HoneyBadgerMPC Documentation

Release 0.0.0

Andrew Miller et al. – Decentralized Systems Lab (UIUC)

Oct 12, 2018

The Honeybadger of MPC Protocols

HoneyBadgerMPC is a research project under active development that aims to be a novel confidentiality layer for consortium blockchains.

HoneyBadgerMPC is the first system implementation of asynchronous MPC.

HoneyBadgerMPC is the first MPC toolkit to provide all the following desired properties:

- Scales to large number of nodes (5 to 50). *Viff fails at this!*
- Guarantees output even if some nodes crash or are compromised recent developments like SPDZ and EMP-Toolkit fail at this.
- Prevents data breaches even if some nodes are compromised.

CHAPTER 1

Motivation Use Case: Supply Chain Tracking

Security Goals

- **Integrity and consistency:**
  - Records cannot be modified or removed without leaving an audit trail.
  - All parties can see a consistent view of records.
- **Availability:**
  - No one can be prevented from submitting or reading a record.
- **Privacy:**
  - Names of participants, business relationships, are proprietary & sensitive.

1.1 Blockchain databases today: Hyperledger Fabric

Todo: Keep going!

CHAPTER 2

Protocol Descriptions

2.1 HBAVSS

An Asynchronous Verifiable Secret Sharing protocol. Allows a dealer to share a secret with n parties such that any $t + 1$ of the honest parties can reconstruct it. For our purposes (achieving optimal Byzantine Fault Tolerance), we will always be using $n = 3t + 1$.

Input: One secret the size of a field element

Output: n parties will have a t-shared share of the secret

2.2 BatchHBAVSS

An altered version of HBAVSS that allows for the more efficient sharing of a batch of secrets. Details to be worked out soon (TM).

2.3 ReliableBroadcast

A protocol which allows a broadcaster to send the same message to n different recipients in a bandwidth-saving way, while being assured that every recipient eventually receives the full correct message.

Input: One message to be broadcasted

Output: n parties will eventually successfully receive the message

2.4 BatchReconstruction

A protocol to reconstruct many secrets at once with fewer messages

Input: At least $t + 1$ parties input their shares of $t + 1$ total t-shared secrets

Output: $t + 1$ reconstructed secrets are output to every participating party

2.5 Rand

A protocal that generates a random secret-shared value, which nobody will know until it is reconstructed

2.6 BatchRand

An effiecient batched version of Rand

2.7 BatchBeaverMultiplication

Perform multiple shared-secret multiplications at once

Input: n t-shared pairs of secrets that one wishes to multiply and n sets of beaver triples. $2n \geq t + 1$

Output: n t-shared secret pairs that have been successfully multiplied

2.8 TripleTransformation

Turns a set of triples into a set of co-related and secret-shared triples. This plus Polynomial Verification together can tell you if all of your input triples are multiplication triples.

> Co-related triples are triples that make up points on polynomials $A()$, $B()$, and $C()$ such that
>
> $$A() \cdot B() = C(),$$

i.e. $A(i) \cdot B(i) = C(i) \forall i$.

Input: m *independent* t-shared triples, where $m = 2d + 1$ and $d \geq t$.

> Note the if we have $d == t$, then this whole process gives 1 multiplication triple, but can tolerate the most dropouts. On the other hand, if $d = 3t/2$, we have $m = 3t + 1$ and hence have the best efficiency but require all parties to provide correct input to proceed.

Output: m t-shared, co-related triples

2.9 PolynomialVerification

Determine if the triples output from TripleTransformation are multiplication triples.

Input: n different t-shared outputs of TripleTransformation

Output: Knowledge of which polynomials one can extract multiplication triples from

2.10 TripleExtraction

Extract unknown random multiplication triples from a set of co-related multiplication triples.

Input: n different polynomials formed by t-shared co-related multiplication triples

Output: $(d + 1 - t)$ t-shared random triples, where d is the degree of polynomials $A()$ and $B()$

CHAPTER 3

Performance

Performance of MPC protocols is determined by:

- How fast can we perform Batch Reconstruction? (Online phase)
- How fast can we generate Beaver triples (Offline phase)

CHAPTER 4

Roadmap

draft

4.1 Benchmarking

- Measure the number of Beaver triples that can be generated per seconds.
- Become aware of where the bottlenecks are, for each subprotocol.
- One performance goal is to be able to have similar performance metrics as viff with better robustness and capability to support a larger number of participants.
- Figure out how well the protocol performs for real-world applications such as the sugar beets auction.
- Identify which parts of the implementation need performance optimizations, and write C or Rust extensions for these parts.

4.2 Project structure

- Refine the project structure over time to reflect the modularity and composability of the protocol.
- Possibly, eventually package and distribute the key sub-protocols as standalone, re-usable software libraries/packages which can be used in diverse larger protocols.

4.3 Development tasks for HoneyBadgerMPC prototype

https://github.com/initc3/HoneyBadgerMPC/issues

CHAPTER 5

Running ACS with an EVM blockchain

Todo: Show how to run the `honeybadgermpc.commonsubset_functionality`.

Todo: Replace functionality with blockchain: *honeybadgermpc.commonsubset_blockchain*

CHAPTER 6

Running passive examples

Todo: Show how to run some `honeybadgermpc.passive` examples.

CHAPTER 7

Integration with Hyperledger Fabric

Todo: Document:

- Motivation
- General idea
- Roadmap (planned iterations, etc)
- Status of the integration work
- etc

CHAPTER 8

honeybadgermpc

HoneyBadgerMPC: Confidentiality Layer for Consortium Blockchains.

8.1 passive

8.2 commonsubset

8.2.1 Ideal functionality

8.2.2 EVM blockchain -based implementation

Implementation of Asynchronous Common Subset using an EVM blockchain

8.3 triple_refinement

8.4 rand

8.4.1 Ideal functionality

Ideal functionality for Random Share This protocol returns a single random share

8.4.2 Protocol

8.4.3 Batch

8.5 secretshare

8.6 field

Modeling of Galois (finite) fields. The GF function creates classes which implements Galois (finite) fields of prime order

All fields work the same: instantiate an object from a field to get hold of an element of that field. Elements implement the normal arithmetic one would expect: addition, multiplication, etc.

Defining a field:

```
>>> Zp = GF.get(19)
```

Defining field elements:

```
>>> x = Zp(10)
>>> y = Zp(15)
>>> z = Zp(1)
```

Addition and subtraction (with modulo reduction):

```
>>> x + y
{6}
>>> x - y
{14}
```

Bitwise xor for field elements:

```
>>> z ^ z
{0}
>>> z ^ 0
{1}
>>> 1 ^ z
{0}
```

Exponentiation:

```
>>> x**3
{12}
```

Square roots can be found for elements based on GF fields with a Blum prime modulus (see `GF()` for more information):

```
>>> x.sqrt()
{3}
```

Field elements from different fields cannot be mixed, you will get a type error if you try:

```
>>> Zq = GF.get(17)
>>> z = Zq(2)
```

(continues on next page)

(continued from previous page)

```
>>> x + z
Traceback (most recent call last):
    ...
TypeError: unsupported operand type(s) for +: 'GFElement' and 'GFElement'
```

The reason for the slightly confusing error message is that `x` and `z` are instances of two *different* classes called `GFElement`.

8.7 polynomial

8.8 router

8.9 ipc

8.10 config

Module for `honeybadgermpc`'s configuration.

This module can be used to:

- define default configuration settings
- load a configuration
- validate a comfiguration

example of config dict:

```
{
    'N': 4,
    't': 1,
    'nodeid': '0',
    'host': 'hbmpc_node_0',
    'port': 23264,
    'peers': {
        '1': hbmpc_node_1:23265,
        '2': hbmpc_node_2:23266,
        '3': hbmpc_node_3:23267,
    },
}
```

CHAPTER 9

Getting started

To start developing and contributing to HoneyBadgerMPC:

1. Fork the HoneyBadgerMPC repository.
2. Clone your fork:

```
$ git clone --branch dev git@github.com:<username>/HoneyBadgerMPC.git
```

3. Add the remote repository initc3/HoneyBadgerMPC:

```
$ git remote add upstream git@github.com:initc3/HoneyBadgerMPC.git
```

Note: The remote name `upstream` is just a convention and you are free to name your remotes whatever you like.

See *Working with Git Remotes* for more information about remotes.

Next step: *setup a development environment*.

9.1 Development environment

You are free to manage your development environment the way you prefer. Two possible approaches are documented:

- *Managing your development environment with docker-compose*
- *Managing your development environment with Pipenv*

Using `docker-compose` has the advantage that you do not need to manage dependencies as everything is taking care of in the `Dockerfile`.

You are encouraged to consult the Your Development Environment section in the The Hitchhiker's Guide to Python for tips and tricks about text editors, IDEs, and interpreter tools.

9.1.1 Managing your development environment with docker-compose

1. Install Docker. (For Linux, see Manage Docker as a non-root user) to run `docker` without `sudo`.)
2. Install docker-compose.
3. Run the tests (the first time will take longer as the image will be built):

   ```
   $ docker-compose run --rm honeybadgermpc
   ```

   The tests should pass, and you should also see a small code coverage report output to the terminal.

If the above went all well, you should be setup for developing **HoneyBadgerMPC**!

Tip: You may find it useful when developing to have the following 3 "windows" opened at all times:

- your text editor or IDE
- an `ipython` session for quickly trying things out
- a shell session for running tests, debugging, and building the docs

You can run the `ipython` and shell session in separate containers:

IPython session:

```
$ docker-compose run --rm honeybadgermpc ipython
```

Shell session:

```
$ docker-compose run --rm honeybadgermpc sh
```

Once in the session (container) you can execute commands just as you would in a non-container session.

Running a specific test in a container (shell session) As an example, to run the tests for `passive.py`, which will generate and open 1000 zero-sharings, $N = 3$ $t = 2$ (so no fault tolerance):

Run a shell session in a container:

```
$ docker-compose run --rm honeybadgermpc sh
```

Run the test:

```
$ pytest -v tests/test_passive.py -s
```

or

```
$ python -m honeybadgermpc.passive
```

About code changes and building the image

When developing, you should not need to rebuild the image nor exit running containers, unless new dependencies were added via the `Dockerfile`. Hence you can modify the code, add breakpoints, add new Python modules (files), and the modifications will be readily available withing the running containers.

9.1.2 Managing your development environment with Pipenv

1. Install pipenv.
2. Install the GMP, MPC and MPFR development packages:

   Debian

   Fedora

   Mac OS X

   Windows

   ```
   $ apt install libgmp-dev libmpc-dev libmpfr-dev
   ```

   ```
   $ dnf install gmp-devel libmpc-devel mpfr-devel
   ```

   ```
   $ brew install gmp libmpc mpfr
   ```

   Should not be needed as pre-compiled versions of `gmpy2` are available on PyPI. See gmpy2 docs for Windows for more information.
3. Install `honeybadgermpc` in editable mode for development:

   ```
   $ cd HoneyBadgerMPC/
   $ pipenv install -e .[dev]
   ```
4. Activate a virtualenv:

   ```
   $ pipenv shell
   ```
5. Run the tests to check that you are well setup:

   ```
   $ pytest -v --cov
   ```

The tests should pass, and you should also see a small code coverage report output to the terminal.

Useful resources on Pipenv

- Pipenv documentation
- Real Python: A Guide to Pipenv

9.2 Running the tests

The tests for `honeybadgermpc` are located under the `tests/` directory and can be run with pytest:

```
$ pytest
```

Running in verbose mode:

```
$ pytest -v
```

Running a specific test:

```
$ pytest -v tests/test_passive.py::test_open_share
```

When debugging, i.e. if one has put breakpoints in the code, use the `-s` option (or its equivalent `--capture=no`):

```
$ pytest -v -s
# or
$ pytest -v --capture=no
```

To exit instantly on first error or failed test:

```
$ pytest -x
```

To re-run only the tests that failed in the last run:

```
$ pytest --lf
```

See `pytest --help` for more options or the pytest docs.

9.2.1 Code coverage

Measuring the code coverage:

```
$ pytest --cov
```

Generating an html coverage report:

```
$ pytest --cov --cov-report html
```

View the report:

```
$ firefox htmlcov/index.html
```

Coverage configuration file

Configuration for code coverage is located under the file `.coveragerc`.

Code coverage tools

The code coverage is measured using the pytest-cov plugin which is based on coverage.py. The documentation of both projects is important when working on code coverage related issues. As an example, documentation for configuration can be first found in pytest-cov configuration but details about the coverage config file need to be looked up in coverage.py configuration docs.

9.2.2 Code quality

In order to keep a minimal level of "code quality" flake8 is used. To run the check:

```
$ flake8
```

Flake8 configuration file

Configuration for flake8 is under the `.flake8` file.

9.3 Building and viewing the documentation

Documentation for `honeybadgermpc` is located under the `docs/` directory. Sphinx is used to build the documentation, which is written using the markup language reStructuredText.

The `docker-compose.yml` can be used to quickly build the docs and view them.

To build the docs:

```
$ docker-compose up builddocs
```

To view the docs:

```
$ docker-compose up -d viewdocs
```

Visit http://localhost:58888/ in a web browser.

Tip: To view the port mapping you can use the command:

```
$ docker-compose port viewdocs 80
```

or, alternatively

```
$ docker-compose ps viewdocs
```

Tip: One may get a `403 Forbidden` error when trying to view the docs at http://localhost:58888/. This may because the generated html docs were removed. Using the `make clean` command under the `docs/` directory, e.g.:

```
$ docker-compose run --rm builddocs make -C docs clean
```

wipes out the `_build/` directory, and one has to restart the `viewdocs` (`nginx`) service, i.e.:

```
$ docker-compose restart viewdocs
```

and then re-build the docs:

```
$ docker-compose up builddocs
```

Or vice-versa: build the docs and restart the server.

9.3.1 Alternative ways to build and view the docs

To build the documentation, one can use the `Makefile` under the `docs/` directory:

```
$ make -C docs html
```

or

```
$ cd docs
$ make html
```

The `Makefile` makes use of the sphinx-build command, which one can also use directly:

```
$ sphinx-build -M html docs docs/_build -c docs -W --keep-going
```

It is possible to set some Sphinx environment variables when using the `Makefile`, and more particularly `SPHINXOPTS` via the shortcut `O`. For instance, to treat warnings as errors and to keep going with building the docs when a warning occurs:

```
$ O='-W --keep-going' make html
```

By default the generated docs are under `docs/_build/html/` and one can view them using a browser, e.g.:

```
$ firefox docs/_build/html/index.html
```

CHAPTER 10

Contributing new code

Since git and github are used to version and host the code, one needs to learn to work with both tools.

10.1 Suggested Git/Github workflow

A small example of a possible workflow is provided here. This is by no means a complete guide on how to work with Git and Github. The Pro Git book can be a very useful reference, whether one is a beginner or an advanced user.

10.1.1 Working with Git Remotes

First make sure your **git remotes** are properly set, and if not consult Configuring a remote for a fork or Git Basics - Working with Remotes for more detailed explanations about remotes. The remote names are just conventions but in order to simplify this documentation we'll adopt the conventions. So by convention, `upstream` should point to the "shared" repository, whereas `origin` should point to your fork. Use `git remote -v` to perform the check:

```
$ git remote -v
origin  git@github.com:<github_username>/HoneyBadgerMPC.git (fetch)
origin  git@github.com:<github_username>/HoneyBadgerMPC.git (push)
upstream        git@github.com:initc3/HoneyBadgerMPC.git (fetch)
upstream        git@github.com:initc3/HoneyBadgerMPC.git (push)
```

10.1.2 Identify the shared remote branch

What should be the base (remote) branch for your work? In many cases, if not most, it'll be the default dev branch, but in other cases you may need to base your work on some other branch, such as jubjub.

It is convenient to have a local copy of the remote shared branch that you need to work on. As an example, if you need to contribute work to the jubjub branch:

```
$ git fetch upstream
$ git checkout -b jubjub upstream/jubjub
```

In order to keep your local copy up-to-date you should periodically sync it with the remote. First switch to the local branch:

```
$ git fetch upstream
$ git rebase upstream/jubjub jubjub
```

There are multiple ways to work with remote branches. See https://git-scm.com/book/en/v2/Git-Branching-Remote-Branches for more information.

For a small discussion regarding the differences between rebasing and merging see https://git-scm.com/book/en/v2/Git-Branching-Rebasing#_rebase_vs_merge.

10.1.3 Create a new branch

Create a new branch from the shared remote branch to which you wish to contribute. As an example, say you are working on issue #23 (Implement jubjub elliptic curve MPC programs), then you could create a new branch like so:

```
$ git checkout -b issue-23-jujub-ec-mpc jubjub
```

You can name the branch whatever you like, but you may find it useful to choose a meaningful name along with the issue number you are working on.

10.1.4 Do you work, backup, and stay in sync

As you are adding new code, making changes etc you may want to push your work to your remote on Github, as this will serve as a backup:

```
$ git push origin issue-23-jujub-ec-mpc
```

In addtion to backing up your work on Github you should stay in sync with the shared remote branch. To do so, periodically `fetch` and `rebase`:

```
$ git fetch upstream
$ git rebase upstream/jubjub issue-23-jujub-ec-mpc
```

10.1.5 Git commit best practices

It is a good idea to familiarize yourself with good practices for the commits you make when preparing a pull request. A few references are provided here for the time being and as the `honeybadgermpc` project evolves we'll document good practices that are most relevant to the project.

- https://en.wikipedia.org/wiki/Separation_of_concerns
- https://wiki.openstack.org/wiki/GitCommitMessages#Information_in_commit_messages
- https://www.slideshare.net/TarinGamberini/commit-messages-goodpractices
- http://who-t.blogspot.com/2009/12/on-commit-messages.html

10.1.6 Signing commits

To sign your commits follow the steps outlined at https://help.github.com/articles/signing-commits/.

Resources

- https://git-scm.com/book/en/v2/Git-Tools-Signing-Your-Work
- https://softwareengineering.stackexchange.com/questions/212192/what-are-the-advantages-and-disadvantages-of-cryptographica

10.1.7 Making a pull request

Once you are done with your work, you have to push it to your remote:

```
$ git push origin issue-23-jujub-ec-mpc
```

and then you can make a pull request to merge your work with the shared remote branch that you have based your work on.

Pull requests go through the following checks:

- unit tests
- code quality
- documentation quality
- code coverage

These checks are performed using Travis CI and Codecov. These checks are there to help keeping the code in good shape and pull requests should ideally pass these 4 checks before being merged.

10.2 Tests

A pull request should ideally be accompanied by some tests. Code coverage is checked on Travis CI via codecov. The coverage requirements are defined in the `.codecov.yaml` file. See codecov's documentation on coverage configuration for more information about the codecov.yaml file.

pytest is the framework used to write tests and it is probably a good idea to consult its documentation once in a while to learn new tricks as it may help a lot when writing tests. For instance, learning to work with pytest fixtures can help greatly to simplify tests, and re-use test components throughout the test code.

Interesting resource on writing unit tests:: https://pylonsproject.org/community-unit-testing-guidelines.html

10.3 Coding Conventions

PEP 8 is used as a guide for coding conventions. The maximum line length is set at 89 characters.

The flake8 tool is used in the continuous integration phase to check the code quality. The configuration file, `.flake8`, is under the project root.

Tip: Recommended reading: The Code Style section in the The Hitchhiker's Guide to Python!.

10.4 Documentation Conventions

PEP 257 is used for docstring conventions. The docstrings are extracted out into the documentation with the autodoc Sphinx extension and should be valid reStructuredText. Here's an example of how a function may be documented:

Todo: Use a HoneyBadgerMPC code sample instead of sample shown below.

```
def send_message(sender, recipient, message_body, [priority=1]):
    """Send a message to a recipient

    :param str sender: The person sending the message
    :param str recipient: The recipient of the message
    :param str message_body: The body of the message
    :param priority: The priority of the message, can be a number 1-5
    :type priority: integer or None
    :return: the message id
    :rtype: int
    :raises ValueError: if the message_body exceeds 160 characters
    :raises TypeError: if the message_body is not a basestring
    """
```

See Sphinx documentation: info field lists, for more information on how to document Python objects.

Tip: **Recommended reading:** The Documentation section in the The Hitchhiker's Guide to Python! is a useful resource.

10.5 Ignoring conventions

The PEP 8 style guide has a very important section at the beginning: A Foolish Consistency is the Hobgoblin of Little Minds. It says:

> *One of Guido's key insights is that code is read much more often than it is written. The guidelines provided here are intended to improve the readability of code and make it consistent across the wide spectrum of Python code. As* PEP 20 *says, "Readability counts".*
>
> *A style guide is about consistency. Consistency with this style guide is important. Consistency within a project is more important. Consistency within one module or function is the most important.*
>
> *However, know when to be inconsistent—sometimes style guide recommendations just aren't applicable. When in doubt, use your best judgment. Look at other examples and decide what looks best. And don't hesitate to ask!*

So if you need to ignore some convention(s), and doing so make one or more checks fail you can ignore the error inline with the `# noqa: <error code>` comment. As an example, say you wanted to ignore `E221` (multiple spaces before operator) errors:

```
coin_recvs = [None] * N
aba_recvs  = [None] * N  # noqa: E221
rbc_recvs  = [None] * N  # noqa: E221
```

See Selecting and Ignoring Violations for more information about ignoring violations reported by flake8.

Error codes

- flake8: http://flake8.pycqa.org/en/latest/user/error-codes.html
- pycodestyle: https://pycodestyle.readthedocs.io/en/latest/intro.html#error-codes
- pydocstyle: http://www.pydocstyle.org/en/2.1.1/error_codes.html

10.6 Logging

Make use of the `logging` module! If you are unsure about whether you should log or print, or when you should log, see When to use logging.

10.6.1 Important resources on logging

- Python documentation: Logging HOWTO
- Python documentation: Logging Cookbook
- The Hitchhiker's Guide to Python!: Logging
- Plumber Jack Stuff about Python's logging package. By Vinay Sajip, main author of the `logging` module.

10.7 Rust bindings

Todo: Document important things to know when contributing to this component.

10.8 References

- Pro Git Book
- The Hitchhiker's Guide to Python!
- On the role of scientific thought by Edsger W. Dijkstra

CHAPTER 11

Reviewing and merging pull requests

Todo: Document some guidelines when reviewing and merging pull requests. See https://cryptography.io/en/latest/development/reviewing-patches/#reviewing-and-merging-patches as an inspiration.

11.1 Merge Requirements

Todo: Come up with merge requirements with the team.

We can use the following from the cryptography project as an inspiration:

- Patches must *never* be pushed directly to `master|dev`, all changes (even the most trivial typo fixes!) must be submitted as a pull request.
- A committer may *never* merge their own pull request, a second party must merge their changes. If multiple people work on a pull request, it must be merged by someone who did not work on it.
- A patch that breaks tests, or introduces regressions by changing or removing existing tests should not be merged. Tests must always be passing on `master|dev`.
- If somehow the tests get into a failing state on `master|dev` (such as by a backwards incompatible release of a dependency) no pull requests may be merged until this is rectified.
- All merged patches must have 100% test coverage.

Merge vs Rebase vs Squash

Todo: Explain the difference between the three and which method should be used. Basically, propose to use rebase or squash in order to simplify the git history. Find a good resource on the topic.

11.2 References

- https://github.community/t5/Support-Protips/Best-practices-for-pull-requests/ba-p/4104
- https://help.github.com/articles/about-pull-request-reviews/A

CHAPTER 12

Continuous integration

Continuous Integration (CI) is a development practice that requires developers to integrate code into a shared repository several times a day. Each check-in is then verified by an automated build, allowing teams to detect problems early.

By integrating regularly, you can detect errors quickly, and locate them more easily.

—ThoughtWorks

`honeybadgermpc` currently uses Travis CI to perform various checks when one wishes to merge new code into a shared branch under the shared repository initc3/HoneyBadgerMPC. The file `.travis.yml` under the root of the project is used to instruct Travis CI on what to do whenever a build is triggered.

12.1 .travis.yml

Whenever a build is triggered, three checks are currently performed:

1. tests
2. code quality via flake8. and
3. documentation generation.

Each of these checks corresponds to a row in the build matrix:

```
matrix:
  include:
    - env: BUILD=tests
    - env: BUILD=flake8
    - env: BUILD=docs
```

Depending on the value of the `BUILD` variable the various steps (e.g.: install, script) of the build lifecycle may differ.

Using Python 3.7 on Travis CI

In order to use Python 3.7 the following workaround is used in `.travis.yml`:

```
os: linux
dist: xenial
language: python
python: 3.7
sudo: true
```

See currently opened issue on this matter: https://github.com/travis-ci/travis-ci/issues/9815

Using Docker on Travis CI

In order to use Docker the following settings are needed in `travis.yml`:

```
sudo: true

services:
  - docker
```

See *.travis.compose.yml* below for more information on how we use `docker` and `docker-compose` on Travis CI to run the tests for `honeybadgermpc`.

12.2 Shell scripts under .ci/

In order to simplify the `.travis.yml` file, shell scripts are invoked for the `install`, `script` and `after_success` steps. These scripts are located under the `.ci` directory and should be edited as needed but with care since it is important that the results of the checks be reliable.

12.3 .travis.compose.yml

For the `docs` and `tests` build jobs (i.e.: `BUILD=docs` and `BUILD=tests` matrix rows), docker-compose is used. The `Dockerfile` used is located under the `.ci/` directory whereas the `docker-compose` file is under the root of the project and is named `.travis.compose.yml`. Both files are similar to the ones used for development. One key difference is that only the docs or tests requirements are installed, depending on the value of the `BUILD` environment variable.

Note: Some work could perhaps be done to limit the duplication accross the two Dockerfiles, by using a base Dockerfile for instance, but this may also complicate things so for now some duplication is tolerated.

12.4 Code coverage

Code coverage is used to check whether code is executed when the tests are run. Making sure that the code is executed when tests are run helps detecting errors.

In the `tests` build job on Travis CI a code coverage report is generated at the end of the `script` step, with the `--cov-report=xml` option:

```
# .ci/travis-install.sh
$BASE_CMD pytest -v --cov --cov-report=term-missing --cov-report=xml
```

If the test run was successful the report is uploaded to codecov in the `after_success` step:

```
# .travis.yml
after_success: .ci/travis-after-success.sh
```

Important: It is important to note that the coverage measurement happens in a docker container meanwhile the report upload happens outside the container. There are different ways to handle this situation and the current approach used is a variation of what is outlined in Codecov Outside Docker.

12.4.1 Configuration

Configuring codecov is done via the `.codecov.yml` file which is in the project root. Consult the codecov documentation for information on how to work with the `.codecov.yml` configuration file. The most relevant sections are About the Codecov yaml and Coverage Configuration.

12.4.2 Github integration

A pull request may fail the code coverage check and if so the pull request will be marked as failing on Github. The Github integration may require having a team bot set up to be fully operational. See issue https://github.com/initc3/HoneyBadgerMPC/issues/66 for more details.

12.5 Recommended readings

- Travis CI: Core Concepts for Beginners
- ThoughtWorks: Continuous Integration
- https://docs.python-guide.org/scenarios/ci/

CHAPTER 13

Packaging and Distributing HoneyBadgerMPC

Todo: Document how to package and distribute `honeybadgermpc` to PyPi.

- https://packaging.python.org/
- https://docs.python-guide.org/shipping/packaging/

Making a PyPI-friendly README

Todo: Explain the importance of keeping the README file "PyPI-friendly".

https://packaging.python.org/guides/making-a-pypi-friendly-readme

CHAPTER 14

Troubleshooting

Some problems may sometimes be resolved by deleting some cached files. You can use the *Makefile* to remove such files:

```
$ make clean
```

Using *docker-compose*:

```
$ docker-compose run --rm honeybadgermpc make clean
```

14.1 Debugging `asyncio`

14.2 `sys.path`/`PYTHONPATH` issues

14.2.1 `ModuleNotFoundError`

If you get an error similar to,

```
    @fixture
    def GaloisField():
>       from honeybadgermpc.field import GF
E       ModuleNotFoundError: No module named 'honeybadgermpc'

tests/conftest.py:31: ModuleNotFoundError
```

when running some tests.

Try running `make clean`, just in case some outdated cached files would be causing the error.

If you are running the tests locally (not in a docker container), you may not have installed the `honeybadgermpc` package, i.e.:

```
pip install --editable .[dev]
```

Make sure to include the option `--editable` or its short verion `-e` so that the project is installed in development mode.

14.2.2 Relevant links

- stackoverflow: PATH issue with pytest 'ImportError: No module named YadaYadaYada'
- pytest import mechanisms and sys.path/PYTHONPATH
- pytest Good Integration Practices

CHAPTER 15

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index