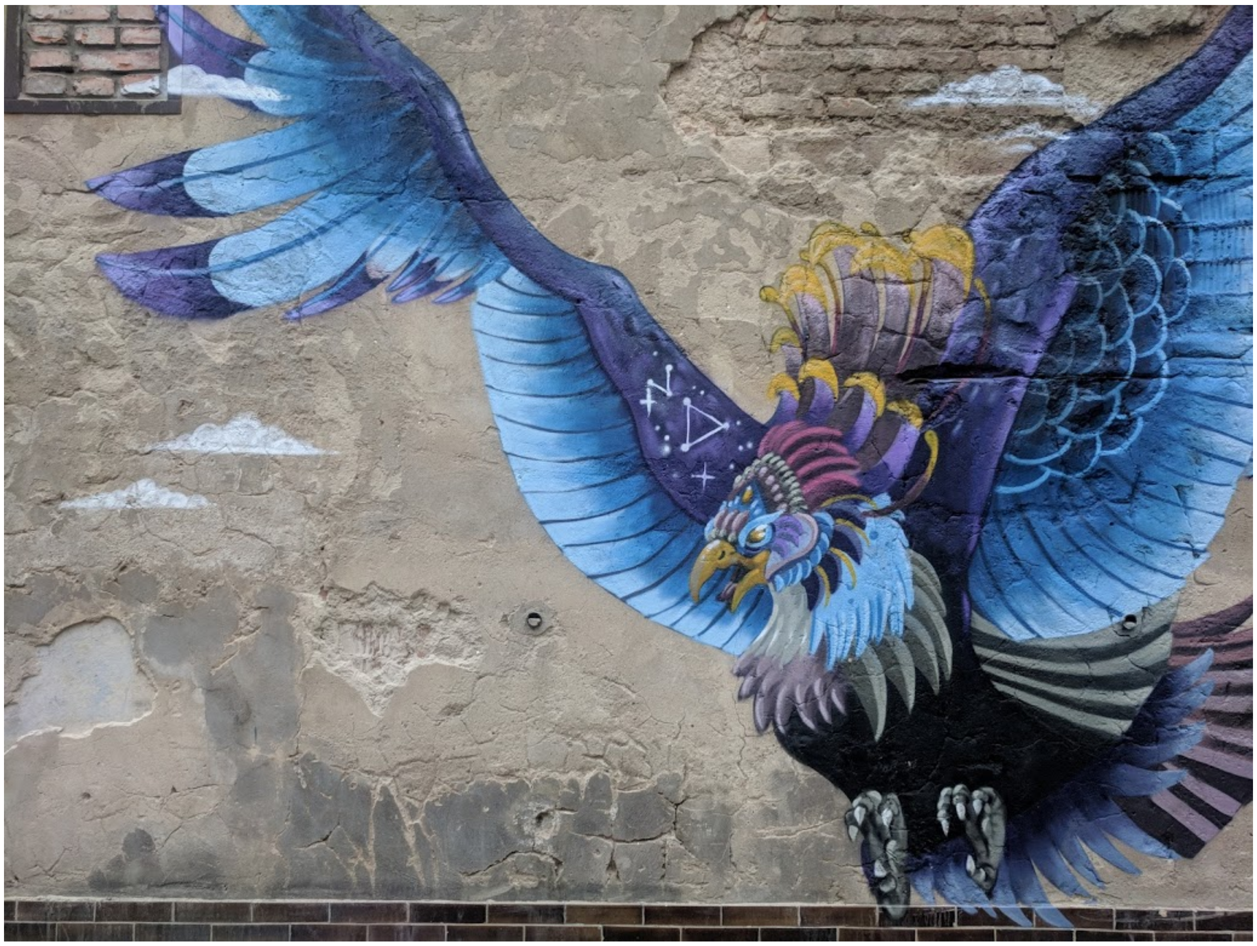
publish(*network_middleware: nucypher.network.middleware.RestMiddleware*) → dict
Spread word of this Policy far and wide.

Base publication method for spreading news of the policy. If this is a blockchain policy, this includes writing to PolicyManager contract storage.

class nucypher.policy.models.**Revocation**(*arrangement_id: bytes*, *signer: SignatureStamp = None*, *signature: umbral.signing.Signature = None*)
Represents a string used by characters to perform a revocation on a specific Ursula. It's a bytestring made of the following format: REVOKE-<arrangement id to revoke><signature of the previous string> This is sent as a payload in a DELETE method to the /KFrag/ endpoint.

verify_signature(*alice_pubkey: umbral.keys.UmbralPublicKey*)
Verifies the revocation was from the provided pubkey.

1.21 NuCypher Release Notes



1.21.1 v0.1.0-alpha.0 (Genesis Release)

1. In the beginning cryptography created the ciphertext and the plaintext.
2. And the ciphertext was without form, and void; and darkness was upon the face of the secret (symmetric) key. And then the spirit of asymmetric crypto moved upon the face of the interwebs.
3. And cryptographers said, Let there be encapsulation: and there was encapsulation.
4. And they saw *the light, that it was good: and divided* the symmetric ciphertext from the encapsulated key.
5. And they called the symmetric ciphertext "bulk data", and the encapsulated key the "*Capsule*". And the encryption and the decryption were the first secure message.
6. And they said, Let there be a re-encryption node in the midst of the interwebs, and let it divide the Bobs from the Bobs.
7. And they made the re-encryption protocol, and divided the *Bobs* which were authorized under the *Policy* from the *Bobs* which were not authorized: and it was so.
8. And they called the re-encryption nodes *Ursula*. And the re-encryption and the decryption were the second secure message.

9. And they said, Let the ciphertext fragments under the interwebs be gathered together unto one place, and let the Capsule appear: and it was so.
10. And they called the total number of fragments `n`, and the number that Bob needed to reassamble `m`: and they saw that it was good.
11. And they said, Let the interwebs bring forth a Character, *Alice*, who yields the *Policy*, and a Character, *Enrico*, who yields the encrypted message after his kind, whose symmetric key is in itself, and let the Ursulas that are participating in the *Policy* be listed in a *TreasureMap*, upon the interwebs: and it was so.
12. And the earth brought forth *Alice*, and *Enrico*, and *Ursula*, and *Bob*: and the decentralized dapp developer community saw that it was good.
13. And the complete access management flow was the third secure message.
14. And they said, Let there be a coherent metaphor and API in the cryptology of the interwebs, and let Policies expire at a specific datetime or upon revocation, and let them be for signs, and for seasons, and for days, and years:
15. And let them be for decentralized re-encryption tooling in the cryptography of the interwebs to give privacy upon the blockchain: and it was so.

CHAPTER 2

Indices and Tables

- genindex
- modindex
- search

Python Module Index

Index