Founsure Software User Guide (version 1.0)

Suayb S. Arslan (arslans@mef.edu.tr)
http://www.suaybarslan.com

https://github.com/suaybarslan/founsure
August 14, 2020

Introduction

Founsure 1.0 is a data coding and erasure/loss correction software library. Founsure 1.0 can be used as a standalone data protection tool or an integral part of software-defined distributed data storage software to achieve extended periods of data retention, available for download at https://github.com/suaybarslan/founsure [1]. You will need a Linux Runtime Environment and a C compiler ($\geq$ 4.8.4)[1] to run it. This user manual is prepared to guide you through its installation as well as the unit, performance and main functionality tests of the software library.

Contents

1 **Installation** 2

2 **Testing Correct Installation and Help Menus** 2

3 **Main Functionality and Performance** 4

3.1 Functions 4

3.2 Parameters of the System 4

3.3 Most typical example use cases 4

3.4 Average Performance tests 5

4 **Unittests for Software Functionality checks** 6

[1] https://gcc.gnu.org/

1 Installation

1. Currently, we only support installation from the source and for different Linux distros (tested on Ubuntu and Centos). Please first download the software from its github home page. There are three main directories of the source code. The rest of folders are created automatically due to `libtool`.
   - The **src** directory contains the founsure source code.
   - The **test** directory contains the some performance test as well as unittest programs.
   - The **man** page for easy guide for the entire founsure software.
2. First, we recommend you to switch to a root account to eliminate any potential file access problems (`sudo -s` (centos)). You also may have to install `autotools` for re-configuration. Please use
   - `sudo yum install autotools autoconf` (for centos),
   - `sudo apt-get install autotools-dev autoconf` (for ubuntu)
3. Make sure you run autreconf before you run the usual installation procedure outline below.
   - **autoreconf –force –install**
   - **./configure**
   - **make**
   - **sudo make install**
4. This will install the library into your machine's `lib` directory, most probably `/usr/local/lib`.
5. The configuration process assumes shared objects are searched for in `/usr/local/lib`. If this is not the case on your system, you can specify a search path at configuration time. Alternatively if you receive the following error when you try to execute one of the main functions of the library to do encoding or decoding, setting the `LD_LIBRARY_PATH` to installed directory (For instance `/usr/local/lib`) and exporting it might help.

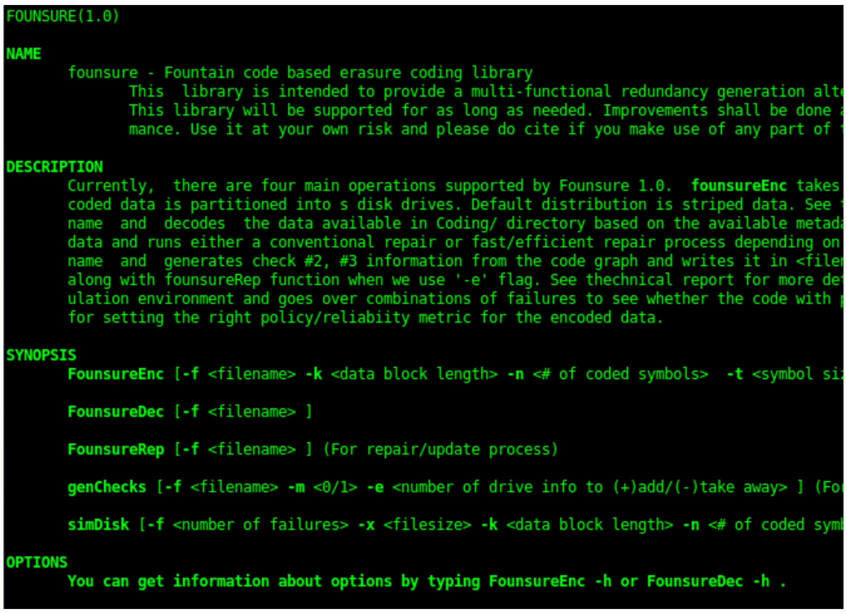
```
error while loading shared libraries: libfounsure.so.2: cannot open shared object file: No such file or directory.
```

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/lib
```

2 Testing Correct Installation and Help Menus

1. Easiest way to test Founsure 1.0 installation is to reach the help documentation installed with the main functionalities of the software library. There are two ways to utilize help options.

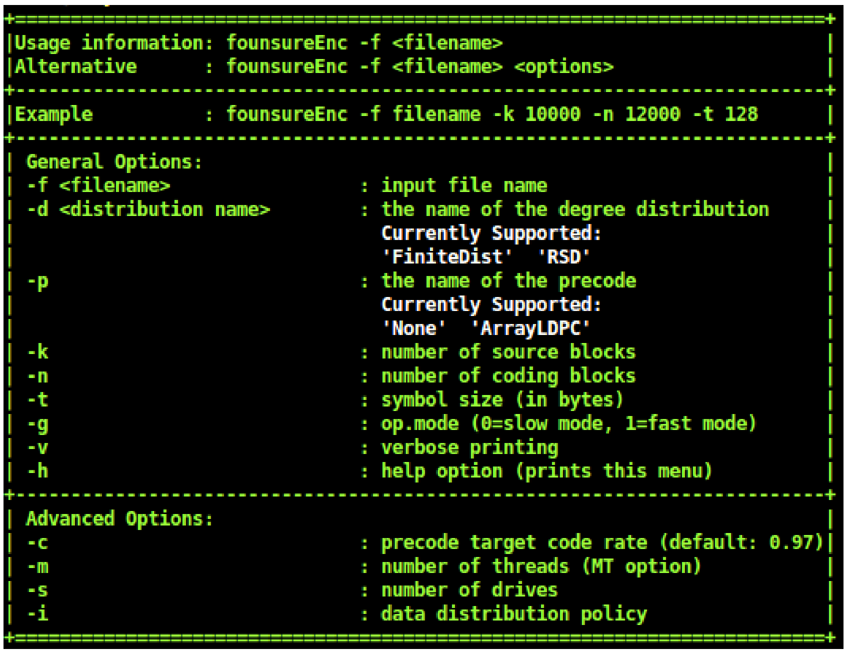
- `man founsure` for man pages. Founsure 1.0 man pages include short description, synopsis and various options associated with the main functionality. See (Figure 5).



```
FOUNSURE(1.0)

NAME
       founsure - Fountain code based erasure coding library
              This  library is intended to provide a multi-functional redundancy generation alte
              This library will be supported for as long as needed. Improvements shall be done a
              mance. Use it at your own risk and please do cite if you make use of any part of t

DESCRIPTION
       Currently,  there are four main operations supported by Founsure 1.0.  founsureEnc takes
       coded data is partitioned into s disk drives. Default distribution is striped data. See t
       name  and  decodes  the data available in Coding/ directory based on the available metada
       data and runs either a conventional repair or fast/efficient repair process depending on
       name  and  generates check #2, #3 information from the code graph and writes it in <filen
       along with founsureRep function when we use '-e' flag. See thechnical report for more det
       ulation environment and goes over combinations of failures to see whether the code with p
       for setting the right policy/reliabiity metric for the encoded data.

SYNOPSIS
       FounsureEnc [-f <filename> -k <data block length> -n <# of coded symbols>  -t <symbol siz

       FounsureDec [-f <filename> ]

       FounsureRep [-f <filename> ] (For repair/update process)

       genChecks [-f <filename> -m <0/1> -e <number of drive info to (+)add/(-)take away> ] (For

       simDisk [-f <number of failures> -x <filesize> -k <data block length> -n <# of coded symb

OPTIONS
       You can get information about options by typing FounsureEnc -h or FounsureDec -h .
```

Figure 1: Man pages

- `founsureEnc -h` for encoding, `founsureDec -h` for decoding and `founsureRep -h` for repair.



```
+=======================================================================+
|Usage information: founsureEnc -f <filename>                           |
|Alternative      : founsureEnc -f <filename> <options>                 |
+-----------------------------------------------------------------------+
|Example          : founsureEnc -f filename -k 10000 -n 12000 -t 128    |
+-----------------------------------------------------------------------+
| General Options:                                                      |
| -f <filename>                : input file name                        |
| -d <distribution name>       : the name of the degree distribution    |
|                                Currently Supported:                   |
|                                'FiniteDist'  'RSD'                    |
| -p                           : the name of the precode                |
|                                Currently Supported:                   |
|                                'None'  'ArrayLDPC'                    |
| -k                           : number of source blocks                |
| -n                           : number of coding blocks                |
| -t                           : symbol size (in bytes)                 |
| -g                           : op.mode (0=slow mode, 1=fast mode)     |
| -v                           : verbose printing                       |
| -h                           : help option (prints this menu)         |
+-----------------------------------------------------------------------+
| Advanced Options:                                                     |
| -c                           : precode target code rate (default: 0.97)|
| -m                           : number of threads (MT option)          |
| -s                           : number of drives                       |
| -i                           : data distribution policy               |
+=======================================================================+
```

Figure 2: Founsure Encoder Help Menu

3 Main Functionality and Performance

3.1 Functions

Founsure 1.0 has the following three executable main components that achieve four important functionalities

- **founsureEnc**: Encoder engine that generates redundancy under a local `Coding` directory and a metadata file that include information about the file and the coding parameters including the seed information.
- **founsureDec**: Decoder engine that requires a local `Coding` directory with enough number of files, a valid file name and an associated metadata file to decode user data.
- **founsureRep**: Data repair engine that also requires a `Coding` directory with enough number of data chunks to either

  – fix/repair one or more data chunks should they have been erased, lost or flagged as unavailable.

  – or generate extra coding blocks should a code update has been requested.

Founsure 1.0 currently supports two utility functions as listed below.

- **simDisk**: This function is used to exhaust all possible combinations of disk failures and code's tolerance to these failures for a given set of coding parameters.
- **genChecks**: This utility function is crucial for two different important functionalities: (1) fast/efficient repair/rebuild of data and (2) seemless on-the-fly update.

3.2 Parameters of the System

Parameters of the system are listed and hard coded in the file

```
/src/include/parameter.h
```

If you change any of the defined variables, make sure to save them from the menu File » Save (assuming a GUI-based editor) or pressing Ctrl + S and re-compile and install the library again. The parameters in this file are mostly critical, mathematically essential and needs to change rarely for the experts only. For more details about the definitions of these parameters, we kindly refer the reader to `https://arxiv.org/pdf/1702.07409.pdf`.

3.3 Most typical example use cases

Here we provide four dependent most typical use cases of the main library functions. The selection of code parameters can be selected differently as you please. From these examples, you can realize with "v" flag, we can read off the pure speed as well as the total time of execution for each function.

`Enocode Example` The following command will encode a test file `testfile.txt` with $k = 500$ data chunks with each chunk occupying $t = 512$ bytes. The encoder generates $n = 1000$ coding chunks using $d =$'FiniteDist' degree distribution and $p =$'ArrayLDPC' precoding. Finally, generated chunks are striped/written to $s = 10$ distinct files for default disk/drive allocation under `Coding` directory.

```
foonsureEnc -f testfile.txt -k 500 -n 1000 -t 512 -d
'FiniteDist' -p 'ArrayLDPC' -s 10 -v
```

`Decode Example` Let us erase one of the coding chunks (`0007`) and run Founsure decoder. The decoder shall generate a decoded file `testfile_decoded.txt` under `Coding` directory. You can use "diff" command to compare this file with the original.

```
rm -rf Coding/testfile_disk0007.txt
founsureDec -f testfile.txt -v
diff testfile.txt Coding/testfile_decoded.txt
```

`Repair Example` The erased chunk (`0007`) can efficiently be repaired using three different types of check relations. First, we need to generate these checks then call repair function to repair the erased chunk.

```
genChecks -f testfile.txt -m 1 -v
founsureRep -f testfile.txt -v
```

`Update Example` Let us assume we want to generate two more chunks (in addition to 10) without re-encoding the entire data. In that case, we need to update the checks (add extra checks on top of the existing ones) and generate two more chunks accordingly.

```
genChecks -f testfile -m 1 -v -e 2
founsureRep -f testfile.txt -v
```

3.4 Average Performance tests

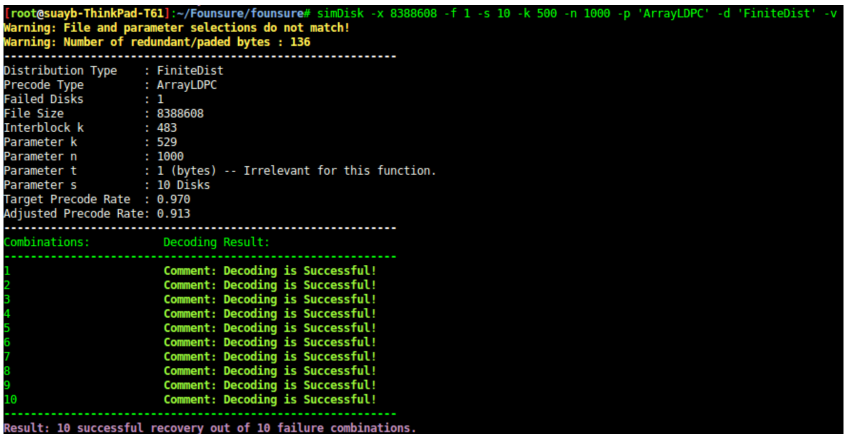
1. Since Founsure 1.0 library is based on fountain codes (non-MDS), it is not straightforward to know how many chunk losses the code can tolerate (especially for small k and n). To be able to test the tolerance to different patterns of chunk losses, we have implemented the utility function `simDisk`. A python version of this function (`perfsim.py`) is also included for those interested.

```
simDisk -x 8388608 -f 1 -s 10 -k 500 -n 1000 -p 'ArrayLDPC'
-d 'FiniteDist' -v
```

2. In Founsure 1.0 , when we run repair function with "`-v`" flag we can monitor the number bytes needed to resurrect a lost data chunk. Depending on the pattern of lost chunks this number may change. We provide a python script (`avgbw.py`) that goes through all `m` lost chunk combinations (out of `n`) and reports the average, standard

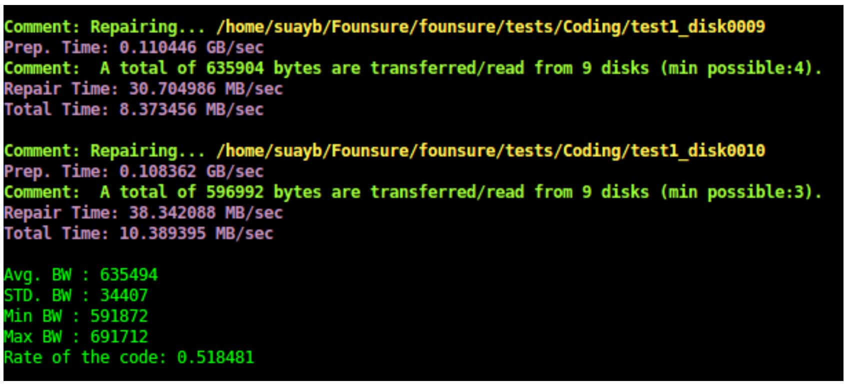
deviation, minimum and maximum values of the required number of bytes for each repair case.

```
python avgbw.py
```



```
[root@suayb-ThinkPad-T61]:~/Founsure/founsure# simDisk -x 8388608 -f 1 -s 10 -k 500 -n 1000 -p 'ArrayLDPC' -d 'FiniteDist' -v
Warning: File and parameter selections do not match!
Warning: Number of redundant/paded bytes : 136
-----------------------------------------------------------
Distribution Type    : FiniteDist
Precode Type         : ArrayLDPC
Failed Disks         : 1
File Size            : 8388608
Interblock k         : 483
Parameter k          : 529
Parameter n          : 1000
Parameter t          : 1 (bytes) -- Irrelevant for this function.
Parameter s          : 10 Disks
Target Precode Rate  : 0.970
Adjusted Precode Rate: 0.913
-----------------------------------------------------------
Combinations:            Decoding Result:
-----------------------------------------------------------
1                        Comment: Decoding is Successful!
2                        Comment: Decoding is Successful!
3                        Comment: Decoding is Successful!
4                        Comment: Decoding is Successful!
5                        Comment: Decoding is Successful!
6                        Comment: Decoding is Successful!
7                        Comment: Decoding is Successful!
8                        Comment: Decoding is Successful!
9                        Comment: Decoding is Successful!
10                       Comment: Decoding is Successful!
-----------------------------------------------------------
Result: 10 successful recovery out of 10 failure combinations.
```

Figure 3: An example run of `simDisk` for one lost chunk.



```
Comment: Repairing... /home/suayb/Founsure/founsure/tests/Coding/test1_disk0009
Prep. Time: 0.110446 GB/sec
Comment:  A total of 635904 bytes are transferred/read from 9 disks (min possible:4).
Repair Time: 30.704986 MB/sec
Total Time: 8.373456 MB/sec

Comment: Repairing... /home/suayb/Founsure/founsure/tests/Coding/test1_disk0010
Prep. Time: 0.108362 GB/sec
Comment:  A total of 596992 bytes are transferred/read from 9 disks (min possible:3).
Repair Time: 38.342088 MB/sec
Total Time: 10.389395 MB/sec

Avg. BW : 635494
STD. BW : 34407
Min BW : 591872
Max BW : 691712
Rate of the code: 0.518481
```

Figure 4: An example run of `avgbw.py` for 1 lost chunk out of 10 for `k=2000` and `n=3950`. See the default parameter selections in `avgbw.py`.

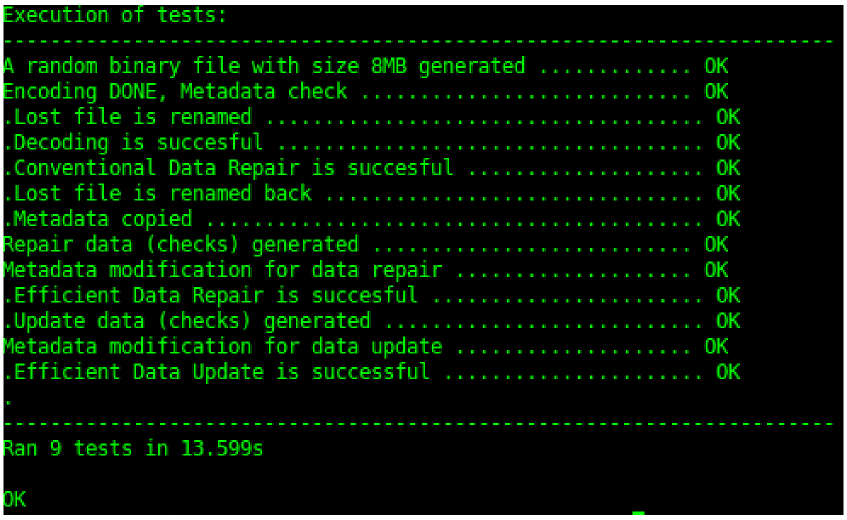
4 Unittests for Software Functionality checks

For automated (and more detailed functionality) testing, we provide a self-contained unittest which does attempt the following set of jobs, check if the system successfully completes these jobs and report either failure or success. Code parameter selections are made (hard coded) inside the unittest file. Different unittests can be generated by changing the code parameters inside this file.

1. A binary file of size `fsize` is generated.

2. A standard encoding function with default code parameters is executed. The process as well as the outputs (including metadata) is checked for accuracy.

3. A file chunk is renamed to simulate loss.
4. Decoder is executed and the output (decoded file) is compared with the original file for consistency check.
5. The lost file is renamed back to its original name.
6. Data for efficient repair is generated (`genCheck`) and output is checked for accuracy. Furthermore, metadata is checked if necessary modifications are made by the function `genCheck`.
7. Data loss is resimulated through renaming. Repair function is invoked to complete repair. Repaired data chunk is compared with the original to check exact repair accuracy.
8. Data for efficient update is generated (`genCheck`) and output is checked for accuracy. Furthermore, metadata is checked if necessary modifications are made by the function `genCheck`.
9. Repair function is invoked to complete update. Updated data chunks are checked if they would be originally generated by the encoder with the modified metadata.



```
Execution of tests:
----------------------------------------------------------------------
A random binary file with size 8MB generated ............ OK
Encoding DONE, Metadata check ........................... OK
.Lost file is renamed .................................... OK
.Decoding is succesful ................................... OK
.Conventional Data Repair is succesful ................... OK
.Lost file is renamed back ............................... OK
.Metadata copied ......................................... OK
Repair data (checks) generated .......................... OK
Metadata modification for data repair ................... OK
.Efficient Data Repair is succesful ...................... OK
.Update data (checks) generated .......................... OK
Metadata modification for data update ................... OK
.Efficient Data Update is successful ..................... OK
.
----------------------------------------------------------------------
Ran 9 tests in 13.599s

OK
```

Figure 5: An example run of `founsure_unittest.py` for one lost chunk.

References

[1] Suayb S. Arslan. Founsure. `https://github.com/suaybarslan/founsure`, 2020.