MP-SPDZ Documentation

Marcel Keller

May 31, 2023

CONTENTS:

1 Compilation process **3**
1.1 Compilation vs run time . . . 4

2 Reference **5**
2.1 High-Level Interface . . . 5
2.1.1 Compiler.types module . . . 5
2.1.1.1 Basic types . . . 5
2.1.1.2 Container types . . . 6
2.1.2 Compiler.GC.types module . . . 48
2.1.3 Compiler.library module . . . 53
2.1.4 Compiler.mpc_math module . . . 59
2.1.5 Compiler.ml module . . . 61
2.1.6 Compiler.circuit module . . . 67
2.1.7 Compiler.program module . . . 68
2.1.8 Compiler.oram module . . . 70
2.2 Virtual Machine . . . 70
2.2.1 Instructions . . . 71
2.2.2 Compiler.instructions module . . . 76
2.2.3 Compiler.GC.instructions module . . . 107
2.3 Low-Level Interface . . . 116
2.3.1 Domain Types . . . 119
2.3.2 Share Types . . . 119
2.3.3 Protocol Setup . . . 120
2.3.4 Protocol Interfaces . . . 121
2.4 Machine Learning . . . 124
2.4.1 Prediction . . . 125
2.5 Networking . . . 126
2.5.1 Internal Infrastructure . . . 126
2.5.1.1 Reference . . . 127
2.6 Input/Output . . . 134
2.6.1 Private Inputs from Computing Parties . . . 134
2.6.2 Compile-Time Data via Private Input . . . 134
2.6.3 Public Inputs . . . 134
2.6.4 Public Outputs . . . 134
2.6.5 Private Outputs to Computing Parties . . . 135
2.6.6 Binary Output . . . 135
2.6.7 Clients (Non-computing Parties) . . . 135
2.6.8 Secret Shares . . . 135
2.6.9 Reference . . . 135
2.7 Non-linear Computation . . . 136

2.7.1 Mixed-Circuit Computation . . . 137
2.7.1.1 Protocol Pairs . . . 137
2.8 Preprocessing . . . 137
2.8.1 Separate preprocessing . . . 138
2.9 Adding a Protocol . . . 139
2.10 Troubleshooting . . . 140
2.10.1 Crash without error message or `bad_alloc` . . . 140
2.10.2 List indices must be integers or slices . . . 140
2.10.3 `compile.py` takes too long or runs out of memory . . . 140
2.10.4 Order of memory instructions not preserved . . . 140
2.10.5 Odd timings . . . 140
2.10.6 Disparities in round figures . . . 141
2.10.7 Handshake failures . . . 141
2.10.8 Connection failures . . . 141
2.10.9 Internally called tape has unknown offline data usage . . . 141
2.10.10 Illegal instruction . . . 141
2.10.11 Invalid instruction . . . 141
2.10.12 Computation used more preprocessing than expected . . . 142
2.10.13 `mac_fail` . . . 142

3 Indices and tables **143**

Python Module Index **145**

Index **147**

If you're new to MP-SPDZ, consider the following:

1. Quickstart tutorial
2. Implemented protocols
3. *Troubleshooting*

Unlike the Readme, this documentation provides a reference for more detailed aspects of the software.

CHAPTER

ONE

COMPILATION PROCESS

The easiest way of using MP-SPDZ is using `compile.py` as described below. If you would like to run compilation directly from Python, see `Scripts/direct_compilation_example.py`. It contains all the necessary setup steps.

After putting your code in `Program/Source/<progname>.mpc`, run the compiler from the root directory as follows

```
./compile.py [options] <progname> [args]
```

The arguments `<progname> [args]` are accessible as list under `program.args` within `progname.mpc`, with `<progname>` as `program.args[0]`.

The following options influence the computation domain:

-F `<integer length>`

--field=`<integer length>`

Compile for computation modulo a prime and the default integer length. This means that secret integers are assumed to have at most said length unless explicitly told otherwise. The compiled output will communicate the minimum length of the prime number to the virtual machine, which will fail if this is not met. This is the default with an *integer length* set to 64. When not specifying the prime, the minimum prime length will be around 40 bits longer than the integer length. Furthermore, the computation will be optimistic in the sense that overflows in the secrets might have security implications.

-P `<prime>`

--prime=`<prime>`

Specify a concrete prime modulus for computation. This can be used together with *-F*, in which case *integer length* has to be at most the prime length minus two. The security implications of overflows in the secrets do not go beyond incorrect results.

-R `<ring size>`

--ring=`<ring size>`

Compile for computation modulo 2^(*ring size*). This will set the assumed length of secret integers to one less because many operations require this. The exact ring size will be communicated to the virtual machine, which will use it automatically if supported.

-B `<integer length>`

--binary=`<integer length>`

Compile for binary computation using *integer length* as default.

For arithmetic computation (*-F*, *-P*, and *-R*) you can set the bit length during execution using `program.set_bit_length(length)`. For binary computation you can do so with `sint = sbitint.get_type(length)`. Use `sfix.set_precision()` to change the range for fixed-point numbers.

The following options switch from a single computation domain to mixed computation when using in conjunction with arithmetic computation:

-X

--mixed

Enables mixed computation using daBits.

-Y

--edabit

Enables mixed computation using edaBits.

The implementation of both daBits and edaBits are explained in this paper.

-Z <number of parties>

--split=<number of parties>

Enables mixed computation using local conversion. This has been used by Mohassel and Rindal and Araki et al. It only works with additive secret sharing modulo a power of two.

The following options change less fundamental aspects of the computation:

-D

--dead-code-elimination

Eliminates unused code. This currently means computation that isn't used for input or output or written to the so-called memory (e.g., *Array*; see *types*).

-b <budget>

--budget=<budget>

Set the budget for loop unrolling with *for_range_opt()* and similar. This means that loops are unrolled up to *budget* instructions. Default is 100,000 instructions.

-C

--CISC

Speed up the compilation of repetitive code at the expense of a potentially higher number of communication rounds. For example, the compiler by default will try to compute a division and a logarithm in parallel if possible. Using this option complex operations such as these will be separated and only multiple divisions or logarithms will be computed in parallel. This speeds up the compilation because of reduced complexity.

-l

--flow-optimization

Optimize simple loops (`for <iterator> in range(<n>)`) by using *for_range_opt()* and defer if statements to the run time.

1.1 Compilation vs run time

The most important thing to keep in mind is that the Python code is executed at compile-time. This means that Python data structures such as `list` and `dict` only exist at compile-time and that all Python loops are unrolled. For run-time loops and lists, you can use *for_range()* (or the more optimizing *for_range_opt()*) and *Array*. For convenient multithreading you can use *for_range_opt_multithread()*, which automatically distributes the computation on the requested number of threads.

This reference uses the term 'compile-time' to indicate Python types (which are inherently known when compiling). If the term 'public' is used, this means both compile-time values as well as public run-time types such as *regint*.

CHAPTER

TWO

REFERENCE

2.1 High-Level Interface

2.1.1 Compiler.types module

This module defines all types available in high-level programs. These include basic types such as secret integers or floating-point numbers and container types. A single instance of the former uses one or more so-called registers in the virtual machine while the latter use the so-called memory. For every register type, there is a corresponding dedicated memory.

Registers are used for computation, allocated on an ongoing basis, and thread-specific. The memory is allocated statically and shared between threads. This means that memory-based types such as *Array* can be used to transfer information between threads. Note that creating memory-based types outside the main thread is not supported.

If viewing this documentation in processed form, many function signatures appear generic because of the use of decorators. See the source code for the correct signature.

2.1.1.1 Basic types

All basic can be used as vectors, that is one instance representing several values, with all operations being executed element-wise. For example, the following computes ten multiplications of integers input by party 0 and 1:

```
sint.get_input_from(0, size=10) * sint.get_input_from(1, size=10)
```

sint	Secret integer in the protocol-specific domain.
cint	Clear integer in same domain as secure computation (depends on protocol).
regint	Clear 64-bit integer.
sfix	Secret fixed-point number represented as secret integer, by multiplying with `2^f` and then rounding.
cfix	Clear fixed-point number represented as clear integer.
sfloat	Secret floating-point number.
sgf2n	Secret $\mathrm{GF}(2^n)$ value.
cgf2n	Clear $\mathrm{GF}(2^n)$ value.

2.1.1.2 Container types

MemValue	Single value in memory.
Array	Array accessible by public index.
Matrix	Matrix.
MultiArray	Multidimensional array.

class Compiler.types.**Array**(*length*, *value_type*, *address=None*, *debug=None*, *alloc=True*)

Array accessible by public index. That is, `a[i]` works for an array `a` and `i` being a *regint*, *cint*, or a Python integer.

Parameters

- **length** – compile-time integer (int) or `None` for unknown length
- **value_type** – basic type
- **address** – if given (regint/int), the array will not be allocated

You can convert between arrays and register vectors by using slice indexing. This allows for element-wise operations as long as supported by the basic type. The following adds 10 secret integers from the first two parties:

```
a = sint.Array(10)
a.input_from(0)
b = sint.Array(10)
b.input_from(1)
a[:] += b[:]
```

assign(*other*, *base=0*)

Assignment.

Parameters

- **other** – vector/Array/Matrix/MultiArray/iterable of compatible type and smaller size
- **base** – index to start assignment at

assign_all(*value*, *use_threads=True*, *conv=True*)

Assign the same value to all entries.

Parameters
value – convertible to basic type

assign_part_vector(*other*, *base=0*)

Assignment.

Parameters

- **other** – vector/Array/Matrix/MultiArray/iterable of compatible type and smaller size
- **base** – index to start assignment at

assign_vector(*other*, *base=0*)

Assignment.

Parameters

- **other** – vector/Array/Matrix/MultiArray/iterable of compatible type and smaller size
- **base** – index to start assignment at

binary_output(*player=None*)

Binary output if supported by type.

Param
player (default all)

classmethod create_from(*l*)

Convert Python iterator or vector to array. Basic type will be taken from first element, further elements must to be convertible to that.

expand_to_vector(*index*, *size*)

Create vector from single entry.

Parameters

- **index** – regint/cint/int
- **size** – int

get(*indices*)

Vector from arbitrary indices.

Parameters
indices – regint vector or array

get_part(*base*, *size*)

Part array.

Parameters

- **base** – start index (regint/cint/int)
- **size** – integer

Returns
Array of same type

get_part_vector(*base=0*, *size=None*)

Return vector with content.

Parameters

- **base** – starting point (regint/cint/int)
- **size** – length (compile-time int)

get_vector(*base=0*, *size=None*)

Return vector with content.

Parameters

- **base** – starting point (regint/cint/int)
- **size** – length (compile-time int)

input_from(*player*, *budget=None*, *raw=False*)

Fill with inputs from player if supported by type.

Parameters
player – public (regint/cint/int)

maybe_get(*condition*, *index*)

Return entry if condition is true.

Parameters

- **condition** – 0/1 (regint/cint/int)
- **index** – regint/cint/int

maybe_set(*condition*, *index*, *value*)

Change entry if condition is true.

Parameters

- **condition** – 0/1 (regint/cint/int)
- **index** – regint/cint/int
- **value** – updated value

print_reveal_nested(*end='\n'*)

Reveal and print as list.

Parameters
end – string to print after (default: line break)

read_from_file(*start*)

Read content from `Persistence/Transactions-P<playerno>.data`. Precision must be the same as when storing if applicable.

Parameters
start – starting position in number of shares from beginning (int/regint/cint)

Returns
destination for final position, -1 for eof reached, or -2 for file not found (regint)

reveal()

Reveal the whole array.

Returns
Array of relevant clear type.

reveal_list()

Reveal as list.

reveal_nested()

Reveal as list.

reveal_to_binary_output(*player=None*)

Reveal to binary output if supported by type.

Param
player to reveal to (default all)

reveal_to_clients(*clients*)

Reveal contents to list of clients.

Parameters
clients – list or array of client identifiers

same_shape()

Array of same length and type.

shuffle()

Insecure shuffle in place.

sort(*n_threads=None*)

Sort in place using Batchers' odd-even merge mergesort with complexity $O(n(\log n)^2)$.

Parameters

n_threads – number of threads to use (single thread by default)

write_to_file()

Write shares of integer representation to `Persistence/Transactions-P<playerno>.data` (appending at the end).

class Compiler.types.**Matrix**(*rows*, *columns*, *value_type*, *debug=None*, *address=None*)

Matrix.

Parameters

- **rows** – compile-time (int)
- **columns** – compile-time (int)
- **value_type** – basic type of entries

assign(*other*)

Assign container to content. Not implemented for floating-point.

Parameters

other – container of matching size and type

assign_all(*value*)

Assign the same value to all entries.

Parameters

value – convertible to relevant basic type

assign_part_vector(*vector*, *base=0*)

Assign vector from range of the first dimension, including all entries in further dimensions.

Parameters

- **vector** – updated entries
- **base** – index in first dimension (regint/cint/int)

assign_vector(*vector*, *base=0*)

Assign vector to content. Not implemented for floating-point.

Parameters

- **vector** – vector of matching size convertible to relevant basic type
- **base** – compile-time (int)

assign_vector_by_indices(*vector*, **indices*)

Assign vector to entries with potential asterisks. See `get_vector_by_indices()` for an example.

diag()

Matrix diagonal.

direct_mul(*other*, *reduce=True*, *indices=None*)

Matrix multiplication in the virtual machine.

Parameters

- **self** – *Matrix* / 2-dimensional *MultiArray*
- **other** – *Matrix* / 2-dimensional *MultiArray*
- **indices** – 4-tuple of *regint* vectors for index selection (default is complete multiplication)

Returns

Matrix as vector of relevant type (row-major)

The following executes a matrix multiplication selecting every third row of `A`:

```
A = sfix.Matrix(7, 4)
B = sfix.Matrix(4, 5)
C = sfix.Matrix(3, 5)
C.assign_vector(A.direct_mul(B, indices=(regint.inc(3, 0, 3),
                                         regint.inc(4),
                                         regint.inc(4),
                                         regint.inc(5)))
```

direct_mul_to_matrix(*other*)

Matrix multiplication in the virtual machine.

Parameters

- **self** – *Matrix* / 2-dimensional *MultiArray*
- **other** – *Matrix* / 2-dimensional *MultiArray*

Returns

Matrix

direct_mul_trans(*other*, *reduce=True*, *indices=None*)

Matrix multiplication with the transpose of `other` in the virtual machine.

Parameters

- **self** – *Matrix* / 2-dimensional *MultiArray*
- **other** – *Matrix* / 2-dimensional *MultiArray*
- **indices** – 4-tuple of *regint* vectors for index selection (default is complete multiplication)

Returns

Matrix as vector of relevant type (row-major)

direct_trans_mul(*other*, *reduce=True*, *indices=None*)

Matrix multiplication with the transpose of `self` in the virtual machine.

Parameters

- **self** – *Matrix* / 2-dimensional *MultiArray*
- **other** – *Matrix* / 2-dimensional *MultiArray*
- **indices** – 4-tuple of *regint* vectors for index selection (default is complete multiplication)

Returns
Matrix as vector of relevant type (row-major)

dot(*other*, *res_params=None*)

Matrix-matrix and matrix-vector multiplication.

Parameters

- **self** – two-dimensional
- **other** – Matrix or Array of matching size and type

get_part(*start*, *size*)

Part multi-array.

Parameters

- **start** – first-dimension index (regint/cint/int)
- **size** – int

get_part_vector(*base=0*, *size=None*)

Vector from range of the first dimension, including all entries in further dimensions.

Parameters

- **base** – index in first dimension (regint/cint/int)
- **size** – size in first dimension (int)

get_slice_vector(*slice*)

Vector from range of indicies of the first dimension, including all entries in further dimensions.

Parameters
slice – regint array

get_vector(*base=0*, *size=None*)

Return vector with content. Not implemented for floating-point.

Parameters

- **base** – public (regint/cint/int)
- **size** – compile-time (int)

get_vector_by_indices(**indices*)

Vector with potential asterisks. The potential retrieves all entry where the first dimension index is 0, and the third dimension index is 1:

```
a.get_vector_by_indices(0, None, 1)
```

iadd(*other*)

Element-wise addition in place.

Parameters
other – container of matching size and type

input_from(*player*, *budget=None*, *raw=False*)

Fill with inputs from player if supported by type.

Parameters
player – public (regint/cint/int)

mul_trans(*other*)

Matrix multiplication with transpose of `other`.

Parameters

- **self** – two-dimensional
- **other** – two-dimensional container of matching type and size

mul_trans_to(*other*, *res*, *n_threads=None*)

Matrix multiplication with the transpose of `other` in the virtual machine.

Parameters

- **self** – *Matrix* / 2-dimensional *MultiArray*
- **other** – *Matrix* / 2-dimensional *MultiArray*
- **res** – matrix of matching dimension to store result
- **n_threads** – number of threads (default: single thread)

plain_mul(*other*, *res=None*)

Alternative matrix multiplication.

Parameters

- **self** – two-dimensional
- **other** – two-dimensional container of matching type and size

print_reveal_nested(*end='\n'*)

Reveal and print as nested list.

Parameters

end – string to print after (default: line break)

read_from_file(*start*)

Read content from `Persistence/Transactions-P<playerno>.data`. Precision must be the same as when storing if applicable.

Parameters

start – starting position in number of shares from beginning (int/regint/cint)

Returns

destination for final position, -1 for eof reached, or -2 for file not found (regint)

reveal_list()

Reveal as list.

reveal_nested()

Reveal as nested list.

reveal_to_binary_output(*player=None*)

Reveal to binary output if supported by type.

Param

player to reveal to (default all)

reveal_to_clients(*clients*)

Reveal contents to list of clients.

Parameters

clients – list or array of client identifiers

same_shape()

Returns
new multidimensional array with same shape and basic type

schur(*other*)

Element-wise product.

Parameters
other – container of matching size and type

Returns
container of same shape and type as `self`

set_column(*index*, *vector*)

Change column.

Parameters

- **index** – regint/cint/int
- **vector** – short enought vector of compatible type

trace()

Matrix trace.

trans_mul(*other*, *reduce=True*, *res=None*)

Matrix multiplication with transpose of `self`

Parameters

- **self** – two-dimensional
- **other** – two-dimensional container of matching type and size

trans_mul_to(*other*, *res*, *n_threads=None*)

Matrix multiplication with the transpose of `self` in the virtual machine.

Parameters

- **self** – *Matrix* / 2-dimensional *MultiArray*
- **other** – *Matrix* / 2-dimensional *MultiArray*
- **res** – matrix of matching dimension to store result
- **n_threads** – number of threads (default: single thread)

transpose()

Matrix transpose.

Parameters
self – two-dimensional

write_to_file()

Write shares of integer representation to `Persistence/Transactions-P<playerno>.data` (appending at the end).

class Compiler.types.**MemValue**(*value*, *address=None*)

Single value in memory. This is useful to transfer information between threads. Operations are automatically read from memory if required, this means you can use any operation with *MemValue* objects as if they were a basic type.

Parameters
value – basic type or int (will be converted to regint)

max(*other*)

Maximum.

Parameters
other – any compatible type

min(*other*)

Minimum.

Parameters
other – any compatible type

read()

Read value.

Returns
relevant basic type instance

reveal()

Reveal value.

Returns
relevant clear type

square()

Square.

write(*value*)

Write value.

Parameters
value – convertible to relevant basic type

class Compiler.types.**MultiArray**(*sizes*, *value_type*, *debug=None*, *address=None*, *alloc=True*)

Multidimensional array. The access operator (`a[i]`) allows to a multi-dimensional array of dimension one less or a simple array for a two-dimensional array.

Parameters

- **sizes** – shape (compile-time list of integers)
- **value_type** – basic type of entries

You can convert between arrays and register vectors by using slice indexing. This allows for element-wise operations as long as supported by the basic type. The following has the first two parties input a 10x10 secret integer matrix followed by storing the element-wise multiplications in the same data structure:

```
a = sint.Tensor([3, 10, 10])
a[0].input_from(0)
a[1].input_from(1)
a[2][:] = a[0][:] * a[1][:]
```

assign(*other*)

Assign container to content. Not implemented for floating-point.

Parameters
other – container of matching size and type

assign_all(*value*)

Assign the same value to all entries.

Parameters
value – convertible to relevant basic type

assign_part_vector(*vector*, *base=0*)

Assign vector from range of the first dimension, including all entries in further dimensions.

Parameters

- **vector** – updated entries
- **base** – index in first dimension (regint/cint/int)

assign_vector(*vector*, *base=0*)

Assign vector to content. Not implemented for floating-point.

Parameters

- **vector** – vector of matching size convertible to relevant basic type
- **base** – compile-time (int)

assign_vector_by_indices(*vector*, **indices*)

Assign vector to entries with potential asterisks. See *get_vector_by_indices()* for an example.

diag()

Matrix diagonal.

direct_mul(*other*, *reduce=True*, *indices=None*)

Matrix multiplication in the virtual machine.

Parameters

- **self** – *Matrix* / 2-dimensional *MultiArray*
- **other** – *Matrix* / 2-dimensional *MultiArray*
- **indices** – 4-tuple of *regint* vectors for index selection (default is complete multiplication)

Returns
Matrix as vector of relevant type (row-major)

The following executes a matrix multiplication selecting every third row of `A`:

```
A = sfix.Matrix(7, 4)
B = sfix.Matrix(4, 5)
C = sfix.Matrix(3, 5)
C.assign_vector(A.direct_mul(B, indices=(regint.inc(3, 0, 3),
                                         regint.inc(4),
                                         regint.inc(4),
                                         regint.inc(5)))
```

direct_mul_to_matrix(*other*)

Matrix multiplication in the virtual machine.

Parameters

- **self** – *Matrix* / 2-dimensional *MultiArray*
- **other** – *Matrix* / 2-dimensional *MultiArray*

Returns
Matrix

direct_mul_trans(*other*, *reduce=True*, *indices=None*)

Matrix multiplication with the transpose of `other` in the virtual machine.

Parameters

- **self** – *Matrix* / 2-dimensional *MultiArray*
- **other** – *Matrix* / 2-dimensional *MultiArray*
- **indices** – 4-tuple of *regint* vectors for index selection (default is complete multiplication)

Returns
Matrix as vector of relevant type (row-major)

direct_trans_mul(*other*, *reduce=True*, *indices=None*)

Matrix multiplication with the transpose of `self` in the virtual machine.

Parameters

- **self** – *Matrix* / 2-dimensional *MultiArray*
- **other** – *Matrix* / 2-dimensional *MultiArray*
- **indices** – 4-tuple of *regint* vectors for index selection (default is complete multiplication)

Returns
Matrix as vector of relevant type (row-major)

dot(*other*, *res_params=None*)

Matrix-matrix and matrix-vector multiplication.

Parameters

- **self** – two-dimensional
- **other** – Matrix or Array of matching size and type

get_part(*start*, *size*)

Part multi-array.

Parameters

- **start** – first-dimension index (regint/cint/int)
- **size** – int

get_part_vector(*base=0*, *size=None*)

Vector from range of the first dimension, including all entries in further dimensions.

Parameters

- **base** – index in first dimension (regint/cint/int)
- **size** – size in first dimension (int)

get_slice_vector(*slice*)

Vector from range of indicies of the first dimension, including all entries in further dimensions.

Parameters
slice – regint array

get_vector(*base=0*, *size=None*)

Return vector with content. Not implemented for floating-point.

Parameters

- **base** – public (regint/cint/int)
- **size** – compile-time (int)

get_vector_by_indices(**indices*)

Vector with potential asterisks. The potential retrieves all entry where the first dimension index is 0, and the third dimension index is 1:

```
a.get_vector_by_indices(0, None, 1)
```

iadd(*other*)

Element-wise addition in place.

Parameters
other – container of matching size and type

input_from(*player*, *budget=None*, *raw=False*)

Fill with inputs from player if supported by type.

Parameters
player – public (regint/cint/int)

mul_trans(*other*)

Matrix multiplication with transpose of `other`.

Parameters

- **self** – two-dimensional
- **other** – two-dimensional container of matching type and size

mul_trans_to(*other*, *res*, *n_threads=None*)

Matrix multiplication with the transpose of `other` in the virtual machine.

Parameters

- **self** – *Matrix* / 2-dimensional *MultiArray*
- **other** – *Matrix* / 2-dimensional *MultiArray*
- **res** – matrix of matching dimension to store result
- **n_threads** – number of threads (default: single thread)

plain_mul(*other*, *res=None*)

Alternative matrix multiplication.

Parameters

- **self** – two-dimensional
- **other** – two-dimensional container of matching type and size

print_reveal_nested(*end='\n'*)

Reveal and print as nested list.

Parameters
end – string to print after (default: line break)

read_from_file(*start*)

Read content from `Persistence/Transactions-P<playerno>.data`. Precision must be the same as when storing if applicable.

Parameters
start – starting position in number of shares from beginning (int/regint/cint)

Returns
destination for final position, -1 for eof reached, or -2 for file not found (regint)

reveal_list()

Reveal as list.

reveal_nested()

Reveal as nested list.

reveal_to_binary_output(*player=None*)

Reveal to binary output if supported by type.

Param
player to reveal to (default all)

reveal_to_clients(*clients*)

Reveal contents to list of clients.

Parameters
clients – list or array of client identifiers

same_shape()

Returns
new multidimensional array with same shape and basic type

schur(*other*)

Element-wise product.

Parameters
other – container of matching size and type

Returns
container of same shape and type as `self`

trace()

Matrix trace.

trans_mul(*other*, *reduce=True*, *res=None*)

Matrix multiplication with transpose of `self`

Parameters

- **self** – two-dimensional
- **other** – two-dimensional container of matching type and size

trans_mul_to(*other*, *res*, *n_threads=None*)

Matrix multiplication with the transpose of `self` in the virtual machine.

Parameters

- **self** – *Matrix* / 2-dimensional *MultiArray*
- **other** – *Matrix* / 2-dimensional *MultiArray*

- **res** – matrix of matching dimension to store result
- **n_threads** – number of threads (default: single thread)

transpose()

Matrix transpose.

Parameters
self – two-dimensional

write_to_file()

Write shares of integer representation to `Persistence/Transactions-P<playerno>.data` (appending at the end).

class Compiler.types.**cfix**(***kwargs*)

Clear fixed-point number represented as clear integer. It supports basic arithmetic (`+, -, *, /`), returning either *cfix* if the other operand is public (cfix/regint/cint/int) or *sfix* if the other operand is an sfix. It also support comparisons (`==, !=, <, <=, >, >=`), returning either *regint* or `sbitint`.

Parameters
v – cfix/float/int

classmethod Array(*size*, **args*, ***kwargs*)

Type-dependent array. Example:

```
a = sint.Array(10)
```

classmethod Matrix(*rows*, *columns*, **args*, ***kwargs*)

Type-dependent matrix. Example:

```
a = sint.Matrix(10, 10)
```

classmethod MemValue(*value*)

classmethod Tensor(*shape*)

Type-dependent tensor of any dimension:

```
a = sfix.Tensor([10, 10])
```

binary_output(**args*, ***kwargs*)

Write double-precision floating-point number to `Player-Data/Binary-Output-P<playerno>-<threadno>`.

Parameters
player – only output on given player (default all)

classmethod load_mem(**args*, ***kwargs*)

Load from memory by public address.

max(*other*)

Maximum.

Parameters
other – any compatible type

min(*other*)

Minimum.

Parameters
other – any compatible type

print_plain(**args*, ***kwargs*)

Clear fixed-point output.

classmethod read_from_socket(**args*, ***kwargs*)

Receive clear fixed-point value(s) from client. The client needs to convert the values to the right integer representation.

Parameters

- **client_id** – Client id (regint)
- **n** – number of values (default 1)

Param
vector size (int)

Returns
cfix (if n=1) or list of cfix

classmethod set_precision(*f*, *k=None*)

Set the precision of the integer representation. Note that some operations are undefined when the precision of *sfix* and *cfix* differs. The initial defaults are chosen to allow the best optimization of probabilistic truncation in computation modulo 2^64 (2*k < 64). Generally, 2*k must be at most the integer length for rings and at most m-s-1 for computation modulo an m-bit prime and statistical security s (default 40).

Parameters

- **f** – bit length of decimal part (initial default 16)
- **k** – whole bit length of fixed point, defaults to twice `f` if not given (initial default 31)

square()

Square.

store_in_mem(*address*)

Store in memory by public address.

classmethod write_to_socket(*client_id*, *values*, *message_type=0*)

Send a list of clear fixed-point values to a client (represented as clear integers).

Parameters

- **client_id** – Client id (regint)
- **values** – list of cint

class Compiler.types.**cfloat**(***kwargs*)

Helper class for printing revealed sfloats.

binary_output(*player=None*)

Write double-precision floating-point number to `Player-Data/Binary-Output-P<playerno>-<threadno>`.

Parameters
player – only output on given player (default all)

print_float_plain(**args*, ***kwargs*)

Output.

class Compiler.types.**cgf2n**(*val=None*, *size=None*)

Clear $\mathrm{GF}(2^n)$ value. n is chosen at runtime. A number operators are supported (`+, -, *, /, **, ^, &, |, ~, ==, !=, <<, >>`), returning either *cgf2n* if the other operand is public (cgf2n/regint/int) or *sgf2n* if the other operand is secret. The following operators require the other operand to be a compile-time integer: `**, <<, >>`. `*, /, **` refer to field multiplication and division.

Parameters

- **val** – initialization (cgf2n/cint/regint/int or list thereof)
- **size** – vector size (int), defaults to 1 or size of list

classmethod Array(*size*, **args*, ***kwargs*)

Type-dependent array. Example:

```
a = sint.Array(10)
```

classmethod Matrix(*rows*, *columns*, **args*, ***kwargs*)

Type-dependent matrix. Example:

```
a = sint.Matrix(10, 10)
```

classmethod MemValue(*value*)

classmethod Tensor(*shape*)

Type-dependent tensor of any dimension:

```
a = sfix.Tensor([10, 10])
```

binary_output(**args*, ***kwargs*)

Write 64-bit signed integer to `Player-Data/Binary-Output-P<playerno>-<threadno>`.

Parameters
player – only output on given player (default all)

bit_and(*other*)

AND in binary circuits.

Parameters
self/other – 0 or 1 (any compatible type)

Return type
depending on inputs (secret if any of them is)

classmethod bit_compose(*bits*, *step=None*)

Clear $\mathrm{GF}(2^n)$ bit composition.

Parameters

- **bits** – list of cgf2n
- **step** – set every `step`-th bit in output (defaults to 1)

bit_decompose(**args*, ***kwargs*)

Clear bit decomposition.

Parameters

- **bit_length** – number of bits (defaults to global $\mathrm{GF}(2^n)$ bit length)
- **step** – extract every `step`-th bit (defaults to 1)

bit_not()

NOT in binary circuits.

bit_or(*other*)

OR in binary circuits.

Parameters
self/other – 0 or 1 (any compatible type)

Returns
type depends on inputs (secret if any of them is)

bit_xor(*other*)

XOR in $\mathrm{GF}(2^n)$ circuits.

Parameters
self/other – 0 or 1 (any compatible type)

Return type
depending on inputs (secret if any of them is)

cond_swap(*a*, *b*, *t=None*)

Swapping in $\mathrm{GF}(2^n)$. Similar to `_int.if_else()`.

half_adder(*other*)

Half adder in binary circuits.

Parameters
self/other – 0 or 1 (any compatible type)

Returns
binary sum, carry

Return type
depending on inputs (secret if any of them is)

if_else(*a*, *b*)

MUX in $\mathrm{GF}(2^n)$ circuits. Similar to `_int.if_else()`.

classmethod load_mem(**args*, ***kwargs*)

Load from memory by public address.

classmethod malloc(*size*, *creator_tape=None*)

Allocate memory (statically).

Parameters
size – compile-time (int)

max(*other*)

Maximum.

Parameters
other – any compatible type

min(*other*)

Minimum.

Parameters
other – any compatible type

print_reg_plain(**args*, ***kwargs*)

Output.

reveal()

Identity.

square()

Square.

store_in_mem(*address*)

Store in memory by public address.

class Compiler.types.**cint**(**kwargs*)

Clear integer in same domain as secure computation (depends on protocol). A number operators are supported (`+, -, *, /, //, **, %, ^, &, |, ~, ==, !=, <<, >>`), returning either *cint* if the other operand is public (cint/regint/int) or *sint* if the other operand is *sint*. Comparison operators (`==, !=, <, <=, >, >=`) are also supported, returning *regint()*. Comparisons and ~ require that the value is within the global bit length. The same holds for `abs()`. `/` runs field division if the modulus is a prime while `//` runs integer floor division. `**` requires the exponent to be compile-time integer or the base to be two.

Parameters

- **val** – initialization (cint/regint/int/cgf2n or list thereof)
- **size** – vector size (int), defaults to 1 or size of list

classmethod Array(*size*, **args*, ***kwargs*)

Type-dependent array. Example:

```
a = sint.Array(10)
```

classmethod Matrix(*rows*, *columns*, **args*, ***kwargs*)

Type-dependent matrix. Example:

```
a = sint.Matrix(10, 10)
```

classmethod MemValue(*value*)

classmethod Tensor(*shape*)

Type-dependent tensor of any dimension:

```
a = sfix.Tensor([10, 10])
```

binary_output(**args*, ***kwargs*)

Write 64-bit signed integer to `Player-Data/Binary-Output-P<playerno>-<threadno>`.

Parameters

player – only output on given player (default all)

static bit_adder(**args*, ***kwargs*)

Binary adder in arithmetic circuits.

Parameters

- **a** – summand (list of 0/1 in compatible type)
- **b** – summand (list of 0/1 in compatible type)
- **carry_in** – input carry (default 0)
- **get_carry** – add final carry to output

Returns
list of 0/1 in relevant type

`bit_and`(*other*)

AND in arithmetic circuits.

Parameters
`self/other` – 0 or 1 (any compatible type)

Return type
depending on inputs (secret if any of them is)

`classmethod bit_compose`(*bits*)

Compose value from bits.

Parameters
`bits` – iterable of any type implementing left shift

`bit_decompose`(**args*, ***kwargs*)

Clear bit decomposition.

Parameters
`bit_length` – number of bits (default is global bit length)

Returns
list of cint

`bit_not`()

NOT in arithmetic circuits.

`bit_or`(*other*)

OR in arithmetic circuits.

Parameters
`self/other` – 0 or 1 (any compatible type)

Returns
type depends on inputs (secret if any of them is)

`bit_xor`(*other*)

XOR in arithmetic circuits.

Parameters
`self/other` – 0 or 1 (any compatible type)

Returns
type depends on inputs (secret if any of them is)

`cond_swap`(*a*, *b*)

Swapping in arithmetic circuits.

Parameters
`a/b` – any type supporting the necessary operations

Returns
`(a, b)` if `self` is 0, `(b, a)` if `self` is 1, and undefined otherwise

Return type
depending on operands, secret if any of them is

`digest`(*num_bytes*)

Clear hashing (libsodium default).

half_adder(*other*)

Half adder in arithmetic circuits.

Parameters
self/other – 0 or 1 (any compatible type)

Returns
binary sum, carry

Return type
depending on inputs, secret if any is

if_else(*a*, *b*)

MUX on bit in arithmetic circuits.

Parameters
a/b – any type supporting the necessary operations

Returns
a if `self` is 1, b if `self` is 0, undefined otherwise

Return type
depending on operands, secret if any of them is

legendre()

Clear Legendre symbol computation.

less_than(**args*, ***kwargs*)

Clear comparison for particular bit length.

Parameters

- **other** – cint/regint/int
- **bit_length** – signed bit length of inputs

Returns
0/1 (regint), undefined if inputs outside range

classmethod load_mem(**args*, ***kwargs*)

Load from memory by public address.

classmethod malloc(*size*, *creator_tape=None*)

Allocate memory (statically).

Parameters
size – compile-time (int)

max(*other*)

Maximum.

Parameters
other – any compatible type

min(*other*)

Minimum.

Parameters
other – any compatible type

mod2m(**args*, ***kwargs*)

Clear modulo a power of two.

Parameters
other – cint/regint/int

print_if(*string*)

Output if value is non-zero.

Parameters
string – Python string

print_reg_plain(**args*, ***kwargs*)

Output.

classmethod read_from_socket(**args*, ***kwargs*)

Receive clear value(s) from client.

Parameters

- **client_id** – Client id (regint)
- **n** – number of values (default 1)
- **size** – vector size (default 1)

Returns
cint (if n=1) or list of cint

reveal()

Identity.

right_shift(**args*, ***kwargs*)

Clear shift.

Parameters
other – cint/regint/int

square()

Square.

store_in_mem(*address*)

Store in memory by public address.

to_regint(**args*, ***kwargs*)

Convert to regint.

Parameters
n_bits – bit length (int)

Returns
regint

classmethod write_to_socket(*client_id*, *values*, *message_type=0*)

Send a list of clear values to a client.

Parameters

- **client_id** – Client id (regint)
- **values** – list of cint

class Compiler.types.**localint**(*value=None*)

Local integer that must prevented from leaking into the secure computation. Uses regint internally.

Parameters
value – initialization, convertible to regint

output()

Output.

class Compiler.types.**regint**(***kwargs*)

Clear 64-bit integer. Unlike *cint* this is always a 64-bit integer. The type supports the following operations with *regint* or Python integers, always returning *regint*: `+, -, *, %, /, //, **, ^, &, |, <<, >>, ==, !=, <, <=, >, >=`. For operations with other types, see the respective descriptions. Both `/` and `//` stand for floor division.

Parameters

- **val** – initialization (cint/cgf2n/regint/int or list thereof)
- **size** – vector size (int), defaults to 1 or size of list

classmethod Array(*size*, **args*, ***kwargs*)

Type-dependent array. Example:

```
a = sint.Array(10)
```

classmethod Matrix(*rows*, *columns*, **args*, ***kwargs*)

Type-dependent matrix. Example:

```
a = sint.Matrix(10, 10)
```

classmethod MemValue(*value*)

classmethod Tensor(*shape*)

Type-dependent tensor of any dimension:

```
a = sfix.Tensor([10, 10])
```

binary_output(*player=None*)

Write 64-bit signed integer to `Player-Data/Binary-Output-P<playerno>-<threadno>`.

Parameters
player – only output on given player (default all)

static bit_adder(**args*, ***kwargs*)

Binary adder in arithmetic circuits.

Parameters

- **a** – summand (list of 0/1 in compatible type)
- **b** – summand (list of 0/1 in compatible type)
- **carry_in** – input carry (default 0)
- **get_carry** – add final carry to output

Returns
list of 0/1 in relevant type

`bit_and`(*other*)

AND in arithmetic circuits.

Parameters
`self/other` – 0 or 1 (any compatible type)

Return type
depending on inputs (secret if any of them is)

`static bit_compose`(*bits*)

Clear bit composition.

Parameters
`bits` – list of regint/cint/int

`bit_decompose`(**args*, ***kwargs*)

Clear bit decomposition.

Parameters
`bit_length` – number of bits (defaults to global bit length)

Returns
list of regint

`bit_not`()

NOT in arithmetic circuits.

`bit_or`(*other*)

OR in arithmetic circuits.

Parameters
`self/other` – 0 or 1 (any compatible type)

Returns
type depends on inputs (secret if any of them is)

`bit_xor`(*other*)

XOR in arithmetic circuits.

Parameters
`self/other` – 0 or 1 (any compatible type)

Returns
type depends on inputs (secret if any of them is)

`cond_swap`(*a*, *b*)

Swapping in arithmetic circuits.

Parameters
`a/b` – any type supporting the necessary operations

Returns
`(a, b)` if `self` is 0, `(b, a)` if `self` is 1, and undefined otherwise

Return type
depending on operands, secret if any of them is

`classmethod get_random`(**args*, ***kwargs*)

Public insecure randomness.

Parameters

- `bit_length` – number of bits (int)

- **size** – vector size (int, default 1)

half_adder(*other*)

Half adder in arithmetic circuits.

Parameters
self/other – 0 or 1 (any compatible type)

Returns
binary sum, carry

Return type
depending on inputs, secret if any is

if_else(*a*, *b*)

MUX on bit in arithmetic circuits.

Parameters
a/b – any type supporting the necessary operations

Returns
a if `self` is 1, b if `self` is 0, undefined otherwise

Return type
depending on operands, secret if any of them is

classmethod inc(*size*, *base=0*, *step=1*, *repeat=1*, *wrap=None*)

Produce *regint* vector with certain patterns. This is particularly useful for `SubMultiArray.direct_mul()`.

Parameters

- **size** – Result size
- **base** – First value
- **step** – Increase step
- **repeat** – Repeate this many times
- **wrap** – Start over after this many increases

The following produces (1, 1, 1, 3, 3, 3, 5, 5, 5, 7):

```
regint.inc(10, 1, 2, 3)
```

classmethod load_mem(**args*, ***kwargs*)

Load from memory by public address.

classmethod malloc(*size*, *creator_tape=None*)

Allocate memory (statically).

Parameters
size – compile-time (int)

max(*other*)

Maximum.

Parameters
other – any compatible type

min(*other*)

Minimum.

Parameters
other – any compatible type

mod2m(**args*, ***kwargs*)

Clear modulo a power of two.

Return type
cint

classmethod pop(**args*, ***kwargs*)

Pop from stack.

print_if(*string*)

Output string if value is non-zero.

Parameters
string – Python string

print_reg_plain()

Output.

classmethod push(**args*, ***kwargs*)

Push to stack.

Parameters
value – any convertible type

classmethod read_from_socket(**args*, ***kwargs*)

Receive clear integer value(s) from client.

Parameters

- **client_id** – Client id (regint)
- **n** – number of values (default 1)
- **size** – vector size (default 1)

Returns
regint (if n=1) or list of regint

reveal()

Identity.

shuffle()

Returns insecure shuffle of vector.

square()

Square.

store_in_mem(*address*)

Store in memory by public address.

classmethod write_to_socket(*client_id*, *values*, *message_type=0*)

Send a list of clear integers to a client.

Parameters

- **client_id** – Client id (regint)

- **values** – list of regint

class Compiler.types.**sfix**(***kwargs*)

Secret fixed-point number represented as secret integer, by multiplying with `2^f` and then rounding. See *sint* for security considerations of the underlying integer operations.

It supports basic arithmetic (`+, -, *, /`), returning *sfix*, and comparisons (`==, !=, <, <=, >, >=`), returning `sbitint`. The other operand can be any of sfix/sint/cfix/regint/cint/int/float. It also supports `abs()` and `**`, the latter for integer exponents.

Note that the default precision (16 bits after the dot, 31 bits in total) only allows numbers up to $2^{31-16-1} \approx 16000$. You can increase this using *set_precision()*.

Params _v
int/float/regint/cint/sint/sfloat

classmethod Array(*size*, **args*, ***kwargs*)

Type-dependent array. Example:

```
a = sint.Array(10)
```

classmethod Matrix(*rows*, *columns*, **args*, ***kwargs*)

Type-dependent matrix. Example:

```
a = sint.Matrix(10, 10)
```

classmethod MemValue(*value*)

classmethod Tensor(*shape*)

Type-dependent tensor of any dimension:

```
a = sfix.Tensor([10, 10])
```

compute_reciprocal(**args*, ***kwargs*)

Secret fixed-point reciprocal.

classmethod dot_product(*x*, *y*, *res_params=None*)

Secret dot product.

Parameters

- **x** – iterable of appropriate secret type
- **y** – iterable of appropriate secret type and same length

classmethod get_input_from(**args*, ***kwargs*)

Secret fixed-point input.

Parameters

- **player** – public (regint/cint/int)
- **size** – vector size (int, default 1)

classmethod get_random(**args*, ***kwargs*)

Uniform secret random number around centre of bounds. Actual range can be smaller but never larger.

Parameters

- **lower** – float

- **upper** – float
- **size** – vector size (int, default 1)

classmethod input_tensor_from(*player*, *shape*)

Input tensor secretly from player.

Parameters

- **player** – int/regint/cint
- **shape** – tensor shape

classmethod input_tensor_from_client(*client_id*, *shape*)

Input tensor secretly from client.

Parameters

- **client_id** – client identifier (public)
- **shape** – tensor shape

classmethod input_tensor_via(*player*, *content*)

Input tensor-like data via a player. This overwrites the input file for the relevant player. The following returns an `sint` matrix of dimension 2 by 2:

```
M = [[1, 2], [3, 4]]
sint.input_tensor_via(0, M)
```

Make sure to copy `Player-Data/Input-P<player>-0` if running on another host.

classmethod load_mem(**args*, ***kwargs*)

Load from memory by public address.

max(*other*)

Maximum.

Parameters
other – any compatible type

min(*other*)

Minimum.

Parameters
other – any compatible type

classmethod read_from_file(**args*, ***kwargs*)

Read shares from `Persistence/Transactions-P<playerno>.data`. Precision must be the same as when storing.

Parameters

- **start** – starting position in number of shares from beginning (int/regint/cint)
- **n_items** – number of items (int)

Returns
destination for final position, -1 for eof reached, or -2 for file not found (regint)

Returns
list of shares

classmethod receive_from_client(**args*, ***kwargs*)

Securely obtain shares of values input by a client. Assumes client has already converted values to integer representation.

Parameters

- **n** – number of inputs (int)
- **client_id** – regint
- **size** – vector size (default 1)

reveal()

Reveal secret fixed-point number.

Returns
relevant clear type

reveal_to(*player*)

Reveal secret value to `player`. Raw representation possibly written to `Player-Data/Private-Output-P<player>`, but not if `player` is a *regint*.

Parameters
player – public integer (int/regint/cint)

Returns
value to be used with *print_ln_to()*

classmethod reveal_to_clients(*clients*, *values*)

Reveal securely to clients.

Parameters

- **clients** – client ids (list or array)
- **values** – list of values of this class

round_nearest = False

Whether to round deterministically to nearest instead of probabilistically, e.g. after fixed-point multiplication.

classmethod set_precision(*f*, *k=None*)

Set the precision of the integer representation. Note that some operations are undefined when the precision of *sfix* and *cfix* differs. The initial defaults are chosen to allow the best optimization of probabilistic truncation in computation modulo 2^64 (2*k < 64). Generally, 2*k must be at most the integer length for rings and at most m-s-1 for computation modulo an m-bit prime and statistical security s (default 40).

Parameters

- **f** – bit length of decimal part (initial default 16)
- **k** – whole bit length of fixed point, defaults to twice `f` if not given (initial default 31)

square()

Square.

store_in_mem(*address*)

Store in memory by public address.

classmethod write_shares_to_socket(**args*, ***kwargs*)

Send shares of integer representations of a list of values to a specified client socket.

Parameters

- **client_id** – regint
- **values** – list of values of this type

classmethod write_to_file(*shares*)

Write shares of integer representation to `Persistence/Transactions-P<playerno>.data` (appending at the end).

Param
shares (list or iterable of sfix)

class Compiler.types.**sfloat**(***kwargs*)

Secret floating-point number. Represents $(1-2s)\cdot(1-z)\cdot v\cdot 2^p$.

v: significand

p: exponent

z: zero flag

s: sign bit

This uses integer operations internally, see `sint` for security considerations.

The type supports basic arithmetic (`+, -, *, /`), returning `sfloat`, and comparisons (`==, !=, <, <=, >, >=`), returning `sint`. The other operand can be any of sint/cfix/regint/cint/int/float.

Parameters
v – initialization (sfloat/sfix/float/int/sint/cint/regint)

classmethod Array(*size*, **args*, ***kwargs*)

Type-dependent array. Example:

```
a = sint.Array(10)
```

classmethod Matrix(*rows*, *columns*, **args*, ***kwargs*)

Type-dependent matrix. Example:

```
a = sint.Matrix(10, 10)
```

classmethod MemValue(*value*)

classmethod Tensor(*shape*)

Type-dependent tensor of any dimension:

```
a = sfix.Tensor([10, 10])
```

classmethod get_input_from(**args*, ***kwargs*)

Secret floating-point input.

Parameters

- **player** – public (regint/cint/int)
- **size** – vector size (int, default 1)

classmethod input_tensor_from(*player*, *shape*)

Input tensor secretly from player.

Parameters

- **player** – int/regint/cint

- **shape** – tensor shape

classmethod input_tensor_from_client(*client_id*, *shape*)

Input tensor secretly from client.

Parameters

- **client_id** – client identifier (public)
- **shape** – tensor shape

classmethod input_tensor_via(*player*, *content*)

Input tensor-like data via a player. This overwrites the input file for the relevant player. The following returns an `sint` matrix of dimension 2 by 2:

```
M = [[1, 2], [3, 4]]
sint.input_tensor_via(0, M)
```

Make sure to copy `Player-Data/Input-P<player>-0` if running on another host.

classmethod load_mem(**args*, ***kwargs*)

Load from memory by public address.

max(*other*)

Maximum.

Parameters
other – any compatible type

min(*other*)

Minimum.

Parameters
other – any compatible type

reveal()

Reveal secret floating-point number.

Returns
cfloat

round_to_int()

Secret floating-point rounding to integer.

Returns
sint

square()

Square.

store_in_mem(*address*)

Store in memory by public address.

class Compiler.types.**sgf2n**(*val=None*, *size=None*)

Secret $\mathrm{GF}(2^n)$ value. n is chosen at runtime. A number operators are supported (`+, -, *, /, **, ^, ~, ==, !=, <<`), `sgf2n`. Operators generally work with cgf2n/regint/cint/int, except `**, <<`, which require a compile-time integer. `/` refers to field division. `*, /, **` refer to field multiplication and division.

Parameters

- **val** – initialization (sgf2n/cgf2n/regint/int/cint or list thereof)

- **size** – vector size (int), defaults to 1 or size of list

classmethod Array(*size*, **args*, ***kwargs*)

Type-dependent array. Example:

```
a = sint.Array(10)
```

classmethod Matrix(*rows*, *columns*, **args*, ***kwargs*)

Type-dependent matrix. Example:

```
a = sint.Matrix(10, 10)
```

classmethod MemValue(*value*)

classmethod Tensor(*shape*)

Type-dependent tensor of any dimension:

```
a = sfix.Tensor([10, 10])
```

bit_and(*other*)

AND in binary circuits.

Parameters
self/other – 0 or 1 (any compatible type)

Return type
depending on inputs (secret if any of them is)

classmethod bit_compose(*bits*)

Compose value from bits.

Parameters
bits – iterable of any type convertible to sint

bit_decompose(**args*, ***kwargs*)

Secret bit decomposition.

Parameters

- **bit_length** – number of bits
- **step** – use every `step`-th bit

Returns
list of sgf2n

bit_not()

NOT in binary circuits.

bit_or(*other*)

OR in binary circuits.

Parameters
self/other – 0 or 1 (any compatible type)

Returns
type depends on inputs (secret if any of them is)

bit_xor(*other*)

XOR in $\mathrm{GF}(2^n)$ circuits.

Parameters
self/other – 0 or 1 (any compatible type)

Return type
depending on inputs (secret if any of them is)

cond_swap(*a*, *b*, *t=None*)

Swapping in $\mathrm{GF}(2^n)$. Similar to `_int.if_else()`.

classmethod dot_product(**args*, ***kwargs*)

Secret dot product.

Parameters

- **x** – Iterable of secret values
- **y** – Iterable of secret values of same length and type

Return type
same as inputs

equal(*other*, *bit_length=None*, *expand=1*)

Secret comparison.

Parameters
other – sgf2n/cgf2n/regint/int

Returns
0/1 (sgf2n)

classmethod get_input_from(**args*, ***kwargs*)

Secret input from player.

Parameters

- **player** – public (regint/cint/int)
- **size** – vector size (int, default 1)

classmethod get_random_bit(**args*, ***kwargs*)

Secret random bit according to security model.

Returns
0/1 50-50

Parameters
size – vector size (int, default 1)

classmethod get_random_inverse(**args*, ***kwargs*)

Secret random inverse tuple according to security model.

Returns
(a, a^{-1})

Parameters
size – vector size (int, default 1)

classmethod get_random_square(**args*, ***kwargs*)

Secret random square according to security model.

Returns
(a, a^2)

Parameters
size – vector size (int, default 1)

classmethod get_random_triple(**args*, ***kwargs*)

Secret random triple according to security model.

Returns
(a, b, ab)

Parameters
size – vector size (int, default 1)

half_adder(*other*)

Half adder in binary circuits.

Parameters
self/other – 0 or 1 (any compatible type)

Returns
binary sum, carry

Return type
depending on inputs (secret if any of them is)

if_else(a, b)

MUX in $\mathrm{GF}(2^n)$ circuits. Similar to `_int.if_else()`.

classmethod input_tensor_from(*player*, *shape*)

Input tensor secretly from player.

Parameters

- **player** – int/regint/cint
- **shape** – tensor shape

classmethod input_tensor_from_client(*client_id*, *shape*)

Input tensor secretly from client.

Parameters

- **client_id** – client identifier (public)
- **shape** – tensor shape

classmethod input_tensor_via(*player*, *content*)

Input tensor-like data via a player. This overwrites the input file for the relevant player. The following returns an *sint* matrix of dimension 2 by 2:

```
M = [[1, 2], [3, 4]]
sint.input_tensor_via(0, M)
```

Make sure to copy `Player-Data/Input-P<player>-0` if running on another host.

classmethod load_mem(**args*, ***kwargs*)

Load from memory by public address.

classmethod malloc(*size*, *creator_tape=None*)

Allocate memory (statically).

Parameters
size – compile-time (int)

max(*other*)

Maximum.

Parameters
other – any compatible type

min(*other*)

Minimum.

Parameters
other – any compatible type

not_equal(*other*, *bit_length=None*)

Secret comparison.

Parameters
other – sgf2n/cgf2n/regint/int

Returns
0/1 (sgf2n)

reveal(**args*, ***kwargs*)

Reveal secret value publicly.

Return type
relevant clear type

reveal_to(**args*, ***kwargs*)

Reveal secret value to `player`. Result written to `Player-Data/Private-Output-P<player>`

Parameters
player – int

Returns
value to be used with *print_ln_to()*

right_shift(**args*, ***kwargs*)

Secret right shift by public value:

Parameters

- **other** – compile-time (int)
- **bit_length** – number of bits of `self` (defaults to $\mathrm{GF}(2^n)$ bit length)

square(**args*, ***kwargs*)

Secret square.

store_in_mem(*address*)

Store in memory by public address.

class Compiler.types.**sint**(***kwargs*)

Secret integer in the protocol-specific domain. It supports operations with *sint*, *cint*, *regint*, and Python integers. Operations where one of the operands is an *sint* either result in an *sint* or an *sintbit*, the latter for comparisons.

The following operations work as expected in the computation domain (modulo a prime or a power of two): `+`, `-`, `*`. `/` denotes the field division modulo a prime. It will reveal if the divisor is zero. Comparisons operators (`==`, `!=`, `<`, `<=`, `>`, `>=`) assume that the element in the computation domain represents a signed integer in a restricted range, see below. The same holds for `abs()`, shift operators (`<<`, `>>`), modulo (`%`), and exponentation (`**`). Modulo only works if the right-hand operator is a compile-time power of two, and exponentiation only works if the base is two or if the exponent is a compile-time integer.

Most non-linear operations require compile-time parameters for bit length and statistical security. They default to the global parameters set by `program.set_bit_length()` and `program.set_security()`. The acceptable minimum for statistical security is considered to be 40. The defaults for the parameters is output at the beginning of the compilation.

If the computation domain is modulo a power of two, the operands will be truncated to the bit length, and the security parameter does not matter. Modulo prime, the behaviour is undefined and potentially insecure if the operands are longer than the bit length.

Parameters

- **val** – initialization (sint/cint/regint/int/cgf2n or list thereof or sbits/sbitvec/sfix)
- **size** – vector size (int), defaults to 1 or size of list

When converting *sbits*, the result is a vector of bits, and when converting *sbitvec*, the result is a vector of values with bit length equal the length of the input.

classmethod Array(*size*, **args*, ***kwargs*)

Type-dependent array. Example:

```
a = sint.Array(10)
```

classmethod Matrix(*rows*, *columns*, **args*, ***kwargs*)

Type-dependent matrix. Example:

```
a = sint.Matrix(10, 10)
```

classmethod MemValue(*value*)

classmethod Tensor(*shape*)

Type-dependent tensor of any dimension:

```
a = sfix.Tensor([10, 10])
```

static bit_adder(**args*, ***kwargs*)

Binary adder in arithmetic circuits.

Parameters

- **a** – summand (list of 0/1 in compatible type)
- **b** – summand (list of 0/1 in compatible type)
- **carry_in** – input carry (default 0)
- **get_carry** – add final carry to output

Returns

list of 0/1 in relevant type

bit_and(*other*)

AND in arithmetic circuits.

Parameters
self/other – 0 or 1 (any compatible type)

Return type
depending on inputs (secret if any of them is)

classmethod bit_compose(*bits*)

Compose value from bits.

Parameters
bits – iterable of any type convertible to sint

bit_decompose(**args*, ***kwargs*)

Secret bit decomposition.

bit_not()

NOT in arithmetic circuits.

bit_or(*other*)

OR in arithmetic circuits.

Parameters
self/other – 0 or 1 (any compatible type)

Returns
type depends on inputs (secret if any of them is)

bit_xor(*other*)

XOR in arithmetic circuits.

Parameters
self/other – 0 or 1 (any compatible type)

Returns
type depends on inputs (secret if any of them is)

cond_swap(*a*, *b*)

Swapping in arithmetic circuits.

Parameters
a/b – any type supporting the necessary operations

Returns
`(a, b)` if `self` is 0, `(b, a)` if `self` is 1, and undefined otherwise

Return type
depending on operands, secret if any of them is

classmethod dot_product(**args*, ***kwargs*)

Secret dot product.

Parameters

- **x** – Iterable of secret values
- **y** – Iterable of secret values of same length and type

Return type
same as inputs

equal(**args*, ***kwargs*)

Secret comparison (signed).

Parameters

- **other** – sint/cint/regint/int
- **bit_length** – bit length of input (default: global bit length)

Returns
0/1 (sintbit)

classmethod get_dabit(**args*, ***kwargs*)

Bit in arithmetic and binary circuit according to security model

classmethod get_edabit(**args*, ***kwargs*)

Bits in arithmetic and binary circuit

classmethod get_input_from(**args*, ***kwargs*)

Secret input.

Parameters

- **player** – public (regint/cint/int)
- **size** – vector size (int, default 1)

classmethod get_random(**args*, ***kwargs*)

Secret random ring element according to security model.

Parameters
size – vector size (int, default 1)

classmethod get_random_bit(**args*, ***kwargs*)

Secret random bit according to security model.

Returns
0/1 50-50

Parameters
size – vector size (int, default 1)

classmethod get_random_int(**args*, ***kwargs*)

Secret random n-bit number according to security model.

Parameters

- **bits** – compile-time integer (int)
- **size** – vector size (int, default 1)

classmethod get_random_inverse(**args*, ***kwargs*)

Secret random inverse tuple according to security model.

Returns
(a, a^{-1})

Parameters
size – vector size (int, default 1)

classmethod get_random_square(**args*, ***kwargs*)

Secret random square according to security model.

Returns
(a, a^2)

Parameters
`size` – vector size (int, default 1)

`classmethod get_random_triple`(**args*, ***kwargs*)

Secret random triple according to security model.

Returns
(a, b, ab)

Parameters
`size` – vector size (int, default 1)

`greater_equal`(*other*, *bit_length=None*, *security=None*)

Secret comparison (signed).

Parameters

- `other` – sint/cint/regint/int
- `bit_length` – bit length of input (default: global bit length)

Returns
0/1 (sintbit)

`greater_than`(**args*, ***kwargs*)

Secret comparison (signed).

Parameters

- `other` – sint/cint/regint/int
- `bit_length` – bit length of input (default: global bit length)

Returns
0/1 (sintbit)

`half_adder`(*other*)

Half adder in arithmetic circuits.

Parameters
`self/other` – 0 or 1 (any compatible type)

Returns
binary sum, carry

Return type
depending on inputs, secret if any is

`if_else`(a, b)

MUX on bit in arithmetic circuits.

Parameters
`a/b` – any type supporting the necessary operations

Returns
a if `self` is 1, b if `self` is 0, undefined otherwise

Return type
depending on operands, secret if any of them is

classmethod input_tensor_from(*player*, *shape*)

Input tensor secretly from player.

Parameters

- **player** – int/regint/cint
- **shape** – tensor shape

classmethod input_tensor_from_client(*client_id*, *shape*)

Input tensor secretly from client.

Parameters

- **client_id** – client identifier (public)
- **shape** – tensor shape

classmethod input_tensor_via(*player*, *content*)

Input tensor-like data via a player. This overwrites the input file for the relevant player. The following returns an `sint` matrix of dimension 2 by 2:

```
M = [[1, 2], [3, 4]]
sint.input_tensor_via(0, M)
```

Make sure to copy `Player-Data/Input-P<player>-0` if running on another host.

int_div(**args*, ***kwargs*)

Secret integer division.

Parameters

- **other** – sint
- **bit_length** – bit length of input (default: global bit length)

left_shift(*other*, *bit_length=None*, *security=None*)

Secret left shift.

Parameters

- **other** – secret or public integer (sint/cint/regint/int)
- **bit_length** – bit length of input (default: global bit length)

less_equal(*other*, *bit_length=None*, *security=None*)

Secret comparison (signed).

Parameters

- **other** – sint/cint/regint/int
- **bit_length** – bit length of input (default: global bit length)

Returns

0/1 (sintbit)

less_than(**args*, ***kwargs*)

Secret comparison (signed).

Parameters

- **other** – sint/cint/regint/int
- **bit_length** – bit length of input (default: global bit length)

Returns
0/1 (sintbit)

classmethod load_mem(**args*, ***kwargs*)

Load from memory by public address.

classmethod malloc(*size*, *creator_tape=None*)

Allocate memory (statically).

Parameters
size – compile-time (int)

max(*other*)

Maximum.

Parameters
other – any compatible type

min(*other*)

Minimum.

Parameters
other – any compatible type

mod2m(**args*, ***kwargs*)

Secret modulo power of two.

Parameters

- **m** – secret or public integer (sint/cint/regint/int)
- **bit_length** – bit length of input (default: global bit length)

not_equal(*other*, *bit_length=None*, *security=None*)

Secret comparison (signed).

Parameters

- **other** – sint/cint/regint/int
- **bit_length** – bit length of input (default: global bit length)

Returns
0/1 (sintbit)

pow2(**args*, ***kwargs*)

Secret power of two.

Parameters
bit_length – bit length of input (default: global bit length)

private_division(*divisor*, *active=True*, *dividend_length=None*, *divisor_length=None*)

Private integer division as per Veugen and Abspoel

Parameters

- **divisor** – public (cint/regint) or personal value thereof
- **active** – whether to check on the party knowing the divisor (active security)
- **dividend_length** – bit length of the dividend (default: global bit length)
- **dividend_length** – bit length of the divisor (default: global bit length)

classmethod read_from_file(*start*, *n_items*)

Read shares from Persistence/Transactions-P<playerno>.data.

Parameters

- **start** – starting position in number of shares from beginning (int/regint/cint)
- **n_items** – number of items (int)

Returns
destination for final position, -1 for eof reached, or -2 for file not found (regint)

Returns
list of shares

classmethod read_from_socket(**args*, ***kwargs*)

Receive secret-shared value(s) from client.

Parameters

- **client_id** – Client id (regint)
- **n** – number of values (default 1)
- **size** – vector size of values (default 1)

Returns
sint (if n=1) or list of sint

classmethod receive_from_client(**args*, ***kwargs*)

Securely obtain shares of values input by a client. This uses the triple-based input protocol introduced by Damgård et al.

Parameters

- **n** – number of inputs (int)
- **client_id** – regint
- **size** – vector size (default 1)

Returns
list of sint

reveal(**args*, ***kwargs*)

Reveal secret value publicly.

Return type
relevant clear type

reveal_to(**args*, ***kwargs*)

Reveal secret value to player. Result potentially written to Player-Data/Private-Output-P<player>, but not if player is a *regint*.

Parameters
player – public integer (int/regint/cint):

Returns
value to be used with *print_ln_to()*

classmethod reveal_to_clients(*clients*, *values*)

Reveal securely to clients.

Parameters

- **clients** – client ids (list or array)
- **values** – list of sint to reveal

right_shift(**args*, ***kwargs*)

Secret right shift.

Parameters

- **other** – secret or public integer (sint/cint/regint/int)
- **bit_length** – bit length of input (default: global bit length)

round(**args*, ***kwargs*)

Truncate and maybe round secret `k`-bit integer by `m` bits. `m` can be secret if `nearest` is false, in which case the truncation will be exact. For public `m`, `nearest` chooses between nearest rounding (rounding half up) and probablistic truncation.

Parameters

- **k** – int
- **m** – secret or compile-time integer (sint/int)
- **kappa** – statistical security parameter (int)
- **nearest** – bool
- **signed** – bool

square(**args*, ***kwargs*)

Secret square.

store_in_mem(*address*)

Store in memory by public address.

classmethod write_shares_to_socket(*client_id*, *values*, *message_type=0*)

Send shares of a list of values to a specified client socket.

Parameters

- **client_id** – regint
- **values** – list of sint

static write_to_file(*shares*)

Write shares to `Persistence/Transactions-P<playerno>.data` (appending at the end).

Param
shares (list or iterable of sint)

classmethod write_to_socket(**args*, ***kwargs*)

Send a list of shares and MAC shares to a client socket.

Parameters

- **client_id** – regint
- **values** – list of sint

class `Compiler.types.`**sintbit**(***kwargs*)

sint holding a bit, supporting binary operations (`&, |, ^`).

2.1.2 Compiler.GC.types module

This modules contains basic types for binary circuits. The fixed-length types obtained by `get_type(n)` are the preferred way of using them, and in some cases required in connection with container types.

class `Compiler.GC.types.`**`bits`**(*value=None*, *n=None*, *size=None*)

Bases: `Register`, `_structure`, `_bit`

Base class for binary registers.

classmethod **`get_type`**(*length*)

Returns a fixed-length type.

`if_else`(*x*, *y*)

Vectorized oblivious selection:

```
sb32 = sbits.get_type(32)
print_ln('%s', sb32(3).if_else(sb32(5), sb32(2)).reveal())
```

This will output 1.

class `Compiler.GC.types.`**`cbits`**(*value=None*, *n=None*, *size=None*)

Bases: *`bits`*

Clear bits register. Helper type with limited functionality.

class `Compiler.GC.types.`**`sbit`**(**args*, ***kwargs*)

Bases: `bit`, *`sbits`*

Single secret bit.

`if_else`(*x*, *y*)

Non-vectorized oblivious selection:

```
sb32 = sbits.get_type(32)
print_ln('%s', sbit(1).if_else(sb32(5), sb32(2)).reveal())
```

This will output 5.

class `Compiler.GC.types.`**`sbitfix`**(***kwargs*)

Bases: `_fix`

Secret signed integer in one binary register. Use *`set_precision()`* to change the precision.

Example:

```
print_ln('add: %s', (sbitfix(0.5) + sbitfix(0.3)).reveal())
print_ln('mul: %s', (sbitfix(0.5) * sbitfix(0.3)).reveal())
print_ln('sub: %s', (sbitfix(0.5) - sbitfix(0.3)).reveal())
print_ln('lt: %s', (sbitfix(0.5) < sbitfix(0.3)).reveal())
```

will output roughly:

```
add: 0.800003
mul: 0.149994
sub: 0.199997
lt: 0
```

Note that the default precision (16 bits after the dot, 31 bits in total) only allows numbers up to $2^{31-16-1} \approx 16000$. You can increase this using *set_precision()*.

classmethod get_input_from(*player*)

Secret input from `player`.

Param
player (int)

classmethod set_precision(*f*, *k=None*)

Set the precision of the integer representation. Note that some operations are undefined when the precision of `sfix` and `cfix` differs. The initial defaults are chosen to allow the best optimization of probabilistic truncation in computation modulo 2^64 (2*k < 64). Generally, 2*k must be at most the integer length for rings and at most m-s-1 for computation modulo an m-bit prime and statistical security s (default 40).

Parameters

- **f** – bit length of decimal part (initial default 16)
- **k** – whole bit length of fixed point, defaults to twice `f` if not given (initial default 31)

class Compiler.GC.types.**sbitfixvec**(*value=None*, **args*, ***kwargs*)

Bases: `_fix`

Vector of fixed-point numbers for parallel binary computation.

Use *set_precision()* to change the precision.

Example:

```
a = sbitfixvec([sbitfix(0.3), sbitfix(0.5)])
b = sbitfixvec([sbitfix(0.4), sbitfix(0.6)])
c = (a + b).elements()
print_ln('add: %s, %s', c[0].reveal(), c[1].reveal())
c = (a * b).elements()
print_ln('mul: %s, %s', c[0].reveal(), c[1].reveal())
c = (a - b).elements()
print_ln('sub: %s, %s', c[0].reveal(), c[1].reveal())
c = (a < b).bit_decompose()
print_ln('lt: %s, %s', c[0].reveal(), c[1].reveal())
```

This should output roughly:

```
add: 0.699997, 1.10001
mul: 0.119995, 0.300003
sub: -0.0999908, -0.100021
lt: 1, 1
```

classmethod get_input_from(*player*)

Secret input from `player`.

Param
player (int)

classmethod set_precision(*f*, *k=None*)

Set the precision of the integer representation. Note that some operations are undefined when the precision of `sfix` and `cfix` differs. The initial defaults are chosen to allow the best optimization of probabilistic truncation in computation modulo 2^64 (2*k < 64). Generally, 2*k must be at most the integer length for rings and at most m-s-1 for computation modulo an m-bit prime and statistical security s (default 40).

Parameters

- **f** – bit length of decimal part (initial default 16)
- **k** – whole bit length of fixed point, defaults to twice f if not given (initial default 31)

class Compiler.GC.types.**sbitint**(**args*, ***kwargs*)

Bases: _bitint, _number, *sbits*, _sbitintbase

Secret signed integer in one binary register. Use *get_type()* to specify the bit length:

```
si32 = sbitint.get_type(32)
print_ln('add: %s', (si32(5) + si32(3)).reveal())
print_ln('sub: %s', (si32(5) - si32(3)).reveal())
print_ln('mul: %s', (si32(5) * si32(3)).reveal())
print_ln('lt: %s', (si32(5) < si32(3)).reveal())
```

This should output:

```
add: 8
sub: 2
mul: 15
lt: 0
```

classmethod get_type(*n*, *other=None*)

Returns a signed integer type with fixed length.

Parameters
n – length

pow2(*k*)

Computer integer power of two.

Parameters
k – bit length of input

class Compiler.GC.types.**sbitintvec**(*elements=None*, *length=None*, *input_length=None*)

Bases: *sbitvec*, _number, _bitint, _sbitintbase

Vector of signed integers for parallel binary computation:

```
sb32 = sbits.get_type(32)
siv32 = sbitintvec.get_type(32)
a = siv32([sb32(3), sb32(5)])
b = siv32([sb32(4), sb32(6)])
c = (a + b).elements()
print_ln('add: %s, %s', c[0].reveal(), c[1].reveal())
c = (a * b).elements()
print_ln('mul: %s, %s', c[0].reveal(), c[1].reveal())
c = (a - b).elements()
print_ln('sub: %s, %s', c[0].reveal(), c[1].reveal())
c = (a < b).bit_decompose()
print_ln('lt: %s, %s', c[0].reveal(), c[1].reveal())
```

This should output:

```
add: 7, 11
mul: 12, 30
sub: -1, 11
lt: 1, 1
```

pow2(*k*)

Computer integer power of two.

Parameters
k – bit length of input

class Compiler.GC.types.**sbits**(**args*, ***kwargs*)

Bases: *bits*

Secret bits register. This type supports basic bit-wise operations:

```
sb32 = sbits.get_type(32)
a = sb32(3)
b = sb32(5)
print_ln('XOR: %s', (a ^ b).reveal())
print_ln('AND: %s', (a & b).reveal())
print_ln('NOT: %s', (~a).reveal())
```

This will output the following:

```
XOR: 6
AND: 1
NOT: -4
```

Instances can be also be initalized from *regint* and *sint*.

static bit_adder(**args*, ***kwargs*)

Binary adder in binary circuits.

Parameters

- **a** – summand (list of 0/1 in compatible type)
- **b** – summand (list of 0/1 in compatible type)
- **carry_in** – input carry (default 0)
- **get_carry** – add final carry to output

Returns
list of 0/1 in relevant type

classmethod get_input_from(*player*, *n_bits=None*)

Secret input from `player`.

Param
player (int)

popcnt()

Population count / Hamming weight.

Returns
sbits of required length

class Compiler.GC.types.**sbitvec**(*elements=None*, *length=None*, *input_length=None*)

Bases: _vec

Vector of registers of secret bits, effectively a matrix of secret bits. This facilitates parallel arithmetic operations in binary circuits. Container types are not supported, use *sbitvec.get_type* for that.

You can access the rows by member v and the columns by calling elements.

There are three ways to create an instance:

1. By transposition:

```
sb32 = sbits.get_type(32)
x = sbitvec([sb32(5), sb32(3), sb32(0)])
print_ln('%s', [x.v[0].reveal(), x.v[1].reveal(), x.v[2].reveal()])
print_ln('%s', [x.elements()[0].reveal(), x.elements()[1].reveal()])
```

   This should output:

```
[3, 2, 1]
[5, 3]
```

2. Without transposition:

```
sb32 = sbits.get_type(32)
x = sbitvec.from_vec([sb32(5), sb32(3)])
print_ln('%s', [x.v[0].reveal(), x.v[1].reveal()])
```

   This should output:

```
[5, 3]
```

3. From *sint*:

```
y = sint(5)
x = sbitvec(y, 3, 3)
print_ln('%s', [x.v[0].reveal(), x.v[1].reveal(), x.v[2].reveal()])
```

   This should output:

```
[1, 0, 1]
```

classmethod get_type(*n*)

Create type for fixed-length vector of registers of secret bits.

As with *sbitvec*, you can access the rows by member v and the columns by calling elements.

popcnt()

Population count / Hamming weight.

Returns
sbitintvec of required length

2.1.3 Compiler.library module

This module defines functions directly available in high-level programs, in particularly providing flow control and output.

`Compiler.library.accept_client_connection(port)`

Accept client connection on specific port base.

Parameters
port – port base (int/regint/cint)

Returns
client id

`Compiler.library.break_loop()`

Break out of loop.

`Compiler.library.break_point(name='')`

Insert break point. This makes sure that all following code will be executed after preceding code.

Parameters
name – Name for identification (optional)

`Compiler.library.check_point()`

Force MAC checks in current thread and all idle threads if the current thread is the main thread. This implies a break point.

`Compiler.library.crash(condition=None)`

Crash virtual machine.

Parameters
condition – crash if true (default: true)

`Compiler.library.do_while(loop_fn, g=None)`

Do-while loop. The loop is stopped if the return value is zero. It must be public. The following executes exactly once:

```
@do_while
def _():
    ...
    return regint(0)
```

`Compiler.library.for_range(start, stop=None, step=None)`

Decorator to execute loop bodies consecutively. Arguments work as in Python `range()`, but they can by any public integer. Information has to be passed out via container types such as *`Array`* or declaring registers as `global`. Note that changing Python data structures such as lists within the loop is not possible, but the compiler cannot warn about this.

Parameters
start/stop/step – regint/cint/int

Example:

```
a = sint.Array(n)
x = sint(0)
@for_range(n)
def _(i):
    a[i] = i
```

(continues on next page)

(continued from previous page)

```
    global x
    x += 1
```

Note that you cannot overwrite data structures such as *Array* in a loop even when using `global`. Use *assign()* instead.

`Compiler.library.`**`for_range_multithread`**(*n_threads*, *n_parallel*, *n_loops*, *thread_mem_req={}*)

Execute `n_loops` loop bodies in up to `n_threads` threads, up to `n_parallel` in parallel per thread.

Parameters

- **`n_threads/n_parallel`** – compile-time (int)
- **`n_loops`** – regint/cint/int

`Compiler.library.`**`for_range_opt`**(*n_loops*, *budget=None*)

Execute loop bodies in parallel up to an optimization budget. This prevents excessive loop unrolling. The budget is respected even with nested loops. Note that the optimization is rather rudimentary for runtime `n_loops` (regint/cint). Consider using *for_range_parallel()* in this case. Using further control flow constructions inside other than *for_range_opt()* (e.g, *for_range()*) breaks the optimization.

Parameters

- **`n_loops`** – int/regint/cint
- **`budget`** – number of instructions after which to start optimization (default is 100,000)

Example:

```
@for_range_opt(n)
def _(i):
    ...
```

Multidimensional ranges are supported as well. The following executes `f(0, 0)` to `f(4, 2)` in parallel according to the budget.

```
@for_range_opt([5, 3])
def f(i, j):
    ...
```

`Compiler.library.`**`for_range_opt_multithread`**(*n_threads*, *n_loops*)

Execute `n_loops` loop bodies in up to `n_threads` threads, in parallel up to an optimization budget per thread similar to *for_range_opt()*. Note that optimization is rather rudimentary for runtime `n_loops` (regint/cint). Consider using *for_range_multithread()* in this case.

Parameters

- **`n_threads`** – compile-time (int)
- **`n_loops`** – regint/cint/int

The following will execute loop bodies 0-9 in one thread, 10-19 in another etc:

```
@for_range_opt_multithread(8, 80)
def _(i):
    ...
```

Multidimensional ranges are supported as well. The following executes `f(0, 0)` to `f(2, 0)` in one thread and `f(2, 1)` to `f(4, 2)` in another.

```
@for_range_opt_multithread(2, [5, 3])
def f(i, j):
    ...
```

Compiler.library.**for_range_parallel**(*n_parallel*, *n_loops*)

Decorator to execute a loop `n_loops` up to `n_parallel` loop bodies in parallel. Using any other control flow instruction inside the loop breaks the optimization.

Parameters

- **n_parallel** – compile-time (int)
- **n_loops** – regint/cint/int or list of int

Example:

```
@for_range_parallel(n_parallel, n_loops)
def _(i):
    a[i] = a[i] * a[i]
```

Multidimensional ranges are supported as well. The following executes `f(0, 0)` to `f(4, 2)`, two calls in parallel.

```
@for_range_parallel(2, [5, 3])
def f(i, j):
    ...
```

Compiler.library.**foreach_enumerate**(*a*)

Run-time loop over public data. This uses `Player-Data/Public-Input/<progname>`. Example:

```
@foreach_enumerate([2, 8, 3])
def _(i, j):
    print_ln('%s: %s', i, j)
```

This will output:

```
0: 2
1: 8
2: 3
```

Compiler.library.**get_arg**(**args*, ***kwargs*)

Returns the thread argument.

Compiler.library.**get_number_of_players**()

Returns
the number of players

Return type
regint

Compiler.library.**get_player_id**()

Returns
player number

Return type
localint (cannot be used for computation)

`Compiler.library.get_thread_number(*args, **kwargs)`

Returns the thread number.

`Compiler.library.get_threshold()`

The threshold is the maximal number of corrupted players.

Return type
regint

`Compiler.library.if_(condition)`

Conditional execution without else block.

Parameters
condition – regint/cint/int

Usage:

```
@if_(x > 0)
def _():
    ...
```

`Compiler.library.if_e(condition)`

Conditional execution with else block. Use `MemValue` to assign values that live beyond.

Parameters
condition – regint/cint/int

Usage:

```
y = MemValue(0)
@if_e(x > 0)
def _():
    y.write(1)
@else_
def _():
    y.write(0)
```

`Compiler.library.listen_for_clients(port)`

Listen for clients on specific port base.

Parameters
port – port base (int/regint/cint)

`Compiler.library.map_sum_opt(n_threads, n_loops, types)`

Multi-threaded sum reduction. The following computes a sum of ten squares in three threads:

```
@map_sum_opt(3, 10, [sint])
def summer(i):
    return sint(i) ** 2

result = summer()
```

Parameters

- **n_threads** – number of threads (int)
- **n_loops** – number of loop runs (regint/cint/int)
- **types** – return type, must match the return statement in the loop

`Compiler.library.multithread(n_threads, n_items=None, max_size=None)`

Distribute the computation of `n_items` to `n_threads` threads, but leave the in-thread repetition up to the user.

Parameters

- **n_threads** – compile-time (int)
- **n_items** – regint/cint/int (default: `n_threads`)

The following executes `f(0, 8)`, `f(8, 8)`, and `f(16, 9)` in three different threads:

```
@multithread(8, 25)
def f(base, size):
    ...
```

`Compiler.library.print_float_precision(n)`

Set the precision for floating-point printing.

Parameters

n – number of digits (int)

`Compiler.library.print_ln(s='', *args)`

Print line, with optional args for adding variables/registers with `%s`. By default only player 0 outputs, but the `-I` command-line option changes that.

Parameters

- **s** – Python string with same number of `%s` as length of `args`
- **args** – list of public values (regint/cint/int/cfix/cfloat/localint)

Example:

```
print_ln('a is %s.', a.reveal())
```

`Compiler.library.print_ln_if(cond, ss, *args)`

Print line if `cond` is true. The further arguments are treated as in *print_str()*/*print_ln()*.

Parameters

- **cond** – regint/cint/int/localint
- **ss** – Python string
- **args** – list of public values

Example:

```
print_ln_if(get_player_id() == 0, 'Player 0 here')
```

`Compiler.library.print_ln_to(player, ss, *args)`

Print line at `player` only. Note that printing is disabled by default except at player 0. Activate interactive mode with *-I* to enable it for all players.

Parameters

- **player** – int
- **ss** – Python string
- **args** – list of values known to `player`

Example:

```
print_ln_to(player, 'output for %s: %s', player, x.reveal_to(player))
```

Compiler.library.**print_str**(*s*, **args*)

Print a string, with optional args for adding variables/registers with `%s`.

Compiler.library.**print_str_if**(*cond*, *ss*, **args*)

Print string conditionally. See *print_ln_if()* for details.

Compiler.library.**public_input**()

Public input read from `Programs/Public-Input/<progname>`.

Compiler.library.**runtime_error**(*msg=''*, **args*)

Print an error message and abort the runtime. Parameters work as in *print_ln()*

Compiler.library.**runtime_error_if**(*condition*, *msg=''*, **args*)

Conditionally print an error message and abort the runtime.

Parameters

- **condition** – regint/cint/int/cbit
- **msg** – message
- **args** – list of public values to fit `%s` in the message

Compiler.library.**start_timer**(*timer_id=0*)

Start timer. Timer 0 runs from the start of the program. The total time of all used timers is output at the end. Fails if already running.

Parameters
timer_id – compile-time (int)

Compiler.library.**stop_timer**(*timer_id=0*)

Stop timer. Fails if not running.

Parameters
timer_id – compile-time (int)

Compiler.library.**while_do**(*condition*, **args*)

While-do loop. The decorator requires an initialization, and the loop body function must return a suitable input for `condition`.

Parameters

- **condition** – function returning public integer (regint/cint/int)
- **args** – arguments given to `condition` and loop body

The following executes an ten-fold loop:

```
@while_do(lambda x: x < 10, regint(0))
def f(i):
    ...
    return i + 1
```

2.1.4 Compiler.mpc_math module

Module for math operations.

Implements trigonometric and logarithmic functions.

This has to imported explicitly.

`Compiler.mpc_math.atan(*args, **kwargs)`

Returns the arctangent (sfix) of any given fractional value.

Parameters
x – fractional input (sfix).

Returns
arctan of **x** (sfix).

`Compiler.mpc_math.acos(x)`

Returns the arccosine (sfix) of any given fractional value.

Parameters
x – fractional input (sfix). $-1 \leq x \leq 1$

Returns
arccos of **x** (sfix).

`Compiler.mpc_math.asin(x)`

Returns the arcsine (sfix) of any given fractional value.

Parameters
x – fractional input (sfix). valid interval is $-1 \leq x \leq 1$

Returns
arcsin of **x** (sfix).

`Compiler.mpc_math.cos(*args, **kwargs)`

Returns the cosine of any given fractional value.

Parameters
x – fractional input (sfix, sfloat)

Returns
cos of **x** (sfix, sfloat)

`Compiler.mpc_math.exp2_fx(self, *args, **kwargs)`

Power of two for fixed-point numbers.

Parameters

- **a** – exponent for 2^a (sfix)
- **zero_output** – whether to output zero for very small values. If not, the result will be undefined.

Returns
2^a if it is within the range. Undefined otherwise

`Compiler.mpc_math.InvertSqrt(self, *args, **kwargs)`

Reciprocal square root approximation by Lu et al.

`Compiler.mpc_math.log2_fx(self, *args, **kwargs)`

Returns the result of $\log_2(x)$ for any unbounded number. This is achieved by changing `x` into $f \cdot 2^n$ where f is bounded by $[0.5, 1]$. Then the polynomials are used to calculate $\log_2(f)$, which is then just added to n.

Parameters
`x` – input for $\log_2$ (sfix, sint).

Returns
(sfix) the value of $\log_2(x)$

`Compiler.mpc_math.log_fx(x, b)`

Returns the value of the expression $\log_b(x)$ where `x` is secret shared. It uses `log2_fx()` to calculate the expression $\log_b(2) \cdot \log_2(x)$.

Parameters

- `x` – (sfix, sint) secret shared coefficient for log.
- `b` – (float) base for log operation.

Returns
(sfix) the value of $log_b(x)$.

`Compiler.mpc_math.pow_fx(x, y)`

Returns the value of the expression x^y where both inputs are secret shared. It uses `log2_fx()` together with `exp2_fx()` to calculate the expression $2^{y \log_2(x)}$.

Parameters

- `x` – (sfix) secret shared base.
- `y` – (sfix, clear types) secret shared exponent.

Returns
x^y (sfix) if positive and in range

`Compiler.mpc_math.sin(*args, **kwargs)`

Returns the sine of any given fractional value.

Parameters
`x` – fractional input (sfix, sfloat)

Returns
sin of `x` (sfix, sfloat)

`Compiler.mpc_math.sqrt(self, *args, **kwargs)`

Returns the square root (sfix) of any given fractional value as long as it can be rounded to a integral value with `f` bits of decimal precision.

Parameters
`x` – fractional input (sfix).

Returns
square root of `x` (sfix).

`Compiler.mpc_math.tan(*args, **kwargs)`

Returns the tangent of any given fractional value.

Parameters
`x` – fractional input (sfix, sfloat)

Returns
tan of `x` (sfix, sfloat)

`Compiler.mpc_math.tanh(x)`

Hyperbolic tangent. For efficiency, accuracy is diminished around $\pm \log(k - f - 2)/2$ where k and f denote the fixed-point parameters.

2.1.5 Compiler.ml module

This module contains machine learning functionality. It is work in progress, so you must expect things to change. The only tested functionality for training is using consecutive layers. This includes logistic regression. It can be run as follows:

```
sgd = ml.SGD([ml.Dense(n_examples, n_features, 1),
              ml.Output(n_examples, approx=True)], n_epochs,
             report_loss=True)
sgd.layers[0].X.input_from(0)
sgd.layers[1].Y.input_from(1)
sgd.reset()
sgd.run()
```

This loads measurements from party 0 and labels (0/1) from party 1. After running, the model is stored in `sgd.layers[0].W` and `sgd.layers[0].b`. The `approx` parameter determines whether to use an approximate sigmoid function. Setting it to 5 uses a five-piece approximation instead of a three-piece one.

A simple network for MNIST using two dense layers can be trained as follows:

```
sgd = ml.SGD([ml.Dense(60000, 784, 128, activation='relu'),
              ml.Dense(60000, 128, 10),
              ml.MultiOutput(60000, 10)], n_epochs,
             report_loss=True)
sgd.layers[0].X.input_from(0)
sgd.layers[1].Y.input_from(1)
sgd.reset()
sgd.run()
```

See this repository for scripts importing MNIST training data and further examples.

Inference can be run as follows:

```
data = sfix.Matrix(n_test, n_features)
data.input_from(0)
res = sgd.eval(data)
print_ln('Results: %s', [x.reveal() for x in res])
```

For inference/classification, this module offers the layers necessary for neural networks such as DenseNet, ResNet, and SqueezeNet. A minimal example using input from player 0 and model from player 1 looks as follows:

```
graph = Optimizer()
graph.layers = layers
layers[0].X.input_from(0)
for layer in layers:
    layer.input_from(1)
graph.forward(1)
res = layers[-1].Y
```

See the readme for an example of how to run MP-SPDZ on TensorFlow graphs.

class Compiler.ml.**Adam**(*layers*, *n_epochs=1*, *approx=False*, *amsgrad=False*, *normalize=False*)

Bases: *Optimizer*

Adam/AMSgrad optimizer.

Parameters

- **layers** – layers of linear graph
- **approx** – use approximation for inverse square root (bool)
- **amsgrad** – use AMSgrad (bool)

class Compiler.ml.**Add**(*inputs*)

Bases: NoVariableLayer

Fixed-point addition layer.

Parameters

inputs – two input layers with same shape (tuple/list)

class Compiler.ml.**Argmax**(*shape*)

Bases: NoVariableLayer

Fixed-point Argmax layer.

Parameters

shape – input shape (tuple/list of two int)

class Compiler.ml.**Concat**(*inputs*, *dimension*)

Bases: NoVariableLayer

Fixed-point concatentation layer.

Parameters

- **inputs** – two input layers (tuple/list)
- **dimension** – dimension for concatenation (must be 3)

class Compiler.ml.**Dense**(*N*, *d_in*, *d_out*, *d=1*, *activation='id'*, *debug=False*)

Bases: DenseBase

Fixed-point dense (matrix multiplication) layer.

Parameters

- **N** – number of examples
- **d_in** – input dimension
- **d_out** – output dimension

class Compiler.ml.**Dropout**(*N*, *d1*, *d2=1*, *alpha=0.5*)

Bases: NoVariableLayer

Dropout layer.

Parameters

- **N** – number of examples
- **d1** – total dimension
- **alpha** – probability (power of two)

class Compiler.ml.**FixAveragePool2d**(*input_shape*, *output_shape*, *filter_size*, *strides=(1, 1)*)

Bases: FixBase, AveragePool2d

Fixed-point 2D AvgPool layer.

Parameters

- **input_shape** – input shape (tuple/list of four int)
- **output_shape** – output shape (tuple/list of four int)
- **filter_size** – filter size (tuple/list of two int)
- **strides** – strides (tuple/list of two int)

class Compiler.ml.**FixConv2d**(*input_shape*, *weight_shape*, *bias_shape*, *output_shape*, *stride*, *padding='SAME'*, *tf_weight_format=False*, *inputs=None*)

Bases: Conv2d, FixBase

Fixed-point 2D convolution layer.

Parameters

- **input_shape** – input shape (tuple/list of four int)
- **weight_shape** – weight shape (tuple/list of four int)
- **bias_shape** – bias shape (tuple/list of one int)
- **output_shape** – output shape (tuple/list of four int)
- **stride** – stride (tuple/list of two int)
- **padding** – 'SAME' (default), 'VALID', or tuple/list of two int
- **tf_weight_format** – weight shape format is (height, width, input channels, output channels) instead of the default (output channels, height, width, input channels)

class Compiler.ml.**FusedBatchNorm**(*shape*, *inputs=None*)

Bases: Layer

Fixed-point fused batch normalization layer.

Parameters

shape – input/output shape (tuple/list of four int)

class Compiler.ml.**MaxPool**(*shape*, *strides=(1, 2, 2, 1)*, *ksize=(1, 2, 2, 1)*, *padding='VALID'*)

Bases: NoVariableLayer

Fixed-point MaxPool layer.

Parameters

- **shape** – input shape (tuple/list of four int)
- **strides** – strides (tuple/list of four int, first and last must be 1)
- **ksize** – kernel size (tuple/list of four int, first and last must be 1)
- **padding** – 'VALID' (default) or 'SAME'

class Compiler.ml.**MultiOutput**(*N*, *d_out*, *approx=False*, *debug=False*)

Bases: MultiOutputBase

Output layer for multi-class classification with softmax and cross entropy.

Parameters

- **N** – number of examples
- **d_out** – number of classes
- **approx** – use ReLU division instead of softmax for the loss

class Compiler.ml.**Optimizer**(*report_loss=None*)

Bases: `object`

Base class for graphs of layers.

backward(***kwargs*)

Compute backward propagation.

eval(***kwargs*)

Compute evaluation after training.

Parameters

- **data** – sample data (`Compiler.types.Matrix` with one row per sample)
- **top** – return top prediction instead of probability distribution

forward(***kwargs*)

Compute graph.

Parameters

- **N** – batch size (used if batch not given)
- **batch** – indices for computation (`Array` or list)
- **keep_intermediate** – do not free memory of intermediate results after use

property layers

Get all layers.

reset()

Initialize weights.

run(***kwargs*)

Run training.

Parameters

- **batch_size** – batch size (defaults to example size of first layer)
- **stop_on_loss** – stop when loss falls below this (default: 0)

set_layers_with_inputs(*layers*)

Construct graph from `inputs` members of list of layers.

class Compiler.ml.**Output**(*N*, *debug=False*, *approx=False*)

Bases: `NoVariableLayer`

Fixed-point logistic regression output layer.

Parameters

- **N** – number of examples
- **approx** – `False` (default) or parameter for `approx_sigmoid`

class Compiler.ml.**Relu**(*shape*, *inputs=None*)

Bases: ElementWiseLayer

Fixed-point ReLU layer.

Parameters
shape – input/output shape (tuple/list of int)

static f(*x*)

ReLU function (maximum of input and zero).

static f_prime(*x*)

ReLU derivative.

prime_type

alias of *sint*

class Compiler.ml.**ReluMultiOutput**(*N*, *d_out*, *approx=False*, *debug=False*)

Bases: MultiOutputBase

Output layer for multi-class classification with back-propagation based on ReLU division.

Parameters

- **N** – number of examples
- **d_out** – number of classes

class Compiler.ml.**SGD**(*layers*, *n_epochs*, *debug=False*, *report_loss=None*)

Bases: *Optimizer*

Stochastic gradient descent.

Parameters

- **layers** – layers of linear graph
- **n_epochs** – number of epochs for training
- **report_loss** – disclose and print loss

reset(***kwargs*)

Reset layer parameters.

Parameters
X_by_label – if given, set training data by public labels for balancing

class Compiler.ml.**Square**(*shape*, *inputs=None*)

Bases: ElementWiseLayer

Fixed-point square layer.

Parameters
shape – input/output shape (tuple/list of int)

prime_type

alias of *sfix*

Compiler.ml.**argmax**(*x*)

Compute index of maximum element.

Parameters
x – iterable

Returns
sint

`Compiler.ml.mr(A, n_iterations, stop=False)`

Iterative matrix inverse approximation.

Parameters

- **A** – matrix to invert
- **n_iterations** – maximum number of iterations
- **stop** – whether to stop when converged (implies revealing)

`Compiler.ml.relu(x)`

ReLU function (maximum of input and zero).

`Compiler.ml.relu_prime(x)`

ReLU derivative.

`Compiler.ml.sigmoid(x)`

Sigmoid function.

Parameters
x – sfix

`Compiler.ml.sigmoid_prime(x)`

Sigmoid derivative.

Parameters
x – sfix

`Compiler.ml.softmax(x)`

Softmax.

Parameters
x – vector or list of sfix

Returns
sfix vector

`Compiler.ml.solve_linear(A, b, n_iterations, progress=False, n_threads=None, stop=False, already_symmetric=False, precond=False)`

Iterative linear solution approximation for $Ax = b$.

Parameters

- **progress** – print some information on the progress (implies revealing)
- **n_threads** – number of threads to use
- **stop** – whether to stop when converged (implies revealing)

`Compiler.ml.var(x)`

Variance.

`Compiler.ml.approx_sigmoid(*args, **kwargs)`

Piece-wise approximate sigmoid as in Dahl et al.

Parameters

- **x** – input
- **n** – number of pieces, 3 (default) or 5

2.1.6 Compiler.circuit module

This module contains functionality using circuits in the so-called Bristol Fashion format. You can download a few examples including the ones used below into `Programs/Circuits` as follows:

```
make Programs/Circuits
```

class Compiler.circuit.**Circuit**(*name*)

Use a Bristol Fashion circuit in a high-level program. The following example adds signed 64-bit inputs from two different parties and prints the result:

```
from circuit import Circuit
sb64 = sbits.get_type(64)
adder = Circuit('adder64')
a, b = [sbitvec(sb64.get_input_from(i)) for i in (0, 1)]
print_ln('%s', adder(a, b).elements()[0].reveal())
```

Circuits can also be executed in parallel as the following example shows:

```
from circuit import Circuit
sb128 = sbits.get_type(128)
key = sb128(0x2b7e151628aed2a6abf7158809cf4f3c)
plaintext = sb128(0x6bc1bee22e409f96e93d7e117393172a)
n = 1000
aes128 = Circuit('aes_128')
ciphertexts = aes128(sbitvec([key] * n), sbitvec([plaintext] * n))
ciphertexts.elements()[n - 1].reveal().print_reg()
```

This executes AES-128 1000 times in parallel and then outputs the last result, which should be `0x3ad77bb40d7a3660a89ecaf32466ef97`, one of the test vectors for AES-128.

class Compiler.circuit.**ieee_float**(*value*)

This gives access IEEE754 floating-point operations using Bristol Fashion circuits. The following example computes the standard deviation of 10 integers input by each of party 0 and 1:

```
from circuit import ieee_float

values = []

for i in range(2):
    for j in range(10):
        values.append(sbitint.get_type(64).get_input_from(i))

fvalues = [ieee_float(x) for x in values]

avg = sum(fvalues) / ieee_float(len(fvalues))
var = sum(x * x for x in fvalues) / ieee_float(len(fvalues)) - avg * avg
stddev = var.sqrt()

print_ln('avg: %s', avg.reveal())
print_ln('var: %s', var.reveal())
print_ln('stddev: %s', stddev.reveal())
```

Compiler.circuit.**sha3_256**(*x*)

This function implements SHA3-256 for inputs of up to 1080 bits:

```
from circuit import sha3_256
a = sbitvec.from_vec([])
b = sbitvec(sint(0xcc), 8, 8)
for x in a, b:
    sha3_256(x).elements()[0].reveal().print_reg()
```

This should output the first two test vectors of SHA3-256 in byte-reversed order:

```
0x4a43f8804b0ad882fa493be44dff80f562d661a05647c15166d71ebff8c6ffa7
0xf0d7aa0ab2d92d580bb080e17cbb52627932ba37f085d3931270d31c39357067
```

Note that `sint` to `sbitvec` conversion is only implemented for computation modulo a power of two.

2.1.7 Compiler.program module

This module contains the building blocks of the compiler such as code blocks and registers. Most relevant is the central *Program* object that holds various properties of the computation.

class Compiler.program.**Program**(*args*, *options=<class 'Compiler.program.defaults'>*)

A program consists of a list of tapes representing the whole computation.

When compiling an `.mpc` file, the single instances is available as `program` in order. When compiling directly from Python code, an instance has to be created before running any instructions.

join_tapes(*thread_numbers*)

Wait for completion of tapes. See *new_tape()* for an example.

Parameters
thread_numbers – list of thread numbers

new_tape(*function*, *args=[]*, *name=None*, *single_thread=False*)

Create a new tape from a function. See *multithread()* and *for_range_opt_multithread()* for easier-to-use higher-level functionality. The following runs two threads defined by two different functions:

```
def f():
    ...
def g():
    ...
tapes = [program.new_tape(x) for x in (f, g)]
thread_numbers = program.run_tapes(tapes)
program.join_tapes(threads_numbers)
```

Parameters

- **function** – Python function defining the thread
- **args** – arguments to the function
- **name** – name used for files
- **single_thread** – Boolean indicating whether tape will never be run in parallel to itself

Returns
tape handle

options_from_args()

Set a number of options from the command-line arguments.

public_input(*x*)

Append a value to the public input file.

run_tapes(*args*)

Run tapes in parallel. See *new_tape()* for an example.

Parameters
args – list of tape handles or tuples of tape handle and extra argument (for *get_arg()*)

Returns
list of thread numbers

property security

The statistical security parameter for non-linear functions.

set_bit_length(*bit_length*)

Change the integer bit length for non-linear functions.

use_dabit

Setting whether to use daBits for non-linear functionality.

use_edabit(*change=None*)

Setting whether to use edaBits for non-linear functionality (default: false).

Parameters
change – change setting if not `None`

Returns
setting if `change` is `None`

use_split(*change=None*)

Setting whether to use local arithmetic-binary share conversion for non-linear functionality (default: false).

Parameters
change – change setting if not `None`

Returns
setting if `change` is `None`

use_square(*change=None*)

Setting whether to use preprocessed square tuples (default: false).

Parameters
change – change setting if not `None`

Returns
setting if `change` is `None`

property use_trunc_pr

Setting whether to use special probabilistic truncation.

2.1.8 Compiler.oram module

This module contains an implementation of the tree-based oblivious RAM as proposed by Shi et al. as well as the straight-forward construction using linear scanning. Unlike `Array`, this allows access by a secret index:

```
a = OptimalORAM(1000)
i = sint.get_input_from(0)
a[i] = sint.get_input_from(1)
```

`Compiler.oram.`**`OptimalORAM`**(*size*, **args*, ***kwargs*)

Create an ORAM instance suitable for the size based on experiments.

Parameters

- **size** – number of elements
- **value_type** – `sint` (default) / `sg2fn`

2.2 Virtual Machine

Calling `compile.py` outputs the computation in a format specific to MP-SPDZ. This includes a schedule file and one or several bytecode files. The schedule file can be found at `Programs/Schedules/<progname>.sch`. It contains the names of all bytecode files found in `Programs/Bytecode` and the maximum number of parallel threads. Each bytecode file represents the complete computation of one thread, also called tape. The computation of the main thread is always `Programs/Bytecode/<progname>-0.bc` when compiled by the compiler.

The bytecode is made up of 32-bit units in big-endian byte order. Every unit represents an instruction code (possibly including vector size), register number, or immediate value.

For example, adding the secret integers in registers 1 and 2 and then storing the result at register 0 leads to the following bytecode (in hexadecimal representation):

```
00 00 00 21  00 00 00 00  00 00 00 01  00 00 00 02
```

This is because `0x021` is the code of secret integer addition. The debugging output (`compile.py -a <prefix>`) looks as follows:

```
adds s0, s1, s2 # <instruction number>
```

There is also a vectorized addition. Adding 10 secret integers in registers 10-19 and 20-29 and then storing the result in registers 0-9 is represented as follows in bytecode:

```
00 00 28 21  00 00 00 00  00 00 00 0a  00 00 00 14
```

This is because the vector size is stored in the upper 22 bits of the first 32-bit unit (instruction codes are up to 10 bits long), and `0x28` equals 40 or 10 shifted by two bits. In the debugging output the vectorized addition looks as follows:

```
vadds 10, s0, s10, s20 # <instruction number>
```

Finally, some instructions have a variable number of arguments to accommodate any number of parallel operations. For these, the first argument usually indicates the number of arguments yet to come. For example, multiplying the secret integers in registers 2 and 3 as well as registers 4 and 5 and the storing the two results in registers 0 and 1 results in the following bytecode:

```
00 00 00 a6  00 00 00 06  00 00 00 00  00 00 00 02
00 00 00 03  00 00 00 01  00 00 00 04  00 00 00 05
```

and the following debugging output:

```
muls 6, s0, s2, s3, s1, s4, s5 # <instruction number>
```

Note that calling instructions in high-level code never is done with the explicit number of arguments. Instead, this is derived from number of function arguments. The example above would this simply be called as follows:

```
muls(s0, s2, s3, s1, s4, s5)
```

2.2.1 Instructions

The following table list all instructions except the ones for $\mathrm{GF}(2^n)$ computation, untested ones, and those considered obsolete.

Name	Code	
CISC	0x0	Meta instruction for emulation
LDI	0x1	Assign (constant) immediate value to clear register (vector)
LDSI	0x2	Assign (constant) immediate value to secret register (vector)
LDMC	0x3	Assign clear memory value(s) to clear register (vector) by immediate address
LDMS	0x4	Assign secret memory value(s) to secret register (vector) by immediate address
STMC	0x5	Assign clear register (vector) to clear memory value(s) by immediate address
STMS	0x6	Assign secret register (vector) to secret memory value(s) by immediate address
LDMCI	0x7	Assign clear memory value(s) to clear register (vector) by register address
LDMSI	0x8	Assign secret memory value(s) to secret register (vector) by register address
STMCI	0x9	Assign clear register (vector) to clear memory value(s) by register address
STMSI	0xa	Assign secret register (vector) to secret memory value(s) by register address
MOVC	0xb	Copy clear register (vector)
MOVS	0xc	Copy secret register (vector)
LDTN	0x10	Store the number of the current thread in clear integer register
LDARG	0x11	Store the argument passed to the current thread in clear integer register
REQBL	0x12	Requirement on computation modulus
STARG	0x13	Copy clear integer register to the thread argument
TIME	0x14	Output time since start of computation
START	0x15	Start timer
STOP	0x16	Stop timer
USE	0x17	Offline data usage
USE_INP	0x18	Input usage
RUN_TAPE	0x19	Start tape/bytecode file in another thread
JOIN_TAPE	0x1a	Join thread
CRASH	0x1b	Crash runtime if the value in the register is not zero
USE_PREP	0x1c	Custom preprocessed data usage
USE_MATMUL	0x1f	Matrix multiplication usage

continues on next page

Table 1 – continued from previous page

Name	Code	
ADDC	0x20	Clear addition
ADDS	0x21	Secret addition
ADDM	0x22	Mixed addition
ADDCI	0x23	Addition of clear register (vector) and (constant) immediate value
ADDSI	0x24	Addition of secret register (vector) and (constant) immediate value
SUBC	0x25	Clear subtraction
SUBS	0x26	Secret subtraction
SUBML	0x27	Subtract clear from secret value
SUBMR	0x28	Subtract secret from clear value
SUBCI	0x29	Subtraction of (constant) immediate value from clear register (vector)
SUBSI	0x2a	Subtraction of (constant) immediate value from secret register (vector)
SUBCFI	0x2b	Subtraction of clear register (vector) from (constant) immediate value
SUBSFI	0x2c	Subtraction of secret register (vector) from (constant) immediate value
MULC	0x30	Clear multiplication
MULM	0x31	Multiply secret and clear value
MULCI	0x32	Multiplication of clear register (vector) and (constant) immediate value
MULSI	0x33	Multiplication of secret register (vector) and (constant) immediate value
DIVC	0x34	Clear division
DIVCI	0x35	Division of secret register (vector) and (constant) immediate value
MODC	0x36	Clear modular reduction
MODCI	0x37	Modular reduction of clear register (vector) and (constant) immediate value
LEGENDREC	0x38	Clear Legendre symbol computation (a/p) over prime p (the computation modulus)
DIGESTC	0x39	Clear truncated hash computation
INV2M	0x3a	Inverse of power of two modulo prime (the computation modulus)
FLOORDIVC	0x3b	Clear integer floor division
TRIPLE	0x50	Store fresh random triple(s) in secret register (vectors)
BIT	0x51	Store fresh random triple(s) in secret register (vectors)
SQUARE	0x52	Store fresh random square(s) in secret register (vectors)
INV	0x53	Store fresh random inverse(s) in secret register (vectors)
PREP	0x57	Store custom preprocessed data in secret register (vectors)
DABIT	0x58	Store fresh random daBit(s) in secret register (vectors)
EDABIT	0x59	Store fresh random loose edaBit(s) in secret register (vectors)
SEDABIT	0x5a	Store fresh random strict edaBit(s) in secret register (vectors)
RANDOMS	0x5b	Store fresh length-restricted random shares(s) in secret register (vectors)
INPUTMASKREG	0x5c	Store fresh random input mask(s) in secret register (vector) and clear register (vector) of the relevant player
RANDOMFULLS	0x5d	Store share(s) of a fresh secret random element in secret register (vectors)
READSOCKETC	0x63	Read a variable number of clear values in internal representation from socket for a specified client id and store them in clear registers
READSOCKETS	0x64	Read a variable number of secret shares (potentially with MAC) from a socket for a client id and store them in registers

continues on next page

Table 1 – continued from previous page

Name	Code	
WRITESOCKETS	0x66	Write a variable number of secret shares (potentially with MAC) from registers into a socket for a specified client id
READSOCKETINT	0x69	Read a variable number of 32-bit integers from socket for a specified client id and store them in clear integer registers
WRITESOCKETSHARE	0x6b	Write a variable number of shares (without MACs) from secret registers into socket for a specified client id
LISTEN	0x6c	Open a server socket on a party-specific port number and listen for client connections (non-blocking)
ACCEPTCLIENTCONNECTION	0x6d	Wait for a connection at the given port and write socket handle to clear integer register
CLOSECLIENTCONNECTION	0x6e	Close connection to client
ANDC	0x70	Logical AND of clear (vector) registers
XORC	0x71	Logical XOR of clear (vector) registers
ORC	0x72	Logical OR of clear (vector) registers
ANDCI	0x73	Logical AND of clear register (vector) and (constant) immediate value
XORCI	0x74	Logical XOR of clear register (vector) and (constant) immediate value
ORCI	0x75	Logical OR of clear register (vector) and (constant) immediate value
NOTC	0x76	Clear logical NOT of a constant number of bits of clear (vector) register
SHLC	0x80	Bitwise left shift of clear register (vector)
SHRC	0x81	Bitwise right shift of clear register (vector)
SHLCI	0x82	Bitwise left shift of clear register (vector) by (constant) immediate value
SHRCI	0x83	Bitwise right shift of clear register (vector) by (constant) immediate value
SHRSI	0x84	Bitwise right shift of secret register (vector) by (constant) immediate value
JMP	0x90	Unconditional relative jump in the bytecode (compile-time parameter)
JMPNZ	0x91	Conditional relative jump in the bytecode
JMPEQZ	0x92	Conditional relative jump in the bytecode
EQZC	0x93	Clear integer zero test
LTZC	0x94	Clear integer less than zero test
LTC	0x95	Clear integer less-than comparison
GTC	0x96	Clear integer greater-than comparison
EQC	0x97	Clear integer equality test
JMPI	0x98	Unconditional relative jump in the bytecode (run-time parameter)
BITDECINT	0x99	Clear integer bit decomposition
LDINT	0x9a	Store (constant) immediate value in clear integer register (vector)
ADDINT	0x9b	Clear integer register (vector) addition
SUBINT	0x9c	Clear integer register (vector) subtraction
MULINT	0x9d	Clear integer register (element-wise vector) multiplication
DIVINT	0x9e	Clear integer register (element-wise vector) division with floor rounding
PRINTINT	0x9f	Output clear integer register
OPEN	0xa5	Reveal secret registers (vectors) to clear registers (vectors)
MULS	0xa6	(Element-wise) multiplication of secret registers (vectors)
MULRS	0xa7	Constant-vector multiplication of secret registers

continues on next page

Table 1 – continued from previous page

Name	Code	
DOTPRODS	0xa8	Dot product of secret registers (vectors)
TRUNC_PR	0xa9	Probabilistic truncation if supported by the protocol
MATMULS	0xaa	Secret matrix multiplication from registers
MATMULSM	0xab	Secret matrix multiplication reading directly from memory
CONV2DS	0xac	Secret 2D convolution
CHECK	0xaf	Force MAC check in current thread and all idle thread if current thread is the main thread
PRINTREG	0xb1	Debugging output of clear register (vector)
RAND	0xb2	Store insecure random value of specified length in clear integer register (vector)
PRINTREGPLAIN	0xb3	Output clear register
PRINTCHR	0xb4	Output a single byte
PRINTSTR	0xb5	Output four bytes
PUBINPUT	0xb6	Store public input in clear register (vector)
PRINTFLOATPLAIN	0xbc	Output floating-number from clear registers
WRITEFILESHARE	0xbd	Write shares to `Persistence/Transactions-P<playerno>.data` (appending at the end)
READFILESHARE	0xbe	Read shares from `Persistence/Transactions-P<playerno>.data`
CONDPRINTSTR	0xbf	Conditionally output four bytes
CONVINT	0xc0	Convert clear integer register (vector) to clear register (vector)
CONVMODP	0xc1	Convert clear integer register (vector) to clear register (vector)
LDMINT	0xca	Assign clear integer memory value(s) to clear integer register (vector) by immediate address
STMINT	0xcb	Assign clear integer register (vector) to clear integer memory value(s) by immediate address
LDMINTI	0xcc	Assign clear integer memory value(s) to clear integer register (vector) by register address
STMINTI	0xcd	Assign clear integer register (vector) to clear integer memory value(s) by register address
PUSHINT	0xce	Pushes clear integer register to the thread-local stack
POPINT	0xcf	Pops from the thread-local stack to clear integer register
MOVINT	0xd0	Copy clear integer register (vector)
INCINT	0xd1	Create incremental clear integer vector
SHUFFLE	0xd2	Randomly shuffles clear integer vector with public randomness
PRINTFLOATPREC	0xe0	Set number of digits after decimal point for *print_float_plain*
CONDPRINTPLAIN	0xe1	Conditionally output clear register (with precision)
NPLAYERS	0xe2	Store number of players in clear integer register
THRESHOLD	0xe3	Store maximal number of corrupt players in clear integer register
PLAYERID	0xe4	Store current player number in clear integer register
USE_EDABIT	0xe5	edaBit usage
INTOUTPUT	0xe6	Binary integer output
FLOATOUTPUT	0xe7	Binary floating-point output
INPUTMIXED	0xf2	Store private input in secret registers (vectors)
INPUTMIXEDREG	0xf3	Store private input in secret registers (vectors)
RAWINPUT	0xf4	Store private input in secret registers (vectors)
INPUTPERSONAL	0xf5	Private input from cint
GMULBITC	0x136	Clear GF(2^n) by clear GF(2) multiplication
GMULBITM	0x137	Modular reduction of clear register (vector) and (constant) immediate value

continues on next page

Table 1 – continued from previous page

Name	Code	
XORS	0x200	Bitwise XOR of secret bit register vectors
XORM	0x201	Bitwise XOR of single secret and clear bit registers
ANDRS	0x202	Constant-vector AND of secret bit registers
BITDECS	0x203	Secret bit register decomposition
BITCOMS	0x204	Secret bit register decomposition
CONVSINT	0x205	Copy clear integer register to secret bit register
LDBITS	0x20a	Store immediate in secret bit register
ANDS	0x20b	Bitwise AND of secret bit register vector
TRANS	0x20c	Secret bit register vector transpose
BITB	0x20d	Copy fresh secret random bit to secret bit register
ANDM	0x20e	Bitwise AND of single secret and clear bit registers
NOTS	0x20f	Bitwise NOT of secret register vector
XORCBI	0x210	Bitwise XOR of single clear bit register and immediate
BITDECC	0x211	Secret bit register decomposition
NOTCB	0x212	Bitwise NOT of secret register vector
CONVCINT	0x213	Copy clear integer register to clear bit register
REVEAL	0x214	Reveal secret bit register vectors and copy result to clear bit register vectors
LDMCB	0x217	Copy clear bit memory cell with compile-time address to clear bit register
STMCB	0x218	Copy clear bit register to clear bit memory cell with compile-time address
XORCB	0x219	Bitwise XOR of two single clear bit registers
ADDCB	0x21a	Integer addition two single clear bit registers
ADDCBI	0x21b	Integer addition single clear bit register and immediate
MULCBI	0x21c	Integer multiplication single clear bit register and immediate
SHRCBI	0x21d	Right shift of clear bit register by immediate
SHLCBI	0x21e	Left shift of clear bit register by immediate
CONVCINTVEC	0x21f	Copy clear register vector by bit to clear bit register vectors
PRINTREGSIGNED	0x220	Signed output of clear bit register
PRINTREGB	0x221	Debug output of clear bit register
PRINTREGPLAINB	0x222	Output clear bit register
PRINTFLOATPLAINB	0x223	Output floating-number from clear bit registers
CONDPRINTSTRB	0x224	Conditionally output four bytes
CONVCBIT	0x230	Copy clear bit register to clear integer register
CONVCBITVEC	0x231	Copy clear bit register vector to clear register by bit
LDMSB	0x240	Copy secret bit memory cell with compile-time address to secret bit register
STMSB	0x241	Copy secret bit register to secret bit memory cell with compile-time address
LDMSBI	0x242	Copy secret bit memory cell with run-time address to secret bit register
STMSBI	0x243	Copy secret bit register to secret bit memory cell with run-time address
MOVSB	0x244	Copy secret bit register
INPUTB	0x246	Copy private input to secret bit register vectors
INPUTBVEC	0x247	Copy private input to secret bit registers bit by bit
SPLIT	0x248	Local share conversion
CONVCBIT2S	0x249	Copy clear bit register vector to secret bit register vector

continues on next page

Table 1 – continued from previous page

Name	Code	
LDMCBI	0x258	Copy clear bit memory cell with run-time address to clear bit register
STMCBI	0x259	Copy clear bit register to clear bit memory cell with run-time address

2.2.2 Compiler.instructions module

This module contains all instruction types for arithmetic computation and general control of the virtual machine such as control flow.

The parameter descriptions refer to the instruction arguments in the right order.

class `Compiler.instructions.`**`acceptclientconnection`**`(`**args*, ***kwargs*`)`

Wait for a connection at the given port and write socket handle to clear integer register.

Param
client id destination (regint)

Param
port number (regint)

class `Compiler.instructions.`**`addc`**`(`**args*, ***kwargs*`)`

Clear addition.

Param
result (cint)

Param
summand (cint)

Param
summand (cint)

class `Compiler.instructions.`**`addci`**`(`**args*, ***kwargs*`)`

Addition of clear register (vector) and (constant) immediate value.

Param
result (cint)

Param
summand (cint)

Param
summand (int)

class `Compiler.instructions.`**`addint`**`(`**args*, ***kwargs*`)`

Clear integer register (vector) addition.

Param
result (regint)

Param
summand (regint)

Param
summand (regint)

op`(`*b*, `/)`

Same as a + b.

class `Compiler.instructions.`**`addm`**(**args*, ***kwargs*)

Mixed addition.

Param
result (sint)

Param
summand (sint)

Param
summand (cint)

class `Compiler.instructions.`**`adds`**(**args*, ***kwargs*)

Secret addition.

Param
result (sint)

Param
summand (sint)

Param
summand (sint)

class `Compiler.instructions.`**`addsi`**(**args*, ***kwargs*)

Addition of secret register (vector) and (constant) immediate value.

Param
result (cint)

Param
summand (cint)

Param
summand (int)

class `Compiler.instructions.`**`andc`**(**args*, ***kwargs*)

Logical AND of clear (vector) registers.

Param
result (cint)

Param
operand (cint)

Param
operand (cint)

class `Compiler.instructions.`**`andci`**(**args*, ***kwargs*)

Logical AND of clear register (vector) and (constant) immediate value.

Param
result (cint)

Param
operand (cint)

Param
operand (int)

class `Compiler.instructions.`**`asm_open`**(**args*, ***kwargs*)

Reveal secret registers (vectors) to clear registers (vectors).

Param
number of argument to follow (multiple of two)

Param
destination (cint)

Param
source (sint)

Param
(repeat the last two)…

class `Compiler.instructions.`**`bit`**(**args*, ***kwargs*)

Store fresh random triple(s) in secret register (vectors).

Param
destination (sint)

class `Compiler.instructions.`**`bitdecint`**(**args*, ***kwargs*)

Clear integer bit decomposition.

Param
number of arguments to follow / number of bits minus one (int)

Param
source (regint)

Param
destination for least significant bit (regint)

Param
(destination for one bit higher)…

class `Compiler.instructions.`**`check`**(**args*, ***kwargs*)

Force MAC check in current thread and all idle thread if current thread is the main thread.

class `Compiler.instructions.`**`cisc`**

Meta instruction for emulation. This instruction is only generated when using `-K` with `compile.py`. The header looks as follows:

Param
number of arguments after name plus one

Param
name (16 bytes, zero-padded)

Currently, the following names are supported:

LTZ
Less than zero.

param
number of arguments in this unit (must be 6)

param
vector size

param
result (sint)

param
input (sint)

param
bit length

param
(ignored)

param
(repeat)…

Trunc
Truncation.

param
number of arguments in this unit (must be 8)

param
vector size

param
result (sint)

param
input (sint)

param
bit length

param
number of bits to truncate

param
(ignored)

param
0 for unsigned or 1 for signed

param
(repeat)…

FPDiv
Fixed-point division. Division by zero results in zero without error.

param
number of arguments in this unit (must be at least 7)

param
vector size

param
result (sint)

param
dividend (sint)

param
divisor (sint)

param
(ignored)

param
fixed-point precision

param
(repeat)…

exp2_fx
Fixed-point power of two.

param
number of arguments in this unit (must be at least 6)

param
vector size

param
result (sint)

param
exponent (sint)

param
(ignored)

param
fixed-point precision

param
(repeat)…

log2_fx
Fixed-point logarithm with base 2.

param
number of arguments in this unit (must be at least 6)

param
vector size

param
result (sint)

param
input (sint)

param
(ignored)

param
fixed-point precision

param
(repeat)…

`class Compiler.instructions.closeclientconnection(*args, **kwargs)`

Close connection to client.

Param
client id (regint)

`class Compiler.instructions.cond_print_plain(*args, **kwargs)`

Conditionally output clear register (with precision). Outputs $x \cdot 2^p$ where p is the precision.

Param
condition (cint, no output if zero)

Param
source (cint)

Param
precision (cint)

class `Compiler.instructions.`**`cond_print_str`**(*cond*, *val*)

Conditionally output four bytes.

Param
condition (cint, no output if zero)

Param
four bytes (int)

class `Compiler.instructions.`**`conv2ds`**(**args*, ***kwargs*)

Secret 2D convolution.

Param
result (sint vector in row-first order)

Param
inputs (sint vector in row-first order)

Param
weights (sint vector in row-first order)

Param
output height (int)

Param
output width (int)

Param
input height (int)

Param
input width (int)

Param
weight height (int)

Param
weight width (int)

Param
stride height (int)

Param
stride width (int)

Param
number of channels (int)

Param
padding height (int)

Param
padding width (int)

Param
batch size (int)

class `Compiler.instructions.`**`convint`**`(`**args*, ***kwargs*`)`

Convert clear integer register (vector) to clear register (vector).

Param
destination (cint)

Param
source (regint)

class `Compiler.instructions.`**`convmodp`**`(`**args*, ***kwargs*`)`

Convert clear integer register (vector) to clear register (vector). If the bit length is zero, the unsigned conversion is used, otherwise signed conversion is used. This makes a difference when computing modulo a prime p. Signed conversion of $p-1$ results in -1 while signed conversion results in $(p-1) \mod 2^{64}$.

Param
destination (regint)

Param
source (cint)

Param
bit length (int)

class `Compiler.instructions.`**`crash`**`(`**args*, ***kwargs*`)`

Crash runtime if the value in the register is not zero.

Param
Crash condition (regint)

class `Compiler.instructions.`**`dabit`**`(`**args*, ***kwargs*`)`

Store fresh random daBit(s) in secret register (vectors).

Param
arithmetic part (sint)

Param
binary part (sbit)

class `Compiler.instructions.`**`digestc`**`(`**args*, ***kwargs*`)`

Clear truncated hash computation.

Param
result (cint)

Param
input (cint)

Param
byte length of hash value used (int)

class `Compiler.instructions.`**`divc`**`(`**args*, ***kwargs*`)`

Clear division.

Param
result (cint)

Param
dividend (cint)

Param
divisor (cint)

class Compiler.instructions.**divci**(**args*, ***kwargs*)

Division of secret register (vector) and (constant) immediate value.

Param
result (cint)

Param
dividend (cint)

Param
divisor (int)

class Compiler.instructions.**divint**(**args*, ***kwargs*)

Clear integer register (element-wise vector) division with floor rounding.

Param
result (regint)

Param
dividend (regint)

Param
divisor (regint)

op(*b*, /)

Same as a // b.

class Compiler.instructions.**dotprods**(**args*)

Dot product of secret registers (vectors). Note that the vectorized version works element-wise.

Param
number of arguments to follow (int)

Param
twice the dot product length plus two (I know…)

Param
result (sint)

Param
first factor (sint)

Param
first factor (sint)

Param
second factor (sint)

Param
second factor (sint)

Param
(remaining factors)…

Param
(repeat from dot product length)…

get_def()

Return the set of registers that are written to in this instruction.

get_used()

Return the set of registers that are read in this instruction.

class Compiler.instructions.**edabit**(**args*, ***kwargs*)

Store fresh random loose edaBit(s) in secret register (vectors). The length is the first argument minus one.

Param
number of arguments to follow / number of bits plus two (int)

Param
arithmetic (sint)

Param
binary (sbit)

Param
(binary)…

class Compiler.instructions.**eqc**(**args*, ***kwargs*)

Clear integer equality test. The result is 1 if the operands are equal and 0 otherwise.

Param
destination (regint)

Param
first operand (regint)

Param
second operand (regint)

op(*b*, /)

Same as a == b.

class Compiler.instructions.**eqzc**(**args*, ***kwargs*)

Clear integer zero test. The result is 1 for true and 0 for false.

Param
destination (regint)

Param
operand (regint)

class Compiler.instructions.**floatoutput**(**args*, ***kwargs*)

Binary floating-point output.

Param
player (int)

Param
significand (cint)

Param
exponent (cint)

Param
zero bit (cint)

Param
sign bit (cint)

class Compiler.instructions.**floordivc**(**args*, ***kwargs*)

Clear integer floor division.

Param
result (cint)

Param
dividend (cint)

Param
divisor (cint)

class Compiler.instructions.**gbitcom**(**args*, ***kwargs*)

Store the bits $cg_j, dots$ as every n-th bit of cg_i.

class Compiler.instructions.**gbitdec**(**args*, ***kwargs*)

Store every n-th bit of cg_i in $cg_j, dots$.

class Compiler.instructions.**gbitgf2ntriple**(**args*, ***kwargs*)

Load secret variables s_i, s_j and s_k with the next GF(2) and GF(2^n) multiplication triple.

class Compiler.instructions.**gbittriple**(**args*, ***kwargs*)

Load secret variables s_i, s_j and s_k with the next GF(2) multiplication triple.

class Compiler.instructions.**gconvgf2n**(**args*, ***kwargs*)

Convert from clear modp register c_j to integer register ci_i.

class Compiler.instructions.**gldmci**(**args*, ***kwargs*)

Assigns register c_i the value in memory verb+C[cj]+.

direct

alias of `GF2N_Instruction`

class Compiler.instructions.**gldmsi**(**args*, ***kwargs*)

Assigns register s_i the value in memory verb+S[cj]+.

direct

alias of `GF2N_Instruction`

class Compiler.instructions.**gmulbitc**(**args*, ***kwargs*)

Clear GF(2^n) by clear GF(2) multiplication

class Compiler.instructions.**gmulbitm**(**args*, ***kwargs*)

Secret GF(2^n) by clear GF(2) multiplication

class Compiler.instructions.**gnotc**(**args*, ***kwargs*)

Clear logical NOT $cg_i = lnot cg_j$

class Compiler.instructions.**gstmci**(**args*, ***kwargs*)

Sets verb+C[cj]+ to be the value c_i.

direct

alias of `GF2N_Instruction`

class Compiler.instructions.**gstmsi**(**args*, ***kwargs*)

Sets verb+S[cj]+ to be the value s_i.

direct

alias of `GF2N_Instruction`

class Compiler.instructions.**gtc**(**args*, ***kwargs*)

Clear integer greater-than comparison. The result is 1 if the first operand is greater and 0 otherwise.

Param
destination (regint)

Param
first operand (regint)

Param
second operand (regint)

op(*b*, /)

Same as a > b.

class Compiler.instructions.**incint**(**args*, ***kwargs*)

Create incremental clear integer vector. For example, vector size 10, base 1, increment 2, repeat 3, and wrap 2 produces the following:

```
(1, 1, 1, 3, 3, 3, 1, 1, 1, 3)
```

This is because the first number is always the `base`, every number is repeated `repeat` times, after which `increment` is added, and after `wrap` increments the number returns to `base`.

Param
destination (regint)

Param
base (non-vector regint)

Param
increment (int)

Param
repeat (int)

Param
wrap (int)

class Compiler.instructions.**inputfix**(**args*, ***kwargs*)

class Compiler.instructions.**inputfloat**(**args*, ***kwargs*)

class Compiler.instructions.**inputmaskreg**(**args*, ***kwargs*)

Store fresh random input mask(s) in secret register (vector) and clear register (vector) of the relevant player.

Param
mask (sint)

Param
mask (cint, player only)

Param
player (regint)

class Compiler.instructions.**inputmixed**(*name*, **args*)

Store private input in secret registers (vectors). The input is read as integer or floating-point number and the latter is then converted to the internal representation using the given precision. This instruction uses compile-time player numbers.

Param
number of arguments to follow (int)

Param
type (0: integer, 1: fixed-point, 2: floating-point)

Param
destination (sint)

Param
destination (sint, only for floating-point)

Param
destination (sint, only for floating-point)

Param
destination (sint, only for floating-point)

Param
fixed-point precision or precision of floating-point significand (int, not with integer)

Param
input player (int)

Param
(repeat from type parameter)...

`class Compiler.instructions.inputmixedreg(name, *args)`

Store private input in secret registers (vectors). The input is read as integer or floating-point number and the latter is then converted to the internal representation using the given precision. This instruction uses run-time player numbers.

Param
number of arguments to follow (int)

Param
type (0: integer, 1: fixed-point, 2: floating-point)

Param
destination (sint)

Param
destination (sint, only for floating-point)

Param
destination (sint, only for floating-point)

Param
destination (sint, only for floating-point)

Param
fixed-point precision or precision of floating-point significand (int, not with integer)

Param
input player (regint)

Param
(repeat from type parameter)...

`class Compiler.instructions.inputpersonal(*args)`

Private input from cint.

Param
vector size (int)

Param
player (int)

Param
destination (sint)

Param
source (cint)

Param
(repeat from vector size)...

class `Compiler.instructions.`**`intoutput`**`(`**args*, ***kwargs*`)`

Binary integer output.

Param
player (int)

Param
regint

class `Compiler.instructions.`**`inv2m`**`(`**args*, ***kwargs*`)`

Inverse of power of two modulo prime (the computation modulus).

Param
result (cint)

Param
exponent (int)

class `Compiler.instructions.`**`inverse`**`(`**args*, ***kwargs*`)`

Store fresh random inverse(s) in secret register (vectors).

Param
value (sint)

Param
inverse (sint)

class `Compiler.instructions.`**`jmp`**`(`**args*, ***kwargs*`)`

Unconditional relative jump in the bytecode (compile-time parameter). The parameter is added to the regular jump of one after every instruction. This means that a jump of 0 results in a no-op while a jump of -1 results in an infinite loop.

Param
number of instructions (int)

class `Compiler.instructions.`**`jmpeqz`**`(`**args*, ***kwargs*`)`

Conditional relative jump in the bytecode. The parameter is added to the regular jump of one after every instruction. This means that a jump of 0 results in a no-op while a jump of -1 results in an infinite loop.

Param
condition (regint, only jump if zero)

Param
number of instructions (int)

class `Compiler.instructions.`**`jmpi`**`(`**args*, ***kwargs*`)`

Unconditional relative jump in the bytecode (run-time parameter). The parameter is added to the regular jump of one after every instruction. This means that a jump of 0 results in a no-op while a jump of -1 results in an infinite loop.

Param
number of instructions (regint)

class Compiler.instructions.**jmpnz**(**args*, ***kwargs*)

Conditional relative jump in the bytecode. The parameter is added to the regular jump of one after every instruction. This means that a jump of 0 results in a no-op while a jump of -1 results in an infinite loop.

Param
condition (regint, only jump if not zero)

Param
number of instructions (int)

class Compiler.instructions.**join_tape**(**args*, ***kwargs*)

Join thread.

Param
virtual machine thread number (int)

class Compiler.instructions.**ldarg**(**args*, ***kwargs*)

Store the argument passed to the current thread in clear integer register.

Param
destination (regint)

class Compiler.instructions.**ldi**(**args*, ***kwargs*)

Assign (constant) immediate value to clear register (vector).

Param
destination (cint)

Param
value (int)

class Compiler.instructions.**ldint**(**args*, ***kwargs*)

Store (constant) immediate value in clear integer register (vector).

Param
destination (regint)

Param
immediate (int)

class Compiler.instructions.**ldmc**(**args*, ***kwargs*)

Assign clear memory value(s) to clear register (vector) by immediate address. The vectorized version starts at the base address and then iterates the memory address.

Param
destination (cint)

Param
memory address base (int)

class Compiler.instructions.**ldmci**(**args*, ***kwargs*)

Assign clear memory value(s) to clear register (vector) by register address. The vectorized version starts at the base address and then iterates the memory address.

Param
destination (cint)

Param
memory address base (regint)

direct

alias of *ldmc*

class Compiler.instructions.**ldmint**(**args*, ***kwargs*)

Assign clear integer memory value(s) to clear integer register (vector) by immediate address. The vectorized version starts at the base address and then iterates the memory address.

Param
destination (regint)

Param
memory address base (int)

class Compiler.instructions.**ldminti**(**args*, ***kwargs*)

Assign clear integer memory value(s) to clear integer register (vector) by register address. The vectorized version starts at the base address and then iterates the memory address.

Param
destination (regint)

Param
memory address base (regint)

direct

alias of *ldmint*

class Compiler.instructions.**ldms**(**args*, ***kwargs*)

Assign secret memory value(s) to secret register (vector) by immediate address. The vectorized version starts at the base address and then iterates the memory address.

Param
destination (sint)

Param
memory address base (int)

class Compiler.instructions.**ldmsi**(**args*, ***kwargs*)

Assign secret memory value(s) to secret register (vector) by register address. The vectorized version starts at the base address and then iterates the memory address.

Param
destination (sint)

Param
memory address base (regint)

direct

alias of *ldms*

class Compiler.instructions.**ldsi**(**args*, ***kwargs*)

Assign (constant) immediate value to secret register (vector).

Param
destination (sint)

Param
value (int)

class Compiler.instructions.**ldtn**(**args*, ***kwargs*)

Store the number of the current thread in clear integer register.

Param
destination (regint)

class Compiler.instructions.**legendrec**(**args*, ***kwargs*)

Clear Legendre symbol computation (a/p) over prime p (the computation modulus).

Param
result (cint)

Param
a (int)

class Compiler.instructions.**listen**(**args*, ***kwargs*)

Open a server socket on a party-specific port number and listen for client connections (non-blocking).

Param
port number (regint)

class Compiler.instructions.**ltc**(**args*, ***kwargs*)

Clear integer less-than comparison. The result is 1 if the first operand is less and 0 otherwise.

Param
destination (regint)

Param
first operand (regint)

Param
second operand (regint)

op(*b*, /)

Same as a < b.

class Compiler.instructions.**ltzc**(**args*, ***kwargs*)

Clear integer less than zero test. The result is 1 for true and 0 for false.

Param
destination (regint)

Param
operand (regint)

class Compiler.instructions.**matmuls**(**args*, ***kwargs*)

Secret matrix multiplication from registers. All matrices are represented as vectors in row-first order.

Param
result (sint vector)

Param
first factor (sint vector)

Param
second factor (sint vector)

Param
number of rows in first factor and result (int)

Param
number of columns in first factor and rows in second factor (int)

Param
number of columns in second factor and result (int)

class Compiler.instructions.**matmulsm**(**args*, ***kwargs*)

Secret matrix multiplication reading directly from memory.

Param
result (sint vector in row-first order)

Param
base address of first factor (regint value)

Param
base address of second factor (regint value)

Param
number of rows in first factor and result (int)

Param
number of columns in first factor and rows in second factor (int)

Param
number of columns in second factor and result (int)

Param
rows of first factor to use (regint vector, length as number of rows in first factor)

Param
columns of first factor to use (regint vector, length below)

Param
rows of second factor to use (regint vector, length below)

Param
columns of second factor to use (regint vector, length below)

Param
number of columns of first / rows of second factor to use (int)

Param
number of columns of second factor to use (int)

class Compiler.instructions.**modc**(**args*, ***kwargs*)

Clear modular reduction.

Param
result (cint)

Param
dividend (cint)

Param
divisor (cint)

class Compiler.instructions.**modci**(**args*, ***kwargs*)

Modular reduction of clear register (vector) and (constant) immediate value.

Param
result (cint)

Param
dividend (cint)

Param
divisor (int)

class `Compiler.instructions.`**`movc`**`(`**args*, ***kwargs*`)`

Copy clear register (vector).

Param
destination (cint)

Param
source (cint)

class `Compiler.instructions.`**`movint`**`(`**args*, ***kwargs*`)`

Copy clear integer register (vector).

Param
destination (regint)

Param
source (regint)

class `Compiler.instructions.`**`movs`**`(`**args*, ***kwargs*`)`

Copy secret register (vector).

Param
destination (cint)

Param
source (cint)

class `Compiler.instructions.`**`mulc`**`(`**args*, ***kwargs*`)`

Clear multiplication.

Param
result (cint)

Param
factor (cint)

Param
factor (cint)

class `Compiler.instructions.`**`mulci`**`(`**args*, ***kwargs*`)`

Multiplication of clear register (vector) and (constant) immediate value.

Param
result (cint)

Param
factor (cint)

Param
factor (int)

class `Compiler.instructions.`**`mulint`**`(`**args*, ***kwargs*`)`

Clear integer register (element-wise vector) multiplication.

Param
result (regint)

Param
factor (regint)

Param
factor (regint)

op(*b*, /)

Same as a * b.

class Compiler.instructions.**mulm**(**args*, ***kwargs*)

Multiply secret and clear value.

Param
result (sint)

Param
factor (sint)

Param
factor (cint)

class Compiler.instructions.**mulrs**(*res*, *x*, *y*)

Constant-vector multiplication of secret registers.

Param
number of arguments to follow (multiple of four)

Param
vector size (int)

Param
result (sint)

Param
vector factor (sint)

Param
constant factor (sint)

Param
(repeat the last four)…

get_def()

Return the set of registers that are written to in this instruction.

get_used()

Return the set of registers that are read in this instruction.

class Compiler.instructions.**muls**(**args*, ***kwargs*)

(Element-wise) multiplication of secret registers (vectors).

Param
number of arguments to follow (multiple of three)

Param
result (sint)

Param
factor (sint)

Param
factor (sint)

Param
(repeat the last three)…

class Compiler.instructions.**mulsi**(**args*, ***kwargs*)

Multiplication of secret register (vector) and (constant) immediate value.

Param
result (sint)

Param
factor (sint)

Param
factor (int)

class Compiler.instructions.**notc**(**args*, ***kwargs*)

Clear logical NOT of a constant number of bits of clear (vector) register.

Param
result (cint)

Param
operand (cint)

Param
bit length (int)

class Compiler.instructions.**nplayers**(**args*, ***kwargs*)

Store number of players in clear integer register.

Param
destination (regint)

class Compiler.instructions.**orc**(**args*, ***kwargs*)

Logical OR of clear (vector) registers.

Param
result (cint)

Param
operand (cint)

Param
operand (cint)

class Compiler.instructions.**orci**(**args*, ***kwargs*)

Logical OR of clear register (vector) and (constant) immediate value.

Param
result (cint)

Param
operand (cint)

Param
operand (int)

class Compiler.instructions.**playerid**(**args*, ***kwargs*)

Store current player number in clear integer register.

Param
destination (regint)

class Compiler.instructions.**popint**(**args*, ***kwargs*)

Pops from the thread-local stack to clear integer register.

Param
destination (regint)

class Compiler.instructions.**prep**(**args*, ***kwargs*)

Store custom preprocessed data in secret register (vectors).

Param
number of arguments to follow (int)

Param
tag (16 bytes / 4 units, cut off at first zero byte)

Param
destination (sint)

Param
(repeat destination)...

class Compiler.instructions.**print_char**(*ch*)

Output a single byte.

Param
byte (int)

class Compiler.instructions.**print_char4**(*val*)

Output four bytes.

Param
four bytes (int)

class Compiler.instructions.**print_float_plain**(**args*, ***kwargs*)

Output floating-number from clear registers.

Param
significand (cint)

Param
exponent (cint)

Param
zero bit (cint, zero output if bit is one)

Param
sign bit (cint, negative output if bit is one)

Param
NaN (cint, regular number if zero)

class Compiler.instructions.**print_float_prec**(**args*, ***kwargs*)

Set number of digits after decimal point for `print_float_plain`.

Param
number of digits (int)

class Compiler.instructions.**print_int**(**args*, ***kwargs*)

Output clear integer register.

Param
source (regint)

class Compiler.instructions.**print_reg**(*reg*, *comment=''*)

Debugging output of clear register (vector).

Param
source (cint)

Param
comment (4 bytes / 1 unit)

class Compiler.instructions.**print_reg_plain**(**args*, ***kwargs*)

Output clear register.

Param
source (cint)

class Compiler.instructions.**pubinput**(**args*, ***kwargs*)

Store public input in clear register (vector).

Param
destination (cint)

class Compiler.instructions.**pushint**(**args*, ***kwargs*)

Pushes clear integer register to the thread-local stack.

Param
source (regint)

class Compiler.instructions.**rand**(**args*, ***kwargs*)

Store insecure random value of specified length in clear integer register (vector).

Param
destination (regint)

Param
length (regint)

class Compiler.instructions.**randomfulls**(**args*, ***kwargs*)

Store share(s) of a fresh secret random element in secret register (vectors).

Param
destination (sint)

class Compiler.instructions.**randoms**(**args*, ***kwargs*)

Store fresh length-restricted random shares(s) in secret register (vectors). This is only implemented for protocols that also implement local share conversion with `split`.

Param
destination (sint)

Param
length (int)

class Compiler.instructions.**rawinput**(**args*, ***kwargs*)

Store private input in secret registers (vectors). The input is read in the internal binary format according to the protocol.

Param
number of arguments to follow (multiple of two)

Param
player number (int)

Param
destination (sint)

class Compiler.instructions.**readsharesfromfile**(**args*, ***kwargs*)

Read shares from Persistence/Transactions-P<playerno>.data.

Param
number of arguments to follow / number of shares plus two (int)

Param
starting position in number of shares from beginning (regint)

Param
destination for final position, -1 for eof reached, or -2 for file not found (regint)

Param
destination for share (sint)

Param
(repeat from destination for share)...

class Compiler.instructions.**readsocketc**(**args*, ***kwargs*)

Read a variable number of clear values in internal representation from socket for a specified client id and store them in clear registers.

Param
number of arguments to follow / number of inputs minus one (int)

Param
client id (regint)

Param
vector size (int)

Param
destination (cint)

Param
(repeat destination)...

class Compiler.instructions.**readsocketint**(**args*, ***kwargs*)

Read a variable number of 32-bit integers from socket for a specified client id and store them in clear integer registers.

Param
number of arguments to follow / number of inputs minus one (int)

Param
client id (regint)

Param
vector size (int)

Param
destination (regint)

Param
(repeat destination)...

class Compiler.instructions.**readsockets**(**args*, ***kwargs*)

Read a variable number of secret shares (potentially with MAC) from a socket for a client id and store them in registers. If the protocol uses MACs, the client should be different for every party.

Param
client id (regint)

Param
vector size (int)

Param
source (sint)

Param
(repeat source)…

class `Compiler.instructions.`**`reqbl`**`(`**args*, ***kwargs*`)`

Requirement on computation modulus. Minimal bit length of prime if positive, minus exact bit length of power of two if negative.

Param
requirement (int)

class `Compiler.instructions.`**`run_tape`**`(`**args*, ***kwargs*`)`

Start tape/bytecode file in another thread.

Param
number of arguments to follow (multiple of three)

Param
virtual machine thread number (int)

Param
tape number (int)

Param
tape argument (int)

Param
(repeat the last three)…

class `Compiler.instructions.`**`sedabit`**`(`**args*, ***kwargs*`)`

Store fresh random strict edaBit(s) in secret register (vectors). The length is the first argument minus one.

Param
number of arguments to follow / number of bits plus two (int)

Param
arithmetic (sint)

Param
binary (sbit)

Param
(binary)…

class `Compiler.instructions.`**`shlc`**`(`**args*, ***kwargs*`)`

Bitwise left shift of clear register (vector).

Param
result (cint)

Param
first operand (cint)

Param
second operand (cint)

class Compiler.instructions.**shlci**(**args*, ***kwargs*)

Bitwise left shift of clear register (vector) by (constant) immediate value.

Param
result (cint)

Param
first operand (cint)

Param
second operand (int)

class Compiler.instructions.**shrc**(**args*, ***kwargs*)

Bitwise right shift of clear register (vector).

Param
result (cint)

Param
first operand (cint)

Param
second operand (cint)

class Compiler.instructions.**shrci**(**args*, ***kwargs*)

Bitwise right shift of clear register (vector) by (constant) immediate value.

Param
result (cint)

Param
first operand (cint)

Param
second operand (int)

class Compiler.instructions.**shrsi**(**args*, ***kwargs*)

Bitwise right shift of secret register (vector) by (constant) immediate value. This only makes sense in connection with protocols allowing local share conversion (i.e., based on additive secret sharing modulo a power of two). Moreover, the result is not a secret sharing of the right shift of the secret value but needs to be corrected using the overflow. This is explained by Dalskov et al. in the appendix.

Param
result (sint)

Param
first operand (sint)

Param
second operand (int)

class Compiler.instructions.**shuffle**(**args*, ***kwargs*)

Randomly shuffles clear integer vector with public randomness.

Param
destination (regint)

Param
source (regint)

class Compiler.instructions.**square**(**args*, ***kwargs*)

Store fresh random square(s) in secret register (vectors).

Param
value (sint)

Param
square (sint)

class Compiler.instructions.**starg**(**args*, ***kwargs*)

Copy clear integer register to the thread argument.

Param
source (regint)

class Compiler.instructions.**start**(**args*, ***kwargs*)

Start timer.

Param
timer number (int)

class Compiler.instructions.**stmc**(**args*, ***kwargs*)

Assign clear register (vector) to clear memory value(s) by immediate address. The vectorized version starts at the base address and then iterates the memory address.

Param
source (cint)

Param
memory address base (int)

class Compiler.instructions.**stmci**(**args*, ***kwargs*)

Assign clear register (vector) to clear memory value(s) by register address. The vectorized version starts at the base address and then iterates the memory address.

Param
source (cint)

Param
memory address base (regint)

direct

alias of *stmc*

class Compiler.instructions.**stmint**(**args*, ***kwargs*)

Assign clear integer register (vector) to clear integer memory value(s) by immediate address. The vectorized version starts at the base address and then iterates the memory address.

Param
source (regint)

Param
memory address base (int)

class Compiler.instructions.**stminti**(**args*, ***kwargs*)

Assign clear integer register (vector) to clear integer memory value(s) by register address. The vectorized version starts at the base address and then iterates the memory address.

Param
source (regint)

Param
memory address base (regint)

direct

alias of *stmint*

class Compiler.instructions.**stms**(**args*, ***kwargs*)

Assign secret register (vector) to secret memory value(s) by immediate address. The vectorized version starts at the base address and then iterates the memory address.

Param
source (sint)

Param
memory address base (int)

class Compiler.instructions.**stmsi**(**args*, ***kwargs*)

Assign secret register (vector) to secret memory value(s) by register address. The vectorized version starts at the base address and then iterates the memory address.

Param
source (sint)

Param
memory address base (regint)

direct

alias of *stms*

class Compiler.instructions.**stop**(**args*, ***kwargs*)

Stop timer.

Param
timer number (int)

class Compiler.instructions.**subc**(**args*, ***kwargs*)

Clear subtraction.

Param
result (cint)

Param
first operand (cint)

Param
second operand (cint)

class Compiler.instructions.**subcfi**(**args*, ***kwargs*)

Subtraction of clear register (vector) from (constant) immediate value.

Param
result (cint)

Param
first operand (int)

Param
second operand (cint)

class Compiler.instructions.**subci**(**args*, ***kwargs*)

Subtraction of (constant) immediate value from clear register (vector).

Param
result (cint)

Param
first operand (cint)

Param
second operand (int)

class Compiler.instructions.**subint**(**args*, ***kwargs*)

Clear integer register (vector) subtraction.

Param
result (regint)

Param
first operand (regint)

Param
second operand (regint)

op(*b*, /)

Same as a - b.

class Compiler.instructions.**subml**(**args*, ***kwargs*)

Subtract clear from secret value.

Param
result (sint)

Param
first operand (sint)

Param
second operand (cint)

class Compiler.instructions.**submr**(**args*, ***kwargs*)

Subtract secret from clear value.

Param
result (sint)

Param
first operand (cint)

Param
second operand (sint)

class Compiler.instructions.**subs**(**args*, ***kwargs*)

Secret subtraction.

Param
result (sint)

Param
first operand (sint)

Param
second operand (sint)

class Compiler.instructions.**subsfi**(**args*, ***kwargs*)

Subtraction of secret register (vector) from (constant) immediate value.

Param
result (sint)

Param
first operand (int)

Param
second operand (sint)

class Compiler.instructions.**subsi**(**args*, ***kwargs*)

Subtraction of (constant) immediate value from secret register (vector).

Param
result (sint)

Param
first operand (sint)

Param
second operand (int)

class Compiler.instructions.**threshold**(**args*, ***kwargs*)

Store maximal number of corrupt players in clear integer register.

Param
destination (regint)

class Compiler.instructions.**time**(**args*, ***kwargs*)

Output time since start of computation.

class Compiler.instructions.**triple**(**args*, ***kwargs*)

Store fresh random triple(s) in secret register (vectors).

Param
factor (sint)

Param
factor (sint)

Param
product (sint)

class Compiler.instructions.**trunc_pr**(**args*, ***kwargs*)

Probabilistic truncation if supported by the protocol.

Param
number of arguments to follow (multiple of four)

Param
destination (sint)

Param
source (sint)

Param
bit length of source (int)

Param
number of bits to truncate (int)

class `Compiler.instructions.`**`use`**(**args*, ***kwargs*)

Offline data usage. Necessary to avoid reusage while using preprocessing from files. Also used to multithreading for expensive preprocessing.

Param
domain (0: integer, 1: $\mathrm{GF}(2^n)$, 2: bit)

Param
type (0: triple, 1: square, 2: bit, 3: inverse, 6: daBit)

Param
number (int, -1 for unknown)

class `Compiler.instructions.`**`use_edabit`**(**args*, ***kwargs*)

edaBit usage. Necessary to avoid reusage while using preprocessing from files. Also used to multithreading for expensive preprocessing.

Param
loose/strict (0/1)

Param
length (int)

Param
number (int, -1 for unknown)

class `Compiler.instructions.`**`use_inp`**(**args*, ***kwargs*)

Input usage. Necessary to avoid reusage while using preprocessing from files.

Param
domain (0: integer, 1: $\mathrm{GF}(2^n)$, 2: bit)

Param
input player (int)

Param
number (int, -1 for unknown)

class `Compiler.instructions.`**`use_matmul`**(**args*, ***kwargs*)

Matrix multiplication usage. Used for multithreading of preprocessing.

Param
number of left-hand rows (int)

Param
number of left-hand columns/right-hand rows (int)

Param
number of right-hand columns (int)

Param
number (int, -1 for unknown)

class `Compiler.instructions.`**`use_prep`**(**args*, ***kwargs*)

Custom preprocessed data usage.

Param
tag (16 bytes / 4 units, cut off at first zero byte)

Param
number of items to use (int, -1 for unknown)

class Compiler.instructions.**writesharestofile**(**args*, ***kwargs*)

Write shares to Persistence/Transactions-P<playerno>.data (appending at the end).

Param
number of shares (int)

Param
source (sint)

Param
(repeat from source)…

class Compiler.instructions.**writesockets**(**args*, ***kwargs*)

Write a variable number of secret shares (potentially with MAC) from registers into a socket for a specified client id. If the protocol uses MACs, the client should be different for every party.

Param
client id (regint)

Param
message type (must be 0)

Param
vector size (int)

Param
source (sint)

Param
(repeat source)…

class Compiler.instructions.**writesocketshare**(**args*, ***kwargs*)

Write a variable number of shares (without MACs) from secret registers into socket for a specified client id.

Param
client id (regint)

Param
message type (must be 0)

Param
vector size (int)

Param
source (sint)

Param
(repeat source)…

class Compiler.instructions.**xorc**(**args*, ***kwargs*)

Logical XOR of clear (vector) registers.

Param
result (cint)

Param
operand (cint)

Param
operand (cint)

class `Compiler.instructions.`**`xorci`**(**args*, ***kwargs*)

Logical XOR of clear register (vector) and (constant) immediate value.

Param
result (cint)

Param
operand (cint)

Param
operand (int)

2.2.3 Compiler.GC.instructions module

This module constrains instructions for binary circuits. Unlike arithmetic instructions, they generally do not use the vector size in the instruction code field. Instead the number of bits affected is given as an extra argument. Also note that a register holds 64 values instead of just one as is the case for arithmetic instructions. Therefore, an instruction for 65-128 bits will affect two registers etc. Similarly, a memory cell holds 64 bits.

class `Compiler.GC.instructions.`**`addcb`**(**args*, ***kwargs*)

Integer addition two single clear bit registers.

Param
result (cbit)

Param
summand (cbit)

Param
summand (cbit)

class `Compiler.GC.instructions.`**`addcbi`**(**args*, ***kwargs*)

Integer addition single clear bit register and immediate.

Param
result (cbit)

Param
summand (cbit)

Param
summand (int)

class `Compiler.GC.instructions.`**`andm`**(**args*, ***kwargs*)

Bitwise AND of single secret and clear bit registers.

Param
number of bits (less or equal 64)

Param
result (sbit)

Param
operand (sbit)

Param
operand (cbit)

class Compiler.GC.instructions.**andrs**(**args*, ***kwargs*)

Constant-vector AND of secret bit registers.

Param
number of arguments to follow (multiple of four)

Param
number of bits (int)

Param
result vector (sbit)

Param
vector operand (sbit)

Param
single operand (sbit)

Param
(repeat from number of bits)…

class Compiler.GC.instructions.**ands**(**args*, ***kwargs*)

Bitwise AND of secret bit register vector.

Param
number of arguments to follow (multiple of four)

Param
number of bits (int)

Param
result (sbit)

Param
operand (sbit)

Param
operand (sbit)

Param
(repeat from number of bits)…

class Compiler.GC.instructions.**bitb**(**args*, ***kwargs*)

Copy fresh secret random bit to secret bit register.

Param
destination (sbit)

class Compiler.GC.instructions.**bitcoms**(**args*, ***kwargs*)

Secret bit register decomposition.

Param
number of arguments to follow / number of bits plus one (int)

Param
destination (sbit)

Param
source for least significant bit (sbit)

Param
(source for one bit higher)…

class Compiler.GC.instructions.**bitdecc**(**args*, ***kwargs*)

Secret bit register decomposition.

Param
number of arguments to follow / number of bits plus one (int)

Param
source (sbit)

Param
destination for least significant bit (sbit)

Param
(destination for one bit higher)…

class Compiler.GC.instructions.**bitdecs**(**args*, ***kwargs*)

Secret bit register decomposition.

Param
number of arguments to follow / number of bits plus one (int)

Param
source (sbit)

Param
destination for least significant bit (sbit)

Param
(destination for one bit higher)…

class Compiler.GC.instructions.**cond_print_strb**(*cond*, *val*)

Conditionally output four bytes.

Param
condition (cbit, no output if zero)

Param
four bytes (int)

class Compiler.GC.instructions.**convcbit**(**args*, ***kwargs*)

Copy clear bit register to clear integer register.

Param
destination (regint)

Param
source (cbit)

class Compiler.GC.instructions.**convcbit2s**(**args*, ***kwargs*)

Copy clear bit register vector to secret bit register vector.

Param
number of bits (int)

Param
destination (sbit)

Param
source (cbit)

class Compiler.GC.instructions.**convcbitvec**(**args*)

Copy clear bit register vector to clear register by bit. This means that every element of the destination register vector will hold one bit.

Param
number of bits / vector length (int)

Param
destination (regint)

Param
source (cbit)

class Compiler.GC.instructions.**convcint**(**args*, ***kwargs*)

Copy clear integer register to clear bit register.

Param
number of bits (int)

Param
destination (cbit)

Param
source (regint)

class Compiler.GC.instructions.**convcintvec**(**args*, ***kwargs*)

Copy clear register vector by bit to clear bit register vectors. This means that the first destination will hold the least significant bits of all inputs etc.

Param
number of arguments to follow / number of bits plus one (int)

Param
source (cint)

Param
destination for least significant bits (sbit)

Param
(destination for bits one step higher)…

class Compiler.GC.instructions.**convsint**(**args*, ***kwargs*)

Copy clear integer register to secret bit register.

Param
number of bits (int)

Param
destination (sbit)

Param
source (regint)

class Compiler.GC.instructions.**inputb**(**args*, ***kwargs*)

Copy private input to secret bit register vectors. The input is read as floating-point number, multiplied by a power of two, and then rounded to an integer.

Param
number of arguments to follow (multiple of four)

Param
player number (int)

Param
number of bits in output (int)

Param
exponent to power of two factor (int)

Param
destination (sbit)

class `Compiler.GC.instructions.`**`inputbvec`**`(`**args*, ***kwargs*`)`

Copy private input to secret bit registers bit by bit. The input is read as floating-point number, multiplied by a power of two, rounded to an integer, and then decomposed into bits.

Param
total number of arguments to follow (int)

Param
number of arguments to follow for one input / number of bits plus three (int)

Param
exponent to power of two factor (int)

Param
player number (int)

Param
destination for least significant bit (sbit)

Param
(destination for one bit higher)…

Param
(repeat from number of arguments to follow for one input)…

class `Compiler.GC.instructions.`**`ldbits`**`(`**args*, ***kwargs*`)`

Store immediate in secret bit register.

Param
destination (sbit)

Param
number of bits (int)

Param
immediate (int)

class `Compiler.GC.instructions.`**`ldmcb`**`(`**args*, ***kwargs*`)`

Copy clear bit memory cell with compile-time address to clear bit register.

Param
destination (cbit)

Param
memory address (int)

class `Compiler.GC.instructions.`**`ldmcbi`**`(`**args*, ***kwargs*`)`

Copy clear bit memory cell with run-time address to clear bit register.

Param
destination (cbit)

Param
memory address (regint)

class Compiler.GC.instructions.**ldmsb**(**args*, ***kwargs*)

Copy secret bit memory cell with compile-time address to secret bit register.

Param
destination (sbit)

Param
memory address (int)

class Compiler.GC.instructions.**ldmsbi**(**args*, ***kwargs*)

Copy secret bit memory cell with run-time address to secret bit register.

Param
destination (sbit)

Param
memory address (regint)

class Compiler.GC.instructions.**movsb**(**args*, ***kwargs*)

Copy secret bit register.

Param
destination (sbit)

Param
source (sbit)

class Compiler.GC.instructions.**mulcbi**(**args*, ***kwargs*)

Integer multiplication single clear bit register and immediate.

Param
result (cbit)

Param
factor (cbit)

Param
factor (int)

class Compiler.GC.instructions.**notcb**(**args*, ***kwargs*)

Bitwise NOT of secret register vector.

Param
number of bits

Param
result (cbit)

Param
operand (cbit)

class Compiler.GC.instructions.**nots**(**args*, ***kwargs*)

Bitwise NOT of secret register vector.

Param
number of bits (less or equal 64)

Param
result (sbit)

Param
operand (sbit)

class Compiler.GC.instructions.**print_float_plainb**(**args*, ***kwargs*)

Output floating-number from clear bit registers.

Param
significand (cbit)

Param
exponent (cbit)

Param
zero bit (cbit, zero output if bit is one)

Param
sign bit (cbit, negative output if bit is one)

Param
NaN (cbit, regular number if zero)

class Compiler.GC.instructions.**print_reg_plainb**(**args*, ***kwargs*)

Output clear bit register.

Param
source (cbit)

class Compiler.GC.instructions.**print_reg_signed**(**args*, ***kwargs*)

Signed output of clear bit register.

Param
bit length (int)

Param
source (cbit)

class Compiler.GC.instructions.**print_regb**(*reg*, *comment=''*)

Debug output of clear bit register.

Param
source (cbit)

Param
comment (4 bytes / 1 unit)

class Compiler.GC.instructions.**reveal**(**args*, ***kwargs*)

Reveal secret bit register vectors and copy result to clear bit register vectors.

Param
number of arguments to follow (multiple of three)

Param
number of bits (int)

Param
destination (cbit)

Param
source (sbit)

Param
(repeat from number of bits)…

class Compiler.GC.instructions.**shlcbi**(**args*, ***kwargs*)

Left shift of clear bit register by immediate.

Param
destination (cbit)

Param
source (cbit)

Param
number of bits to shift (int)

class Compiler.GC.instructions.**shrcbi**(**args*, ***kwargs*)

Right shift of clear bit register by immediate.

Param
destination (cbit)

Param
source (cbit)

Param
number of bits to shift (int)

class Compiler.GC.instructions.**split**(**args*, ***kwargs*)

Local share conversion. This instruction use the vector length in the instruction code field.

Param
number of arguments to follow (number of bits times number of additive shares plus one)

Param
source (sint)

Param
first share of least significant bit

Param
second share of least significant bit

Param
(remaining share of least significant bit)…

Param
(repeat from first share for bit one step higher)…

class Compiler.GC.instructions.**stmcb**(**args*, ***kwargs*)

Copy clear bit register to clear bit memory cell with compile-time address.

Param
source (cbit)

Param
memory address (int)

class Compiler.GC.instructions.**stmcbi**(**args*, ***kwargs*)

Copy clear bit register to clear bit memory cell with run-time address.

Param
source (cbit)

Param
memory address (regint)

class Compiler.GC.instructions.**stmsb**(**args*, ***kwargs*)

Copy secret bit register to secret bit memory cell with compile-time address.

Param
source (sbit)

Param
memory address (int)

class Compiler.GC.instructions.**stmsbi**(**args*, ***kwargs*)

Copy secret bit register to secret bit memory cell with run-time address.

Param
source (sbit)

Param
memory address (regint)

class Compiler.GC.instructions.**trans**(**args*)

Secret bit register vector transpose. The first destination vector will contain the least significant bits of all source vectors etc.

Param
number of arguments to follow (int)

Param
number of outputs (int)

Param
destination for least significant bits (sbit)

Param
(destination for bits one step higher)...

Param
source (sbit)

Param
(source)...

class Compiler.GC.instructions.**xorcb**(**args*, ***kwargs*)

Bitwise XOR of two single clear bit registers.

Param
result (cbit)

Param
operand (cbit)

Param
operand (cbit)

class Compiler.GC.instructions.**xorcbi**(**args*, ***kwargs*)

Bitwise XOR of single clear bit register and immediate.

Param
result (cbit)

Param
operand (cbit)

Param
immediate (int)

class Compiler.GC.instructions.**xorm**(**args*, ***kwargs*)

Bitwise XOR of single secret and clear bit registers.

Param
number of bits (less or equal 64)

Param
result (sbit)

Param
operand (sbit)

Param
operand (cbit)

class Compiler.GC.instructions.**xors**(**args*, ***kwargs*)

Bitwise XOR of secret bit register vectors.

Param
number of arguments to follow (multiple of four)

Param
number of bits (int)

Param
result (sbit)

Param
operand (sbit)

Param
operand (sbit)

Param
(repeat from number of bits)...

2.3 Low-Level Interface

In the following we will explain the basic of the C++ interface by walking trough `Utils/paper-example.cpp`.

```
template<class T>
void run(char** argv, int prime_length);
```

MP-SPDZ heavily uses templating to allow to reuse code between different protocols. `run()` is a simple example of this. The entire virtual machine in the `Processor` directory is built on the same principle. The central type is a type representing a share in a particular type.

```
// bit length of prime
const int prime_length = 128;

// compute number of 64-bit words needed
const int n_limbs = (prime_length + 63) / 64;
```

Computation modulo a prime requires to fix the number of limbs (64-bit words) at compile time. This allows for optimal memory usage and computation.

```
if (protocol == "MASCOT")
    run<Share<gfp_<0, n_limbs>>>(argv, prime_length);
else if (protocol == "CowGear")
    run<CowGearShare<gfp_<0, n_limbs>>>(argv, prime_length);
```

Share types for computation module a prime (and in $\mathrm{GF}(2^n)$) generally take one parameter for the computation domain. `gfp_` in turn takes two parameters, a counter and the number of limbs. The counter allows to use several instances with different parameters. It can be chosen freely, but the convention is to use 0 for the online phase and 1 for the offline phase where required.

```
else if (protocol == "SPDZ2k")
    run<Spdz2kShare<64, 64>>(argv, 0);
```

Share types for computation modulo a power of two simply take the exponent as parameter, and some take an additional security parameter.

```
int my_number = atoi(argv[1]);
int n_parties = atoi(argv[2]);
int port_base = 9999;
Names N(my_number, n_parties, "localhost", port_base);
```

All implemented protocols require point-to-point connections between all parties. *`Names`* objects represent a setup of hostnames and IPs used to set up the actual connections. The chosen initialization provides a way where every party connects to party 0 on a specified location (localhost in this case), which then broadcasts the locations of all parties. The base port number is used to derive the port numbers for the parties to listen on (base + party number). See the the *`Names`* class for other possibilities such as a text file containing hostname and port number for each party.

```
CryptoPlayer P(N);
```

The networking setup is used to set up the actual connections. *`CryptoPlayer`* uses encrypted connection while *`PlainPlayer`* does not. If you use several instances (for several threads for example), you must use an integer identifier as the second parameter, which must differ from any other by at least the number of parties.

```
ProtocolSetup<T> setup(P, prime_length);
```

We have to use a specific prime for computation modulo a prime. This deterministically generates one of the desired length if necessary. For computation modulo a power of two, this does not do anything. Some protocols use an information-theoretic tag that is constant throughout the protocol. This code reads it from storage if available or generates a fresh one otherwise.

```
ProtocolSet<T> set(P, setup);
auto& input = set.input;
auto& protocol = set.protocol;
auto& output = set.output;
```

The `ProtocolSet` contains one instance for every essential protocol step.

```
int n = 1000;
vector<T> a(n), b(n);
T c;
typename T::clear result;
```

Remember that `T` stands for a share in the protocol. The derived type `T::clear` stands for the cleartext domain. Share types support linear operations such as addition, subtraction, and multiplication with a constant. Use `T::constant()` to convert a constant to a share type.

```
input.reset_all(P);
for (int i = 0; i < n; i++)
    input.add_from_all(i);
input.exchange();
for (int i = 0; i < n; i++)
{
    a[i] = input.finalize(0);
    b[i] = input.finalize(1);
}
```

The interface for all protocols proceeds in four stages:

1. Initialization. This is required to initialize and reset data structures in consecutive use.
2. Local data preparation
3. Communication
4. Output extraction

This blueprint allows for a minimal number of communication rounds.

```
protocol.init_dotprod(&processor);
for (int i = 0; i < n; i++)
    protocol.prepare_dotprod(a[i], b[i]);
protocol.next_dotprod();
protocol.exchange();
c = protocol.finalize_dotprod(n);
```

The initialization of the multiplication sets the preprocessing and output instances to use in Beaver multiplication. `next_dotprod()` separates dot products in the data preparation phase.

```
protocol.check();
```

Some protocols require a check of all multiplications up to a certain point. To guarantee that outputs do not reveal secret information, it has to be run before using the output protocol.

```
output.init_open(P);
output.prepare_open(c);
output.exchange(P);
result = output.finalize_open();

cout << "result: " << result << endl;
output.Check(P);
```

The output protocol follows the same blueprint as the multiplication protocol.

```
T::LivePrep::teardown();
```

This frees the memory used for global key material when using homomorphic encryption. Otherwise, this does not do anything.

2.3.1 Domain Types

gfp_<X, L>	Computation modulo a prime. L is the number of 64-bit limbs, that is, it covers primes of bit length $64(L-1)+1$ to $64L$. The type has to be initialized using init_field() or init_default(). The latter picks a prime given a bit length.
SignedZ2<K> / Z2<K>	Computation modulo 2^K. This is not a field.
gf2n_short / gf2n_long / gf2n_<T>	$GF(2^n)$. T denotes a type that is used to store the values. It must support a variety of integer operations. The type has to be initialized using init_field(). The choice of degrees is limited. At the time of writing, 4, 8, 28, 40, 63, and 128 are supported if the storage type is large enough.

2.3.2 Share Types

Type	Protocol
AtlasShare<T>	Semi-honest version of ATLAS (Section 4.2). T must represent a field.
ChaiGearShare<T>	HighGear with covert key setup. T must be gfp_<X, L> or gf2n_short.
CowGearShare<T>	LowGear with covert key setup. T must be gfp_<X, L> or gf2n_short.
HemiShare<T>	Semi-honest protocol with Beaver multiplication based on semi-homomorphic encryption. T must be gfp_<X, L> or gf2n_short.
HighGearShare<T>	HighGear. T must be gfp_<X, L> or gf2n_short.
LowGearShare<T>	LowGear. T must be gfp_<X, L> or gf2n_short.
MaliciousShamirShare<T>	Shamir secret sharing with Beaver multiplication and sacrifice. T must represent a field.
MamaShare<T, N>	MASCOT with multiple MACs. T must represent a field, N is the number of MACs.
PostSacriRepFieldShare<T>	Post-sacrifice protocol using three-party replicated secret sharing with T representing a field.
PostSacriRepRingShare<K, S>	Post-sacrifice protocol using replicated three-party secret sharing modulo 2^K with security parameter S.
Rep3Share2<K>	Three-party semi-honest protocol using replicated secret sharing modulo 2^K.
Rep4Share<T>	Four-party malicious protocol using replicated secret sharing over a field.
Rep4Share2<K>	Four-party malicious protocol using replicated secret sharing modulo 2^K.
SemiShare2<K>	Semi-honest dishonest-majority protocol using Beaver multiplication based on oblivious transfer modulo 2^K.
SemiShare<T>	Semi-honest dishonest-majority protocol using Beaver multiplication based on oblivious transfer in a field.
ShamirShare<T>	Semi-honest protocol based on Shamir's secret sharing. T must represent a field.
Share<T>	MASCOT. T must represent a field.
SohoShare<T>	Semi-honest protocol with Beaver multiplication based on somewhat homomorphic encryption. T must be gfp_<X, L> or gf2n_short.
Spdz2kShare<K, S>	SPDZ2k computing modulo 2^K with security parameter S.
SpdzWiseShare<K, S>	SPDZ-wise computing modulo 2^K with security parameter S.
SpdzWiseShare<T>	SPDZ-wise. T must be MaliciousShamirShare or MaliciousRep3Share.

2.3.3 Protocol Setup

template<class **T**>

class **ProtocolSetup**

Global setup for an arithmetic share type

Subclassed by *MixedProtocolSetup< T >*

Public Functions

inline **ProtocolSetup**(*Player* &P, int prime_length = 0, string directory = "")

Parameters

- **P** – communication instance (used for MAC generation if needed)
- **prime_length** – length of prime if computing modulo a prime
- **directory** – location to read MAC if needed

template<class **T**>

class **ProtocolSet**

Input, multiplication, and output protocol instance for an arithmetic share type

Public Functions

inline **ProtocolSet**(*Player* &P, const *ProtocolSetup*<*T*> &setup)

Parameters

- **P** – communication instance
- **setup** – one-time setup instance

template<class **T**>

class **BinaryProtocolSetup**

Global setup for a binary share type

Public Functions

inline **BinaryProtocolSetup**(*Player* &P, string directory = "")

Parameters

- **P** – communication instance (used for MAC generation if needed)
- **directory** – location to read MAC if needed

template<class **T**>

class **BinaryProtocolSet**

Input, multiplication, and output protocol instance for a binary share type

Public Functions

inline **BinaryProtocolSet**(*Player* &P, const *BinaryProtocolSetup*<*T*> &setup)

Parameters

- **P** – communication instance
- **setup** – one-time setup instance

template<class **T**>

class **MixedProtocolSetup** : public *ProtocolSetup*<*T*>

Global setup for an arithmetic share type and the corresponding binary one

Public Functions

inline **MixedProtocolSetup**(*Player* &P, int prime_length = 0, string directory = "")

Parameters

- **P** – communication instance (used for MAC generation if needed)
- **prime_length** – length of prime if computing modulo a prime
- **directory** – location to read MAC if needed

template<class **T**>

class **MixedProtocolSet**

Input, multiplication, and output protocol instance for an arithmetic share type and the corresponding binary one

Public Functions

inline **MixedProtocolSet**(*Player* &P, const *MixedProtocolSetup*<*T*> &setup)

Parameters

- **P** – communication instance
- **setup** – one-time setup instance

2.3.4 Protocol Interfaces

template<class **T**>

class **ProtocolBase**

Abstract base class for multiplication protocols

Subclassed by Replicated< T >

Public Functions

T **mul**(const *T* &x, const *T* &y)

Single multiplication.

inline virtual void **init**(*Preprocessing*<*T*>&, typename *T*::MAC_Check&)

Initialize protocol if needed (repeated call possible)

virtual void **init_mul**() = 0

Initialize multiplication round.

virtual void **prepare_mul**(const *T* &x, const *T* &y, int n = -1) = 0

Schedule multiplication of operand pair.

virtual void **exchange**() = 0

Run multiplication protocol.

virtual *T* **finalize_mul**(int n = -1) = 0

Get next multiplication result.

virtual void **finalize_mult**(*T* &res, int n = -1)

Store next multiplication result in `res`

inline void **init_dotprod**()

Initialize dot product round.

inline void **prepare_dotprod**(const *T* &x, const *T* &y)

Add operand pair to current dot product.

inline void **next_dotprod**()

Finish dot product.

T **finalize_dotprod**(int length)

Get next dot product result.

template<class **T**>

class **InputBase**

Abstract base for input protocols

Subclassed by Input< T >

Public Functions

virtual void **reset**(int player) = 0

Initialize input round for `player`

void **reset_all**(*Player* &P)

Initialize input round for all players.

virtual void **add_mine**(const typename *T*::open_type &input, int n_bits = -1) = 0

Schedule input from me.

virtual void **add_other**(int player, int n_bits = -1) = 0

Schedule input from other player.

void **add_from_all**(const clear &input, int n_bits = -1)

Schedule input from all players.

virtual void **send_mine**() = 0

Send my inputs.

virtual void **exchange**()

Run input protocol for all players.

virtual *T* **finalize_mine**() = 0

Get share for next input of mine.

virtual void **finalize_other**(int player, *T* &target, *octetStream* &o, int n_bits = -1) = 0

Store share for next input from `player` from buffer `o` in `target`

virtual *T* **finalize**(int player, int n_bits = -1)

Get share for next input from `player`

template<class **T**>

class **MAC_Check_Base**

Abstract base class for opening protocols

Public Functions

inline virtual void **Check**(const *Player* &P)

Run checking protocol.

inline const *T*::mac_key_type::Scalar &**get_alphai**() const

Get MAC key.

virtual void **POpen**(vector<typename *T*::open_type> &values, const vector<*T*> &S, const *Player* &P)

Open values in `S` and store results in `values`

inline *T*::open_type **open**(const *T* &secret, const *Player* &P)

Open single value.

virtual void **init_open**(const *Player* &P, int n = 0)

Initialize opening round.

virtual void **prepare_open**(const *T* &secret)

Add value to be opened.

virtual void **exchange**(const *Player* &P) = 0

Run opening protocol.

virtual *T*::open_type **finalize_open**()

Get next opened value.

virtual void **CheckFor**(const typename *T*::open_type &value, const vector<*T*> &shares, const *Player* &P)

Check whether all `shares` are `value`

template<class **T**>

class **Preprocessing** : public PrepBase

Abstract base class for preprocessing

Subclassed by *BufferPrep< T >*, Sub_Data_Files< T >

Public Functions

virtual array<*T*, 3> **get_triple**(int n_bits)
Get fresh random multiplication triple.

virtual *T* **get_bit**()
Get fresh random bit.

virtual *T* **get_random**()
Get fresh random value in domain.

virtual void **get_dabit**(*T* &a, typename *T*::bit_type &b)
Store fresh daBit in a (arithmetic part) and b (binary part)

template<int>
edabitvec<*T*> **get_edabitvec**(bool strict, int n_bits)
Get fresh edaBit chunk.

template<class **T**>

class **BufferPrep** : public *Preprocessing*<*T*>
Abstract base class for live preprocessing

Subclassed by BitPrep< T >

Public Functions

virtual *T* **get_random**()
Get fresh random value.

Public Static Functions

static inline void **basic_setup**(*Player* &P)
Key-independent setup if necessary (cryptosystem parameters)

static inline void **setup**(*Player* &P, typename *T*::mac_key_type alphai)
Generate keys if necessary.

static inline void **teardown**()
Free memory of global cryptosystem parameters.

2.4 Machine Learning

MP-SPDZ supports a limited subset of the Keras interface for machine learning. This includes the SGD and Adam optimizers and the following layer types: dense, 2D convolution, 2D max-pooling, and dropout.

In the following we will walk through the example code in `keras_mnist_dense.mpc`, which trains a dense neural network for MNIST. It starts by defining tensors to hold data:

```
training_samples = sfix.Tensor([60000, 28, 28])
training_labels = sint.Tensor([60000, 10])

test_samples = sfix.Tensor([10000, 28, 28])
test_labels = sint.Tensor([10000, 10])
```

The tensors are then filled with inputs from party 0 in the order that is used by the preparation script:

```
training_labels.input_from(0)
training_samples.input_from(0)

test_labels.input_from(0)
test_samples.input_from(0)
```

This is followed by Keras-like code setting up the model and training it:

```
from Compiler import ml
tf = ml

layers = [
  tf.keras.layers.Flatten(),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dense(10,  activation='softmax')
]

model = tf.keras.models.Sequential(layers)

optim = tf.keras.optimizers.SGD(momentum=0.9, learning_rate=0.01)

model.compile(optimizer=optim)

opt = model.fit(
  training_samples,
  training_labels,
  epochs=1,
  batch_size=128,
  validation_data=(test_samples, test_labels)
)
```

Lastly, the model is stored on disk in secret-shared form:

```
for var in model.trainable_variables:
  var.write_to_file()
```

2.4.1 Prediction

The example code in `keras_mnist_dense_predict.mpc` uses the model stored above for prediction. Much of the setup is the same, but instead of training it reads the model from disk:

```
model.build(test_samples.sizes)

start = 0
for var in model.trainable_variables:
  start = var.read_from_file(start)
```

Then it runs the prediction:

```
guesses = model.predict(test_samples)
```

Using `var.input_from(player)` instead the model would be input privately by a party.

2.5 Networking

All protocols in MP-SPDZ rely on point-to-point connections between all pairs of parties. This is realized using TCP, which means that every party must be reachable under at least one TCP port. The default is to set this port to a base plus the player number. This allows for easily running all parties on the same host. The base defaults to 5000, which can be changed with the command-line option `--portnumbase`. However, the scripts in `Scripts` use a random base port number, which can be changed using the same option.

There are two ways of communicating hosts and individually setting ports:

1. All parties first to connect to a coordination server, which broadcasts the data for all parties. This is the default with the coordination server being run as a thread of party 0. The hostname of the coordination server has to be given with the command-line parameter `--hostname`, and the coordination server runs on the base port number, thus defaulting to 5000. Furthermore, you can specify a party's listening port using `--my-port`.
2. The parties read the information from a local file, which needs to be the same everywhere. The file can be specified using `--ip-file-name` and has the following format:

   ```
   <host0>[:<port0>]
   <host1>[:<port1>]
   ...
   ```

   The hosts can be both hostnames and IP addresses. If not given, the ports default to base plus party number.

Whether or not encrypted connections are used depends on the security model of the protocol. Honest-majority protocols default to encrypted whereas dishonest-majority protocols default to unencrypted. You change this by either using `--encrypted/-e` or `--unencrypted/-u`.

If using encryption, the certificates (`Player-Data/*.pem`) must be the same on all hosts, and you have to run `c_rehash Player-Data` on all of them. `Scripts/setup-ssl.sh` can be used to generate the necessary certificates. The common name has to be `P<player number>` for computing parties and `C<client number>` for clients.

2.5.1 Internal Infrastructure

The internal networking infrastructure of MP-SPDZ reflects the needs of the various multi-party computation. For example, some protocols require a simultaneous broadcast from all parties whereas other protocols require that every party sends different information to different parties (include none at all). The infrastructure makes sure to send and receive in parallel whenever possible.

All communication is handled through two subclasses of *`Player`* defined in `Networking/Player.h`. *`PlainPlayer`* communicates in cleartext while *`CryptoPlayer`* uses TLS encryption. The former uses the same BSD socket for sending and receiving but the latter uses two different connections for sending and receiving. This is because TLS communication is never truly one-way due key renewals etc., so the only way for simultaneous sending and receiving we found was to use two connections in two threads.

If you wish to use a different networking facility, we recommend to subclass *`Player`* and fill in the virtual-only functions required by the compiler (e.g., `send_to_no_stats()` for sending to one other party). Note that not all protocols require all functions, so you only need to properly implement those you need. You can then replace uses of *`PlainPlayer`* or *`CryptoPlayer`* by your own class. Furthermore, you might need to extend the `Names` class to suit your purpose. By default, *`Names`* manages one TCP port that a party is listening on for connections. If this suits you, you don't need to change anything

2.5.1.1 Reference

class **Names**

Network setup (hostnames and port numbers)

Public Functions

void **init**(int player, int pnb, int my_port, const char *servername, bool setup_socket = true)

Initialize with central server

Parameters

- **player** – my number
- **pnb** – base port number (server listens one below)
- **my_port** – my port number (DEFAULT_PORT for default, which is base port number plus player number)
- **servername** – location of server
- **setup_socket** – whether to start listening

inline **Names**(int player, int pnb, int my_port, const char *servername)

Names(int player, int nplayers, const string &servername, int pnb, int my_port = *DEFAULT_PORT*)

Initialize with central server running on player 0

Parameters

- **player** – my number
- **nplayers** – number of players
- **servername** – location of player 0
- **pnb** – base port number
- **my_port** – my port number (DEFAULT_PORT for default, which is base port number plus player number)

void **init**(int player, int pnb, vector<string> Nms)

Initialize without central server

Parameters

- **player** – my number
- **pnb** – base port number
- **Nms** – locations of all parties

inline **Names**(int player, int pnb, vector<string> Nms)

void **init**(int player, int pnb, const string &hostsfile, int players = 0)

Initialize from file. One party per line, format <hostname>[:<port>]

Parameters

- **player** – my number
- **pnb** – base port number

- **hostsfile** – filename
- **players** – number of players (0 to take from file)

inline **Names**(int player, int pnb, const string &hostsfile)

Names(ez::ezOptionParser &opt, int argc, const char **argv, int default_nplayers = 2)

Initialize from command-line options

Parameters

- **opt** – option parser instance
- **argc** – number of command-line arguments
- **argv** – command-line arguments
- **default_nplayers** – default number of players (used if not given in arguments)

inline **Names**()

Names(const *Names* &other)

~Names()

inline int **num_players**() const

inline int **my_num**() const

inline const string **get_name**(int i) const

inline int **get_portnum_base**() const

Public Static Attributes

static const int **DEFAULT_PORT** = -1

class **Player** : public PlayerBase

Abstract class for multi-player communication. `*_no_stats` functions are called by their equivalents after accounting for communications statistics.

Subclassed by *MultiPlayer< T >*, *MultiPlayer< int >*, *MultiPlayer< ssl_socket *>*

Public Functions

inline virtual int **num_players**() const

Get number of players

inline virtual int **my_num**() const

Get my player number

virtual void **send_all**(const *octetStream* &o) const

Send the same to all other players

void **send_to**(int player, const *octetStream* &o) const

Send to a specific player

void **receive_all**(vector<*octetStream*> &os) const

Receive from all other players. Information from player 0 at `os[0]` etc.

void **receive_player**(int i, *octetStream* &o) const

Receive from a specific player

void **send_relative**(const vector<*octetStream*> &o) const

Send to all other players by offset. `o[0]` gets sent to the next player etc.

void **receive_relative**(vector<*octetStream*> &o) const

Receive from all other players by offset. `o[0]` will contain data from the next player etc.

void **receive_relative**(int offset, *octetStream* &o) const

Receive from other player specified by offset. 1 stands for the next player etc.

void **exchange**(int other, const *octetStream* &to_send, *octetStream* &ot_receive) const

Exchange information with one other party, reusing the buffer if possible.

void **exchange**(int other, *octetStream* &o) const

Exchange information with one other party, reusing the buffer.

void **exchange_relative**(int offset, *octetStream* &o) const

Exchange information with one other party specified by offset, reusing the buffer if possible.

inline virtual void **pass_around**(*octetStream* &o, int offset = 1) const

Send information to a party while receiving from another by offset, The default is to send to the next party while receiving from the previous. The buffer is reused.

void **pass_around**(*octetStream* &to_send, *octetStream* &to_receive, int offset) const

Send information to a party while receiving from another by offset. The default is to send to the next party while receiving from the previous.

virtual void **unchecked_broadcast**(vector<*octetStream*> &o) const

Broadcast and receive data to/from all players. Assumes o[player_no] contains the data to be broadcast by me.

virtual void **Broadcast_Receive**(vector<*octetStream*> &o) const

Broadcast and receive data to/from all players with eventual verification. Assumes o[player_no] contains the data to be broadcast by me.

virtual void **Check_Broadcast**() const

Run protocol to verify broadcast is correct

void **send_receive_all**(const vector<*octetStream*> &to_send, vector<*octetStream*> &to_receive) const

Send something different to each player.

void **send_receive_all**(const vector<bool> &senders, const vector<*octetStream*> &to_send, vector<*octetStream*> &to_receive) const

Specified senders only send something different to each player.

Parameters

- **senders** – set whether a player sends or not, must be equal on all players
- **to_send** – data to send by player number
- **to_receive** – received data by player number

void **send_receive_all**(const vector<vector<bool>> &channels, const vector<*octetStream*> &to_send, vector<*octetStream*> &to_receive) const

Send something different only one specified channels.

Parameters

- **channels** – channel[i][j] indicates whether party i sends to party j
- **to_send** – data to send by player number
- **to_receive** – received data by player number

virtual void **partial_broadcast**(const vector<bool> &senders, const vector<bool> &receivers, vector<*octetStream*> &os) const

Specified senders broadcast information to specified receivers.

Parameters

- **senders** – specify which parties do send
- **receivers** – specify which parties do send
- **os** – data to send at os[my_number], received data elsewhere

template<class **T**>

class **MultiPlayer** : public *Player*

Multi-player communication helper class. T = int for unencrypted BSD sockets and T = ssl_socket* for Boost SSL streams.

class **PlainPlayer** : public *MultiPlayer*<int>

Plaintext multi-player communication

Subclassed by ThreadPlayer

Public Functions

PlainPlayer(const *Names* &Nms, const string &id)

Start a new set of unencrypted connections.

Parameters

- **Nms** – network setup
- **id** – unique identifier

class **CryptoPlayer** : public *MultiPlayer*<ssl_socket*>

Encrypted multi-party communication. Uses OpenSSL and certificates issued to "P<player_no>". Sending and receiving is done in separate threads to allow for bidirectional communication.

Public Functions

CryptoPlayer(const *Names* &Nms, const string &id)

Start a new set of encrypted connections.

Parameters

- **Nms** – network setup
- **id** – unique identifier

virtual void **partial_broadcast**(const vector<bool> &my_senders, const vector<bool> &my_receivers, vector<*octetStream*> &os) const

Specified senders broadcast information to specified receivers.

Parameters

- **senders** – specify which parties do send
- **receivers** – specify which parties do send
- **os** – data to send at `os[my_number]`, received data elsewhere

class **octetStream**

Buffer for network communication with a pointer for sequential reading. When sent over the network or stored in a file, the length is prefixed as eight bytes in little-endian order.

Public Functions

inline void **resize**(size_t l)

Increase allocation if needed.

void **clear**()

Free memory.

octetStream(size_t maxlen)

Initial allocation.

octetStream(size_t len, const octet *source)

Initial buffer.

octetStream(const string &other)

Initial buffer.

inline size_t **get_ptr**() const

Number of bytes already read.

inline size_t **get_length**() const

Length.

inline size_t **get_max_length**() const

Allocation.

inline octet ***get_data**() const

Data pointer.

inline octet ***get_data_ptr**() const

Read pointer.

inline bool **done**() const

Whether done reading.

inline bool **empty**() const

Whether empty.

inline size_t **left**() const

Bytes left to read.

string **str**() const

Convert to string.

octetStream **hash**() const

Hash content.

void **concat**(const *octetStream* &os)

Append other buffer.

inline void **reset_read_head**()

Reset reading.

inline void **reset_write_head**()

Set length to zero but keep allocation.

inline bool **operator==**(const *octetStream* &a) const

Equality test.

void **append_random**(size_t num)

Append `num` random bytes.

inline void **append**(const octet *x, const size_t l)

Append `l` bytes from `x`

inline void **consume**(octet *x, const size_t l)

Read `l` bytes to `x`

inline void **store**(unsigned int a)

Append 4-byte integer.

void **store**(int a)

Append 4-byte integer.

inline void **get**(unsigned int &a)

Read 4-byte integer.

void **get**(int &a)

Read 4-byte integer.

inline void **store**(size_t a)

Append 8-byte integer.

inline void **get**(size_t &a)

Read 8-byte integer.

inline void **store_int**(size_t a, int n_bytes)

Append integer of `n_bytes` bytes.

inline size_t **get_int**(int n_bytes)

Read integer of `n_bytes` bytes.

template<int **N_BYTES**>
inline void **store_int**(size_t a)

Append integer of `N_BYTES` bytes.

template<int **N_BYTES**>
inline size_t **get_int**()

Read integer of `N_BYTES` bytes.

void **store**(const bigint &x)

Append big integer.

void **get**(bigint &ans)

Read big integer.

template<class **T**>
void **store**(const *T* &x)

Append instance of type implementing `pack`

template<class **T**>
T **get**()

Read instance of type implementing `unpack`

template<class **T**>
void **get**(*T* &ans)

Read instance of type implementing `unpack`

template<class **T**>
void **store**(const vector<*T*> &v)

Append vector of type implementing `pack`

template<class **T**>
void **get**(vector<*T*> &v, const *T* &init = { })

Read vector of type implementing `unpack`

Parameters

- **v** – results
- **init** – initialization if required

template<class **T**>
void **get_no_resize**(vector<*T*> &v)

Read vector of type implementing `unpack` if vector already has the right size

inline void **consume**(*octetStream* &s, size_t l)

Read `l` bytes into separate buffer.

template<class **T**>
inline void **Send**(*T* socket_num) const

Send on `socket_num`

template<class **T**>
inline void **Receive**(*T* socket_num)

Receive on `socket_num`, overwriting current content.

void **input**(istream &s)

Input from stream, overwriting current content.

void **output**(ostream &s)

Output to stream.

template<class **T**>
inline void **exchange**(*T* send_socket, *T* receive_socket)

Send on `socket_num` while receiving on `receiving_socket`, overwriting current content

template<class **T**>
void **exchange**(*T* send_socket, *T* receive_socket, *octetStream* &receive_stream) const

Send this buffer on `socket_num` while receiving to `receive_stream` on `receiving_socket`

2.6 Input/Output

This section gives an overview over the input/output facilities.

2.6.1 Private Inputs from Computing Parties

All secret types have an input function (e.g. *Compiler.types.sint.get_input_from()* or *Compiler.types.sfix.get_input_from()*). Inputs are read as whitespace-separated text in order (independent of the data type) from `Player-Data/Input-P<player>-<thread>`, where `thread` is `0` for the main thread. You can change the prefix (`Player-Data/Input`) using the `-IF` option on the virtual machine binary. You can also use `-I` to read inputs from the command line. *Compiler.types.sint.input_tensor_from()* and *Compiler.types.sfix.input_tensor_from()* allow inputting a tensor.

2.6.2 Compile-Time Data via Private Input

input_tensor_via() is a convenience function that allows to use data available at compile-time via private input.

2.6.3 Public Inputs

All types can be assigned a hard-coded value at compile time, e.g. `sint(1)`. This is impractical for larger amounts of data. *foreach_enumerate()* provides a facility for this case. It uses *public_input* internally, which reads from `Programs/Public-Input/<progname>`.

2.6.4 Public Outputs

By default, *print_ln()* and related functions only output to the terminal on party 0. This allows to run several parties in one terminal without spoiling the output. You can use interactive mode with option `-I` in order to output on all parties. Note that this also to reading inputs from the command line unless you specify `-IF` as well. You can also specify a file prefix with `-OF`, so that outputs are written to `<prefix>-P<player>-<thread>`.

2.6.5 Private Outputs to Computing Parties

Some types provide a function to reveal a value only to a specific party (e.g., *Compiler.types.sint.reveal_to()*). It can be used conjunction with *print_ln_to()* in order to output it.

2.6.6 Binary Output

Most types returned by `reveal()` or `reveal_to()` feature a `binary_output()` method, which writes to `Player-Data/Binary-Output-P<playerno>-<threadno>`. The format is either signed 64-bit integer or double-precision floating-point in machine byte order (usually little endian).

2.6.7 Clients (Non-computing Parties)

Compiler.types.sint.receive_from_client() and *Compiler.types.sint.reveal_to_clients()* allow communicating securely with the clients. See this example covering both client code and server-side high-level code. *Compiler.types.sint.input_tensor_from_client()* and *Compiler.types.MultiArray.reveal_to_clients()*. The same functions are available for *sfix* and *Array*, respectively. See also *Reference* below.

2.6.8 Secret Shares

Compiler.types.sint.read_from_file() and *Compiler.types.sint.write_to_file()* allow reading and writing secret shares to and from files. These instructions use `Persistence/Transactions-P<playerno>.data`. The format depends on the protocol with the following principles.

- One share follows the other without metadata.
- If there is a MAC, it comes after the share.
- Numbers are stored in little-endian format.
- Numbers modulo a power of two are stored with the minimal number of bytes.
- Numbers modulo a prime are stored in Montgomery representation in blocks of eight bytes.

Another possibility for persistence between program runs is to use the fact that the memory is stored in `Player-Data/Memory-<protocol>-P<player>` at the end of a run. The best way to use this is via the memory access functions like *store_in_mem()* and *load_mem()*. Make sure to only use addresses below `USER_MEM` specified in `Compiler/config.py` to avoid conflicts with the automatic allocation used for arrays etc. Note also that all types based on *sint* (e.g., *sfix*) share the same memory, and that the address is only a base address. This means that vectors will be written to the memory starting at the given address.

2.6.9 Reference

class **Client**

Client-side interface

Public Functions

Client(const vector<string> &hostnames, int port_base, int my_client_id)

Start a new set of connections to computing parties.

Parameters

- **hostnames** – location of computing parties
- **port_base** – port base
- **my_client_id** – client identifier

template<class **T**>
void **send_private_inputs**(const vector<*T*> &values)

Securely input private values.

Parameters
values – vector of integer-like values

template<class **T**>
vector<*T*> **receive_outputs**(int n)

Securely receive output values.

Parameters
n – number of values

Returns
vector of integer-like values

Public Members

vector<ssl_socket*> **sockets**

Sockets for cleartext communication

octetStream **specification**

Specification of computation domain

2.7 Non-linear Computation

While the computation of addition and multiplication varies from protocol, non-linear computation such as comparison in arithmetic domains (modulus other than two) only comes in three flavors throughout MP-SPDZ:

Unknown prime modulus
This approach goes back to Catrina and de Hoogh. It crucially relies on the use of secret random bits in the arithmetic domain. Enough such bits allow to mask a secret value so that it is secure to reveal the masked value. This can then be split in bits as it is public. The public bits and the secret mask bits are then used to compute a number of non-linear functions. The same idea has been used to implement fixed-point and floating-point computation. We call this method "unknown prime modulus" because it only mandates a minimum modulus size for a given cleartext range, which is roughly the cleartext bit length plus a statistical security parameter. It has the downside that there is implicit enforcement of the cleartext range.

Known prime modulus
Damgård et al. have proposed non-linear computation that involves an exact prime modulus. We have implemented the refined bit decomposition by Nishide and Ohta, which enables further non-linear computation. Our

assumption with this method is that the cleartext space is slightly smaller the full range modulo the prime. This allows for comparison by taking a difference and extracting the most significant bit, which is different than the above works that implement comparison between two positive numbers modulo the prime. We also used an idea by Makri et al., namely that a random k-bit number is indistinguishable from a random number modulo p if the latter is close enough to 2^k.

Power-of-two modulus

In the context of non-linear computation, there are two important differences to prime modulus setting:

1. Multiplication with a power of two effectively erases some of the most significant bits.
2. There is no right shift using multiplication. Modulo a prime, multiplying with a power of the inverse of two allows to right-shift numbers with enough zeros as least significant bits.

Taking this differences into account, Dalskov et al. have adapted the mask-and-reveal approach above to the setting of computation modulo a power of two.

2.7.1 Mixed-Circuit Computation

Another approach to non-linear computation is switching to binary computation for parts of the computation. MP-SPDZ implements protocols proposed for particular security models by a number of works: Demmler et al., Mohassel and Rindal, and Dalskov et al. MP-SPDZ also implements more general methods such as daBits and edaBits.

2.7.1.1 Protocol Pairs

The following table lists the matching arithmetic and binary protocols.

Arithmetic	Binary
MASCOT, SPDZ2k, LowGear, HighGear, CowGear, ChaiGear	Tinier with improved cut-and-choose analysis by Furukawa et al.
Semi, Hemi, Soho, Semi2k	SemiBin (Beaver triples modulo 2 using OT)
Malicious Shamir	Malicious Shamir over $GF(2^{40})$ for secure sacrificing
Malicious Rep3, Post-Sacrifice, SPDZ-wise replicated	Malicious Rep3 modulo 2
Rep4	Rep4 modulo 2
Shamir	Shamir over $GF(2^8)$
ATLAS	ATLAS over $GF(2^8)$
Rep3	Rep3

2.8 Preprocessing

Many protocols in MP-SPDZ use preprocessing, that is, producing secret shares that are independent of the actual data but help with the computation. Due to the independence, this can be done in batches to save communication rounds and even communication when using homomorphic encryption that works with large vectors such as LWE-based encryption.

Generally, preprocessing is done on demand and per computation threads. On demand means that batches of preprocessing data are computed whenever there is none in storage, and a computation thread is a thread created by control flow instructions such as `for_range_multithread()`.

The exceptions to the general rule are edaBit generation with malicious security and AND triples with malicious security and honest majority, both when using bucket size three. Bucket size three implies batches of over a million to achieve 40-bit statistical security, and in honest-majority binary computation the item size is 64, which makes the actual batch size 64 million triples. In multithreaded programs, the preprocessing is run centrally using the threads as helpers.

The batching means that the cost in terms of time and communication jump whenever another batch is generated. Note that, while some protocols are flexible with the batch size and can thus be controlled using `-b`, others mandate a batch size, which can be as large as a million.

2.8.1 Separate preprocessing

It is possible to separate out the preprocessing from the input-dependent ("online") phase. This is done by either option `-F` or `-f` on the virtual machines. In both cases, the preprocessing data is read from files, either all data per type from a single file (`-F`) or one file per thread (`-f`). The latter allows to use named pipes.

The file name depends on the protocol and the computation domain. It is generally `<prefix>/<number of players>-<protocol shorthand>-<domain length>/<preprocessing type>-<protocol shorthand>-P<player number>[-T<thread number>]`. For example, the triples for party 1 in SPDZ modulo a 128-bit prime can be found in `Player-Data/2-p-128/Triples-p-P1`. The protocol shorthand can be found by calling `<share type>::type_short()`. See *Share Types* for a description of the share types.

Preprocessing files start with a header describing the protocol and computation domain to avoid errors due to mismatches. The header is as follows:

- Length to follow (little-endian 8-byte number)
- Protocol descriptor
- Domain descriptor

The protocol descriptor is defined by `<share type>::type_string()`. For SPDZ modulo a prime it is `SPDZ gfp`.

The domain descriptor depends on the kind of domain:

Modulo a prime
Serialization of the prime

- Sign bit (0 as 1 byte)
- Length to follow (little-endian 4-byte number)
- Prime (big-endian)

Modulo a power of two:
Exponent (little-endian 4-byte number)

$GF(2^n)$

- Storage size in bytes (little-endian 8-byte number). Default is 16.
- n (little-endian 4-byte number)

As an example, the following output of `hexdump -C` describes SPDZ modulo the default 128-bit prime (170141183460469231731687303715885907969):

```
00000000  1d 00 00 00 00 00 00 00  53 50 44 5a 20 67 66 70  |........SPDZ gfp|
00000010  00 10 00 00 00 80 00 00  00 00 00 00 00 00 00 00  |................|
00000020  00 00 1b 80 01                                    |.....|
00000025
```

`Fake-Offline.x` generates preprocessing data insecurely for a range of protocols, and `{mascot,cowgear,mal-shamir}-offline.x` generate sufficient preprocessing data for a specific high-level program with MASCOT, CowGear, and malicious Shamir secret sharing, respectively.

2.9 Adding a Protocol

In order to illustrate how to create a virtual machine for a new protocol, we have created one with blanks to be filled in. It is defined in the following files:

Machines/no-party.cpp
Contains the main function.

Protocols/NoShare.h
Contains the `NoShare` class, which is supposed to hold one share. `NoShare` takes the cleartext type as a template parameter.

Protocols/NoProtocol.h
Contains a number of classes representing instances of protocols:

NoInput
Private input.

NoProtocol
Multiplication protocol.

NoOutput
Public output.

Protocols/NoLivePrep.h
Contains the `NoLivePrep` class, representing a preprocessing instance.

The number of blanks can be overwhelming. We therefore recommend the following approach to get started. If the desired protocol resembles one that is already implemented, you can check its code for inspiration. The main function of `<protocol>-party.x` can be found in `Machines/<protocol>-party.cpp`, which in turns contains the name of the share class. For example `replicated-ring-party.x` is implemented in `Machines/replicated-ring-party.cpp`, which refers to `Rep3Share2()` in `Protocols/Rep3Share2.h`. There you will find that it uses `Replicated()` for multiplication, which is found in `Protocols/Replicated.h`.

1. Fill in the `constant()` static member function of `NoShare` as well as the `exchange()` member function of `NoOutput`. Check out `DirectSemiMC<T>::exchange_()` in `Protocols/SemiMC.hpp` for a simple example. It opens an additive secret sharing by sending all shares to all other parties and then summing up the received. See *this reference* for documentation on the necessary infrastructure. Constant sharing and public output allows to execute the following program:

```
print_ln('%s', sint(123).reveal())
```

   This allows to check the correct execution of further functionality.

2. Fill in the operator functions in `NoShare` and check them:

```
print_ln('%s', (sint(2) + sint(3)).reveal())
print_ln('%s', (sint(2) - sint(3)).reveal())
print_ln('%s', (sint(2) * cint(3)).reveal())
```

   Many protocols use these basic operations, which makes it beneficial to check the correctness

3. Fill in `NoProtocol`. Alternatively, if the desired protocol is based on Beaver multiplication, you can specify the following in `NoShare`:

```
typedef Beaver<This> Protocol;
```

   Then add the desired triple generation to `NoLivePrep::buffer_triples()`. In any case you should then be able to execute:

```
print_ln('%s', (sint(2) * sint(3)).reveal())
```

4. In order to execute many kinds of non-linear computation, random bits are needed. After filling in `NoLivePrep::buffer_bits()`, you should be able to execute:

```
print_ln('%s', (sint(2) < sint(3)).reveal()
```

2.10 Troubleshooting

This section shows how to solve some common issues.

2.10.1 Crash without error message or `bad_alloc`

Some protocols require several gigabytes of memory, and the virtual machine will crash if there is not enough RAM. You can reduce the memory usage for some malicious protocols with `-B 5`. Furthermore, every computation thread requires separate resources, so consider reducing the number of threads with `for_range_multithreads()` and similar.

2.10.2 List indices must be integers or slices

You cannot access Python lists with runtime variables because the lists only exists at compile time. Consider using *`Array`*.

2.10.3 `compile.py` takes too long or runs out of memory

If you use Python loops (`for`), they are unrolled at compile-time, resulting in potentially too much virtual machine code. Consider using *`for_range()`* or similar. You can also use `-l` when compiling, which will replace simple loops by an optimized version.

2.10.4 Order of memory instructions not preserved

By default, the compiler runs optimizations that in some corner case can introduce errors with memory accesses such as accessing an *`Array`*. If you encounter such errors, you can fix this either with `-M` when compiling or placing *break_point()* around memory accesses.

2.10.5 Odd timings

Many protocols use preprocessing, which means they execute expensive computation to generates batches of information that can be used for computation until the information is used up. An effect of this is that computation can seem oddly slow or fast. For example, one multiplication has a similar cost then some thousand multiplications when using homomorphic encryption because one batch contains information for more than than 10,000 multiplications. Only when a second batch is necessary the cost shoots up. Other preprocessing methods allow for a variable batch size, which can be changed using `-b`. Smaller batch sizes generally reduce the communication cost while potentially increasing the number of communication rounds. Try adding `-b 10` to the virtual machine (or script) arguments for very short computations.

2.10.6 Disparities in round figures

The number of virtual machine rounds given by the compiler are not an exact prediction of network rounds but the number of relevant protocol calls (such as multiplication, input, output etc) in the program. The actual number of network rounds is determined by the choice of protocol, which might use several rounds per protocol call. Furthermore, communication at the beginning and the end of a computation such as random key distribution and MAC checks further increase the number of network rounds.

2.10.7 Handshake failures

If you run on different hosts, the certificates (`Player-Data/*.pem`) must be the same on all of them. Furthermore, party <i> requires `Player-Data/P<i>.key` that must match `Player-Data/P<i>.pem`, that is, they have to be generated to together. The easiest way of setting this up is to run `Scripts/setup-ssl.sh` on one host and then copy all `Player-Data/*.{pem,key}` to all other hosts. This is *not* secure but it suffices for experiments. A secure setup would generate every key pair locally and then distributed only the public keys. Finally, run `c_rehash Player-Data` on all hosts. The certificates generated by `Scripts/setup-ssl.sh` expire after a month, so you need to regenerate them. The same holds for `Scripts/setup-client.sh` if you use the client facility.

2.10.8 Connection failures

MP-SPDZ requires one TCP port per party to be open to other parties. In the default setting, it's 5000 on party 0, and 5001 on party 1 etc. You change change the base port (5000) using `--portnumbase` and individual ports for parties using `--my-port`. The scripts in use a random base port number, which you can also change with `--portnumbase`.

2.10.9 Internally called tape has unknown offline data usage

Certain computations are not compatible with reading preprocessing from disk. You can compile the binaries with `MY_CFLAGS += -DINSECURE` in `CONFIG.mine` in order to execute the computation in a way that reuses preprocessing.

2.10.10 Illegal instruction

By default, the binaries are optimized for the machine they are compiled on. If you try to run them an another one, make sure set `ARCH` in `CONFIG` accordingly. Furthermore, if you run on an x86 processor without AVX (produced before 2011), you need to set `AVX_OT = 0` to run dishonest-majority protocols.

2.10.11 Invalid instruction

The compiler code and the virtual machine binary have to be from the same version because most version slightly change the bytecode. This mean you can only use the precompiled binaries with the Python code in the same release.

2.10.12 Computation used more preprocessing than expected

This indicates an error in the internal accounting of preprocessing. Please file a bug report.

2.10.13 `mac_fail`

This is a catch-all failure in protocols with malicious protocols that can be caused by something being wrong at any level. Please file a bug report with the specifics of your case.

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

C

Compiler.circuit, 67
Compiler.GC.instructions, 107
Compiler.GC.types, 48
Compiler.instructions, 76
Compiler.library, 53
Compiler.ml, 61
Compiler.mpc_math, 59
Compiler.oram, 70
Compiler.program, 68
Compiler.types, 5

INDEX

Symbols

-B
    command line option, 3
-C
    command line option, 4
-D
    command line option, 4
-F
    command line option, 3
-P
    command line option, 3
-R
    command line option, 3
-X
    command line option, 3
-Y
    command line option, 4
-Z
    command line option, 4
--CISC
    command line option, 4
--binary
    command line option, 3
--budget
    command line option, 4
--dead-code-elimination
    command line option, 4
--edabit
    command line option, 4
--field
    command line option, 3
--flow-optimization
    command line option, 4
--mixed
    command line option, 3
--prime
    command line option, 3
--ring
    command line option, 3
--split
    command line option, 4
-b
    command line option, 4
-l
    command line option, 4

A

accept_client_connection() (*in module Compiler.library*), 53
acceptclientconnection (*class in Compiler.instructions*), 76
acos() (*in module Compiler.mpc_math*), 59
Adam (*class in Compiler.ml*), 61
Add (*class in Compiler.ml*), 62
addc (*class in Compiler.instructions*), 76
addcb (*class in Compiler.GC.instructions*), 107
addcbi (*class in Compiler.GC.instructions*), 107
addci (*class in Compiler.instructions*), 76
addint (*class in Compiler.instructions*), 76
addm (*class in Compiler.instructions*), 76
adds (*class in Compiler.instructions*), 77
addsi (*class in Compiler.instructions*), 77
andc (*class in Compiler.instructions*), 77
andci (*class in Compiler.instructions*), 77
andm (*class in Compiler.GC.instructions*), 107
andrs (*class in Compiler.GC.instructions*), 107
ands (*class in Compiler.GC.instructions*), 108
approx_sigmoid() (*in module Compiler.ml*), 66
Argmax (*class in Compiler.ml*), 62
argmax() (*in module Compiler.ml*), 65
Array (*class in Compiler.types*), 6
Array() (*Compiler.types.cfix class method*), 19
Array() (*Compiler.types.cgf2n class method*), 21
Array() (*Compiler.types.cint class method*), 23
Array() (*Compiler.types.regint class method*), 27
Array() (*Compiler.types.sfix class method*), 31
Array() (*Compiler.types.sfloat class method*), 34
Array() (*Compiler.types.sgf2n class method*), 36
Array() (*Compiler.types.sint class method*), 40
asin() (*in module Compiler.mpc_math*), 59
asm_open (*class in Compiler.instructions*), 77
assign() (*Compiler.types.Array method*), 6
assign() (*Compiler.types.Matrix method*), 9
assign() (*Compiler.types.MultiArray method*), 14

assign_all() (*Compiler.types.Array method*), 6
assign_all() (*Compiler.types.Matrix method*), 9
assign_all() (*Compiler.types.MultiArray method*), 14
assign_part_vector() (*Compiler.types.Array method*), 6
assign_part_vector() (*Compiler.types.Matrix method*), 9
assign_part_vector() (*Compiler.types.MultiArray method*), 15
assign_vector() (*Compiler.types.Array method*), 6
assign_vector() (*Compiler.types.Matrix method*), 9
assign_vector() (*Compiler.types.MultiArray method*), 15
assign_vector_by_indices() (*Compiler.types.Matrix method*), 9
assign_vector_by_indices() (*Compiler.types.MultiArray method*), 15
atan() (*in module Compiler.mpc_math*), 59

B

backward() (*Compiler.ml.Optimizer method*), 64
binary_output() (*Compiler.types.Array method*), 6
binary_output() (*Compiler.types.cfix method*), 19
binary_output() (*Compiler.types.cfloat method*), 20
binary_output() (*Compiler.types.cgf2n method*), 21
binary_output() (*Compiler.types.cint method*), 23
binary_output() (*Compiler.types.regint method*), 27
BinaryProtocolSet (*C++ class*), 120
BinaryProtocolSet::BinaryProtocolSet (*C++ function*), 121
BinaryProtocolSetup (*C++ class*), 120
BinaryProtocolSetup::BinaryProtocolSetup (*C++ function*), 120
bit (*class in Compiler.instructions*), 78
bit_adder() (*Compiler.GC.types.sbits static method*), 51
bit_adder() (*Compiler.types.cint static method*), 23
bit_adder() (*Compiler.types.regint static method*), 27
bit_adder() (*Compiler.types.sint static method*), 40
bit_and() (*Compiler.types.cgf2n method*), 21
bit_and() (*Compiler.types.cint method*), 24
bit_and() (*Compiler.types.regint method*), 27
bit_and() (*Compiler.types.sgf2n method*), 36
bit_and() (*Compiler.types.sint method*), 40
bit_compose() (*Compiler.types.cgf2n class method*), 21
bit_compose() (*Compiler.types.cint class method*), 24
bit_compose() (*Compiler.types.regint static method*), 28
bit_compose() (*Compiler.types.sgf2n class method*), 36
bit_compose() (*Compiler.types.sint class method*), 41
bit_decompose() (*Compiler.types.cgf2n method*), 21
bit_decompose() (*Compiler.types.cint method*), 24
bit_decompose() (*Compiler.types.regint method*), 28
bit_decompose() (*Compiler.types.sgf2n method*), 36
bit_decompose() (*Compiler.types.sint method*), 41
bit_not() (*Compiler.types.cgf2n method*), 21
bit_not() (*Compiler.types.cint method*), 24
bit_not() (*Compiler.types.regint method*), 28
bit_not() (*Compiler.types.sgf2n method*), 36
bit_not() (*Compiler.types.sint method*), 41
bit_or() (*Compiler.types.cgf2n method*), 21
bit_or() (*Compiler.types.cint method*), 24
bit_or() (*Compiler.types.regint method*), 28
bit_or() (*Compiler.types.sgf2n method*), 36
bit_or() (*Compiler.types.sint method*), 41
bit_xor() (*Compiler.types.cgf2n method*), 22
bit_xor() (*Compiler.types.cint method*), 24
bit_xor() (*Compiler.types.regint method*), 28
bit_xor() (*Compiler.types.sgf2n method*), 36
bit_xor() (*Compiler.types.sint method*), 41
bitb (*class in Compiler.GC.instructions*), 108
bitcoms (*class in Compiler.GC.instructions*), 108
bitdecc (*class in Compiler.GC.instructions*), 108
bitdecint (*class in Compiler.instructions*), 78
bitdecs (*class in Compiler.GC.instructions*), 109
bits (*class in Compiler.GC.types*), 48
break_loop() (*in module Compiler.library*), 53
break_point() (*in module Compiler.library*), 53
BufferPrep (*C++ class*), 124
BufferPrep::basic_setup (*C++ function*), 124
BufferPrep::get_random (*C++ function*), 124
BufferPrep::setup (*C++ function*), 124
BufferPrep::teardown (*C++ function*), 124

C

cbits (*class in Compiler.GC.types*), 48
cfix (*class in Compiler.types*), 19
cfloat (*class in Compiler.types*), 20
cgf2n (*class in Compiler.types*), 20
check (*class in Compiler.instructions*), 78
check_point() (*in module Compiler.library*), 53
cint (*class in Compiler.types*), 23
Circuit (*class in Compiler.circuit*), 67
cisc (*class in Compiler.instructions*), 78
Client (*C++ class*), 135
Client::Client (*C++ function*), 136
Client::receive_outputs (*C++ function*), 136
Client::send_private_inputs (*C++ function*), 136
Client::sockets (*C++ member*), 136
Client::specification (*C++ member*), 136
closeclientconnection (*class in Compiler.instructions*), 80
command line option
    -B, 3
    -C, 4
    -D, 4
    -F, 3
    -P, 3

-R, 3
-X, 3
-Y, 4
-Z, 4
--CISC, 4
--binary, 3
--budget, 4
--dead-code-elimination, 4
--edabit, 4
--field, 3
--flow-optimization, 4
--mixed, 3
--prime, 3
--ring, 3
--split, 4
-b, 4
-l, 4
Compiler.circuit
module, 67
Compiler.GC.instructions
module, 107
Compiler.GC.types
module, 48
Compiler.instructions
module, 76
Compiler.library
module, 53
Compiler.ml
module, 61
Compiler.mpc_math
module, 59
Compiler.oram
module, 70
Compiler.program
module, 68
Compiler.types
module, 5
compute_reciprocal() (*Compiler.types.sfix method*), 31
Concat (*class in Compiler.ml*), 62
cond_print_plain (*class in Compiler.instructions*), 80
cond_print_str (*class in Compiler.instructions*), 81
cond_print_strb (*class in Compiler.GC.instructions*), 109
cond_swap() (*Compiler.types.cgf2n method*), 22
cond_swap() (*Compiler.types.cint method*), 24
cond_swap() (*Compiler.types.regint method*), 28
cond_swap() (*Compiler.types.sgf2n method*), 37
cond_swap() (*Compiler.types.sint method*), 41
conv2ds (*class in Compiler.instructions*), 81
convcbit (*class in Compiler.GC.instructions*), 109
convcbit2s (*class in Compiler.GC.instructions*), 109
convcbitvec (*class in Compiler.GC.instructions*), 109
convcint (*class in Compiler.GC.instructions*), 110
convcintvec (*class in Compiler.GC.instructions*), 110
convint (*class in Compiler.instructions*), 82
convmodp (*class in Compiler.instructions*), 82
convsint (*class in Compiler.GC.instructions*), 110
cos() (*in module Compiler.mpc_math*), 59
crash (*class in Compiler.instructions*), 82
crash() (*in module Compiler.library*), 53
create_from() (*Compiler.types.Array class method*), 7
CryptoPlayer (*C++ class*), 130
CryptoPlayer::CryptoPlayer (*C++ function*), 131
CryptoPlayer::partial_broadcast (*C++ function*), 131

D

dabit (*class in Compiler.instructions*), 82
Dense (*class in Compiler.ml*), 62
diag() (*Compiler.types.Matrix method*), 9
diag() (*Compiler.types.MultiArray method*), 15
digest() (*Compiler.types.cint method*), 24
digestc (*class in Compiler.instructions*), 82
direct (*Compiler.instructions.gldmci attribute*), 85
direct (*Compiler.instructions.gldmsi attribute*), 85
direct (*Compiler.instructions.gstmci attribute*), 85
direct (*Compiler.instructions.gstmsi attribute*), 85
direct (*Compiler.instructions.ldmci attribute*), 89
direct (*Compiler.instructions.ldminti attribute*), 90
direct (*Compiler.instructions.ldmsi attribute*), 90
direct (*Compiler.instructions.stmci attribute*), 101
direct (*Compiler.instructions.stminti attribute*), 102
direct (*Compiler.instructions.stmsi attribute*), 102
direct_mul() (*Compiler.types.Matrix method*), 9
direct_mul() (*Compiler.types.MultiArray method*), 15
direct_mul_to_matrix() (*Compiler.types.Matrix method*), 10
direct_mul_to_matrix() (*Compiler.types.MultiArray method*), 15
direct_mul_trans() (*Compiler.types.Matrix method*), 10
direct_mul_trans() (*Compiler.types.MultiArray method*), 16
direct_trans_mul() (*Compiler.types.Matrix method*), 10
direct_trans_mul() (*Compiler.types.MultiArray method*), 16
divc (*class in Compiler.instructions*), 82
divci (*class in Compiler.instructions*), 83
divint (*class in Compiler.instructions*), 83
do_while() (*in module Compiler.library*), 53
dot() (*Compiler.types.Matrix method*), 11
dot() (*Compiler.types.MultiArray method*), 16
dot_product() (*Compiler.types.sfix class method*), 31
dot_product() (*Compiler.types.sgf2n class method*), 37
dot_product() (*Compiler.types.sint class method*), 41
dotprods (*class in Compiler.instructions*), 83

Dropout (*class in Compiler.ml*), 62

E

edabit (*class in Compiler.instructions*), 84
eqc (*class in Compiler.instructions*), 84
equal() (*Compiler.types.sgf2n method*), 37
equal() (*Compiler.types.sint method*), 41
eqzc (*class in Compiler.instructions*), 84
eval() (*Compiler.ml.Optimizer method*), 64
exp2_fx() (*in module Compiler.mpc_math*), 59
expand_to_vector() (*Compiler.types.Array method*), 7

F

f() (*Compiler.ml.Relu static method*), 65
f_prime() (*Compiler.ml.Relu static method*), 65
FixAveragePool2d (*class in Compiler.ml*), 62
FixConv2d (*class in Compiler.ml*), 63
floatoutput (*class in Compiler.instructions*), 84
floordivc (*class in Compiler.instructions*), 84
for_range() (*in module Compiler.library*), 53
for_range_multithread() (*in module Compiler.library*), 54
for_range_opt() (*in module Compiler.library*), 54
for_range_opt_multithread() (*in module Compiler.library*), 54
for_range_parallel() (*in module Compiler.library*), 55
foreach_enumerate() (*in module Compiler.library*), 55
forward() (*Compiler.ml.Optimizer method*), 64
FusedBatchNorm (*class in Compiler.ml*), 63

G

gbitcom (*class in Compiler.instructions*), 85
gbitdec (*class in Compiler.instructions*), 85
gbitgf2ntriple (*class in Compiler.instructions*), 85
gbittriple (*class in Compiler.instructions*), 85
gconvgf2n (*class in Compiler.instructions*), 85
get() (*Compiler.types.Array method*), 7
get_arg() (*in module Compiler.library*), 55
get_dabit() (*Compiler.types.sint class method*), 42
get_def() (*Compiler.instructions.dotprods method*), 83
get_def() (*Compiler.instructions.mulrs method*), 94
get_edabit() (*Compiler.types.sint class method*), 42
get_input_from() (*Compiler.GC.types.sbitfix class method*), 49
get_input_from() (*Compiler.GC.types.sbitfixvec class method*), 49
get_input_from() (*Compiler.GC.types.sbits class method*), 51
get_input_from() (*Compiler.types.sfix class method*), 31
get_input_from() (*Compiler.types.sfloat class method*), 34
get_input_from() (*Compiler.types.sgf2n class method*), 37
get_input_from() (*Compiler.types.sint class method*), 42
get_number_of_players() (*in module Compiler.library*), 55
get_part() (*Compiler.types.Array method*), 7
get_part() (*Compiler.types.Matrix method*), 11
get_part() (*Compiler.types.MultiArray method*), 16
get_part_vector() (*Compiler.types.Array method*), 7
get_part_vector() (*Compiler.types.Matrix method*), 11
get_part_vector() (*Compiler.types.MultiArray method*), 16
get_player_id() (*in module Compiler.library*), 55
get_random() (*Compiler.types.regint class method*), 28
get_random() (*Compiler.types.sfix class method*), 31
get_random() (*Compiler.types.sint class method*), 42
get_random_bit() (*Compiler.types.sgf2n class method*), 37
get_random_bit() (*Compiler.types.sint class method*), 42
get_random_int() (*Compiler.types.sint class method*), 42
get_random_inverse() (*Compiler.types.sgf2n class method*), 37
get_random_inverse() (*Compiler.types.sint class method*), 42
get_random_square() (*Compiler.types.sgf2n class method*), 37
get_random_square() (*Compiler.types.sint class method*), 42
get_random_triple() (*Compiler.types.sgf2n class method*), 38
get_random_triple() (*Compiler.types.sint class method*), 43
get_slice_vector() (*Compiler.types.Matrix method*), 11
get_slice_vector() (*Compiler.types.MultiArray method*), 16
get_thread_number() (*in module Compiler.library*), 55
get_threshold() (*in module Compiler.library*), 56
get_type() (*Compiler.GC.types.bits class method*), 48
get_type() (*Compiler.GC.types.sbitint class method*), 50
get_type() (*Compiler.GC.types.sbitvec class method*), 52
get_used() (*Compiler.instructions.dotprods method*), 83
get_used() (*Compiler.instructions.mulrs method*), 94
get_vector() (*Compiler.types.Array method*), 7
get_vector() (*Compiler.types.Matrix method*), 11
get_vector() (*Compiler.types.MultiArray method*), 16

get_vector_by_indices() (*Compiler.types.Matrix method*), 11
get_vector_by_indices() (*Compiler.types.MultiArray method*), 17
gldmci (*class in Compiler.instructions*), 85
gldmsi (*class in Compiler.instructions*), 85
gmulbitc (*class in Compiler.instructions*), 85
gmulbitm (*class in Compiler.instructions*), 85
gnotc (*class in Compiler.instructions*), 85
greater_equal() (*Compiler.types.sint method*), 43
greater_than() (*Compiler.types.sint method*), 43
gstmci (*class in Compiler.instructions*), 85
gstmsi (*class in Compiler.instructions*), 85
gtc (*class in Compiler.instructions*), 85

H

half_adder() (*Compiler.types.cgf2n method*), 22
half_adder() (*Compiler.types.cint method*), 24
half_adder() (*Compiler.types.regint method*), 29
half_adder() (*Compiler.types.sgf2n method*), 38
half_adder() (*Compiler.types.sint method*), 43

I

iadd() (*Compiler.types.Matrix method*), 11
iadd() (*Compiler.types.MultiArray method*), 17
ieee_float (*class in Compiler.circuit*), 67
if_() (*in module Compiler.library*), 56
if_e() (*in module Compiler.library*), 56
if_else() (*Compiler.GC.types.bits method*), 48
if_else() (*Compiler.GC.types.sbit method*), 48
if_else() (*Compiler.types.cgf2n method*), 22
if_else() (*Compiler.types.cint method*), 25
if_else() (*Compiler.types.regint method*), 29
if_else() (*Compiler.types.sgf2n method*), 38
if_else() (*Compiler.types.sint method*), 43
inc() (*Compiler.types.regint class method*), 29
incint (*class in Compiler.instructions*), 86
input_from() (*Compiler.types.Array method*), 7
input_from() (*Compiler.types.Matrix method*), 11
input_from() (*Compiler.types.MultiArray method*), 17
input_tensor_from() (*Compiler.types.sfix class method*), 32
input_tensor_from() (*Compiler.types.sfloat class method*), 34
input_tensor_from() (*Compiler.types.sgf2n class method*), 38
input_tensor_from() (*Compiler.types.sint class method*), 43
input_tensor_from_client() (*Compiler.types.sfix class method*), 32
input_tensor_from_client() (*Compiler.types.sfloat class method*), 35
input_tensor_from_client() (*Compiler.types.sgf2n class method*), 38
input_tensor_from_client() (*Compiler.types.sint class method*), 44
input_tensor_via() (*Compiler.types.sfix class method*), 32
input_tensor_via() (*Compiler.types.sfloat class method*), 35
input_tensor_via() (*Compiler.types.sgf2n class method*), 38
input_tensor_via() (*Compiler.types.sint class method*), 44
inputb (*class in Compiler.GC.instructions*), 110
InputBase (*C++ class*), 122
InputBase::add_from_all (*C++ function*), 122
InputBase::add_mine (*C++ function*), 122
InputBase::add_other (*C++ function*), 122
InputBase::exchange (*C++ function*), 123
InputBase::finalize (*C++ function*), 123
InputBase::finalize_mine (*C++ function*), 123
InputBase::finalize_other (*C++ function*), 123
InputBase::reset (*C++ function*), 122
InputBase::reset_all (*C++ function*), 122
InputBase::send_mine (*C++ function*), 123
inputbvec (*class in Compiler.GC.instructions*), 111
inputfix (*class in Compiler.instructions*), 86
inputfloat (*class in Compiler.instructions*), 86
inputmaskreg (*class in Compiler.instructions*), 86
inputmixed (*class in Compiler.instructions*), 86
inputmixedreg (*class in Compiler.instructions*), 87
inputpersonal (*class in Compiler.instructions*), 87
int_div() (*Compiler.types.sint method*), 44
intoutput (*class in Compiler.instructions*), 88
inv2m (*class in Compiler.instructions*), 88
inverse (*class in Compiler.instructions*), 88
InvertSqrt() (*in module Compiler.mpc_math*), 59

J

jmp (*class in Compiler.instructions*), 88
jmpeqz (*class in Compiler.instructions*), 88
jmpi (*class in Compiler.instructions*), 88
jmpnz (*class in Compiler.instructions*), 89
join_tape (*class in Compiler.instructions*), 89
join_tapes() (*Compiler.program.Program method*), 68

L

layers (*Compiler.ml.Optimizer property*), 64
ldarg (*class in Compiler.instructions*), 89
ldbits (*class in Compiler.GC.instructions*), 111
ldi (*class in Compiler.instructions*), 89
ldint (*class in Compiler.instructions*), 89
ldmc (*class in Compiler.instructions*), 89
ldmcb (*class in Compiler.GC.instructions*), 111
ldmcbi (*class in Compiler.GC.instructions*), 111
ldmci (*class in Compiler.instructions*), 89
ldmint (*class in Compiler.instructions*), 90

ldminti (*class in Compiler.instructions*), 90
ldms (*class in Compiler.instructions*), 90
ldmsb (*class in Compiler.GC.instructions*), 111
ldmsbi (*class in Compiler.GC.instructions*), 112
ldmsi (*class in Compiler.instructions*), 90
ldsi (*class in Compiler.instructions*), 90
ldtn (*class in Compiler.instructions*), 90
left_shift() (*Compiler.types.sint method*), 44
legendre() (*Compiler.types.cint method*), 25
legendrec (*class in Compiler.instructions*), 91
less_equal() (*Compiler.types.sint method*), 44
less_than() (*Compiler.types.cint method*), 25
less_than() (*Compiler.types.sint method*), 44
listen (*class in Compiler.instructions*), 91
listen_for_clients() (*in module Compiler.library*), 56
load_mem() (*Compiler.types.cfix class method*), 19
load_mem() (*Compiler.types.cgf2n class method*), 22
load_mem() (*Compiler.types.cint class method*), 25
load_mem() (*Compiler.types.regint class method*), 29
load_mem() (*Compiler.types.sfix class method*), 32
load_mem() (*Compiler.types.sfloat class method*), 35
load_mem() (*Compiler.types.sgf2n class method*), 38
load_mem() (*Compiler.types.sint class method*), 45
localint (*class in Compiler.types*), 26
log2_fx() (*in module Compiler.mpc_math*), 59
log_fx() (*in module Compiler.mpc_math*), 60
ltc (*class in Compiler.instructions*), 91
ltzc (*class in Compiler.instructions*), 91

M

MAC_Check_Base (*C++ class*), 123
MAC_Check_Base::Check (*C++ function*), 123
MAC_Check_Base::CheckFor (*C++ function*), 123
MAC_Check_Base::exchange (*C++ function*), 123
MAC_Check_Base::finalize_open (*C++ function*), 123
MAC_Check_Base::get_alphai (*C++ function*), 123
MAC_Check_Base::init_open (*C++ function*), 123
MAC_Check_Base::open (*C++ function*), 123
MAC_Check_Base::POpen (*C++ function*), 123
MAC_Check_Base::prepare_open (*C++ function*), 123
malloc() (*Compiler.types.cgf2n class method*), 22
malloc() (*Compiler.types.cint class method*), 25
malloc() (*Compiler.types.regint class method*), 29
malloc() (*Compiler.types.sgf2n class method*), 38
malloc() (*Compiler.types.sint class method*), 45
map_sum_opt() (*in module Compiler.library*), 56
matmuls (*class in Compiler.instructions*), 91
matmulsm (*class in Compiler.instructions*), 91
Matrix (*class in Compiler.types*), 9
Matrix() (*Compiler.types.cfix class method*), 19
Matrix() (*Compiler.types.cgf2n class method*), 21
Matrix() (*Compiler.types.cint class method*), 23
Matrix() (*Compiler.types.regint class method*), 27
Matrix() (*Compiler.types.sfix class method*), 31
Matrix() (*Compiler.types.sfloat class method*), 34
Matrix() (*Compiler.types.sgf2n class method*), 36
Matrix() (*Compiler.types.sint class method*), 40
max() (*Compiler.types.cfix method*), 19
max() (*Compiler.types.cgf2n method*), 22
max() (*Compiler.types.cint method*), 25
max() (*Compiler.types.MemValue method*), 14
max() (*Compiler.types.regint method*), 29
max() (*Compiler.types.sfix method*), 32
max() (*Compiler.types.sfloat method*), 35
max() (*Compiler.types.sgf2n method*), 39
max() (*Compiler.types.sint method*), 45
MaxPool (*class in Compiler.ml*), 63
maybe_get() (*Compiler.types.Array method*), 7
maybe_set() (*Compiler.types.Array method*), 8
MemValue (*class in Compiler.types*), 13
MemValue() (*Compiler.types.cfix class method*), 19
MemValue() (*Compiler.types.cgf2n class method*), 21
MemValue() (*Compiler.types.cint class method*), 23
MemValue() (*Compiler.types.regint class method*), 27
MemValue() (*Compiler.types.sfix class method*), 31
MemValue() (*Compiler.types.sfloat class method*), 34
MemValue() (*Compiler.types.sgf2n class method*), 36
MemValue() (*Compiler.types.sint class method*), 40
min() (*Compiler.types.cfix method*), 19
min() (*Compiler.types.cgf2n method*), 22
min() (*Compiler.types.cint method*), 25
min() (*Compiler.types.MemValue method*), 14
min() (*Compiler.types.regint method*), 29
min() (*Compiler.types.sfix method*), 32
min() (*Compiler.types.sfloat method*), 35
min() (*Compiler.types.sgf2n method*), 39
min() (*Compiler.types.sint method*), 45
MixedProtocolSet (*C++ class*), 121
MixedProtocolSet::MixedProtocolSet (*C++ function*), 121
MixedProtocolSetup (*C++ class*), 121
MixedProtocolSetup::MixedProtocolSetup (*C++ function*), 121
mod2m() (*Compiler.types.cint method*), 25
mod2m() (*Compiler.types.regint method*), 30
mod2m() (*Compiler.types.sint method*), 45
modc (*class in Compiler.instructions*), 92
modci (*class in Compiler.instructions*), 92
module
    Compiler.circuit, 67
    Compiler.GC.instructions, 107
    Compiler.GC.types, 48
    Compiler.instructions, 76
    Compiler.library, 53
    Compiler.ml, 61
    Compiler.mpc_math, 59

Compiler.oram, 70
Compiler.program, 68
Compiler.types, 5
movc (*class in Compiler.instructions*), 92
movint (*class in Compiler.instructions*), 93
movs (*class in Compiler.instructions*), 93
movsb (*class in Compiler.GC.instructions*), 112
mr() (*in module Compiler.ml*), 66
mul_trans() (*Compiler.types.Matrix method*), 11
mul_trans() (*Compiler.types.MultiArray method*), 17
mul_trans_to() (*Compiler.types.Matrix method*), 12
mul_trans_to() (*Compiler.types.MultiArray method*), 17
mulc (*class in Compiler.instructions*), 93
mulcbi (*class in Compiler.GC.instructions*), 112
mulci (*class in Compiler.instructions*), 93
mulint (*class in Compiler.instructions*), 93
mulm (*class in Compiler.instructions*), 94
mulrs (*class in Compiler.instructions*), 94
muls (*class in Compiler.instructions*), 94
mulsi (*class in Compiler.instructions*), 94
MultiArray (*class in Compiler.types*), 14
MultiOutput (*class in Compiler.ml*), 63
MultiPlayer (*C++ class*), 130
multithread() (*in module Compiler.library*), 57

N

Names (*C++ class*), 127
Names::~Names (*C++ function*), 128
Names::DEFAULT_PORT (*C++ member*), 128
Names::get_name (*C++ function*), 128
Names::get_portnum_base (*C++ function*), 128
Names::init (*C++ function*), 127
Names::my_num (*C++ function*), 128
Names::Names (*C++ function*), 127, 128
Names::num_players (*C++ function*), 128
new_tape() (*Compiler.program.Program method*), 68
not_equal() (*Compiler.types.sgf2n method*), 39
not_equal() (*Compiler.types.sint method*), 45
notc (*class in Compiler.instructions*), 95
notcb (*class in Compiler.GC.instructions*), 112
nots (*class in Compiler.GC.instructions*), 112
nplayers (*class in Compiler.instructions*), 95

O

octetStream (*C++ class*), 131
octetStream::append (*C++ function*), 132
octetStream::append_random (*C++ function*), 132
octetStream::clear (*C++ function*), 131
octetStream::concat (*C++ function*), 132
octetStream::consume (*C++ function*), 132, 133
octetStream::done (*C++ function*), 131
octetStream::empty (*C++ function*), 132
octetStream::exchange (*C++ function*), 134
octetStream::get (*C++ function*), 132, 133
octetStream::get_data (*C++ function*), 131
octetStream::get_data_ptr (*C++ function*), 131
octetStream::get_int (*C++ function*), 132, 133
octetStream::get_length (*C++ function*), 131
octetStream::get_max_length (*C++ function*), 131
octetStream::get_no_resize (*C++ function*), 133
octetStream::get_ptr (*C++ function*), 131
octetStream::hash (*C++ function*), 132
octetStream::input (*C++ function*), 133
octetStream::left (*C++ function*), 132
octetStream::octetStream (*C++ function*), 131
octetStream::operator== (*C++ function*), 132
octetStream::output (*C++ function*), 134
octetStream::Receive (*C++ function*), 133
octetStream::reset_read_head (*C++ function*), 132
octetStream::reset_write_head (*C++ function*), 132
octetStream::resize (*C++ function*), 131
octetStream::Send (*C++ function*), 133
octetStream::store (*C++ function*), 132, 133
octetStream::store_int (*C++ function*), 132, 133
octetStream::str (*C++ function*), 132
op() (*Compiler.instructions.addint method*), 76
op() (*Compiler.instructions.divint method*), 83
op() (*Compiler.instructions.eqc method*), 84
op() (*Compiler.instructions.gtc method*), 86
op() (*Compiler.instructions.ltc method*), 91
op() (*Compiler.instructions.mulint method*), 93
op() (*Compiler.instructions.subint method*), 103
OptimalORAM() (*in module Compiler.oram*), 70
Optimizer (*class in Compiler.ml*), 64
options_from_args() (*Compiler.program.Program method*), 68
orc (*class in Compiler.instructions*), 95
orci (*class in Compiler.instructions*), 95
Output (*class in Compiler.ml*), 64
output() (*Compiler.types.localint method*), 27

P

plain_mul() (*Compiler.types.Matrix method*), 12
plain_mul() (*Compiler.types.MultiArray method*), 17
PlainPlayer (*C++ class*), 130
PlainPlayer::PlainPlayer (*C++ function*), 130
Player (*C++ class*), 128
Player::Broadcast_Receive (*C++ function*), 129
Player::Check_Broadcast (*C++ function*), 129
Player::exchange (*C++ function*), 129
Player::exchange_relative (*C++ function*), 129
Player::my_num (*C++ function*), 128
Player::num_players (*C++ function*), 128
Player::partial_broadcast (*C++ function*), 130
Player::pass_around (*C++ function*), 129
Player::receive_all (*C++ function*), 128

Player::receive_player (*C++ function*), 129
Player::receive_relative (*C++ function*), 129
Player::send_all (*C++ function*), 128
Player::send_receive_all (*C++ function*), 129
Player::send_relative (*C++ function*), 129
Player::send_to (*C++ function*), 128
Player::unchecked_broadcast (*C++ function*), 129
playerid (*class in Compiler.instructions*), 95
pop() (*Compiler.types.regint class method*), 30
popcnt() (*Compiler.GC.types.sbits method*), 51
popcnt() (*Compiler.GC.types.sbitvec method*), 52
popint (*class in Compiler.instructions*), 95
pow2() (*Compiler.GC.types.sbitint method*), 50
pow2() (*Compiler.GC.types.sbitintvec method*), 51
pow2() (*Compiler.types.sint method*), 45
pow_fx() (*in module Compiler.mpc_math*), 60
prep (*class in Compiler.instructions*), 96
Preprocessing (*C++ class*), 123
Preprocessing::get_bit (*C++ function*), 124
Preprocessing::get_dabit (*C++ function*), 124
Preprocessing::get_edabitvec (*C++ function*), 124
Preprocessing::get_random (*C++ function*), 124
Preprocessing::get_triple (*C++ function*), 124
prime_type (*Compiler.ml.Relu attribute*), 65
prime_type (*Compiler.ml.Square attribute*), 65
print_char (*class in Compiler.instructions*), 96
print_char4 (*class in Compiler.instructions*), 96
print_float_plain (*class in Compiler.instructions*), 96
print_float_plain() (*Compiler.types.cfloat method*), 20
print_float_plainb (*class in Compiler.GC.instructions*), 112
print_float_prec (*class in Compiler.instructions*), 96
print_float_precision() (*in module Compiler.library*), 57
print_if() (*Compiler.types.cint method*), 26
print_if() (*Compiler.types.regint method*), 30
print_int (*class in Compiler.instructions*), 96
print_ln() (*in module Compiler.library*), 57
print_ln_if() (*in module Compiler.library*), 57
print_ln_to() (*in module Compiler.library*), 57
print_plain() (*Compiler.types.cfix method*), 19
print_reg (*class in Compiler.instructions*), 96
print_reg_plain (*class in Compiler.instructions*), 97
print_reg_plain() (*Compiler.types.cgf2n method*), 22
print_reg_plain() (*Compiler.types.cint method*), 26
print_reg_plain() (*Compiler.types.regint method*), 30
print_reg_plainb (*class in Compiler.GC.instructions*), 113
print_reg_signed (*class in Compiler.GC.instructions*), 113
print_regb (*class in Compiler.GC.instructions*), 113
print_reveal_nested() (*Compiler.types.Array method*), 8
print_reveal_nested() (*Compiler.types.Matrix method*), 12
print_reveal_nested() (*Compiler.types.MultiArray method*), 17
print_str() (*in module Compiler.library*), 58
print_str_if() (*in module Compiler.library*), 58
private_division() (*Compiler.types.sint method*), 45
Program (*class in Compiler.program*), 68
ProtocolBase (*C++ class*), 121
ProtocolBase::exchange (*C++ function*), 122
ProtocolBase::finalize_dotprod (*C++ function*), 122
ProtocolBase::finalize_mul (*C++ function*), 122
ProtocolBase::finalize_mult (*C++ function*), 122
ProtocolBase::init (*C++ function*), 122
ProtocolBase::init_dotprod (*C++ function*), 122
ProtocolBase::init_mul (*C++ function*), 122
ProtocolBase::mul (*C++ function*), 122
ProtocolBase::next_dotprod (*C++ function*), 122
ProtocolBase::prepare_dotprod (*C++ function*), 122
ProtocolBase::prepare_mul (*C++ function*), 122
ProtocolSet (*C++ class*), 120
ProtocolSet::ProtocolSet (*C++ function*), 120
ProtocolSetup (*C++ class*), 120
ProtocolSetup::ProtocolSetup (*C++ function*), 120
pubinput (*class in Compiler.instructions*), 97
public_input() (*Compiler.program.Program method*), 69
public_input() (*in module Compiler.library*), 58
push() (*Compiler.types.regint class method*), 30
pushint (*class in Compiler.instructions*), 97

R

rand (*class in Compiler.instructions*), 97
randomfulls (*class in Compiler.instructions*), 97
randoms (*class in Compiler.instructions*), 97
rawinput (*class in Compiler.instructions*), 97
read() (*Compiler.types.MemValue method*), 14
read_from_file() (*Compiler.types.Array method*), 8
read_from_file() (*Compiler.types.Matrix method*), 12
read_from_file() (*Compiler.types.MultiArray method*), 17
read_from_file() (*Compiler.types.sfix class method*), 32
read_from_file() (*Compiler.types.sint class method*), 45
read_from_socket() (*Compiler.types.cfix class method*), 20
read_from_socket() (*Compiler.types.cint class method*), 26

read_from_socket() (*Compiler.types.regint class method*), 30
read_from_socket() (*Compiler.types.sint class method*), 46
readsharesfromfile (*class in Compiler.instructions*), 98
readsocketc (*class in Compiler.instructions*), 98
readsocketint (*class in Compiler.instructions*), 98
readsockets (*class in Compiler.instructions*), 98
receive_from_client() (*Compiler.types.sfix class method*), 32
receive_from_client() (*Compiler.types.sint class method*), 46
regint (*class in Compiler.types*), 27
Relu (*class in Compiler.ml*), 64
relu() (*in module Compiler.ml*), 66
relu_prime() (*in module Compiler.ml*), 66
ReluMultiOutput (*class in Compiler.ml*), 65
reqbl (*class in Compiler.instructions*), 99
reset() (*Compiler.ml.Optimizer method*), 64
reset() (*Compiler.ml.SGD method*), 65
reveal (*class in Compiler.GC.instructions*), 113
reveal() (*Compiler.types.Array method*), 8
reveal() (*Compiler.types.cgf2n method*), 22
reveal() (*Compiler.types.cint method*), 26
reveal() (*Compiler.types.MemValue method*), 14
reveal() (*Compiler.types.regint method*), 30
reveal() (*Compiler.types.sfix method*), 33
reveal() (*Compiler.types.sfloat method*), 35
reveal() (*Compiler.types.sgf2n method*), 39
reveal() (*Compiler.types.sint method*), 46
reveal_list() (*Compiler.types.Array method*), 8
reveal_list() (*Compiler.types.Matrix method*), 12
reveal_list() (*Compiler.types.MultiArray method*), 18
reveal_nested() (*Compiler.types.Array method*), 8
reveal_nested() (*Compiler.types.Matrix method*), 12
reveal_nested() (*Compiler.types.MultiArray method*), 18
reveal_to() (*Compiler.types.sfix method*), 33
reveal_to() (*Compiler.types.sgf2n method*), 39
reveal_to() (*Compiler.types.sint method*), 46
reveal_to_binary_output() (*Compiler.types.Array method*), 8
reveal_to_binary_output() (*Compiler.types.Matrix method*), 12
reveal_to_binary_output() (*Compiler.types.MultiArray method*), 18
reveal_to_clients() (*Compiler.types.Array method*), 8
reveal_to_clients() (*Compiler.types.Matrix method*), 12
reveal_to_clients() (*Compiler.types.MultiArray method*), 18
reveal_to_clients() (*Compiler.types.sfix class method*), 33
reveal_to_clients() (*Compiler.types.sint class method*), 46
right_shift() (*Compiler.types.cint method*), 26
right_shift() (*Compiler.types.sgf2n method*), 39
right_shift() (*Compiler.types.sint method*), 47
round() (*Compiler.types.sint method*), 47
round_nearest (*Compiler.types.sfix attribute*), 33
round_to_int() (*Compiler.types.sfloat method*), 35
run() (*Compiler.ml.Optimizer method*), 64
run_tape (*class in Compiler.instructions*), 99
run_tapes() (*Compiler.program.Program method*), 69
runtime_error() (*in module Compiler.library*), 58
runtime_error_if() (*in module Compiler.library*), 58

S

same_shape() (*Compiler.types.Array method*), 8
same_shape() (*Compiler.types.Matrix method*), 12
same_shape() (*Compiler.types.MultiArray method*), 18
sbit (*class in Compiler.GC.types*), 48
sbitfix (*class in Compiler.GC.types*), 48
sbitfixvec (*class in Compiler.GC.types*), 49
sbitint (*class in Compiler.GC.types*), 50
sbitintvec (*class in Compiler.GC.types*), 50
sbits (*class in Compiler.GC.types*), 51
sbitvec (*class in Compiler.GC.types*), 51
schur() (*Compiler.types.Matrix method*), 13
schur() (*Compiler.types.MultiArray method*), 18
security (*Compiler.program.Program property*), 69
sedabit (*class in Compiler.instructions*), 99
set_bit_length() (*Compiler.program.Program method*), 69
set_column() (*Compiler.types.Matrix method*), 13
set_layers_with_inputs() (*Compiler.ml.Optimizer method*), 64
set_precision() (*Compiler.GC.types.sbitfix class method*), 49
set_precision() (*Compiler.GC.types.sbitfixvec class method*), 49
set_precision() (*Compiler.types.cfix class method*), 20
set_precision() (*Compiler.types.sfix class method*), 33
sfix (*class in Compiler.types*), 31
sfloat (*class in Compiler.types*), 34
SGD (*class in Compiler.ml*), 65
sgf2n (*class in Compiler.types*), 35
sha3_256() (*in module Compiler.circuit*), 67
shlc (*class in Compiler.instructions*), 99
shlcbi (*class in Compiler.GC.instructions*), 113
shlci (*class in Compiler.instructions*), 99
shrc (*class in Compiler.instructions*), 100
shrcbi (*class in Compiler.GC.instructions*), 114

shrci (*class in Compiler.instructions*), 100
shrsi (*class in Compiler.instructions*), 100
shuffle (*class in Compiler.instructions*), 100
shuffle() (*Compiler.types.Array method*), 8
shuffle() (*Compiler.types.regint method*), 30
sigmoid() (*in module Compiler.ml*), 66
sigmoid_prime() (*in module Compiler.ml*), 66
sin() (*in module Compiler.mpc_math*), 60
sint (*class in Compiler.types*), 39
sintbit (*class in Compiler.types*), 47
softmax() (*in module Compiler.ml*), 66
solve_linear() (*in module Compiler.ml*), 66
sort() (*Compiler.types.Array method*), 9
split (*class in Compiler.GC.instructions*), 114
sqrt() (*in module Compiler.mpc_math*), 60
square (*class in Compiler.instructions*), 100
Square (*class in Compiler.ml*), 65
square() (*Compiler.types.cfix method*), 20
square() (*Compiler.types.cgf2n method*), 23
square() (*Compiler.types.cint method*), 26
square() (*Compiler.types.MemValue method*), 14
square() (*Compiler.types.regint method*), 30
square() (*Compiler.types.sfix method*), 33
square() (*Compiler.types.sfloat method*), 35
square() (*Compiler.types.sgf2n method*), 39
square() (*Compiler.types.sint method*), 47
starg (*class in Compiler.instructions*), 101
start (*class in Compiler.instructions*), 101
start_timer() (*in module Compiler.library*), 58
stmc (*class in Compiler.instructions*), 101
stmcb (*class in Compiler.GC.instructions*), 114
stmcbi (*class in Compiler.GC.instructions*), 114
stmci (*class in Compiler.instructions*), 101
stmint (*class in Compiler.instructions*), 101
stminti (*class in Compiler.instructions*), 101
stms (*class in Compiler.instructions*), 102
stmsb (*class in Compiler.GC.instructions*), 114
stmsbi (*class in Compiler.GC.instructions*), 115
stmsi (*class in Compiler.instructions*), 102
stop (*class in Compiler.instructions*), 102
stop_timer() (*in module Compiler.library*), 58
store_in_mem() (*Compiler.types.cfix method*), 20
store_in_mem() (*Compiler.types.cgf2n method*), 23
store_in_mem() (*Compiler.types.cint method*), 26
store_in_mem() (*Compiler.types.regint method*), 30
store_in_mem() (*Compiler.types.sfix method*), 33
store_in_mem() (*Compiler.types.sfloat method*), 35
store_in_mem() (*Compiler.types.sgf2n method*), 39
store_in_mem() (*Compiler.types.sint method*), 47
subc (*class in Compiler.instructions*), 102
subcfi (*class in Compiler.instructions*), 102
subci (*class in Compiler.instructions*), 102
subint (*class in Compiler.instructions*), 103
subml (*class in Compiler.instructions*), 103
submr (*class in Compiler.instructions*), 103
subs (*class in Compiler.instructions*), 103
subsfi (*class in Compiler.instructions*), 103
subsi (*class in Compiler.instructions*), 104

T

tan() (*in module Compiler.mpc_math*), 60
tanh() (*in module Compiler.mpc_math*), 60
Tensor() (*Compiler.types.cfix class method*), 19
Tensor() (*Compiler.types.cgf2n class method*), 21
Tensor() (*Compiler.types.cint class method*), 23
Tensor() (*Compiler.types.regint class method*), 27
Tensor() (*Compiler.types.sfix class method*), 31
Tensor() (*Compiler.types.sfloat class method*), 34
Tensor() (*Compiler.types.sgf2n class method*), 36
Tensor() (*Compiler.types.sint class method*), 40
threshold (*class in Compiler.instructions*), 104
time (*class in Compiler.instructions*), 104
to_regint() (*Compiler.types.cint method*), 26
trace() (*Compiler.types.Matrix method*), 13
trace() (*Compiler.types.MultiArray method*), 18
trans (*class in Compiler.GC.instructions*), 115
trans_mul() (*Compiler.types.Matrix method*), 13
trans_mul() (*Compiler.types.MultiArray method*), 18
trans_mul_to() (*Compiler.types.Matrix method*), 13
trans_mul_to() (*Compiler.types.MultiArray method*), 18
transpose() (*Compiler.types.Matrix method*), 13
transpose() (*Compiler.types.MultiArray method*), 19
triple (*class in Compiler.instructions*), 104
trunc_pr (*class in Compiler.instructions*), 104

U

use (*class in Compiler.instructions*), 104
use_dabit (*Compiler.program.Program attribute*), 69
use_edabit (*class in Compiler.instructions*), 105
use_edabit() (*Compiler.program.Program method*), 69
use_inp (*class in Compiler.instructions*), 105
use_matmul (*class in Compiler.instructions*), 105
use_prep (*class in Compiler.instructions*), 105
use_split() (*Compiler.program.Program method*), 69
use_square() (*Compiler.program.Program method*), 69
use_trunc_pr (*Compiler.program.Program property*), 69

V

var() (*in module Compiler.ml*), 66

W

while_do() (*in module Compiler.library*), 58
write() (*Compiler.types.MemValue method*), 14
write_shares_to_socket() (*Compiler.types.sfix class method*), 33

write_shares_to_socket() (*Compiler.types.sint class method*), 47
write_to_file() (*Compiler.types.Array method*), 9
write_to_file() (*Compiler.types.Matrix method*), 13
write_to_file() (*Compiler.types.MultiArray method*), 19
write_to_file() (*Compiler.types.sfix class method*), 34
write_to_file() (*Compiler.types.sint static method*), 47
write_to_socket() (*Compiler.types.cfix class method*), 20
write_to_socket() (*Compiler.types.cint class method*), 26
write_to_socket() (*Compiler.types.regint class method*), 30
write_to_socket() (*Compiler.types.sint class method*), 47
writesharestofile (*class in Compiler.instructions*), 105
writesockets (*class in Compiler.instructions*), 106
writesocketshare (*class in Compiler.instructions*), 106

X

xorc (*class in Compiler.instructions*), 106
xorcb (*class in Compiler.GC.instructions*), 115
xorcbi (*class in Compiler.GC.instructions*), 115
xorci (*class in Compiler.instructions*), 106
xorm (*class in Compiler.GC.instructions*), 115
xors (*class in Compiler.GC.instructions*), 116