Final secure block device

D2.7

Project reference no. 653884

February 2018

European Commission

Horizon 2020
European Union funding
for Research & Innovation

Document information

Scheduled delivery	28.02.2018
Actual delivery	28.02.2018
Version	1.2
Responsible Partner	UniNE

Dissemination level

Public

Revision history

Date	Editor	Status	Version	Changes
20.02.2018	D.Burihabwa	Draft	0.1	Initial version
27.02.2018	D.Burihabwa	Draft	0.2	Amended version
28.02.2018	D.Burihabwa	Final	1.0	Final version
28.02.2018	D.Burihabwa	Final	1.1	Internal reviewers correction
28.02.2018	H. Mercier	Final	1.2	Final version with C&H comments

Contributors

D. Burihabwa (UniNE)
H. Mercier (UniNE)

Internal reviewers

J. Paulo (INESC TEC)
S. Schmerler (Cloud&Heat)

Acknowledgements

This project is partially funded by the European Commission Horizon 2020 work programme under grant agreement no. 653884.

More information

Additional information and public deliverables of SafeCloud can be found at http://www.safecloud-project.eu

Table of contents

Executive summary

The past few years have seen the emergence of competitive commercial storage solutions. Individuals and companies looking for storage space to share documents with the world or host their remote backups now have plenty of cloud alternatives to turn to. But whenever a user chooses one of these solutions, it entrusts the cloud provider with the durability of their data. To do so at a reasonable cost, most providers perform multi-site replication. While this enables recovery from accidental loss it does not prevent tampering from malicious outsiders and even the provider itself. To solve this problem, the SafeCloud project provides solutions as part of WP2.

To satisfy different storage needs, WP2 builds upon existing self-hosted and commercial storage components (SS1) to offer two solutions: the secure data archive (SS2) and the secure file system (SS3). SS3 is a POSIX-compliant distributed file system on top of cloud storage providers that focuses on the security of the data. In contrast, SS2 is a RESTful object store focusing on durability. With different objectives, SS2 and SS3 both leverage existing bricks to split the data and trust in such a way that they can recover from independent failures. In particular, SS2 can recover from failures beyond the capabilities of typical replication and erasure coding schemes. By building recursive links between documents through data entanglement, the secure data archive can detect and repair targeted censoring attempts. In the rest of this document we present RECAST, the final prototype version of the secure data archive. More precisely, we describe how long-term protection (D2.2) and short-term protection (D2.5) are blended together in a final implementation built on top of playcloud (D2.2). This document serves as a user manual for the deployment and experimentation of RECAST. Parts of the codebase are publicly available at https://github.com/safecloud-project.

This document is organized in two parts. In Section 1, we summarize STEP-archives introduced first in D2.2 and extended in D2.5. From Section 2 and on, we present a user-oriented manual describing how to deploy, configure and manage an instance of RECAST.

Secure storage		
SS1 **Secure block storage**	**SS2** **Secure data archive**	**SS3** **Secure file system**

Table 1: secure storage solutions.

1 SafeCloud archival using data entanglement

As part of SafeCloud, we introduce STEP-archives, a storage system for archiving coded documents. STEP-archives were first presented in [MAL16], extensively discussed in D2.2, further discussed in D7.10 and finally extended in D2.5. We summarize them here for completeness. Using data entanglement and erasure-correcting codes, we develop a data storage architecture where a stored document can only be deleted or modified by compromising the integrity of other documents in the system.

There are two main objectives behind this work. The first objective is data integrity. We want to provide guarantees to users that their data cannot be deleted or corrupted without compromising other data stored by themselves or other users. The second objective is to provide censorship resistance by forcing a censor who wants to tamper with data to do so noisily, i.e., being forced to corrupt a large number of other documents in the system. An ancillary result deriving from the two objectives is increased protection against failures, which can be seen as attacks from random or failure-specific censors.

Definition 1. A (s,t,e,p)-archive is a storage system where each archived document consists of a codeword with s source blocks, t tangled blocks, p parity blocks and that can correct $e = p - s$ block erasures.

When a document is archived, it is split into $s \geq 1$ source blocks. Using the s source blocks with t distinct old blocks already archived, a systematic maximum distance separable (MDS) code [LC04] is used to create $p \geq s$ parity blocks which are then archived on the system.

An archived document can be recovered from $s + t$ or more of its blocks. The code can correct p block erasures per document codeword, but since the source blocks are not archived and are considered as erased, at most $e = p - s$ block erasures per document on the storage medium can be corrected. Note that increasing t does not increase storage utilization overhead or error-correcting capability but does increase coding and decoding complexity.

An attacker can censor a document d_k by erasing more than e of its blocks. However, by entangling new documents with documents already archived, it might be possible for the system to recover the deleted blocks by recursively decoding other documents that use them.

The challenging part of our approach is thus to choose the pointers to entangled blocks. In D2.5 we improved upon D2.2 by combining uniform random selection and normal distribution-based selection centred on the tail of the archive. As a result, we maintained randomness in the structure preventing the attacker from planning the attack in advance while insuring a short-term protection of the newer documents through replication.

For detailed information on STEP -archives, please refer to D2.2 and D2.5.

1.1 To delete or not to delete

The conflict between legitimate and illegitimate deletion is always present: if it is easy to delete files for a legitimate reason, then the protection offered by entangled data is lost and illegitimate tampering becomes easier. This is unavoidable, and in this sense similar to distributed ledgers such as blockchains: tampering, including deletion, must be very difficult by design. Throughout the project, we studied mechanisms to legitimately delete entangled files from a STEP-archive. The conclusion is that although deletion mechanisms are indeed possible, they require major design changes, are technically expensive, and always decrease the anti-tampering properties of a STEP-archive. This undesirable tradeoff was not worth the benefits of an implementation, thus we decided not to implement any of these deletion techniques. This clearly signals once again that our STEP-archives should only be used to store immutable data.

2 Architecture

2.1 Design

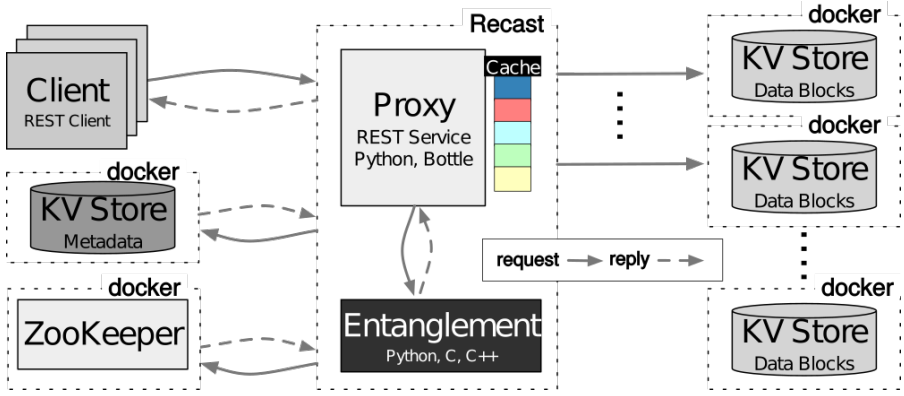

Figure 2 The architecture of RECAST.

Our implementation of the step-archive, named RECAST, is based on playcloud (introduced in D2.2 and used for experimental evaluation in D2.5). As such its architecture is similar to playcloud's with the addition of a few components for metadata management and coordination. The main components of the system are: a proxy/encoder, a metadata server, a coordination service and one or more storage nodes.

The proxy mediates interactions between clients and the system communicating over HTTP through a REST interface. Users may insert new documents using a PUT request and retrieve them by issuing at GET request.

The encoder entangles the new documents with older ones already present in the system. Note that depending on the use cases and resources available, the proxy can take care of the encoding without delegating to a dedicated encoder instance.

The metadata server keeps track of the stored document. The information helps looking up blocks and checking their integrity as they are fetched from the storage nodes. In practice, the metadata is stored in a centralized key-value store.

The coordination service enables separate processes such as the proxy and the repair daemon to operate simultaneously in the same live instance.

Finally, storage nodes serve as backends. They are considered independent and leveraged by the proxy to balance the reading and writing load evenly (as the blocks are placed randomly).

2.2 Implementation

The actual implementation of RECAST is a combination of existing pieces of software and new code. The following paragraph is a brief presentation of the software used.

The proxy listens through a REST API written in Python(2.7) [python] exposed by uwsgi 2.0.15 [uwsgi] application server. The entanglement is implemented using PyEClib[pyec] and liberasurecode[libec] and Intel ISA-L [isa]. Communication between separate proxy and encoder instances is done using grpc[grpc].

The metadata server is a Redis[redis] server, an in-memory key-value store, whose writes are logged on disk. The coordination service is Zookeeper[zkp]. Finally, storage nodes can either be redis or minio[min] nodes.

Each component is packaged as a separate docker [docker] container and can be deployed either locally, using docker-compose [dc] or on multiple machines using docker swarm mode [ds] (see Deployment section).

3 Deployment

This section details the deployment procedure for RECAST. As RECAST mostly relies on docker for ease of build and deployment, please make sure that you have the following requirements installed on your machine.

3.1 Local deployment

A local deployment only requires docker[docker] and docker-compose[dc]. This process follows 2 steps:

- Build the docker images
- Deploy locally

3.1.1 Build the docker images

At the top of the project, make sure file docker-compose.yml is present and then run:

```
docker-compose build
```

This command will build all the images that are not pre-built in the dockerhub. At the end of the process you should be ready to deploy recast.

3.1.2 Deploy locally

Having made sure that the images were built properly, run

```
docker-compose up --detach=true
```

You should now have a running instance of RECAST listening on port 3000. You can interact with this instance by trying to upload or read files as described in the basic usage section.
To stop your instance, run:

```
docker-compose down
```

3.2 Distributed deployment

RECAST can be deployed in a distributed mode where the components are spread over multiple machines. To deploy to multiple machines, RECAST uses docker and docker-swarm [ds]. A typical deployment follows 3 steps:

- Create a docker swarm
- Build, push and download the docker images
- Deploy on the swarm

3.2.1 Create a docker swarm

First ensure that all the machines that are to be added to the cluster have the docker-engine installed. One of these machines is to be chosen as the leader of the swarm. On that machine, run:

```
docker-swarm init
```

Take good note of the leader token given as output of the command as it is going to be used by the other machines to join the cluster. On the other machines, run docker-swarm join with the token to create the swarm.

```
docker swarm join <leader-token-here>
```

By the end of the process, the swarm is assembled and ready to move on image building and deployment.

3.2.2 Build, push and download the docker images

In order run in swarm mode, the docker images used by RECAST must be built and then pushed to a docker registry. More specifically, the proxy image must be pushed for RECAST to work. For this process, make sure that you have an account on the docker hub.
First, build the image:

```
docker build --file pyproxy/Dockerfile --tag <your-docker-
username>/proxy
```

Enter your docker credentials if you have not yet been authenticated.

```
docker login
```

Then push the image to the docker hub.

```
docker push <your-docker-username>/proxy
```

Finally, edit the *docker-compose-production.yml* to replace the username prefix in the proxy service.

```
…
version: "3"
services:
    proxy:
        …
        image: <your-docker-username>/proxy
        …
```

Figure 3 Section to edit in *docker-compoe-production.yml*.

3.2.3 Deploy on the swarm

After successfully building and pushing the docker image to the docker hub, it is time to get RECAST running. At the top of the project, run:

```
docker stack deploy --compose-file docker-compose-production.yml recast
```

While you will get control of your console prompt quickly after running the command above, the complete deployment of RECAST might take some time. Indeed, on the first deployment over the swarm the images required to the container must be pulled by the machines running the matching service.

4 Configuration

RECAST can be launched without changes to the original configuration but the default settings may not suit all use cases. Further fine-tuning may be needed to match the desired configuration. In particular, RECAST's settings can be tweaked along two axes:

1. Entanglement parameters
2. Storage parameters

The tweaked parameters ultimately determine the storage overhead (in terms of disk utilization) to expect when launching a RECAST instance:

$$storage\ overhead = \frac{p}{s} * replication\ factor$$

The system operator must therefore be careful that the chosen RECAST configuration matches the resources available.

4.1 Entanglement parameters

RECAST implements the (s,t,e,p)-archive scheme and lets the operator set the coding parameters. The number of source blocks (s), of pointers (t) and of parities (p) can be tuned by editing the content of *configuration.json* (at the top of the project) in the entanglement section.

```
{
  …
  "entanglement": {
    "type": "step",
    "configuration": {
      "s": 1,
      "t": 10,
      "p": 3
    }
  }
  …
}
```

Figure 4 Content to modify in *configuration.json*.

The file *configuration.json* serves as the central configuration file. To spread the changes to other configuration files, the operator needs to run the following command from the top of the project.

```
./scripts/configure.py
```

The configuration changes should now have spread to the other configuration files: *pyproxy/dispatcher.json* and *pyproxy/pycoder.cfg*.

```
{
…
    "entanglement": {
        "configuration": {
            "p": 3,
            "s": 1,
            "t": 10
        },
        "type": "step",
…
}
```

```
[main]
…
splitter = entanglement
…
[entanglement]
type = step
…
[step]
s = 1
t = 10
p = 3
```

Figure 5 Excerpts of *pyproxy/dispatcher.json* and *pyproxy/pycoder.cfg* after changes in entanglement configuration.

4.2 Storage parameters

For local or distributed experimentation, the number and type of storage nodes (minio [min] or redis[redis]) can be specified in *configuration.json*. You can also change the replication factor, which will impact the storage overhead before replica management.

```
…
  "storage": {
    "nodes": 42,
    "type": "redis",
    "replication_factor": 2
  },
…
```

The change can then be propagated to the other configuration files by running:

```
scripts/configure.py
```

This will result in a matching change in files *docker-compose.yml*, *docker-compose-production.yml* and *pyproxy/dispatcher.json*.

```
…
storage-node-42:
  image: redis:3.2.8
  volumes:
    - ./volumes/storage-node-
42/:/data/
   container_name: storage-node-42
…
```

```
…
storage-node-42:
  image: redis:3.2.8
  deploy:
    placement:
      constraints:
        - node.role == worker
…
```

```
    …
    "replication_factor": 2,
    "providers": {
        …
        "storage-node-42": {
            "host": "storage-node-42",
            "type": "redis",
            "port": 6379
        },
        …,
     },
     …
```

Figure 6 Excerpts of *docker-compose.yml*, *docker-compose-production.yml* and *pyproxy/dispatcher.json* after changes in storage configuration.

Please note that *pyproxy/dispatcher.json* is the actual configuration file read by RECAST at startup time (*configuration.json* is ignored at runtime and its changes must be spread by running *./scripts/configure.py* beforehand). In consequence, *pyproxy/dispatcher.json* can be manually configured to connect to other storage backends that may not be containerized, deployed locally or even part of the swarm. The operator should take care of deleting the unnecessary containers from *docker-compose.yml* and *docker-compose-production.yml* before starting RECAST when their storage backends are not managed by docker.

5 Basic usage

Users can interact with a RECAST instance through the REST interface using an HTTP client such as curl [curl] or any other software library of their choice.

5.1 Insert a document

Clients can upload files to the archive using a simple command.

```
curl --request PUT http://proxy-ip:3000/name --upload-file path/to/file
```

In this case, the client uploads the document located at *path/to/file* under the key *name*. On a successful request, RECAST replies to the request with the chosen document name. In case of the failure RECAST replies with the appropriate HTTP error code (400 on empty requests or 409 when trying to overwrite a document).
In the case where a document cannot be stored under a user-prefered name, the client can choose to upload without a filename and let RECAST pick one for them.

```
curl --request PUT http://proxy-ip:3000/ --upload-file path/to/file
```

RECAST will then pick a random UUIDv4 as the name and send it in the reply to the client.

5.2 Read a document

Once uploaded to RECAST, files can be read using the filename picked by the client or the user at upload time.

```
curl --request GET http://proxy-ip:3000/name --output path/to/file
```

With this command, a user can recover a copy of a document stored in RECAST under the key *name* in *path/to/file*.

```
curl --request GET http://proxy-ip:3000/name/__meta
```

If a user is only interested in reading metadata about a given document, they can issue a request on the document and get result as a JSON object detailing information about the document, its blocks and the pointers used for entanglement.

5.3 Repair a document

In case of loss of one or several blocks, the repair can be operated by attaching to a proxy instance and running the following command:

```
docker exec --interactive --tty proxy ./repair.py <block ids>
```

When auditing the entire archive to check for corruption of blocks, run:

```
docker exec --interactive --tty proxy ./repair.py
```

6 Replica management

The right selection of pointers for entanglement is paramount to long-term protection of documents in the archive. D2.5 described how to achieve better selection results using a mixed approach by picking one part of the pointers over the entire archive and the other in a sliding window over the most recent documents that entered the archive (aka the tail). In addition, we proposed a similar window-based approach to provide short-term protection to documents in the tail of the archive through replication. To deal with the storage overhead of this replication (see Section 4), our implementation provides two strategies for replica management: reference-counting and window-based. The rest of this section describes these strategies and their practical uses.

6.1 Replica management by reference counting

A new document entering the archive is entangled and split into blocks. These blocks are immediately replicated, and the copies are spread across the storage nodes for redundancy. From there on, the blocks can be used as pointers for the entanglement of new documents. Thus, the reference count, or the number of times a block has been selected as a pointer, is bound to increase over time. As this reference count also informs us on the possibility of recursive reconstruction in case of failure, we can leverage it to decide how to deal with the explosion in storage overhead.
Using a threshold value *th*, defined by the system operator, the replica management script can crawl the metadata looking for blocks that have been pointed at *th* or more times and delete their copies to lower the storage overhead across the storage nodes.

This strategy guarantees recovery of all blocks in case of block loss but requires expert fine-tuning. Indeed, poor settings can lead to an archive where the storage overhead is not mitigated as best as possible or worse where some documents cannot be recovered in case of failure.

To run the replica management script using a reference-counting base in a RECAST instance on Docker, run:

```
docker exec --interactive --tty  proxy ./scrub.py --pointers <th>
```

Or as a daemon, running at fixed intervals:

```
docker run --interactive --tty  proxy --entrypoint ./scrub.py --pointers <th> \
                                                              --interval <seconds>
```

6.2 Replica management by sliding window

A new document entering the archive is entangled and split into blocks. As a recent document, it is still part of the tail of the archive made of the *w* most recent documents in the archive. In a system that uses mixed pointer selection, and is thus biased towards documents in the tail, we can assume that as documents exit the window, they can have been used as pointers and copies of their blocks can be discarded (see D2.5 for more details).

The sliding window strategy offers a way to maintain a predictable storage overhead and even guarantees that the size occupied by replicas decreases over time relative to the size of the archive. But it does so with little regard to the actual recoverability of blocks themselves. Indeed, a document could exit the window without all his blocks being pointed at least once.

To run the replica management script using a sliding window base in a RECAST instance on Docker, run:

```
docker exec --interactive –-tty  proxy ./scrub.py  --window <w>
```

Or as a daemon, running at fixed intervals:

```
docker run --interactive --tty  proxy --entrypoint ./scrub.py --window <w> \
                                                              --interval <seconds>
```

7 Metadata management

Both long-term and short-term protection schemes require the maintenance of metadata by the system operator. Random or mixed entanglement imply keeping a list of pointers and blocks location. Replication management can only be performed when copies of blocks can be identified and located. In our implementation, a centralized metadata server is set up to keep track of all this information. Please note that a centralized metadata server is only necessary because block placement is random. If the blocks were placed in a deterministic fashion, we could remove this component. However as (s,t,e,p)-archives build upon randomness to prevent pre-computed attacks, we have chosen not to implement such a block placement model.
In the rest of this section, we describe metadata management in our prototype, its strength and limitations, and how they can be mitigated.

For operational purposes, the secure archive needs to keep track of blocks' location and the entanglement graph. This metadata is maintained in a Redis [redis] database and serves as the source of truth for the system. Components of the system that need to read or write metadata interact with the database using a Python [python] client. Such components include the proxy or the replica management and block repair scripts.

7.1 Metadata storage overhead

In terms of growth, the storage overhead is predictable. Variations can be observed depending on the configuration of STeP but storing 1 document adds around 1kB to the metadata server regardless of the size of the documents stored in the archive.

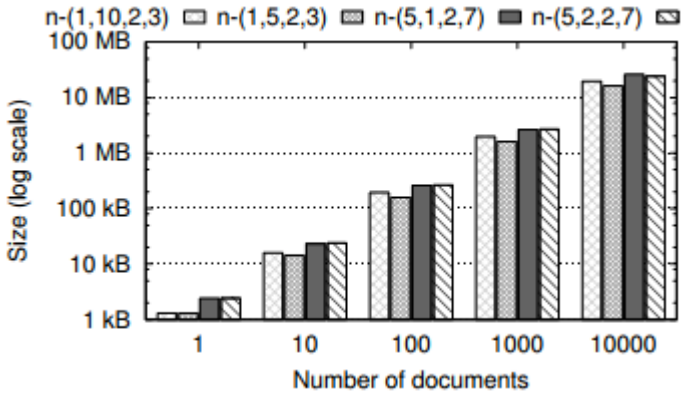

Figure 7 Metadata storage overhead with an increasing number of documents in the archive with different STeP configurations.

The storage overhead of the metadata can further be reduced through several means. Better normalization would reduce the overall size by avoiding repetition in document entries. This would be achievable at the cost of longer and more complex queries.

7.2 Metadata reconstruction

Our system leverages randomness in the selection of tangled blocks and in the placement of new blocks into storage nodes. If metadata is unavailable, damaged or lost, stored blocks become meaningless. To mitigate this issue, we implement a mechanism to reduce risks of complete loss of access to the metadata server. This procedure enables the system to reconstruct the metadata from the data itself. Under the assumptions of

available and honest storage nodes as well as pristine data blocks, we scan the storage nodes, examining the hosted blocks and reconstruct the associated metadata. This solution is possible because we prepend the entanglement information to each block before sending it to the storage nodes. More specifically, given a block, this metadata-overhead includes a reference to all the pointers selected during the entanglement of the document. In our prototype, it consists of a fixed 80 Bytes per block (erasure coding information) and an additional t * (average block name length). As blocks are named according to the document they belong to and their position in the codeword, this average length depends on the naming patterns in the archive.

Figure 7 shows the availability of documents during metadata reconstruction in a 16 nodes archive with different configurations and replication factors. If all blocks are replicated, we can expect to be able to serve all documents before reading from all the storage nodes as depicted by the full points. In contrast, if none of the blocks are replicated, many storage nodes will have to be crawled through until we gather enough knowledge about the entire system. When running an instance of our system with mixed entanglement and replication management enabled, the number of replicated blocks is constant as the archive grows. Following this principle, the number of nodes that need to be explored to rebuild a functional archive increases.

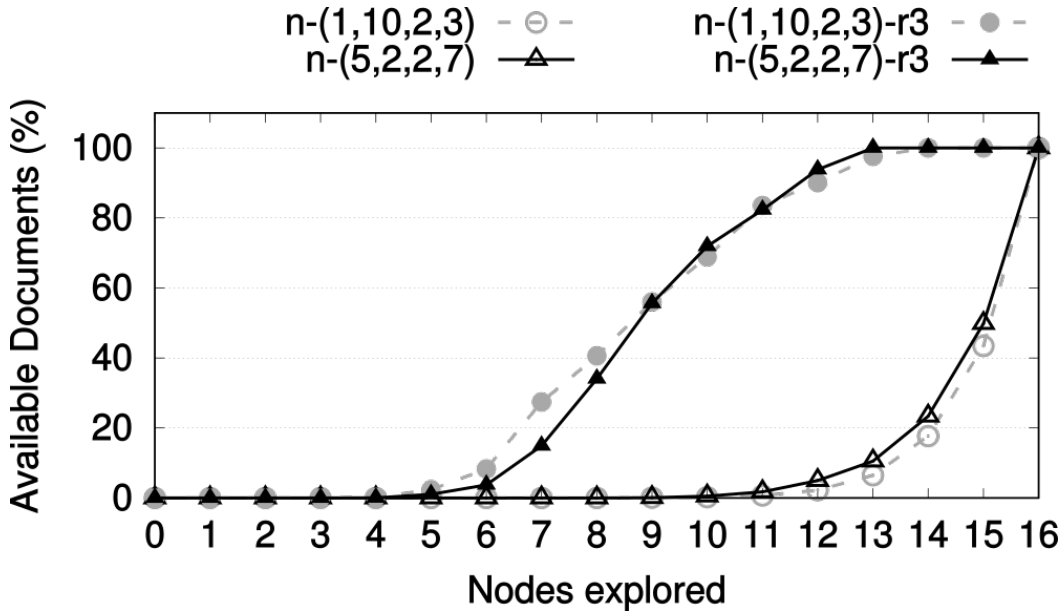

Figure 8 Availability of documents during metadata reconstruction in a 16 nodes archive with different STeP configurations and replication factors.

To rebuild the metadata after a loss of the metadata server, some information about the RECAST instance must be gathered. The hostname (or IP address) and the port number used by the metadata server as well as the path to *dispatcher.json*. If you have a running RECAST using Docker, attach to the proxy container and run the following command:

```
docker exec --interactive --tty proxy ./rebuild_metadata.py --host
<host> --port <port> --conf </path/to/dispatcher>
```

8 Summary

This deliverable presents the final implementation of the secure block device which leverages the system introduced in D2.2 and extended to protect the whole archive in D2.5. Re-introducing (s,t,e,p)-archives as the cornerstone of this prototype, it covers the architecture of the final system, RECAST, and presents a user-oriented manual for its deployment, use and maintenance.

9 References

[curl] https://curl.haxx.se/

[docker] https://www.docker.com/

[dc] https://docs.docker.com/compose/

[ds] https://docs.docker.com/engine/swarm/

[grpc] http://www.grpc.io/

[isa] https://github.com/01org/isa-l

[LC04] Shu Lin and Daniel J. Costello. Error Control Coding. Paerson Prentice Hall, second edition, 2004.

[libec] https://github.com/openstack/liberasurecode/

[MAL16] Hugues Mercier, Maxime Augier and Arjen K. Lenstra, STEP-archival: Storage Integrity and Tamper Resistance using Data Entanglement, Submitted to the IEEE Transactions on Information Theory, 2016 (revised 2017). Available upon request.

[min] https://minio.io/

[pyec] https://github.com/openstack/pyeclib

[python] https://www.python.org/

[redis] https://redis.io

[uwsgi] https://github.com/unbit/uwsgi

[zkp] https://zookeeper.apache.org/