Report for Assignment 4

SOFTWARE ENGINEERING FUNDAMENTALS

Group 2

Aron Strandberg - Eysteinn Gunnlaugsson - Robert Kindwall - Karl Kvarnfors - Erik Comstedt

Project Description

Name: Paramiko
It's a module for Python 2.7/3.4+ that implements the SSH2 protocol for secure (encrypted and authenticated) connections to remote machines.

URL: `https://github.com/paramiko/paramiko`

Complexity

1. What are your results for the ten most complex functions? Did all tools/methods get the same result? Are the results clear?

The obtained results can be seen in Table 1. We only tried one tool, Lizard, which was recommended in the assignment. For most functions, we got the same result when hand-counting as Lizard. In the cases where we got different answers, we suspect the tool is right as our own hand-counted results were different in these cases as well. We believe the discrepancy is due to calculating errors and different treatment of certain types of statements such as try/catch, assertions and other less clear-cut cases.

CCN (Lizard)	Length (NLOC)	Name	File
35	95	`_process`	`sftp_server.py`
35	69	`run`	`transport.py`
31	73	`readline`	`file.py`
28	110	`_parse_userauth_request`	`auth_handler.py`
21	142	`_handle_request`	`channel.py`
20	144	`_parse_service_accept`	`auth_handler.py`
19	108	`_parse_channel_open`	`transport.py`
17	142	`read_message`	`packet.py`
16	120	`_parse_signing_key_data`	`ed25519key.py`
16	74	`_parse_kex_init`	`transport.py`

Table 1: Ten most complex functions

2. Are the functions just complex, or also long?

We consider all our listed functions to be long. The smallest one is 69 non-comment lines of code, and the longest 144.

3. What is the purpose of the functions?

- `_process` in `sfpt_server.py`
  This method is undocumented, but based on code inspection, its purpose is to handle various SFTP requests. As there are a lot of different types of SFTP requests, it is natural that the method has a quite high cyclomatic complexity.

- `run` in `transport.py`
  This method also lacks documentation, but it seems to open a new thread that handles incoming packets received by the application. As there is some multiplexing and error handling inherent in this process, some cyclomatic complexity is to be expected.

- `readline` in `file.py`
  This method has a good docstring that states that this method reads an entire line from the file that corresponding file object represents. Although some complexity is to expected for this task, the complexity of this function is way larger than one expects it to be.

- `_parse_userauth_request` in `auth_handler.py`
  As this function is private, it is undocumented. Based on the name and the code, it parses a user authentication request. As the function contains some input sanitation and multiplexing, it is to be expected that this function is quite complex.

- `_handle_request` in `channel.py`
  This function seems to handle an incoming request to the SSH channel and will perform various different tasks depending on which type of request was received. Since the function performs several tasks, it is going to be quite long and complex.

- `_parse_service_accept` in `auth_handler.py`
  This method also lacks documentation but it seems to accept or deny different kinds of user authentication. It handles the different authentication methods in different ways and some are more complex than others with nested if statements.

- `_parse_channel_open` in `transport.py`
  Based on the name and the code, this function parses a request to open a transport layer channel such as a TCP channel. As this function handles several protocols which all require a certain amount of input validation, the complexity of this function is naturally quite high.

- `read_message` in `packet.py`
  The documentation for this function is very sparse, but judging from the function name, it reads a message from the SSH channel. Since this is a complex task involving handling of raw binary data and sequence number checking, it is not unthinkable that this would be a function of high complexity if it is hard to split the function into smaller subfunctions.

- `_parse_signing_key_data` in `ed25519key.py` This function parses the data needed for constructing an Ed25519 signature. As the function extracts the signature from a encrypted packet, the function is quite complex as it has to deal with several protocols.

- `_parse_kex_init` in `transport.py`
  The function does not have a lot of documentation, just a few lines for some specific rows that they want to make more clear. The method initializes the key exchange procedure and has multiple ifs followed by elses that could be refactored.

4. Are exceptions taken into account in the given measurements?

We took exceptions into account when we counted the complexity by hand. We were unable to ascertain whether Lizard does.

5. Is the documentation clear w.r.t. all the possible outcomes?

No, at least not in all cases. It varies a fair bit between different parts of the project. Where certain parts are more well documented than others.

Coverage

Tools

We used to coverage tool `coverage.py` for calculating the branch coverage. It was easy enough to setup and use. The documentation pointed to a plugin which provided integration with the project's testing framework, but we did not try this out.

Ad-Hoc Coverage tool

The coverage tool we created can be found in `test_parser.py`. The tools requires that a line containing a unique ID is manually added before the first statement of each branch in the functions of interest. Since the tests in this project are not called from a central test-handler it was decided that it would be best to write the unique-ID for each line directly to a file. When the testing is done the `test_parser.py` generates a report from the file.

The tool has a few limitations. It does not work for ternary operators without adding code to the function itself. Also since the lines are manually added it is very prone to error by forgetting to add a line or not having the unique ID's unique.

Both `coverage.py` and *test_parser.py* give almost identical results. There are however two functions, `_parse_channel_open` and `_parse_userauth_request` where the two coverage tools give slightly different results. Why this is happening is hard for us to tell since both functions are very complex.

To view the changes to the code use the command:
`git diff master..ad_hoc_coverage`
Or look at the branch directly by clicking here.

Added test cases

Four test cases were added per group member, which makes for twenty tests in total. All tests increased the branch coverage by executing previously untested branches. To find untested branches, we used `coverage.py`.

The added tests can be found through the links listed in the appendix.

Evaluation

Report of old coverage: Go to table

Report of new coverage: Go to table

As can be seen in the tables, the test coverage was improved quite substantially for some files (up to 21 %) and was at least slightly improved in others. The overall coverage was increased by 2 %, which is not bad in big project like this one.

Refactoring

Refactoring plan for the five functions of highest complexity

`_process` in `sfpt_server.py`
The function basically consists of several larger branches that are executed based on the value of the argument `t`. A refactoring strategy would therefore be to split the function into several smaller functions that are called instead of the corresponding blocks of code in `_process`, leaving only the logic for which function to call in `_process`.

`_run` in `transport.py`
Most of the complexity in `run` is in a single while loop. This loop is 66 lines of code with multiple `if/else` statements. Refactoring this loop to use helper functions would reduce its complexity by a significant amount.

`_readline` in `file.py`
The complexity of this function is absurdly high. The developers even left a comment in the code saying *"it's almost silly how complex this function is"*. As in the other functions, one way to reduce its complexity would be to introduce helper functions that are called from `readline`. The function contains a large while loop with multiple different branches of code. This loop could be refactored by introducing said helper functions. Creating these helper functions will however be very hard without significant domain knowledge and a potentially large increase in data flow complexity.

`_parse_userauth_request` in `auth_handler.py`
This function consists of several large branches that are executed based on the value of the variable `method`. A refactoring strategy would therefor be to split the function into several helper functions that handle the logic for the possible cases of the variable `method`.

`_handle_request` in `channel.py`
Like `_parse_userauth_request` and `_process` this function consists of several branches that are executed based on a specific variable. In this function the variable is called `key`. A refactoring strategy could therefor be to create specific helper functions for each of the values `key` can take.

Refactoring carried out

We have refactored two functions, `_process` in `sfpt_server.py` and `_handle_request` in `channel.py`.

`_process`
As described above, the contents of the branches contained within the switch statement were moved to helper functions. The new cyclomatic complexity after the refactoring is 20, which corresponds to a 42.9 % reduction in complexity from the original 35. The refactored code itself can be found by clicking here.

`_handle_request`
This function was refactored in a very similar manner to `_process`. The largest difference is that `_handle_request` has a status flag that is set in each branch, requiring the helper functions to return the value of this flag, which is set when the function is called in `_handle_request`. In this fashion, the cyclomatic complexity was reduced by 38 % (from 21 to 13). The refactored code itself can be found by clicking here.

Effort spent

The contributions of each member is listed below

Aron Strandberg

- Looking for projects: 3h
- Discussions / meetings: 1h
- Reading documentation: 3h
- Configuring project: 1h
- Reading and analyzing code: 2h
- Writing tests: 5h
- Writing documentation: 1h

Eysteinn Gunnlaugsson

- Looking for projects: 5h
- Discussions / meetings: 4h
- Reading documentation: 2h
- Configuring project: 1h
- Code coverage: 4h
- Reading and analyzing code: 5h
- Writing tests: 5h
- Writing documentation: 4h

Robert Kindwall

- Looking for projects: 4h
- Discussions / meetings: 4h
- Reading documentation: 3h
- Configuring project: 1h
- Reading and analyzing code: 5h
- Writing tests: 7h
- Writing documentation: 4h

Karl Kvarnfors

- Looking for projects: 4 h
- Discussions / meetings: 4 h
- Reading documentation: 2 h
- Configuring project: 2 h
- Reading and analyzing code: 5 h
- Writing tests: 5 h
- Writing documentation: 8 h

Erik Comstedt

- Looking for projects: 5h
- Discussions / meetings: 4h
- Reading documentation: 2h
- Configuring project: 2h
- Reading and analyzing code: 4h
- Writing tests: 7h
- Writing documentation: 8h

Overall experience

One of our main take-aways from this project is that it is vital to have a well-structured and not too complex code base in an open source project like this. When we were about to get into this project, it was quite difficult and challenging to get an understanding of what certain functions were actually doing. Additionally the projects documentation was not very clear at certain parts, which made things even more challenging. In particular, we saw that it is not

at all a bad idea to provide docstrings in private functions as it can sometimes be tricky to figure out what a private function is actually supposed to do by just looking at the code. These aspects are especially important for open source projects, where it is of great importance for the project to be well structured and documented. Otherwise it might be too confusing and possibly discourage new developers from getting involved with the project, thus halting the project's progression.

Our tests check some previously untested parts of the code. In any project it is important to make sure that most parts of the code is covered by tests as you progress forward with development. For each new function or method created, make sure that relevant test cases for the newly implemented code are created as well. This makes sure the code and tests are always up to date with each other. This was not the case in this project, as several files had quite low coverage despite the high overall coverage level. We feel that our contributions to the project has at the very least been a step in the right direction.

Additionally, we refactored two functions within the project, and by this reducing their overall complexity. The two functions were previously quite complex and hard to grasp. After our refactoring, we have been able to reduce the complexity of these functions significantly. Although the work done is in some sense trivial, the code is now easier to read despite being longer than before.

We have also added a pull request in order to integrate our work into the project, but at the time of writing it has not been accepted. The pull request is found on this link:

`https://github.com/paramiko/paramiko/pull/1170`

Appendix

Test Coverage before

Table 2: Test Coverage before changes

Name	Stmts	Miss	Branch	BrPart	Cover
__init__.py	34	0	0	0	100 %
_version.py	2	0	0	0	100 %
agent.py	223	131	50	5	36 %
auth_handler.py	510	231	134	23	55 %
ber.py	83	36	36	5	50 %
buffered_pipe.py	93	6	30	3	93 %
channel.py	585	121	160	35	76 %
client.py	275	75	120	26	71 %
common.py	92	2	2	1	97 %
compress.py	12	0	0	0	100 %
config.py	120	6	72	6	92 %
dsskey.py	115	12	18	5	87 %
ecdsakey.py	157	12	38	6	91 %
ed25519key.py	120	16	50	14	82 %
file.py	252	22	102	10	90 %
hostkeys.py	197	40	98	14	76 %
kex_ecdh_nist.py	86	1	10	1	98 %
kex_gex.py	182	13	50	13	89 %
kex_group1.py	86	4	14	4	92 %
kex_group14.py	8	0	0	0	100 %
kex_gss.py	347	296	86	0	13 %
message.py	104	1	18	0	99 %
packet.py	341	24	124	21	89 %
pipe.py	84	25	14	1	69 %
pkey.py	179	33	44	7	80 %
primes.py	70	59	32	0	11 %
proxy.py	49	33	12	0	26 %
py3compat.py	101	54	24	2	46 %
rsakey.py	87	4	16	2	94 %
server.py	98	33	10	2	64 %
sftp.py	90	13	26	9	79 %
sftp_attr.py	150	20	80	15	81 %
sftp_client.py	355	39	113	28	85 %
sftp_file.py	227	29	80	11	86 %
sftp_handle.py	67	10	20	4	84 %
sftp_server.py	321	50	146	30	82 %
sftp_si.py	42	13	4	1	70 %
ssh_exception.py	52	4	4	0	93 %
ssh_gss.py	187	143	42	0	19 %
transport.py	1189	244	390	89	75 %
util.py	176	35	62	4	79 %
win_pageant.py	57	57	10	0	0 %
TOTAL	7605	1947	2341	397	72 %

Test Coverage after

Table 3: Test Coverage after changes

Name	Stmts	Miss	Branch	BrPart	Cover
__init__.py	34	0	0	0	100%
_version.py	2	0	0	0	100%
agent.py	223	99	50	9	49%
auth_handler.py	510	219	134	24	58%
ber.py	83	20	36	8	71%
buffered_pipe.py	93	6	30	3	93%
channel.py	616	108	162	36	79%
client.py	275	75	120	26	71%
common.py	92	2	2	1	97%
compress.py	12	0	0	0	100%
config.py	120	6	72	6	92%
dsskey.py	115	12	18	5	87%
ecdsakey.py	157	12	38	6	91%
ed25519key.py	120	16	50	14	82%
file.py	252	19	102	10	91%
hostkeys.py	197	29	98	14	81%
kex_ecdh_nist.py	86	1	10	1	98%
kex_gex.py	182	13	50	13	89%
kex_group1.py	86	4	14	4	92%
kex_group14.py	8	0	0	0	100%
kex_gss.py	347	296	86	0	13%
message.py	104	1	18	0	99%
packet.py	341	23	124	21	90%
pipe.py	84	25	14	1	69%
pkey.py	179	33	44	7	80%
primes.py	70	48	32	1	25%
proxy.py	49	28	12	0	34%
py3compat.py	101	54	24	2	46%
rsakey.py	87	4	16	2	94%
server.py	98	31	10	2	66%
sftp.py	90	13	26	9	79%
sftp_attr.py	150	20	80	15	81%
sftp_client.py	355	39	113	28	85%
sftp_file.py	227	29	80	11	86%
sftp_handle.py	67	10	20	4	84%
sftp_server.py	358	50	146	30	84%
sftp_si.py	42	13	4	1	70%
ssh_exception.py	52	4	4	0	93%
ssh_gss.py	187	143	42	0	19%
transport.py	1189	235	390	89	76%
util.py	176	34	62	3	79%
win_pageant.py	57	57	10	0	0%
TOTAL	7673	1831	2343	406	74%

Links to added test cases

Here is a list of links to the pull requests containing the added tests.

https://github.com/eysteinn13/paramiko/pull/48/files

https://github.com/eysteinn13/paramiko/pull/45/files

https://github.com/eysteinn13/paramiko/pull/40/files

https://github.com/eysteinn13/paramiko/pull/37/files

https://github.com/eysteinn13/paramiko/pull/36/files

https://github.com/eysteinn13/paramiko/pull/35/files

https://github.com/eysteinn13/paramiko/pull/33/files

https://github.com/eysteinn13/paramiko/pull/31/files

https://github.com/eysteinn13/paramiko/pull/30/files

https://github.com/eysteinn13/paramiko/pull/29/files

https://github.com/eysteinn13/paramiko/pull/28/files

https://github.com/eysteinn13/paramiko/pull/27/files

https://github.com/eysteinn13/paramiko/pull/24/files

https://github.com/eysteinn13/paramiko/pull/22/files

https://github.com/eysteinn13/paramiko/pull/21/files