Report assignment 3, by Group 9

Project

Name: BorgBackup

URL: https://github.com/borgbackup/borg

Description:
BorgBackup is a deduplicating backup program for various Unix-like operating systems.

Onboarding experience

The first project we picked was BorgBackup and we ended up going with it. We installed it on three different OS systems: Linux based, MacOS and Windows. The documentation covered all OS systems and went smoothly for every system. However, some parts of the test suite fail on Mac. Mainly parts related to low level functions that work with file systems that are not compatible with Mac.

Complexity

1. What are your results for five complex functions?

- **Did all methods (tools vs. manual count) get the same result?**

Function	**Tool Count**	**Manual Count**
yes	24	29
check	25	26
rst_to_text	38	30
ChunkerParams	26	12
do_transfer	32	28

- **Are the results clear?**

The results are generally clear in the sense that both the tool-based measurements and the manual counts consistently identify the same functions as complex, all show relatively high complexity values compared to simpler helper functions. This indicates that both approaches agree on which functions are structurally complex.

However, the numerical results are not identical between the automated tool (Lizard) and the manual count. In some cases, the difference is small (e.g., check: tool = 25, manual = 26), while in others the difference is larger (e.g., ChunkerParams: tool = 26, manual = 12). Therefore, the methods did not always produce the same exact result.

The differences arise mainly because automated tools and manual counting apply slightly different interpretations of cyclomatic complexity. Tools such as Lizard count decision points according to their own parsing rules. They typically include:

- if, elif
- loops (for, while)
- except blocks
- boolean operators (&&, ||, and, or)
- sometimes conditional expressions or other constructs

Manual counting, on the other hand, depends on how strictly the theoretical definition is applied. For example:

- Some group members may or may not count boolean operators inside a single condition as separate decisions.
- There may be inconsistencies in counting nested conditions.
- Exception handling (try/except) may be interpreted differently.
- Internal helper functions defined inside a function (such as nested functions in check) may or may not be included in the count.

In particular, larger discrepancies (such as for ChunkerParams) likely stem from differences in how compound boolean expressions and validation logic were counted manually compared to how the tool parses them Overall, while the absolute numbers differ slightly, the results are consistent in identifying which functions are more complex. The differences highlight that manual complexity calculation can be sensitive to interpretation, whereas automated tools apply a fixed and consistent counting rule.

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

Function	Complex or just long
yes	Complex, but not very long. It does a lot of things though (read input, produce output, classify answer and run the loop which continually runs through asking for answers and giving output).
check	Complex and long function
rst_to_text	Even if the function takes only three parameters, it contains many if conditions and checks in a long while loop.The function is also long. It's NLOC is 93.
ChunkerParams	The function is relatively long and also has a high complexity.However, length alone is not enough. The complexity mainly comes from the number of decisions and validations, not only from the size.but functions with many branches often become longer.
do_transfer	The function has both a high complexity and is very long (114 LOC).

3. What is the purpose of the functions?

Function	Purpose
yes	Helper function to handle IO and classify input as True/False.
check	It verifies repository integrity by validating object checksums and sizes. If corruption is found, it logs errors and optionally deletes damaged objects when repair=True
rst_to_text	It performs a loose conversion of reStructuredText (rST) into simple and readable text for the eyes .
ChunkerParams	parse and validate user input for chunking parameters.It checks which algorithm the user wants and verifies that the parameters are valid.
do_transfer	Transfer a borg repository to a different repository. It both validates the compatibility of the two repositories and does the actual transfer

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

Yes, exceptions are taken into account in the given measurements, but how they are handled depends on the method used (tool-based vs. manual counting).

When using an automated tool such as Lizard, exception handling constructs like *try* and *except* blocks are typically considered additional decision points. In particular, each *except* clause introduces an alternative execution path and therefore increases the cyclomatic complexity. The tool automatically counts these constructs based on its parsing rules, ensuring consistent treatment across all functions.

In the manual counting approach, exceptions were also considered, but the inclusion depended on how strictly the theoretical definition of cyclomatic complexity was applied. According to the standard definition, cyclomatic complexity increases with each additional independent control-flow path. Since *except* blocks represent alternative execution flows, they should be counted as decision points. However, differences can occur if group members interpret *try* blocks differently or disagree on whether to count specific constructs such as raise statements or nested exception handling.

Therefore, exceptions were included in both measurement approaches, but minor discrepancies between tool-based and manual results may partly be explained by differences in how exception-related constructs were interpreted during manual counting.

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

Function	**Documentation clear for all outcomes**
yes	It is convoluted, but yes. It does detail them.
check	There are not many details about the function, only a few comments.
rst_to_text	There is a comment, the docstring, which says the output is human-friendly text. There is also a # comment which describes the input. It didn't feel very descriptive.
ChunkerParams	The function contains some useful comments explaining the validation rules and constraints. However, the overall documentation is not fully clear. There is no complete explanation of all valid formats and all possible errors in one place.
do_transfer	The function is not documented as only public functions seem to be

	documented in the repository.

Refactoring

Plan for refactoring complex functions:

Check function : The check() function in src/borg/repository.py is complex mainly because it mixes several responsibilities in one long method: iterating repository objects, validating object structure and checksums, handling partial-check checkpoint logic, tracking statistics, and implementing repair behavior (reload and delete). A refactoring plan would focus on separating these concerns into smaller units with clear inputs/outputs.
A practical plan is to extract three main parts: (1) a pure validation function that inspects a repository object and returns a list of validation errors instead of mutating shared state, (2) a repair handler that decides whether to retry loading or delete the object, and (3) a checkpoint/progress helper responsible for storing and resuming LAST_KEY_CHECKED. This would reduce nesting, make the code easier to test, and allow branch-specific tests without heavy mocking.

rst_to_text Function:
In order to refactor this function to get lower CC, one would need to divide the comprehensive while loop and its if/else statements into different functions. There is a while loop in a while loop, and the inner while loop should not be there. The main loop should ask what state the code is in, and depending on state, the correct function should be called. So, there should be one while true loop that reads one character at the time. And depending on what character type is being parsed, the correct helper function should be called.

Removing nested while loops that contain only conditions will also lower the CC. Although maybe not very much. But there are several calls to .peek(). If we use a cache for previous peeks we can save some computational cost by simply reusing the previous peeks.

ChunkerParams:
The function is currently more complex than needed because all the logic is written in one long block and some validation rules are repeated. It is possible to reduce this complexity by splitting the function into smaller and clearer units. The main idea is to turn `ChunkerParams` into a simple dispatcher that selects the correct helper function depending on the algorithm. Each algorithm, such as `fixed`, `fail`, `buzhash64`, and the compatibility mode, can be handled in its own small function. In addition, common validation rules, like checking the mask range, minimum and maximum limits, and window size, can be extracted into shared helper functions to avoid **duplicated code**.

Estimated impact of refactoring (lower CC, but other drawbacks?).

rst_to_text Function:
Refactoring this function would lower CC. It is reasonable to assume that it would be more difficult to make changes in the code. Now all of rST conversion is handled in the same function, so you don't need to look up other functions. A potential drawback is that there will be a lot more function calls. It may be convenient to write everything in one function and have everything in one place. Refactoring into smaller functions may also cause unpredictable behaviour. Since there are now more functions that may depend on each other, more rigorous testing may be needed.

Check function:

Refactoring would likely reduce cyclomatic complexity of check() because several nested conditional blocks would move into helper functions (e.g., corruption detection, repair, checkpointing). Even if the total complexity across the module remains similar, the complexity would be distributed across smaller functions, making each unit easier to understand and test. The most important benefit is improved readability and testability: smaller pure functions allow deterministic unit tests and reduce the need for integration-style setups.

Potential drawbacks include slightly higher code volume (more helper functions), the risk of changing behavior if refactoring is not done carefully, and minor overhead in passing state between functions. There is also a maintainability trade-off: spreading logic across multiple functions can make the full workflow harder to follow if naming and structure are not done well. However, with clear naming and documentation, the benefits outweigh the drawbacks for a function of this size.

ChunkerParams:

Refactoring `ChunkerParams` into a dispatcher + small helpers would likely lower cyclomatic complexity a lot . The total project complexity doesn't disappear, but it becomes spread across smaller units, so each function is easier to read/test and lizard will report lower CC for `ChunkerParams` .

Carried out refactoring (optional, P+):

git diff …

Yes function refactoring

- Complexity comes from
    - The function doing many things related to the input message loop
    - Input handling
    - Input classification
    - Output formatting
    - The retry control loop

- Remove the nested function of output
    - Create it as a separate function
    - Will also need to pass some more things
        - ofile, msgid
    - What it does?
        - Same thing!
        - Helps give output
- Extract answer retrieval
    - Input
        - def _get_answer(env_var_override, env_msg, input_func, default, prompt):
    - Does
        - Gets answer and throws exceptions if needed
        - Also uses env variable if needed
    - Return
        - answer
- Extract the answer classification
    - Input
        - def _classify_answer(answer, truish, falsish, defaultish, default):
    - Does
        - Classifies the answer based on definitions of truish etc. that are feeded to it
    - Return
        - True/False/ other for invalid
- Main function
    - Mainly runs loop to
    - Get answer
    - Classify answer
    - Decides if it should output (using the separate function)
    - Or rerun the loop looking for a valid answer

This is estimated to reduce complexity by roughly 50% since we reduce how often to run many of the if branches.

Refactor do_transform
That the cyclomatic complexity of `do_transform` is that high is caused by multiple factors. Some of them are inherent to the problem of transferring one repository to another while accounting for a large number of different possible cases (borg 1.x to 2.x repository, mismatching Hashes etc) but the reason why the cyclomatic complexity is that high also is because the do_transform function does multiple things at once:

- Validating Compatibility between the two repositories
- Validating the metadata of the repository
- Upgrading a borg 1.x repository to a borg 2 repository
- Transferring the actual chunks.

To reduce the complexity of this method through refactoring, we should split off parts of the method into helper methods to make it clearer, more readable and follow the *Separation of Concerns* pattern.

Coverage

Tools

Document your experience in using a "new"/different coverage tool.

In this assignment, we used coverage.py together with pytest to measure statement and branch coverage for five different functions in the Borg project: yes, check, rst_to_text, ChunkerParams, and do_transfer. These functions vary significantly in size and complexity, which allowed us to evaluate how the tool behaves for both small helper functions and highly complex logic.

Our experience with coverage.py was overall very positive. The tool integrates naturally with pytest and provides detailed reports that include statement coverage, branch coverage, and partially covered branches. It allowed us to clearly identify which control-flow paths were executed and which were not

How well was the tool documented? Was it possible/easy/difficult to integrate it with your build environment?

The documentation of coverage.py is comprehensive and clear. The official documentation includes installation instructions, command-line usage examples, explanation of report formats, and integration details with common test frameworks such as pytest.

Integration into our build environment was straightforward. After installing the tool via pip, we executed the tests using: *coverage run -m pytest*

And generated a detailed reports using: *coverage report*

The only challenge was ensuring that we used consistent test selection when comparing coverage before and after adding new tests. Once this was clarified, integration was smooth and required no modification of the build system itself. Overall, the integration process was easy.

Your own coverage tool

We implemented a DIY branch coverage tool by manually instrumenting the code and printing a coverage report at the end of the pytest session.

Branch used: **`1-create-diy-coverage-checker`**

How to obtain the patch (git diff command):

git diff origin/master...origin/1-create-diy-coverage-checker -- src/borg/helpers/coverage_diy.py src/borg/conftest.py

What the tool does

- We created a small helper module src/borg/helpers/coverage_diy.py.
- We manually inserted mark("BRANCH_ID") calls at the start of each branch outcome.
- We also registered all branch IDs so missing branches can be reported.
- We hooked the report printing into pytest using pytest_sessionfinish() in conftest.py, so the report prints automatically after all tests finish.

Supported constructs

Our DIY tool supports **explicit branch outcomes** that we manually instrument, such as:

- if / else branches (both True and False paths)
- boolean conditions with and / or **as a single decision**

Accuracy of the output

- The output is **accurate for the branches we instrumented**, because it reports exactly which mark() IDs were executed during the test run.
- It is **not automatic**: if a branch exists in code but we forget to instrument it, the tool will not see it.
- It measures coverage only at the places where we inserted markers.

- For complex boolean expressions (especially with or / and), the tool counts the whole expression as one branch.

Evaluation

1. How detailed is your coverage measurement?

Our Coverage tool checks whatever we manually instrument with the `mark` function.
In our case those are manually identified branches. We could in theory go into more detail to get information about line coverage with it even though this would be tedious.

2. What are the limitations of your own tool?

First and foremost the biggest limitation is that our tool requires manual instrumentation, which is time consuming and modifies the actual codebase which clutters up the actual code.
Furthermore currently it cannot detect path coverage since we do not measure which combinations of ids occurred after another.

3. Are the results of your tool consistent with existing coverage tools?

Yes in terms of branch coverage our tool is mostly consistent with `pytest-coverage`.
But since `pytest-coverage` uses interpreter level instrumentation it can identify slightly different branches than we manually did which can lead to inconsistencies between them.

Test cases added:

- **Davide added 4 tests to: Yes()**
  - test_yes_prompt_false():
    - Line 91
  - test_yes_unicode_decode_error():
    - Line 96
  - test_yes_invalid_msg():
    - Line 115
  - test_yes_default_msg():
    - Line 103

- **Anton Trappe added 4 tests to: do_transfer**
    - test_transfer_id_hash_mismatch
        - Tests that mismatching hashes raise a exception
    - test_transfer_invalid_archive_names
        - Tests that invalid archive names raise exception
    - test_transfer_skips_part_files_branch
        - Tests that invalid file parts are skipped in the transfer
    - test_transfer_skips_legacy_parts
        - Tests that legacy parts that cannot be upgraded are skipped

- **Yasmine added 2 tests to: rst_to_text()**
    - test_code_block_coverage()
        - Tests that that rst_to_text has "::" handled correctly (that it enters literal mode)
    - test_bold_text_coverage()
        - Tests that rst_to_text enters bold state and exits bold state

- **Gabriel added 4 test for check()**
    - def test_check_repair_deletes_corrupt_object
        - Verifies that objects smaller than the required header size are detected and deleted.
    - def test_check_stops_after_timeout
        - Confirms the check process stops executing once the `max_duration` time limit is reached
    - def test_check_repair_deletes_invalid_hash
        - Ensures objects with data not matching their checksum are identified and removed.
    - def test_check_empty_repository
        - Validates that the check completes successfully without errors when the store contains no objects

- **Jribi added 4 test for chunkerparams in parseformat_test.py**

The function **`ChunkerParams`** parses and validates chunking parameters provided as a string. There are two main possible outcomes. The parameters can either be **valid**, in which case the function returns the parsed values depending on the selected algorithm, or they can be **invalid**, in which case the function should raise an error.

Because this function supports several algorithms and different parameter formats, there are many possible input cases. Each input corresponds to an expected behaviour, either a correct return value or an exception. Therefore, each input with its expected output can be considered as a **separate unit test**.

In the existing test suite, there are already two global test functions:

- `test_valid_chunkerparams(chunker_params, expected_return)` checks that the function correctly parses and returns valid parameters.

- `test_invalid_chunkerparams(invalid_chunker_params)` verifies that the function raises an error for invalid inputs.

However, these two lists of inputs do not cover all possible branches of the function. From the DIY branch coverage, we identified that some branches were not executed during testing. To improve coverage, we added new inputs that specifically target these uncovered branches.

We added **four new test cases**, each corresponding to a previously untested branch:

1. **("fail,4096,bitmap", ("fail", 4096, "bitmap")) (line 614)**
   This test covers the case where the algorithm is `fail` with three parameters. This is a valid case, and the expected output is the parsed tuple.

2. **"Buzhash64,5,6,5,17" (line 633)**
   This test checks that the function raises an error when the algorithm is `buzhash64` and the minimum chunk size is too small (`chunk_min < 6`) .

3. **"buzhash64,6,24,6,17"(line 634)**
   This verifies that the function raises an error when the maximum chunk size exceeds the allowed limit (`chunk_max > 23`).

4. **"buzhash64,10,12,9,17"(line 635)**
   This checks that the function raises an error when the constraint `min ≤ mask ≤ max` is violated.

In conclusion, these four additional inputs correspond to four new unit tests because each one validates a previously uncovered branch. Instead of creating separate test functions, we integrated them into the existing parameterized tests. This approach avoids redundancy and keeps the test suite clean and maintainable while improving branch coverage.

Link to tests:
https://github.com/DillwynKnox/borg/compare/master...DillwynKnox:borg:4-feat-add-2-unit-tests-per-person

Number of test cases added: two per team member (P) or at least four (P+):
Every group Member did at least two tests, the following members did 4:

- Anton Trappe (trappe@kth.se)
- Davide
- Gabriel Arias (gaag2@kth.se)
- Jribi Mohamed Aziz (jribi@kth.se)

Coverage improvement

The added tests let to the following improvements in Coverage found with the pytest-cov tool

Function	Coverage before	Coverage after	Additional Branches Covered
yes	85%	95%	6
do_transfer	82%	90%	7
rst_to_text	70%	79%	15
ChunkerParams	50%	51%	10
check	85%	87%	6

Report of old coverage:
https://github.com/DillwynKnox/borg/blob/4-feat-add-2-unit-tests-per-person/coverage_before.txt

Report of new coverage:

https://github.com/DillwynKnox/borg/blob/4-feat-add-2-unit-tests-per-person/coverage_after.txt

Self-assessment: Way of working

Current state according to the Essence standard:

Table 8.8 – Checklist for Way-of-Working

State	Checklist
Principles Established	Principles and constraints are committed to by the team. Principles and constraints are agreed to by the stakeholders. The tool needs of the work and its stakeholders are agreed. A recommendation for the approach to be taken is available. The context within which the team will operate is understood. The constraints that apply to the selection, acquisition, and use of practices and tools are known.
Foundation Established	The key practices and tools that form the foundation of the way-of-working are selected. Enough practices for work to start are agreed to by the team. All non-negotiable practices and tools have been identified. The gaps that exist between the practices and tools that are needed and the practices and tools that are available have been analyzed and understood. The capability gaps that exist between what is needed to execute the desired way of working and the capability levels of the team have been analyzed and understood. The selected practices and tools have been integrated to form a usable way-of-working.
In Use	The practices and tools are being used to do real work. The use of the practices and tools selected are regularly inspected. The practices and tools are being adapted to the team's context. The use of the practices and tools is supported by the team. Procedures are in place to handle feedback on the team's way of working. The practices and tools support team communication and collaboration.
In Place	The practices and tools are being used by the whole team to perform their work. All team members have access to the practices and tools required to do their work. The whole team is involved in the inspection and adaptation of the way-of-working.
Working well	Team members are making progress as planned by using and adapting the way-of-working to suit their current context. The team naturally applies the practices without thinking about them. The tools naturally support the way that the team works. The team continually tunes their use of the practices and tools.

Was the self-assessment unanimous? Any doubts about certain items?

The self-assessment was unanimous.

How have you improved so far?
Yes, we have become comfortable with the tools we use such as git.

Where is potential for improvement?
In order to reach **Working Well** we need to: Continue to work with these tools to ensure that they are applied naturally. Git and discord now work well, but we still need to become more natural with the tools we use. How to divide work in git and link real world problems to meaningful issues.

Overall experience

What are your main take-aways from this project? What did you learn?
Even big open source projects can have poor code. We learned how big projects are organized and what folder structure they use. We also learned more about git usage, since we merged branches into other branches, instead of main. Furthermore, using a forked git repo made it look slightly different.

P+ work per person

Davide Attebrant

- P+ criteria 1 (4 tests for yes function)
- P+ criteria 2 (issue tracker, see production branches: master, 1, 4)
- P+ criteria - relevant mini-assignments done

Anton Trappe

- P+ criteria 4 tests added
- P+ criteria refactored `do_transfer` (https://github.com/DillwynKnox/borg/pull/28)
- P+ criteria Issue tracker

Jribi Mohamed Aziz

- P+ criteria 1 (4 tests for ChunkerParams function)
- P+ criteria 2 (issue tracker, see production branches: master, 1, 4)
- P+ criteria refactored `ChunkerParams` (https://github.com/DillwynKnox/borg/pull/36)