PySNARK Documentation

Release

Meilof Veeningen

Jan 10, 2018

CONTENTS:

PySNARK is a Python-based system for performing verifiable computations based on the Pinocchio system and the Geppetri extension for proofs on authenticated data.

CHAPTER

ONE

LICENSE

Some files in this package have their own copyright conditions. Please refer to the respective files for details:

- everything under `qaptools/xbyak` is copyright by MITSUNARI Shigeo, see `qaptools/xbyak/COPYRIGHT`
- everything under `qaptools/ate-pairing` is released under the BSD 3-Clause License, see `qaptools/ate-pairing/readme.md`
- `pysnark/contracts/Pairing.sol` is copyright by Christian Reitwiessner, see the file itself
- `examples/DRBG.py` is copyright by David Lazar, see the file itself

CHAPTER

TWO

INSTALLATION GUIDE

2.1 Unix (Linux/MacOS/. . .)

First, download the PySNARK dependencies, `ate-pairing` and `xbyak`:

```
git submodule init
git submodule update
```

Build the `ate-pairing` library:

```
cd qaptools/ate-pairing
make SUPPORT_SNARK=1
```

Build the `qaptools` library:

```
cd qaptools
make
```

Build and install the *pysnark* library:

```
python setup.py install
```

2.2 Windows

PySNARK comes with precompiled Windows executables of `qaptools`, meaning it is possible to build an install PySNARK by just running:

```
python setup.py install
```

To recompiling `qaptools` from source, set up a Unix-like build environment such as Mingw with MSYS and use the Unix instructions above.

2.3 Using without installation

It is also possible to run PySNARK applications without installing PySNARK. For this, follow the above steps but run `python setup.py build` instead of `python setup.py install`. This makes sure all files are compiled and put in their correct locations. Then, run the application with the `PYTHONPATH` environment variable set to the PySNARK library, e.g.:

```
PYTHONPATH=/path/to/pysnark/sources python script.py
```

CHAPTER

THREE

THE PYSNARK TOOLCHAIN

We discuss the usage of the PySNARK toolchain based on running one of the provided examples acting as each of the different types of parties in a verifiable computation: trusted party, prover, or verifier.

3.1 As trusted party

To try out running PySNARK as trusted party performing key generation, do the following:

```
cd examples
python cube.py 3
```

If PySNARK has been correctly installed, this will perform a verifiable computation that will compute the cube of the input value, `3`. At the same time, it will generate all key material needed to verifiably perform the computation in the script. (Performing an example computation is the only way to generate this key material.) PySNARK produces the following files:

- **Files that should be kept secret by the trusted party generating the key material:**
  - `pysnark_mastersk`: zk-SNARK master secret key
- **Files that the trusted party should distribute to provers along with the Python script (i.e., *cube.py* in this case):**
  - `pysnark_schedule`: schedule of functions called in the computation
  - `pysnark_masterek`: master evaluation key
  - `pysnark_ek_main`: zk-SNARK evaluation key for the main function of the computation
  - `pysnark_eqs_main`: equations for the main function of the computation
- **Files that the trusted party should distribute to verifiers:**
  - `pysnark_schedule`: schedule of functions called in the computation
  - `pysnark_masterpk`: master public key
  - `pysnark_vk_main`: verificaiton key for the main function
- **Files that the prover should distribute to verifiers:**
  - `pysnark_proof`: proof that the particular computation was performed correctly
  - `pysnark_values`: input/output values of the computation
- **Files that are not needed anymore after the execution:**
  - `pysnark_eqs`: equations for the zk-SNARK

    – pysnark_wires: wire values of the computation

Note that the PySNARK master evaluation key (pysnark_mastersk) and master public key (pysnark_masterpk) grow as required. As a consequence, if a small computation is executed after a large computation, these files are larger than needed for the small computation. PySNARK will indicate this with a warning such as the following:

```
** Evaluation/public keys larger than needed for function: 128>8 or 2>2.
** To re-create, remove pysnark_masterek and pysnark_masterpk and run again.
```

To obtain smaller keys that are sufficient for distribution to provers that only need to execute the small computation, remove the master evaluation and public key (but not the master secret key). The master evaluation and public key will be re-generated and will now have the minimum size needed to succesfully execute the small computation.

3.2 As prover

To try out running PySNARK as a prover, put the files discussed above (i.e., pysnark_schedule, pysnark_masterek, pysnark_ek_main, and pysnark_eqs_main) together with cube.py in a directory and run the same command:

```
cd examples
python cube.py 3
```

This will perform a verifiable computation based on the previously generated key material.

3.3 As verifier

To try out running PySNARK as a verifier, put the files discussed above (i.e., pysnark_schedule, pysnark_masterpk and pysnark_vk_main received from the trusted party, and pysnark_proof and pysnark_values received from the prover) in a folder and run:

```
PYTHONPATH=../.. python -m pysnark.qaptools.runqapver
```

This will verify the computation proof with respect to the input/output values from the pysnark_values file, e.g,:

```
# PySNARK i/o
main/o_in: 21
main/o_out: 9261
```

In this case, we have verifiably computed the fact that the cube of 21 is 9261. See also the other provided examples.

3.4 Environment variables

Operation of the PySNARK toolchain can be configured by means of the following environment variables:

- PYSNARK_ENABLED – if set to another value than 1, this will disable the PySNARK runtime: no pysnark_* files will be written (except for *pysnark.runtime.exportcomm()*, which will still write a wire file but no commitment; this will also not disable function calls outside of *pysnark.runtime* such as *pysnark.qaptools.runqapinput.gencomm()*)

- `PYSNARK_REBUILD` – if set to another value than `0`, no key material will be (re)generated (normally, key material is generated either if a master secret key is present, or there is no PySNARK key material at all) but proofs will stilbe produced
- `PYSNARK_KEYDIR` – if set, all PySNARK key material will be written to and read from the given directory
- `PYSNARK_PROOFDIR` – if set, all PySNARK proofs, wires, and other material relating to the present computation will be written to and read from the given directory
- `QAPTOOLS_BIN` – if set, the `qaptools` binaries from the given directory will be used

3.5 Using commitments

PySNARK allows proofs to refer to committed data using Geppetri. This has three applications:

- it allows proofs to refer to external private inputs from parties other than the trusted third party;
- it allows different verifiable computations to share secret data with each other; and
- it allows to divide a verifiable computation into multiple subcomputations, each with their own evaluation and verification keys (but all based on the same master secret key)

See `examples/testcomm.py` for examples.

3.5.1 External secret inputs

To commit to data, use *`pysnark.qaptools.runqapinput`*, e.g., to commit to values 1, 2, and 3 using a commitment named `test`, use:

```
python -m pysnark.qaptools.runqapinput test 1 2 3
```

Share `pysnark_wires_test` with any prover who wants to perform a computation with respect to this committed data, and `pysnark_comm_test` to any verifier. Alternatively, use *`pysnark.qaptools.runqapinput.gencomm()`* from a Python script.

Import this data into the verifiable computation with:

```
[one,two,three] = pysnark.runtime.importcomm("test")
```

3.5.2 Sharing data between verifiable computations

In the first computation, do:

```
pysnark.runtime.exportcomm([Var(1),Var(2),Var(3)], "test")
```

and share `pysnark_wires_test` and `pysnark_comm_test` with the other prover and the verifier, respectively.

In the second verifiable computation, do:

```
[one,two,three] = pysnark.runtime.importcomm("test")
```

3.5.3 Sharing data between different parts of a verifiable computation

This is implicitly used whenever a function is called that is decorated with `@pysnark.runtime.subqap`. When a particular functon is used multiple times in a verifiable computation, using `@pysnark.runtime.subqap` prevents the circuit for the function to be replicated, resulting in smaller key material (but slower verification).

CHAPTER

FOUR

PYSNARK & SMART CONTRACTS

PySNARK supports the automatic generation of smart contracts that verify the correctness of the given zk-SNARK. These smart contracts are written in Solidity and require support for the recent zkSNARK verification opcodes (EIP 196, EIP 197). To test them out, install a development version of Truffle using these instructions.

Continuing the above example, as verifier first run:

```
truffle init
```

to initialise Truffle (to just see the Solidity code without installing truffle, create two empty directories `contracts` and `test`).

Next, run:

```
python -m pysnark.contract
```

to generate smart contract `contracts/Pysnark.sol` to verify computations of the `cube.py` script (using library `contracts/Pairing.sol` that is also copied into the directory), and test script `test/TestPysnark.sol` that gives a test case for the contract based on the current I/O and proof. Finally, run:

```
truffle test
```

to run the test script and check that the given proof can indeed be verified in Solidity.

Note that `test/TestPysnark.sol` indeed contains the I/O from the computation:

```
pragma solidity ^0.4.2;

import "truffle/Assert.sol";
import "../contracts/Pysnark.sol";

contract TestPysnark {
    function testVerifies() public {
        Pysnark ps = new Pysnark();
        uint[] memory proof = new uint[](22);
        uint[] memory io = new uint[](2);
        proof[0] = ...;
        ...
        proof[21] = ...;
        io[0] = 21; // main/o_in
        io[1] = 9261; // main/o_out
        Assert.equal(ps.verify(proof, io), true, "Proof should verify");
    }
}
```

4.1 Commitments in smart contracts

Smart contracts can also refer to commitments, e.g., as imported with the *pysnark.runtime.importcomm()* API call. In this case, the commitment becomes an argument to the verification function (a six-valued integer array), and the test case shows how the commitment used in the present computation should be used as value for that argument, e.g.:

```
pragma solidity ^0.4.2;

import "truffle/Assert.sol";
import "../contracts/Pysnark.sol";

contract TestPysnark {
    function testVerifies() public {
        Pysnark ps = new Pysnark();
        uint[] memory pysnark_comm_test = new uint[](6);
        pysnark_comm_test[0] = ...;
        ...
        Assert.equal(ps.verify(proof, io, pysnark_comm_test), true, "Proof should␣
↪verify");
    }
}
```

4.2 Running out of gas

Wen testing PySNARK smart contracts using Truffle (especially when using commitments or large amounts of I/O), one may get the message that Truffle runs out of gas. In this case, it is possible to increase Truffle's gas limit by editing the `deployer.deploy` line in `migrations/1_initial_migration.js`, e.g.:

```
deployer.deploy(Migrations, {gas: 6700000});
```

CHAPTER

FIVE

PROGRAMMING GUIDE

This section discusses how to specify verifiable computations using PySNARK.

5.1 Introduction to verifiable computation

In a verifiable computation, a prover proves to a verifier that certain I/O known by the verifier satisfies a certain relation. For instance, the I/O may be the two values 2 and 8 and the relation may be that the second I/O value is the cube of the first I/O value. The prover creates a crypgraphic proof (using an evaluation key depending on the computation performed and generated by a trusted third party) that the verifier uses to verify (using a verification key generated by the same trusted third party) that the I/O it received (2 and 8, in this case), satisfy the given relation (in this case, that the second I/O value is the cube of the first I/O value).

Apart from referring to public I/O, the proof may also refer to secret values known only to the verifier, so-called witnesses. For instance, the public I/O may be the image of a cryptographic hash function (see `pysnark.lib.ggh`) in which case the prover may prove that it knows the pre-image that gives te resulting value when applying the hash function. In this case, the pre-image is the witness. Note that PySNARK will also generate many witnesses implicitly during the computation, e.g., to compute `y=x*x*x` PySNARK will actually internally create a witness `z` such that `z=x*x` and `y=x*z`.

Finally, the prover may in a proof refer to externally committed data, typically provided by a third party other than the trusted third party or the prover itself. For instance, party A publishes a commitment to a sensitive dataset, allowing B (who has access to the dataset) to prove properties about this dataset to party C (who does not learn the dataset, only that B's proof was correct with respect to the data committed to by A), all based on key material generated once by a trusted third party D.

5.2 Basics: variables and proof generation

Verifiable computations are programmed using the PySNARK runtime, `pysnark.runtime`.

Values used in a verifiable computation are called variables and are instances of `pysnark.runtime.Var`. Instances of this class can represent I/O, witnessess, or committed data.

To create an I/O variable with a given value, use `pysnark.runtime.Var.__init__()` with a value and optionally a name for the variable (this name will appear in the I/O file that PySNARK outputs), e.g.:

```
from pysnark.runtime import Var

in = Var(int(sys.argv[1]), "in")
```

PySNARK variables can be manipulated using normal operators, e.g., for addition, multiplication, and sutraction. (These computations are actually performed modulo a large prime number `pysnark.options.vc_p` but this usually does not make a difference in practice.) For instance, one computes the cube of `in` as follows:

```
cube = in*in*in
```

Here, `cube` is a witness of the computation. To output this value, i.e., to turn it into a piece of I/O, use the `pysnark.runtime.Var.val()` function, optionally with a name to use for the variable:

```
print "The cubed value is", cube.val("out")
```

To create a witness, call `pysnark.runtime.Var.__init__()` with `True` as variable name, e.g.:

```
wit = Var(33, True) # this is not I/O
```

Just instantiating Var itself does not cause PySNARK to create verifiable computation proofs. For this, import the `pysnark.prove` module: importing this module will register an exit handler causing PySNARK to build and verify a proof when the Python script terminates.

5.3 Division into subcomputations

Use the `pysnark.runtime.subqap()` decorator.

5.4 How PySNARK works internally

5.5 Keeping computations secret

If the computation is supposed to be secret, use `pysnark.runtime.Var.constant()` instead of plain constants: the values of any constants applied to I/O values can be derived from the public verification key. Also, keep the verification key secret.

CHAPTER

SIX

PYSNARK PACKAGE DOCS

6.1 Subpackages

6.1.1 pysnark.lib package

Submodules

pysnark.lib.array module

`class pysnark.lib.array.`**`Array`**`(`*vals*`)`

> **`__add__`**`(`*other*`)`
>
> **`__getitem__`**`(`*item*`)`
>
> **`__init__`**`(`*vals*`)`
>
> **`__module__ = 'pysnark.lib.array'`**
>
> **`__radd__`**`(`*other*`)`
>
> **`__repr__`**`()`
>
> **`__rmul__`**`(`*other*`)`
>
> **`__setitem__`**`(`*item, value*`)`
>
> **`__sub__`**`(`*other*`)`
>
> **`assert_equals`**`(`*other*`)`
>
> **`joined`**`()`

pysnark.lib.base module

`pysnark.lib.base.`**`if_then_else`**`(`*cond, trueval, falseval*`)`

Returns one of two values depending on choice bit :param cond: Choice bit (function does not ensure that this is a bit) :param trueval: Value if choice bit is 1 :param falseval: Value if choice bit is 0 :return: Value given by choice bit

`pysnark.lib.base.`**`input_bit_array`**`(`*bits, nm=None*`)`

Imports bitstring as a single input of the program. This checks that all provided values are actually bits.

Parameters

- **`bits`** – String consisting of (at most 253) 0s and 1s (starting with most significant)
- **`nm`** – Name of input variable (None for automatic)

Returns Array of bits

`pysnark.lib.base.`**`lin_comb`**`(`*cofs*, *vals*`)`

Returns linear combination of given values with given coefficients :param cofs: Array of variable coefficients :param vals: Array of variable values :return: Variable representing the linear combination

`pysnark.lib.base.`**`lin_comb_pub`**`(`*cofs*, *vals*`)`

Returns linear combination of given values with given coefficients. This can be executed more efficiently than computing the sum by hand but introduces an additional equation to the program.

Parameters

- **`cofs`** – Array of integer coefficients
- **`vals`** – Array of variable values

Returns Variable representing the linear combination

`pysnark.lib.base.`**`output_bit_array`**`(`*bits*, *nm=None*`)`

Exports bitstring as a single output of the QAP. This does not check that all provided values are actually bits; use *`pysnark.runtime.Var.assert_bit()`* for that.

Parameters

- **`bits`** – Array of Vars representing bits
- **`nm`** – Name of output variable (None for automatic)

Returns Bitstring representing the value

pysnark.lib.fixedpoint module

`class` `pysnark.lib.fixedpoint.`**`VarFxp`**`(`*val*, *sig=None*`)`

Bases: *`pysnark.runtime.Var`*

Variable representing a fixed-point number. Number x is represented as integer $x * 2^r$, where r is the resolution *`VarFxp.res`*

`__add__``(`*other*`)`

`__div__``(`*other*`)`

Fixed-point division. This is an expensive operation: it costs approximately *`maxden()`* equations :param other: Other fixed-point number :return: Result of division

`__init__``(`*val*, *sig=None*`)`

Constructor, see `Var.__init__()`

Parameters

- **`val`** – Value for the variable; accepts floats or ints (interpr
- **`sig`** – See `Var.__init__()`

`__module__ = 'pysnark.lib.fixedpoint'`

`__mul__``(`*other*`)`

`__neg__``()`

`__repr__``()`

__rsub__(*other*)

__str__()

__sub__(*other*)

floatval()
Returns floating-point value represented by this variable.

Returns value

classmethod fromvar(*var*)
Convers a non-fixed-point variable to fixed point

Parameters **var** – A non-fixed-point variable

Returns A new fixed-point variable representing the same value

classmethod fromvar_noconv(*var*)

maxden = 40
Maximal length of denominiators for `__div__()`, including resolution

res = 20
Resulution for fixed-point numbers

val(*nm=None*)

pysnark.lib.ggh module

This module provides efficient hashes using the "Ajtai-GGH" hash function due to Ajtai and Goldreich/Goldwasser/Halevi (Goldreich, Goldwasser, Halevi, "Collision-Free Hashing from Lattice Problems").

This implementation uses parameters N=64, Q=524288, M=7296 and translates a 7296-bit input into a 1216-bit output (i.e., it has a compression ratio of 6). These parameters are as suggested by Chris Peikert and used in Pepper, see here.

The coefficients of the hash function have been generated using a PRF.

pysnark.lib.ggh.**ggh_hash**(*plain*)
Computes a GGH hash of the given input bits. This function does not ensure that the inputs are actually bits, but it guarantees that the outputs are bits

Parameters **plain** – Plaintext: array of 7296 bits

Returns Hash: array of 1216 bits

pysnark.lib.ggh.**ggh_hash_packed**(*plain_packed*)

pysnark.lib.ggh_plain module

pysnark.lib.ggh_plain.**fromint**(*val*, *len*)

pysnark.lib.ggh_plain.**ggh_hash**(*plain*)

pysnark.lib.ggh_plain.**packin**(*vals*)

pysnark.lib.ggh_plain.**packout**(*vals*)

pysnark.lib.ggh_plain.**toint**(*vals*)

pysnark.lib.ggh_plain.**unpackin**(*vals*)

pysnark.lib.ggh_plain.**unpackout**(*vals*)

Module contents

6.1.2 pysnark.qaptools package

Submodules

pysnark.qaptools.runqapgen module

`pysnark.qaptools.runqapgen.`**`ensure_mkey`**`(`*eksize*, *pksize*`)`

Ensures that there are master evaluation and public keys of the given sizes.

If master evaluation/public keys exist but are to small, and there is no master secret key, this raises an error.

If there is no key material at all, a fresh master secret key will be generated.

Parameters

- **`eksize`** – Minimal evaluation key size (-1 if not needed)
- **`pksize`** – Minimal public key size (-1 if not needed)

Returns Actual evaluation key, public key size after key generation

`pysnark.qaptools.runqapgen.`**`get_mekey_size`**`()`

Get the size (maximal exponent) of the current master evaluation key

Returns Size, or -1 if key does not exist

`pysnark.qaptools.runqapgen.`**`get_mpkey_size`**`()`

Get the size (maximal exponent) of the current master public key

Returns Size, or -1 if key does not exist

`pysnark.qaptools.runqapgen.`**`run`**`(`*eksize*, *pksize*, *genmk=False*`)`

Run the qapgen tool

Parameters

- **`eksize`** – Desired master evaluation key size
- **`pksize`** – Desired master public key size
- **`genmk`** – True if a new master secret key should be generated, False otherwise

Returns None

pysnark.qaptools.runqapgenf module

`pysnark.qaptools.runqapgenf.`**`ensure_ek`**`(`*nm*, *sig*, *eksz*`)`

Ensure that up-to-date evaluation key for the function is available, corresponding to the given signature

Parameters

- **`nm`** – Function name
- **`sig`** – Signature as returned by *`pysnark.qapsplit.qapsplit()`*
- **`eksz`** – Function size as returned by *`pysnark.qapsplit.qapsplit()`*

Returns None

`pysnark.qaptools.runqapgenf.`**`get_ekfile_sig`**`(`*ekfile*`)`

Get function signature from evaluation key file

Parameters **ekfile** – Evaluation key file

Returns Function signature (first token in the file), or empty string if file does not exist

`pysnark.qaptools.runqapgenf.`**`run`**(*nm*, *sig*, *sz=None*)

Run the qapgenf tool to generate evaluation/verification keys for the given function.

Parameters

- **nm** – Function name to generate key material for
- **sig** – Signature of the function (as returned by *`pysnark.qapsplit.qapsplit()`*
- **sz** – If None, use the master secret key; else, use the coefficient cache of the given size

Returns None

pysnark.qaptools.runqapinput module

`pysnark.qaptools.runqapinput.`**`gencomm`**(*blocknm*, *vals*, *rnd=None*)

Generate commitment file and commitment

Parameters

- **blocknm** – Block name
- **vals** – List of integer values to commit to
- **rnd** – Randomness for the commitment (or generate if not given)

Returns None

`pysnark.qaptools.runqapinput.`**`run`**(*bname*)

Run the qapinput tool to build a commitment file representing some data. The input file (given by *`pysnark.options.get_block_file()`*) consists of one value per line, plus a last line of randomness. The output file generated is given by *`pysnark.options.get_block_comm()`*.

Parameters **bname** – Block name

Returns None

`pysnark.qaptools.runqapinput.`**`writecomm`**(*blocknm*, *vals*, *rnd=None*)

Write values to a commitment file

Parameters

- **bfile** – Block name
- **data** – List of integer values to commit to
- **rnd** – Randomness for the commitment (or generate if not given)

Returns None

pysnark.qaptools.runqapprove module

`pysnark.qaptools.runqapprove.`**`run`**()

Run the qapprove tool to generate a computation proof

Returns None

pysnark.qaptools.runqapver module

`pysnark.qaptools.runqapver.`**`getcommand`**`()`

Returns the command line to the qapver tool

Returns Command line

`pysnark.qaptools.runqapver.`**`run`**`()`

Run the qapver tool to verify a computation proof

Returns The command line to the "qapver" tool

Module contents

6.2 Submodules

6.2.1 pysnark.atexitmaybe module

`class` `pysnark.atexitmaybe.`**`ExitOverrider`**

Bases: `object`

`__dict__ = dict_proxy({'__module__': 'pysnark.atexitmaybe', '__dict__': <attribute '`

`__init__``()`

`__module__ = 'pysnark.atexitmaybe'`

`__weakref__`

list of weak references to the object (if defined)

`excepthook``(`*tp*, *ex*, **args*`)`

`exit``(`*exitcode=0*`)`

`pysnark.atexitmaybe.`**`maybe`**`(`*fn*`)`

6.2.2 pysnark.contract module

`class` `pysnark.contract.`**`QapVk`**`(`*fn*`)`

`__init__``(`*fn*`)`

`__module__ = 'pysnark.contract'`

`pysnark.contract.`**`contract`**`()`

`pysnark.contract.`**`readg1`**`(`*fl*`)`

`pysnark.contract.`**`readg2`**`(`*fl*`)`

`pysnark.contract.`**`strg1`**`(`*val*`)`

`pysnark.contract.`**`strg1p`**`(`*ix*`)`

`pysnark.contract.`**`strg1pp`**`(`*tp*, *ix*`)`

`pysnark.contract.`**`strg2`**`(`*val*`)`

`pysnark.contract.`**`strg2p`**`(`*ix*`)` `pysnark.contract.`**`strg2pp`**`(`*tp*, *ix*`)`

`pysnark.contract.`**`tog1`**`(`*tok*`)`

`pysnark.contract.`**`tog2`**`(`*tok*`)`

6.2.3 pysnark.import module

6.2.4 pysnark.options module

`pysnark.options.`**`do_proof`**`()`

`pysnark.options.`**`do_pysnark`**`()`

`pysnark.options.`**`do_rebuild`**`()`

`pysnark.options.`**`get_block_comm`**`(`*bname*`)`

`pysnark.options.`**`get_block_file`**`(`*bname*`)`

`pysnark.options.`**`get_cache_file`**`(`*sz*`)`

`pysnark.options.`**`get_contract_dir`**`()`

`pysnark.options.`**`get_conttest_dir`**`()`

`pysnark.options.`**`get_ek_file`**`(`*fn*`)`

`pysnark.options.`**`get_eqs_file`**`()`

`pysnark.options.`**`get_eqs_file_fn`**`(`*fn*`)`

`pysnark.options.`**`get_io_file`**`()`

`pysnark.options.`**`get_mkey_file`**`()`

`pysnark.options.`**`get_mpkey_file`**`()`

`pysnark.options.`**`get_mskey_file`**`()`

`pysnark.options.`**`get_proof_file`**`()`

`pysnark.options.`**`get_qaptool_exe`**`(`*tool*`)`

`pysnark.options.`**`get_schedule_file`**`()`

`pysnark.options.`**`get_vk_file`**`(`*fn*`)`

`pysnark.options.`**`get_wire_file`**`()`

`pysnark.options.`**`vc_p = 2188824287183927522224640574525727508854836440041603434369820418657`**

The modulus used in the verifiable computation. All computations are performed using modular arithmetic with this modulus.

6.2.5 pysnark.prove module

`pysnark.prove.`**`prove`**`()`

6.2.6 pysnark.qapsplit module

pysnark.qapsplit.**contextualize**(*lst*)

pysnark.qapsplit.**getqap**(*nm*)

pysnark.qapsplit.**qapsplit**()

Returns (maximum qap size, maximum input block size) encountered

6.2.7 pysnark.runtime module

class pysnark.runtime.**Var**(**args*, ***kwargs*)
A variable of the verifiable computation

__add__(*other*)
Add VcShare or constant to self.

__div__(*other*)

__init__(**args*, ***kwargs*)

__module__ = 'pysnark.runtime'

__mul__(*other*)
Multiply VcShare with other VcShare or constant.

__neg__()
Returns negated VcShare.

__radd__(*other*)
Add VcShare or constant to self.

__repr__()
Return string representation of this VcShare.

__rmul__(*other*)
Multiply VcShare with other VcShare or constant.

__rsub__(*other*)

__sub__(*other*)
Subtract VcShare or constant from self.

assert_bit()
Assert that this variable contains a bit, i.e., 0 or 1 :return: None

assert_equals(*other*)

assert_nonzero()

assert_positive(*bl*)
Assert that the present VcShare represents a positive value, that is, a value in [0,2^bl] with bl the given bit length.

assert_smaller(*val*)

assert_zero()
Assert that the present VcShare represents the value zero.

bit_decompose(*bl*)
Assert that the present VcShare represents a positive value, that is, a value in [0,2^bl] with bl the given bit length.

classmethod constant(*val*)
Return a VcShare representing the given constant value.

classmethod constname(**args*, ***kwargs*)

divmod(*divisor*, *maxquotbl*)
Divide by public value and return quotient and remainder :param divisor: Divisor (integer) :param maxquotbl: Maximal bitlength of the resulting quotient :return: Quotient and remainder

ensure_single()
Return a VcShare with the same value that is guaranteed to refer to one witness, by making a new VcShare and adding the required equation if necessary.

equals(*other*)

isnonzero()
Returns VcShare equal to 1 if self is not zero, and 0 if self is zero.

iszero()

classmethod random()
Return a VcShare representing a random value.

strsig()
Return string representation of linear combination represented by this VcShare.

classmethod tovar(*val*, *nm=None*)

val(**args*, ***kwargs*)

classmethod vals(*vars*, *nm*)

classmethod vars(*vals*, *nm*, *dim=1*)

classmethod zero()
Return a VcShare representing the value zero.

`pysnark.runtime.`**continuefn**(**args*, ***kwargs*)

`pysnark.runtime.`**enterfn**(*fname*, *call=None*)
Start a new call of the given function type :param fname: Function name. All instances of the same function should execute the exact same sequence of instructions :param call: Call name. Should be globally unique (autogenerated if not given) :return: Call name

`pysnark.runtime.`**exportcomm**(**args*, ***kwargs*)

`pysnark.runtime.`**for_each_in**(*cls*, *f*, *struct*)
Recursively traversing all lists and tuples in struct, apply f to each element that is an instance of cls. Returns structure with f applied.

`pysnark.runtime.`**importcomm**(**args*, ***kwargs*)

`pysnark.runtime.`**init**()

`pysnark.runtime.`**inited**(*fn*)

`pysnark.runtime.`**printwire**(**args*, ***kwargs*)

`pysnark.runtime.`**printwireout**(**args*, ***kwargs*)

`pysnark.runtime.`**subqap**(*nm*)

`pysnark.runtime.`**vc_assert_mult**(**args*, ***kwargs*)

`pysnark.runtime.`**vc_declare_block**(**args*, ***kwargs*)

`pysnark.runtime.`**vc_glue**(*ctx1*, *ctx2*, *vals*)

6.2.8 pysnark.schedule module

`pysnark.schedule.`**`lines`**`()`

`pysnark.schedule.`**`oftype`**`(`*type*`)`

6.2.9 pysnark.testqap module

This tool tests whether the wire file in the current location (as given by *`pysnark.options.get_wire_file()`*) satisfies all equations of the current Quadratic Arithmetic Program (as given by *`pysnark.options.get_eqs_file()`*)

Run with

> python -m pysnark.testqap

6.3 Module contents

The PySNARK package contains the main functionality of PySNARK.

CHAPTER

SEVEN

PYSNARK EXAMPLES

PySNARK comes with the following examples that demonstrate particular aspects of its functionality:

7.1 `cube.py`: basic example computing a cube

7.2 `gc.py`: execute a given binary circuit

Executes a binary circuit, given as a text file, on two given inputs. If an input is not given, zeros are assumed. Inputs are right-padded with zeros.

The binary circuit should be given in the format described here.

For instance, to execute Nigel Smart's addition circuit, download `adder_32bit.txt` and run the script, e.g. as follows:

```
python gc.py adder_32bit.txt 010001 00111
```

(Note that the inputs Nigel Smart's circuits for arithmetic circuits are given in inverse binary representation, e.g., 010001 is 100010_2=34 and 00111 is 11100_2=28 so this computation returns 62=111110_2, i.e., 0111110000. . . .)

7.3 `kaplanmeier.py`: execute statistical test on survival data

7.4 `tesarray.py`: test array functionality

7.5 `testcomm.py`: tests with committed data

7.6 PySNARK GGH hashing examples

The examples in this folder demonstrate the use of the GGH hash function.

7.6.1 `hashsnark.py`: tests hash function

`examples.ggh.hashsnark.`**`bitstohex`**`(`*str*`)`

`examples.ggh.hashsnark.`**`readhexbits`**`(`*fl*`)`

7.6.2 `hashtree.py`: hash tree verification

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX