additive

Release 0.6.0

Nth Party, Ltd.

Jun 02, 2023

CONTENTS

Data structure for representing additive secret shares of integers, designed for use within secure multi-party computation (MPC) protocol implementations.

CHAPTER

ONE

PURPOSE

This library provides a data structure and methods that make it possible work with n-out-of-n additive secret shares of integers within secure multi-party computation (MPC) protocol implementations. Secret shares of signed and unsigned integers can be represented using elements from finite fields, with support currently limited to fields having a power-of-two order.

CHAPTER

TWO

INSTALLATION AND USAGE

This library is available as a package on PyPI:

```
python -m pip install additive
```

The library can be imported in the usual ways:

```
import additive
from additive import *
```

2.1 Examples

This library makes it possible to concisely construct multiple secret shares from an integer:

```
>>> from additive import shares
>>> (a, b) = shares(123)
>>> (c, d) = shares(456)
>>> ((a + c) + (b + d)).to_int()
579
```

It is possible to specify the exponent in the order of the finite field used to represent secret shares, as well as whether the encoding of the integer should support signed integers:

```
>>> (s, t) = shares(-123, exponent=8, signed=True)
>>> (s + t).to_int()
-123
```

The number of shares can be specified explicitly (the default is two shares):

```
>>> (r, s, t) = shares(123, quantity=3)
```

The `share` data structure supports Python's built-in addition operators, enabling both operations on shares and concise reconstruction of values from a collection of shares:

```
>>> (r + s + t).to_int()
123
>>> sum([r, s, t]).to_int()
123
```

In addition, conversion methods for Base64 strings and bytes-like objects are included to support encoding and decoding of `share` objects:

```
>>> from additive import share
>>> share.from_base64('HgEA').to_bytes().hex()
'1e0100'
>>> [s.to_base64() for s in shares(123)]
['PvmKMG8=', 'PoJ1z5A=']
```

CHAPTER

THREE

DEVELOPMENT

All installation and development dependencies are fully specified in `pyproject.toml`. The `project.optional-dependencies` object is used to specify optional requirements for various development tasks. This makes it possible to specify additional options (such as `docs`, `lint`, and so on) when performing installation using pip:

```
python -m pip install .[docs,lint]
```

3.1 Documentation

The documentation can be generated automatically from the source files using Sphinx:

```
python -m pip install .[docs]
cd docs
sphinx-apidoc -f -E --templatedir=_templates -o _source .. && make html
```

3.2 Testing and Conventions

All unit tests are executed and their coverage is measured when using pytest (see the `pyproject.toml` file for configuration details):

```
python -m pip install .[test]
python -m pytest
```

Alternatively, all unit tests are included in the module itself and can be executed using doctest:

```
python src/additive/additive.py -v
```

Style conventions are enforced using Pylint:

```
python -m pip install .[lint]
python -m pylint src/additive
```

3.3 Contributions

In order to contribute to the source code, open an issue or submit a pull request on the GitHub page for this library.

3.4 Versioning

The version number format for this library and the changes to the library associated with version number increments conform with Semantic Versioning 2.0.0.

3.5 Publishing

This library can be published as a package on PyPI by a package maintainer. First, install the dependencies required for packaging and publishing:

```
python -m pip install .[publish]
```

Ensure that the correct version number appears in `pyproject.toml`, and that any links in this README document to the Read the Docs documentation of this package (or its dependencies) have appropriate version numbers. Also ensure that the Read the Docs project for this library has an automation rule that activates and sets as the default all tagged versions. Create and push a tag for this version (replacing `?.?.?` with the version number):

```
git tag ?.?.?
git push origin ?.?.?
```

Remove any old build/distribution files. Then, package the source into a distribution archive:

```
rm -rf build dist src/*.egg-info
python -m build --sdist --wheel .
```

Finally, upload the package distribution archive to PyPI:

```
python -m twine upload dist/*
```

3.5.1 additive module

Data structure for representing additive secret shares of integers, designed for use within secure multi-party computation (MPC) protocol implementations.

class additive.additive.**share**(*value*, *exponent=32*, *signed=False*)

Bases: `object`

Data structure for representing an additive secret share of an integer.

Parameters

- **value** (`int`) – Integer value to be split into secret shares.
- **exponent** (`Optional[int]`) – Exponent in finite field order of `2 ** exponent` that is at least `8`, at most `128`, and is a multiple of `8`.

- **signed** (Optional[bool]) – Flag indicating whether `value` is a signed integer; this flag affects the specific way in which a secret share is represented internally and shifts the range of integer values that can be represented from `range(0, 2 ** exponent)` to `range(-(2 ** (exponent - 1)), (2 ** (exponent - 1)))`.

Normally, the *shares* function should be used to construct a list of *share* objects that have correct internal structure.

```
>>> ((a, b), (c, d)) = (shares(123), shares(456))
>>> ((a + c) + (b + d)).to_int()
579
```

Direct construction of *share* objects is made available to enable other use cases, protocols, and/or extensions.

```
>>> share(123)
share(123, 32, False)
>>> share(2**32 - 1, 32)
share(4294967295, 32, False)
>>> share(-(2**31), exponent=32, signed=True)
share(0, 32, True)
```

Some compatibility and validity checks of the supplied parameter values are performed.

```
>>> share(2**32, 32)
Traceback (most recent call last):
  ...
ValueError: value is not in range that can be represented using supplied parameters
>>> share(123, 12)
Traceback (most recent call last):
  ...
ValueError: exponent must be a positive multiple of 8 that is at most 128
```

static from_bytes(*bs*)

Convert a secret share represented as a bytes-like object into a *share* object.

```
>>> share.from_bytes(bytes([30, 1] + ([0] * 31)))
share(1, 16, False)
```

An attempt to decode an invalid binary representation raises an exception.

```
>>> share.from_bytes(bytes([12, 1] + ([0] * 31)))
Traceback (most recent call last):
  ...
ValueError: invalid exponent in binary encoding of share
```

Return type *share*

static from_base64(*s*)

Convert a secret share represented as a Base64 encoding of a bytes-like object into a *share* object.

```
>>> share.from_base64('HgEA')
share(1, 16, False)
```

Return type *share*

__add__(*other*)

Add two secret shares (represented as *share* objects); `0` is supported as an input to accommodate the base case required by the Python `sum` operator.

```
>>> (s, t) = shares(123)
>>> s + t
share(123, 32, False)
>>> (s + t) + 0
share(123, 32, False)
>>> ((a, b), (c, d)) = (shares(123), shares(456))
>>> ((a + c) + (b + d)).to_int()
579
>>> (ss, ts) = (shares(123, 3, signed=True), shares(-100, 3, signed=True))
>>> ((ss[0] + ts[0]) + (ss[1] + ts[1]) + (ss[2] + ts[2])).to_int()
23
>>> ts = [shares(-n, 10, signed=True) for n in [123, 456, 789]]
>>> sum(sum(ss) for ss in zip(*ts)).to_int()
-1368
```

When secret shares are added, it is not possible to determine whether the sum of the values they represent exceeds the maximum value that can be represented. If the sum does exceed the value, then the value reconstructed from the shares will wrap around. In the case of unsigned integer values, this corresponds to the usual behavior of field elements.

```
>>> (a, b) = shares(255, exponent=8) # Unsigned one-byte integer.
>>> (c, d) = shares(123, exponent=8) # Unsigned one-byte integer.
>>> ((a + c) + (b + d)).to_int() == (255 + 123) % 256 == 122
True
```

In the case of signed integers, the sum will wrap from positive to negative (in a manner similar to that of typical implementations of signed integer addition in other popular languages and libraries).

```
>>> (a, b) = shares(127, exponent=8, signed=True)
>>> (c, d) = shares(2, exponent=8, signed=True)
>>> ((a + c) + (b + d)).to_int() == -128 + ((127 + 2) % 128) == -127
True
```

An attempt to add secret shares that are represented using different finite fields (or are not all signed/unsigned) raises an exception.

```
>>> share(0, 8) + share(0, 16)
Traceback (most recent call last):
  ...
ValueError: shares must have compatible parameters to be added
>>> share(0, 8, signed=True) + share(0, 8, signed=False)
Traceback (most recent call last):
  ...
ValueError: shares must have compatible parameters to be added
```

The examples below test this addition method for a range of share quantities and addition operation counts.

```
>>> for quantity in range(2, 20):
...     for operations in range(2, 20):
...         vs = [
```

(continues on next page)

(continued from previous page)

```
...             int.from_bytes(secrets.token_bytes(2), 'little')
...             for _ in range(operations)
...         ]
...         sss = [shares(v, quantity, signed=True) for v in vs]
...         assert(sum([sum(ss) for ss in zip(*sss)]).to_int() == sum(vs))
```

Return type *share*

__radd__(*other*)

Add two secret shares (represented as *share* objects); `0` is supported as an input to accommodate the base case required by the Python `sum` operator.

```
>>> (s, t) = shares(123)
>>> s + t
share(123, 32, False)
>>> 0 + (s + t)
share(123, 32, False)
>>> sum(shares(123, 10))
share(123, 32, False)
```

Return type *share*

__mul__(*scalar*)

Multiply this secret share by an integer scalar. Note that all secret shares must be multiplied by the same integer scalar in order for the reconstructed value to reflect the correct result.

```
>>> (s, t) = shares(123)
>>> s = s * 2
>>> t = t * 2
>>> (s + t).to_int()
246
>>> (s, t) = shares(123, exponent=16, signed=True)
>>> s = s * 2
>>> t = t * 2
>>> (s + t).to_int()
246
```

Multiplication of shares of signed integers by negative scalars is supported.

```
>>> (s, t) = shares(123, exponent=16, signed=True)
>>> s = s * -3
>>> t = t * -3
>>> (s + t).to_int()
-369
>>> (s, t) = shares(123, exponent=16, signed=True)
>>> s = s * -1
>>> t = t * -1
>>> (s + t).to_int()
-123
```

When secret shares are multiplied by a scalar, it is not possible to determine whether the result exceeds the range of values that can be represented. If the result does fall outside the range, then the value reconstructed

from the shares will wrap around. In the case of unsigned integer values, this corresponds to the usual behavior of field elements.

```
>>> (s, t) = shares(129, exponent=8)
>>> s = s * 2
>>> t = t * 2
>>> (s + t).to_int() == (129 * 2) % (2 ** 8) == 2
True
```

In the case of signed integers, the result will wrap around the upper or lower boundary of the range that can be represented (in a manner similar to that of typical implementations of signed integer multiplication in other popular languages and libraries).

```
>>> (a, b) = shares(65, exponent=8, signed=True)
>>> ((2 * a) + (2 * b)).to_int() == -128 + (130 % 128) == -126
True
>>> (a, b) = shares(65, exponent=8, signed=True)
>>> ((-2 * a) + (-2 * b)).to_int() == (-130) % 128 == 126
True
>>> (a, b) = shares(-65, exponent=8, signed=True)
>>> ((-2 * a) + (-2 * b)).to_int() == -128 + (130 % 128) == -126
True
```

The scalar argument must be an integer.

```
>>> (s, t) = shares(123)
>>> s = s * 2.0
Traceback (most recent call last):
  ...
TypeError: scalar must be an integer
```

Shares of unsigned integers cannot be multiplied by a negative scalar.

```
>>> (s, t) = shares(123, signed=False)
>>> s = s * -2
Traceback (most recent call last):
  ...
ValueError: shares of unsigned integers cannot be multiplied by a negative␣
↪scalar
```

The examples below test this scalar multiplication method for a range of share quantities and a number of random scalar values.

```
>>> for quantity in range(2, 20):
...     for _ in range(100):
...         v = int.from_bytes(secrets.token_bytes(2), 'little')
...         c = -128 + int.from_bytes(secrets.token_bytes(1), 'little')
...         ss = shares(v, quantity, signed=True)
...         assert(sum([c * s for s in ss]).to_int() == c * v)
```

Return type *share*

__rmul__(*scalar*)

Multiply this secret share by an integer scalar. Note that all secret shares must be multiplied by the same integer scalar in order for the reconstructed value to reflect the correct result.

```
>>> (s, t) = shares(123)
>>> s = 2 * s
>>> t = 2 * t
>>> (s + t).to_int()
246
>>> (r, s, t) = shares(123, 3, signed=True)
>>> r = -2 * r
>>> s = -2 * s
>>> t = -2 * t
>>> (r + s + t).to_int()
-246
```

Return type *share*

to_int()

Obtain the integer value represented by a fully reconstructed aggregate of secret shares (no checking is performed that the *share* object represents a complete reconstruction).

```
>>> (s, t) = shares(123)
>>> (s + t).to_int()
123
>>> (r, s, t) = shares(-123, 3, signed=True)
>>> sum([r, s, t]).to_int()
-123
>>> (q, r, s, t) = shares(-123, 4, signed=True)
>>> sum([q, r, s, t]).to_int()
-123
>>> all([
...     sum(shares(-123, q, signed=True)).to_int() == -123
...     for q in range(2, 7)
... ])
True
```

Return type int

to_bytes()

Return a bytes-like object that encodes this *share* object.

```
>>> share.from_base64('HgEA').to_bytes().hex()
'1e0100'
>>> ss = [s.to_bytes() for s in shares(123)]
>>> sum(share.from_bytes(s) for s in ss).to_int()
123
```

Return type bytes

to_base64()

Return a Base64 string representation of this *share* object.

```
>>> share(123, 128).to_base64()
'/nsAAAAAAAAAAAAAAAAAAA='
>>> ss = [s.to_base64() for s in shares(-123, signed=True)]
```

(continues on next page)

(continued from previous page)

```
>>> sum(share.from_base64(s) for s in ss).to_int()
-123
```

Return type str

__str__()
Return the string representation of this *share* object.

```
>>> str(share(123))
'share(123, 32, False)'
```

Return type str

__repr__()
Return the string representation of this *share* object.

```
>>> share(123)
share(123, 32, False)
```

Return type str

additive.additive.**shares**(*value*, *quantity=2*, *exponent=32*, *signed=False*)
Convert an integer into two or more secret shares constructed according to the supplied parameters.

Parameters

- **value** (int) – Integer value to be split into secret shares.
- **quantity** (Optional[int]) – Number of secret shares (at least two) to construct and return.
- **exponent** (Optional[int]) – Exponent in finite field order of `2 ** exponent` that is at least `8`, at most `128`, and is a multiple of `8`.
- **signed** (Optional[bool]) – Flag indicating whether `value` is a signed integer; this flag affects the specific way in which a secret share is represented internally and shifts the range of integer values that can be represented from `{0, ..., (2 ** exponent) - 1}` to `{-(2 ** (exponent - 1)), ..., (2 ** (exponent - 1)) - 1}`.

```
>>> (s, t) = shares(123)
>>> (s + t).to_int()
123
>>> ss = shares(123, 20)
>>> len(ss)
20
>>> sum(ss).to_int()
123
>>> all(isinstance(s, share) for s in shares(123))
True
```

Some compatibility and validity checks of the integer value and the parameter values are performed.

```
>>> shares(123, 2, exponent=129)
Traceback (most recent call last):
  ...
```

(continues on next page)

(continued from previous page)

```
ValueError: exponent must be a positive multiple of 8 that is at most 128
>>> shares(256, 2, exponent=8)
Traceback (most recent call last):
  ...
ValueError: value is not in range that can be represented using supplied parameters
>>> shares(128, 2, exponent=8, signed=True)
Traceback (most recent call last):
  ...
ValueError: value is not in range that can be represented using supplied parameters
>>> shares(-129, 2, exponent=8, signed=True)
Traceback (most recent call last):
  ...
ValueError: value is not in range that can be represented using supplied parameters
```

Return type Sequence[*share*]

PYTHON MODULE INDEX

INDEX

Symbols

A

F

M

S

T