MP-SPDZ Documentation

Marcel Keller

May 17, 2021

Contents:

This documentation provides a reference to the most important high-level functionality provided by the MP-SPDZ compiler. For a tutorial and documentation on how to run programs, the implemented protocols etc. see https://github.com/data61/MP-SPDZ.

CHAPTER 1

Compilation process

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.

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 2

Reference

2.1 High-Level Interface

2.1.1 Compiler.types module

This module defines all types availabe 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

Basic types contain many special methods such as `__add__()`. This is used for operator overloading in Python. It is not recommend to use them, use the plain operators instead, such as + instead of `__add__()`. See https://docs.python.org/3/reference/datamodel.html#special-method-names for a translation to operators.

In some operations such as secure comparison, the secure computation protocols allows for more parameters than just the operands which influence the performance. In this case, we provide an alias for better code readability. For example, *sint.greater_than()* is an alias of *sint.__gt__()*. When using operator overloading, the parameters default to the globally defined ones.

Methods of basic types generally return instances of the respective type.

Note that the data model of Python operates with reverse operators such as `__radd__()`. This means that if for the usual operator of the first operand does not support the second operand, the reverse operator of the second operand is used. For example, *_clear.__sub__()* does not support secret values as second operand but *_secret.__rsub__()* does support clear values, so `cint(3) - sint(2)` will result in a secret integer of value 1.

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.
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*)

Bases: object

Array accessible by public index.

__add__(*other*)

Vector addition.

Parameters **other** – vector or container of same length and type that supports operations with type of this array

__getitem__(*index*)

Reading from array.

Parameters **index** – public (regint/cint/int/slice)

Returns array if slice is given, basic type otherwise

__init__(*length*, *value_type*, *address=None*, *debug=None*, *alloc=True*)

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

__mul__(*value*)

Vector multiplication.

Parameters **other** – vector or container of same length and type that supports operations with type of this array

__pow__(*value*)

Vector power-of computation.

Parameters **other** – compile-time integer (int)

__radd__(*other*)

Vector addition.

Parameters **other** – vector or container of same length and type that supports operations with type of this array

__rmul__(*value*)
Vector multiplication.

Parameters **other** – vector or container of same length and type that supports operations with type of this array

__setitem__(*index*, *value*)
Writing to array.

Parameters

- **index** – public (regint/cint/int)
- **value** – convertible for relevant basic type

__str__()
Return str(self).

__sub__(*other*)
Vector subtraction.

Parameters **other** – vector or container of same length and type that supports operations with type of this array

assign_all(*value*, *use_threads=True*, *conv=True*)
Assign the same value to all entries.

Parameters **value** – convertible to basic type

classmethod create_from(*l*)
Convert Python iterator to array. Basic type will be taken from first element, further elements must to be convertible to that.

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)

reveal()
Reveal the whole array.

Returns Array of relevant clear type.

reveal_list()
Reveal as list.

reveal_nested()
Reveal as list.

shuffle()
Insecure shuffle in place.

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

Matrix.

__init__(*rows*, *columns*, *value_type*, *debug=None*, *address=None*)

Parameters

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

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

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.

__init__(*value*, *address=None*)

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

read()
Read value.

Returns relevant basic type instance

reveal()
Reveal value.

Returns relevant clear type

write(*value*)
Write value.

Parameters **value** – convertible to relevant basic type

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

Multidimensional array.

__init__(*sizes*, *value_type*, *debug=None*, *address=None*, *alloc=True*)

Parameters

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

class Compiler.types.**SubMultiArray**(*sizes*, *value_type*, *address*, *index*, *debug=None*)
Bases: object

Multidimensional array functionality.

__add__(*other*)
Element-wise addition.

Parameters **other** – container of matching size and type

Returns container of same shape and type as `self`

__getitem__(*index*)
Part access.

Parameters **index** – public (regint/cint/int)

Returns *Array* if one-dimensional, *SubMultiArray* otherwise

__init__(*sizes*, *value_type*, *address*, *index*, *debug=None*)
Do not call this, use *MultiArray* instead.

__len__()
Size of top dimension.

__mul__(*other*)
Matrix-matrix and matrix-vector multiplication.

Parameters

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

__radd__(*other*)
Element-wise addition.

Parameters **other** – container of matching size and type

Returns container of same shape and type as `self`

__setitem__(*index*, *other*)
Part assignment.

Parameters

- **index** – public (regint/cint/int)
- **other** – container of matching size and type

__str__()
Return str(self).

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_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)

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)

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)

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

plain_mul(*other*, *res=None*)
Alternative matrix multiplication.

Parameters

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

reveal_list()
Reveal as list.

reveal_nested()
Reveal as nested list.

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`

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

transpose()
Matrix transpose.

Parameters **self** – two-dimensional

class Compiler.types.**_bit**
Bases: `object`

Binary functionality.

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)

bit_not()
NOT in binary circuits.

bit_xor(*other*)
XOR in binary circuits.

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

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

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)

class Compiler.types.**_clear**(**args*, ***kwargs*)
Bases: *Compiler.types._register*

Clear domain-dependent type.

__and__(*other*)
Bit-wise AND of public values.

Parameters **other** – convertible type (at least same as `self` and regint/int)

__eq__(*other*)
Equality check of public values.

Parameters **other** – convertible type (at least same as `self` and regint/int)

Returns 0/1 (regint)

__ne__(*other*)
Equality check of public values.

Parameters **other** – convertible type (at least same as `self` and regint/int)

Returns 0/1 (regint)

__or__(*other*)
Bit-wise OR of public values.

Parameters **other** – convertible type (at least same as `self` and regint/int)

__rand__(*other*)
Bit-wise AND of public values.

Parameters **other** – convertible type (at least same as `self` and regint/int)

__ror__(*other*)
Bit-wise OR of public values.

Parameters **other** – convertible type (at least same as `self` and regint/int)

__rsub__(*other*)
Subtraction of public values.

Parameters **other** – convertible type (at least same as `self` and regint/int)

__rtruediv__(*other*)
Field division of public values. Not available for computation modulo a power of two.

Parameters **other** – convertible type (at least same as `self` and regint/int)

__rxor__(*other*)
Bit-wise XOR of public values.

Parameters **other** – convertible type (at least same as `self` and regint/int)

__sub__(*other*)
Subtraction of public values.

Parameters **other** – convertible type (at least same as `self` and regint/int)

__truediv__(*other*)
Field division of public values. Not available for computation modulo a power of two.

Parameters **other** – convertible type (at least same as `self` and regint/int)

__xor__(*other*)
Bit-wise XOR of public values.

Parameters **other** – convertible type (at least same as `self` and regint/int)

add(*other*)
Addition of public values.

Parameters **other** – convertible type (at least same as `self` and regint/int)

mul(*other*)
Multiplication of public values.

Parameters **other** – convertible type (at least same as `self` and regint/int)

print_reg_plain(**args*, ***kwargs*)
Output.

reveal()
Identity.

class Compiler.types.**_fix**(***kwargs*)
Bases: *Compiler.types._single*

Secret fixed point type.

__init__(***kwargs*)

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

__neg__(**args*, ***kwargs*)
Secret fixed-point negation.

__rtruediv__(**args*, ***kwargs*)
Secret fixed-point division.

Parameters **other** – sfix/cfix/sint/cint/regint/int

__truediv__(**args*, ***kwargs*)
Secret fixed-point division.

Parameters **other** – sfix/cfix/sint/cint/regint/int

add(**args*, ***kwargs*)
Secret fixed-point addition.

Parameters **other** – sfix/cfix/sint/cint/regint/int

compute_reciprocal(**args*, ***kwargs*)
Secret fixed-point reciprocal.

classmethod from_sint(*other*, *k=None*, *f=None*)
Convert secret integer.

Parameters **other** – sint

mul(*other*)
Secret fixed-point multiplication.

Parameters **other** – sfix/cfix/sint/cint/regint/int

reveal()
Reveal secret fixed-point number.

Returns relevant clear type

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.types.**_gf2n**
Bases: *Compiler.types._bit*

GF(2^n) functionality.

bit_not()
NOT in binary circuits.

bit_xor(*other*)
XOR in 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 GF(2^n). Similar to *_int.if_else()*.

if_else(*a*, *b*)
MUX in GF(2^n) circuits. Similar to *_int.if_else()*.

class Compiler.types.**_int**
Bases: `object`

Integer functionality.

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)

bit_not()
NOT in arithmetic circuits.

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

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

class Compiler.types.**_mem**
Bases: *Compiler.types._number*

class Compiler.types.**_number**
Bases: `object`

Number functionality.

__abs__()
Absolute value.

__add__(*other*)
Optimized addition.

Parameters **other** – any compatible type

__mul__(*other*)
Optimized multiplication.

Parameters **other** – any compatible type

__pow__(**args*, ***kwargs*)
Exponentation through square-and-multiply.

Parameters **exp** – any type allowing bit decomposition

__radd__(*other*)
Optimized addition.

Parameters **other** – any compatible type

__rmul__(*other*)
Optimized multiplication.

Parameters **other** – any compatible type

max(*other*)
Maximum.

Parameters **other** – any compatible type

min(*other*)
Minimum.

Parameters **other** – any compatible type

square()
Square.

class Compiler.types.**_register**(**args*, ***kwargs*)
Bases: Compiler.program.Register, *Compiler.types._number*, *Compiler.types._structure*

static bit_compose(*bits*)
Compose value from bits.

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

classmethod malloc(*size*, *creator_tape=None*)
Allocate memory (statically).

Parameters **size** – compile-time (int)

class Compiler.types.**_secret**(***kwargs*)
Bases: *Compiler.types._register*

__rsub__(*other*)
Secret subtraction.

Parameters **other** – any compatible type

__rtruediv__(**args*, ***kwargs*)
Secret field division.

Parameters **other** – any compatible type

__sub__(*other*)
Secret subtraction.

Parameters **other** – any compatible type

__truediv__(**args*, ***kwargs*)
Secret field division.

Parameters **other** – any compatible type

add(*other*)
Secret addition.

Parameters **other** – any compatible type

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

classmethod get_input_from(**args*, ***kwargs*)
Secret input from player.

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

classmethod get_random_bit(**args*, ***kwargs*)
Secret random bit according to security model.

Returns 0/1 50-50

classmethod get_random_inverse(**args*, ***kwargs*)
Secret random inverse tuple according to security model.

Returns (a, a^{-1})

classmethod get_random_square(**args*, ***kwargs*)
Secret random square according to security model.

Returns (a, a^2)

classmethod get_random_triple(**args*, ***kwargs*)
Secret random triple according to security model.

Returns (a, b, ab)

mul(**args*, ***kwargs*)
Secret multiplication. Either both operands have the same size or one size 1 for a value-vector multiplication.

Parameters **other** – any compatible type

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()*

square(**args*, ***kwargs*)
Secret square.

class Compiler.types.**_single**
Bases: *Compiler.types._number*, *Compiler.types._structure*

Representation as single integer preserving the order

__eq__(**args*, ***kwargs*)
Comparison.

Parameters **other** – appropriate public or secret (incl. sint/cint/regint/int)

Returns 0/1

Return type same as internal representation

__ge__(**args*, ***kwargs*)
Comparison.

Parameters **other** – appropriate public or secret (incl. sint/cint/regint/int)

Returns 0/1

Return type same as internal representation

__gt__(**args*, ***kwargs*)
Comparison.

Parameters **other** – appropriate public or secret (incl. sint/cint/regint/int)

Returns 0/1

Return type same as internal representation

__le__(**args*, ***kwargs*)
Comparison.

Parameters **other** – appropriate public or secret (incl. sint/cint/regint/int)

Returns 0/1

Return type same as internal representation

__len__()
Vector length.

__lt__(**args*, ***kwargs*)
Comparison.

Parameters **other** – appropriate public or secret (incl. sint/cint/regint/int)

Returns 0/1

Return type same as internal representation

__ne__(**args*, ***kwargs*)
Comparison.

Parameters **other** – appropriate public or secret (incl. sint/cint/regint/int)

Returns 0/1

Return type same as internal representation

__rsub__(*other*)
Subtraction.

Parameters **other** – appropriate public or secret (incl. sint/cint/regint/int)

__sub__(**args*, ***kwargs*)
Subtraction.

Parameters **other** – appropriate public or secret (incl. sint/cint/regint/int)

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 receive_from_client(*n*, *client_id*, *message_type=0*)
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

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

class Compiler.types.**_structure**
Bases: object

Interface for type-dependent container types.

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*)
Type-dependent memory value.

class Compiler.types.**cfix**(***kwargs*)
Bases: *Compiler.types._number*, *Compiler.types._structure*

Clear fixed-point number represented as clear integer.

__eq__(**args*, ***kwargs*)
Clear fixed-point comparison.

Parameters **other** – cfix/cint/regint/int

Returns 0/1

Return type *regint*

__ge__(**args*, ***kwargs*)
Clear fixed-point comparison.

Parameters **other** – cfix/cint/regint/int

Returns 0/1

Return type *regint*

__gt__(**args*, ***kwargs*)
Clear fixed-point comparison.

Parameters **other** – cfix/cint/regint/int

Returns 0/1

Return type *regint*

__init__(***kwargs*)

Parameters **v** – cfix/float/int

__le__(**args*, ***kwargs*)
Clear fixed-point comparison.

Parameters **other** – cfix/cint/regint/int

Returns 0/1

Return type *regint*

__lt__(**args*, ***kwargs*)
Clear fixed-point comparison.

Parameters **other** – cfix/cint/regint/int

Returns 0/1

Return type *regint*

__ne__(**args*, ***kwargs*)
Clear fixed-point comparison.

Parameters **other** – cfix/cint/regint/int

Returns 0/1

Return type *regint*

__neg__(**args*, ***kwargs*)
Clear fixed-point negation.

__rsub__(*other*)
Clear fixed-point subtraction.

Parameters **other** – cfix/cint/regint/int

__sub__(**args*, ***kwargs*)
Clear fixed-point subtraction.

Parameters **other** – cfix/cint/regint/int

__truediv__(**args*, ***kwargs*)
Clear fixed-point division.

Parameters **other** – cfix/cint/regint/int

add(**args*, ***kwargs*)
Clear fixed-point addition.

Parameters **other** – cfix/cint/regint/int

mul(**args*, ***kwargs*)
Clear fixed-point multiplication.

Parameters **other** – cfix/cint/regint/int/sint

print_plain()
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)

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)

classmethod write_to_socket(**args*, ***kwargs*)
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**(*v*, *p=None*, *z=None*, *s=None*, *nan=0*)
Bases: `object`

Helper class for printing revealed sfloats.

__init__(*v*, *p=None*, *z=None*, *s=None*, *nan=0*)
Parameters as with *sfloat* but public.

print_float_plain()
Output.

class Compiler.types.**cgf2n**(*val=None*, *size=None*)
Bases: *Compiler.types._clear*, *Compiler.types._gf2n*

Clear $\mathrm{GF}(2^n)$ value. n is 40 or 128, depending on USE_GF2N_LONG compile-time variable.

__init__(*val=None*, *size=None*)

Parameters

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

__invert__(**args*, ***kwargs*)
Clear bit-wise inversion.

__lshift__(**args*, ***kwargs*)
Left shift.

Parameters **other** – compile-time (int)

__mul__(*other*)
Clear $\mathrm{GF}(2^n)$ multiplication.

Parameters **other** – cgf2n/regint/int

__neg__()
Identity.

__rshift__(**args*, ***kwargs*)
Right shift.

Parameters **other** – compile-time (int)

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)

class Compiler.types.**cint**(*val=None*, *size=None*)
Bases: *Compiler.types._clear*, *Compiler.types._int*

Clear integer in same domain as secure computation (depends on protocol).

__abs__()
Clear absolute.

__eq__(**args*, ***kwargs*)
Clear equality test.

Parameters **other** – cint/regint/int

Returns 0/1 (regint)

__ge__(*other*)
Clear 64-bit comparison.

Parameters **other** – cint/regint/int

Returns 0/1 (regint)

__gt__(*other*)
Clear 64-bit comparison.

Parameters **other** – cint/regint/int

Returns 0/1 (regint)

__init__(*val=None*, *size=None*)

Parameters

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

__invert__(**args*, ***kwargs*)
Clear inversion using global bit length.

__le__(*other*)
Clear 64-bit comparison.

Parameters **other** – cint/regint/int

Returns 0/1 (regint)

__lshift__(*other*)
Clear left shift.

Parameters **other** – cint/regint/int

__lt__(*other*)
Clear 64-bit comparison.

Parameters **other** – cint/regint/int

Returns 0/1 (regint)

__mod__(*other*)
Clear modulo.

Parameters **other** – cint/regint/int

__neg__()
Clear negation.

__rlshift__(**args*, ***kwargs*)
Clear shift.

Parameters **other** – cint/regint/int

__rmod__(*other*)
Clear modulo.

Parameters **other** – cint/regint/int

__rpow__(*base*)
Clear power of two.

Parameters **other** – 2

__rrshift__(**args*, ***kwargs*)
Clear shift.

Parameters **other** – cint/regint/int

__rshift__(*other*)
Clear right shift.

Parameters **other** – cint/regint/int

bit_decompose(**args*, ***kwargs*)
Clear bit decomposition.

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

Returns list of cint

digest(*num_bytes*)
Clear hashing (libsodium default).

legendre()
Clear Legendre symbol computation.

less_than(*other*, *bit_length*)
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

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

classmethod read_from_socket(**args*, ***kwargs*)
Receive clear value(s) from client.

Parameters

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

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

right_shift(**args*, ***kwargs*)
Clear shift.

Parameters **other** – cint/regint/int

to_regint(**args*, ***kwargs*)
Convert to regint.

Parameters **n_bits** – bit length (int)

Returns regint

classmethod write_to_socket(**args*, ***kwargs*)
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*)
Bases: object

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

__init__(*value=None*)

Parameters **value** – initialization, convertible to regint

output()
Output.

class Compiler.types.**regint**(***kwargs*)
Bases: *Compiler.types._register*, *Compiler.types._int*

Clear 64-bit integer. Unlike *cint* this is always a 64-bit integer.

__and__(*other*)
Clear bit-wise AND.

Parameters **other** – regint/cint/int

__eq__(*other*)
Clear comparison.

Parameters **other** – regint/cint/int

Returns 0/1

__floordiv__(*other*)
Clear integer division (rounding to floor).

Parameters **other** – regint/cint/int

__ge__(*other*)
Clear comparison.

Parameters **other** – regint/cint/int

Returns 0/1

__gt__(*other*)
Clear comparison.

Parameters **other** – regint/cint/int

Returns 0/1

__init__(***kwargs*)

Parameters

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

__le__(*other*)
Clear comparison.

Parameters **other** – regint/cint/int

Returns 0/1

__lshift__(*other*)
Clear shift.

Parameters **other** – regint/cint/int

__lt__(*other*)
Clear comparison.

Parameters **other** – regint/cint/int

Returns 0/1

__mod__(*other*)
Clear modulo computation.

Parameters **other** – regint/cint/int

__ne__(*other*)
Clear comparison.

Parameters **other** – regint/cint/int

Returns 0/1

__neg__()
Clear negation.

__or__(*other*)
Clear bit-wise OR.

Parameters **other** – regint/cint/int

__rand__(*other*)
Clear bit-wise AND.

Parameters **other** – regint/cint/int

__rfloordiv__(*other*)
Clear integer division (rounding to floor).

Parameters **other** – regint/cint/int

__rlshift__(*other*)
Clear shift.

Parameters **other** – regint/cint/int

__rmod__(*other*)
Clear modulo computation.

Parameters **other** – regint/cint/int

__ror__(*other*)
Clear bit-wise OR.

Parameters **other** – regint/cint/int

__rpow__(*other*)
Clear power of two computation.

Parameters **other** – regint/cint/int

Return type *cint*

__rrshift__(*other*)
Clear shift.

Parameters **other** – regint/cint/int

__rshift__(*other*)
Clear shift.

Parameters **other** – regint/cint/int

__rsub__(*other*)
Clear subtraction.

Parameters **other** – regint/cint/int

__rtruediv__(*other*)
Clear integer division (rounding to floor).

Parameters **other** – regint/cint/int

__rxor__(*other*)
Clear bit-wise XOR.

Parameters **other** – regint/cint/int

__sub__(*other*)
Clear subtraction.

Parameters **other** – regint/cint/int

__truediv__(*other*)
Clear integer division (rounding to floor).

Parameters **other** – regint/cint/int

__xor__(*other*)
Clear bit-wise XOR.

Parameters **other** – regint/cint/int

add(*other*)
Clear addition.

Parameters **other** – regint/cint/int

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

classmethod get_random(**args*, ***kwargs*)
Public insecure randomness.

Parameters **bit_length** – number of bits (int)

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)
```

mod2m(**args*, ***kwargs*)
Clear modulo a power of two.

Return type *cint*

mul(*other*)
Clear multiplication.

Parameters **other** – regint/cint/int

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)

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

reveal()
Identity.

shuffle()
Returns insecure shuffle of vector.

classmethod write_to_socket(**args*, ***kwargs*)
Send a list of clear integers to a client.

Parameters

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

class Compiler.types.**sfix**(***kwargs*)
Bases: *Compiler.types._fix*

Secret fixed-point number represented as secret integer. This uses integer operations internally, see *sint* for security considerations.

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)

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

reveal_to(*player*)
Reveal secret value to `player`. Raw representation possibly written to `Player-Data/Private-Output-P<player>`.

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

Returns value to be used with *print_ln_to()*

class Compiler.types.**sfloat**(***kwargs*)
Bases: *Compiler.types._number*, *Compiler.types._structure*

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.

__eq__(**args*, ***kwargs*)
Secret floating-point comparison.

Parameters **other** – sfloat/float/sfix/sint/cint/regint/int

Returns 0/1 (sint)

__ge__(*other*)
Secret floating-point comparison.

Parameters **other** – sfloat/float/sfix/sint/cint/regint/int

Returns 0/1 (sint)

__gt__(*other*)
Secret floating-point comparison.

Parameters **other** – sfloat/float/sfix/sint/cint/regint/int

Returns 0/1 (sint)

__init__(***kwargs*)

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

__le__(*other*)
Secret floating-point comparison.

Parameters **other** – sfloat/float/sfix/sint/cint/regint/int

Returns 0/1 (sint)

__lt__(**args*, ***kwargs*)
Secret floating-point comparison.

Parameters **other** – sfloat/float/sfix/sint/cint/regint/int

Returns 0/1 (sint)

__ne__(*other*)
Secret floating-point comparison.

Parameters **other** – sfloat/float/sfix/sint/cint/regint/int

Returns 0/1 (sint)

__neg__(**args*, ***kwargs*)
Secret floating-point negation.

__rsub__(*other*)
Secret floating-point subtraction.

Parameters **other** – sfloat/float/sfix/sint/cint/regint/int

__rtruediv__(*other*)
Secret floating-point division.

Parameters **other** – sfloat/float/sfix/sint/cint/regint/int

__sub__(*other*)
Secret floating-point subtraction.

Parameters **other** – sfloat/float/sfix/sint/cint/regint/int

__truediv__(*other*)
Secret floating-point division.

Parameters **other** – sfloat/float/sfix/sint/cint/regint/int

add(**args*, ***kwargs*)
Secret floating-point addition.

Parameters **other** – sfloat/float/sfix/sint/cint/regint/int

classmethod get_input_from(**args*, ***kwargs*)
Secret floating-point input.

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

mul(**args*, ***kwargs*)
Secret floating-point multiplication.

Parameters **other** – sfloat/float/sfix/sint/cint/regint/int

reveal()
Reveal secret floating-point number.

Returns cfloat

round_to_int()
Secret floating-point rounding to integer.

Returns sint

class Compiler.types.**sgf2n**(*val=None*, *size=None*)
Bases: *Compiler.types._secret*, *Compiler.types._gf2n*

Secret $\mathrm{GF}(2^n)$ value.

__and__(**args*, ***kwargs*)
Secret bit-wise AND.

Parameters **other** – sg2fn/cgf2n/regint/int

__eq__(*other*, *bit_length=None*, *expand=1*)
Secret comparison.

Parameters **other** – sgf2n/cgf2n/regint/int

Returns 0/1 (sgf2n)

__init__(*val=None*, *size=None*)

Parameters

- **val** – initialization (sgf2n/cgf2n/regint/int/cint or list thereof)
- **size** – vector size (int), defaults to 1 or size of list

__invert__(**args*, ***kwargs*)
Secret bit-wise inversion.

__lshift__(**args*, ***kwargs*)
Secret left shift py public value.

Parameters **other** – regint/cint/int

__ne__(*other*, *bit_length=None*)
Secret comparison.

Parameters **other** – sgf2n/cgf2n/regint/int

Returns 0/1 (sgf2n)

__neg__()
Identity.

__rand__(**args*, ***kwargs*)
Secret bit-wise AND.

Parameters **other** – sg2fn/cgf2n/regint/int

__rxor__(*other*)
Secret bit-wise XOR.

Parameters **other** – sg2fn/cgf2n/regint/int

__xor__(*other*)
Secret bit-wise XOR.

Parameters **other** – sg2fn/cgf2n/regint/int

add(*other*)
Secret $\mathrm{GF}(2^n)$ addition (XOR).

Parameters **other** – sg2fn/cgf2n/regint/int

bit_decompose(**args*, ***kwargs*)
Secret bit decomposition.

Parameters

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

Returns list of sgf2n

equal(*other*, *bit_length=None*, *expand=1*)
Secret comparison.

Parameters **other** – sgf2n/cgf2n/regint/int

Returns 0/1 (sgf2n)

mul(*other*)
Secret $\mathrm{GF}(2^n)$ multiplication.

Parameters **other** – sg2fn/cgf2n/regint/int

not_equal(*other*, *bit_length=None*)
Secret comparison.

Parameters **other** – sgf2n/cgf2n/regint/int

Returns 0/1 (sgf2n)

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)

class Compiler.types.**sint**(*val=None*, *size=None*)
Bases: *Compiler.types._secret*, *Compiler.types._int*

Secret integer in the protocol-specific domain.

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.

__abs__(**args*, ***kwargs*)
Secret absolute. Uses global parameters for comparison.

__eq__(**args*, ***kwargs*)
Secret comparison (signed).

Parameters **other** – sint/cint/regint/int

Returns 0/1 (sint)

__ge__(*other*, *bit_length=None*, *security=None*)
Secret comparison (signed).

Parameters **other** – sint/cint/regint/int

Returns 0/1 (sint)

__gt__(**args*, ***kwargs*)
Secret comparison (signed).

Parameters **other** – sint/cint/regint/int

Returns 0/1 (sint)

__init__(*val=None*, *size=None*)

Parameters

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

__le__(*other*, *bit_length=None*, *security=None*)
Secret comparison (signed).

Parameters **other** – sint/cint/regint/int

Returns 0/1 (sint)

__lshift__(*other*, *bit_length=None*, *security=None*)
Secret left shift.

Parameters **other** – secret or public integer (sint/cint/regint/int)

__lt__(**args*, ***kwargs*)
Secret comparison (signed).

Parameters **other** – sint/cint/regint/int

Returns 0/1 (sint)

__mod__(**args*, ***kwargs*)
Secret modulo computation. Uses global parameters for bit length and security.

Parameters **modulus** – power of two (int)

__ne__(*other*, *bit_length=None*, *security=None*)
Secret comparison (signed).

Parameters **other** – sint/cint/regint/int

Returns 0/1 (sint)

__neg__(**args*, ***kwargs*)
Secret negation.

__rlshift__(*other*)
Secret left shift. Bit length of `self` uses global value.

Parameters **other** – secret or public integer (sint/cint/regint/int)

__rpow__(**args*, ***kwargs*)
Secret power computation. Base must be two. Uses global parameters for bit length and security.

__rrshift__(**args*, ***kwargs*)
Secret right shift.

Parameters **other** – secret or public integer (sint/cint/regint/int) of globale bit length if secret

__rshift__(**args*, ***kwargs*)
Secret right shift.

Parameters **other** – secret or public integer (sint/cint/regint/int)

bit_decompose(**args*, ***kwargs*)
Secret bit decomposition.

equal(**args*, ***kwargs*)
Secret comparison (signed).

Parameters **other** – sint/cint/regint/int

Returns 0/1 (sint)

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)

classmethod get_random(**args*, ***kwargs*)
Secret random ring element according to security model.

classmethod get_random_int(**args*, ***kwargs*)
Secret random n-bit number according to security model.

Parameters **bits** – compile-time integer (int)

greater_equal(*other*, *bit_length=None*, *security=None*)
Secret comparison (signed).

Parameters **other** – sint/cint/regint/int

Returns 0/1 (sint)

greater_than(**args*, ***kwargs*)
Secret comparison (signed).

Parameters **other** – sint/cint/regint/int

Returns 0/1 (sint)

int_div(**args*, ***kwargs*)
Secret integer division.

Parameters **other** – sint

left_shift(*other*, *bit_length=None*, *security=None*)
Secret left shift.

Parameters **other** – secret or public integer (sint/cint/regint/int)

less_equal(*other*, *bit_length=None*, *security=None*)
Secret comparison (signed).

Parameters **other** – sint/cint/regint/int

Returns 0/1 (sint)

less_than(**args*, ***kwargs*)
Secret comparison (signed).

Parameters **other** – sint/cint/regint/int

Returns 0/1 (sint)

mod2m(**args*, ***kwargs*)
Secret modulo power of two.

Parameters **m** – secret or public integer (sint/cint/regint/int)

not_equal(*other*, *bit_length=None*, *security=None*)
Secret comparison (signed).

Parameters **other** – sint/cint/regint/int

Returns 0/1 (sint)

pow2(**args*, ***kwargs*)
Secret power of two.

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)

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

classmethod receive_from_client(*n*, *client_id*, *message_type=0*)
Securely obtain shares of values input by a client.

Parameters

- **n** – number of inputs (int)
- **client_id** – regint

reveal_to(**args*, ***kwargs*)
Reveal secret value to `player`. Result potentially written to `Player-Data/Private-Output-P<player>`.

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

Returns value to be used with *print_ln_to()*

right_shift(**args*, ***kwargs*)
Secret right shift.

Parameters **other** – secret or public integer (sint/cint/regint/int)

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

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

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: `Compiler.program.Register`, *Compiler.types._structure*, *Compiler.types._bit*

Base class for binary registers.

classmethod get_type(*length*)
Returns a fixed-length type.

class Compiler.GC.types.**cbits**(*value=None*, *n=None*, *size=None*)
Bases: *Compiler.GC.types.bits*

Clear bits register. Helper type with limited functionality.

class Compiler.GC.types.**sbit**(**args*, ***kwargs*)
Bases: Compiler.GC.types.bit, *Compiler.GC.types.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: *Compiler.types._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
```

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: *Compiler.types._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: `Compiler.types._bitint`, *Compiler.types._number*, *Compiler.GC.types.sbits*, `Compiler.GC.types._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: *Compiler.GC.types.sbitvec*, *Compiler.types._number*, Compiler.types._bitint, Compiler.GC.types._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: *Compiler.GC.types.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*.

classmethod **get_input_from**(*player*, *n_bits=None*)
Secret input from player.

Param player (int)

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.

popcnt()
Population count / Hamming weight.

Returns *sbits* of required length

to_sint(*n_bits*)
Convert the n_bits least significant bits to *sint*.

class Compiler.GC.types.**sbitvec**(*elements=None*, *length=None*, *input_length=None*)
Bases: Compiler.types._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.**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.**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
    global x
    x += 1
```

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 optimization is rather rudimentary for runtime `n_loops` (regint/cint). Consider using *for_range_parallel()* in this case.

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):
    ...
```

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.

Parameters

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

Example:

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

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.

Parameters **condition** – regint/cint/int

Usage:

```
@if_e(x > 0)
def _():
    ...
@else_
def _():
    ...
```

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.**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 explicitely.

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.**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)

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 consective dense/fully-connected 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[1].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. 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.

See also this repository for an example of how to train a model for MNIST.

class Compiler.ml.**Add**(*inputs*)
Fixed-point addition layer.

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

class Compiler.ml.**Argmax**(*shape*)
Fixed-point Argmax layer.

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

class Compiler.ml.**Concat**(*inputs*, *dimension*)
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*)
Fixed-point dense (matrix multiplication) layer.

Parameters

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

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

class Compiler.ml.**FusedBatchNorm**(*shape*, *inputs=None*)
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'*)
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*)
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**
Base class for graphs of layers.

layers
Get all layers.

set_layers_with_inputs(*layers*)
Construct graph from `inputs` members of list of layers.

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

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

Parameters

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

class Compiler.ml.**Square**(*shape*, *inputs=None*)
Fixed-point square layer.

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

prime_type
alias of *Compiler.types.sfix*

Compiler.ml.**argmax**(*x*)
Compute index of maximum element.

Parameters **x** – iterable

Returns sint

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.**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)
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:

```
0x5375f6fb6aa989b0c287a923afe81e79ff875921cacc956666d71ebff8c6ffa7
0x17c7e0d65c285af8406d4f21c071851a312b739a8ecdf25c1270d31c39357067
```

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():
    ...
```

(continues on next page)

(continued from previous page)

```
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

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 = None
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`

use_trunc_pr = None
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	
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
USE_PREP	0x1c	Custom preprocessed data usage
ADDC	0x20	Clear addition
ADDS	0x21	Secret addition

Continued on next page

Table 3 – continued from previous page

Name	Code	
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)
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
READSOCKETINT	0x69	Read a variable number of 32-bit integers from socket for a specified client id and store them in clear integer registers

Continued on next page

Table 3 – continued from previous page

Name	Code	
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
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

Continued on next page

Table 3 – continued from previous page

Name	Code	
CONV2DS	0xac	Secret 2D convolution
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
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)
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

Continued on next page

Table 3 – continued from previous page

Name	Code	
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
CONVCINT	0x213	Copy clear integer register to clear bit register

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 (int)

Compiler.instructions.**addc**(**args*, ***kwargs*)
Clear addition.

Param result (cint)

Param summand (cint)

Param summand (cint)

Compiler.instructions.**addci**(**args*, ***kwargs*)
Addition of clear register (vector) and (constant) immediate value.

Param result (cint)

Param summand (cint)

Param summand (int)

Compiler.instructions.**addint**(**args*, ***kwargs*)
Clear integer register (vector) addition.

Param result (regint)

Param summand (regint)

Param summand (regint)

Compiler.instructions.**addm**(**args*, ***kwargs*)
Mixed addition.

Param result (sint)

Param summand (sint)

Param summand (cint)

Compiler.instructions.**adds**(**args*, ***kwargs*)
Secret addition.

Param result (sint)

Param summand (sint)

Param summand (sint)

Compiler.instructions.**addsi**(**args*, ***kwargs*)
Addition of secret register (vector) and (constant) immediate value.

Param result (cint)

Param summand (cint)

Param summand (int)

Compiler.instructions.**andc**(**args*, ***kwargs*)
Logical AND of clear (vector) registers.

Param result (cint)

Param operand (cint)

Param operand (cint)

Compiler.instructions.**andci**(**args*, ***kwargs*)
Logical AND of clear register (vector) and (constant) immediate value.

Param result (cint)

Param operand (cint)

Param operand (int)

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)...

Compiler.instructions.**bit**(**args*, ***kwargs*)
Store fresh random triple(s) in secret register (vectors).

Param destination (sint)

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.**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)

Compiler.instructions.**convint**(**args*, ***kwargs*)
Convert clear integer register (vector) to clear register (vector).

Param destination (cint)

Param source (regint)

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.

Compiler.instructions.**dabit**(**args*, ***kwargs*)
Store fresh random daBit(s) in secret register (vectors).

Param arithmetic part (sint)

Param binary part (sbit)

Compiler.instructions.**digestc**(**args*, ***kwargs*)
Clear truncated hash computation.

Param result (cint)

Param input (cint)

Param byte length of hash value used (int)

Compiler.instructions.**divc**(**args*, ***kwargs*)
Clear division.

Param result (cint)

Param dividend (cint)

Param divisor (cint)

Compiler.instructions.**divci**(**args*, ***kwargs*)
Division of secret register (vector) and (constant) immediate value.

Param result (cint)

Param dividend (cint)

Param divisor (int)

Compiler.instructions.**divint**(**args*, ***kwargs*)
Clear integer register (element-wise vector) division with floor rounding.

Param result (regint)

Param dividend (regint)

Param divisor (regint)

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)...

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)...

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)

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)

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)

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)

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)

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)...

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)...

Compiler.instructions.**inv2m**(**args*, ***kwargs*)

Inverse of power of two modulo prime (the computation modulus).

Param result (cint)

Param exponent (int)

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)

Compiler.instructions.**ldarg**(**args*, ***kwargs*)
Store the argument passed to the current thread in clear integer register.

Param destination (regint)

Compiler.instructions.**ldi**(**args*, ***kwargs*)
Assign (constant) immediate value to clear register (vector).

Param destination (cint)

Param value (int)

Compiler.instructions.**ldint**(**args*, ***kwargs*)
Store (constant) immediate value in clear integer register (vector).

Param destination (regint)

Param immediate (int)

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)

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)

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)

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)

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)

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)

Compiler.instructions.**ldsi**(**args*, ***kwargs*)
Assign (constant) immediate value to secret register (vector).

Param destination (sint)

Param value (int)

Compiler.instructions.**ldtn**(**args*, ***kwargs*)
Store the number of the current thread in clear integer register.

Param destination (regint)

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 (int)

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)

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)

`Compiler.instructions.`**`modc`**(**args*, ***kwargs*)
Clear modular reduction.

Param result (cint)

Param dividend (cint)

Param divisor (cint)

`Compiler.instructions.`**`modci`**(**args*, ***kwargs*)
Modular reduction of clear register (vector) and (constant) immediate value.

Param result (cint)

Param dividend (cint)

Param divisor (int)

`Compiler.instructions.`**`movc`**(**args*, ***kwargs*)
Copy clear register (vector).

Param destination (cint)

Param source (cint)

`Compiler.instructions.`**`movint`**(**args*, ***kwargs*)
Copy clear integer register (vector).

Param destination (regint)

Param source (regint)

`Compiler.instructions.`**`movs`**(**args*, ***kwargs*)
Copy secret register (vector).

Param destination (cint)

Param source (cint)

`Compiler.instructions.`**`mulc`**(**args*, ***kwargs*)
Clear multiplication.

Param result (cint)

Param factor (cint)

Param factor (cint)

Compiler.instructions.**mulci**(**args*, ***kwargs*)
Multiplication of clear register (vector) and (constant) immediate value.

Param result (cint)

Param factor (cint)

Param factor (int)

Compiler.instructions.**mulint**(**args*, ***kwargs*)
Clear integer register (element-wise vector) multiplication.

Param result (regint)

Param factor (regint)

Param factor (regint)

Compiler.instructions.**mulm**(**args*, ***kwargs*)
Multiply secret and clear value.

Param result (sint)

Param factor (sint)

Param factor (cint)

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)...

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)...

Compiler.instructions.**mulsi**(**args*, ***kwargs*)
Multiplication of secret register (vector) and (constant) immediate value.

Param result (sint)

Param factor (sint)

Param factor (int)

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)

Compiler.instructions.**orc**(**args*, ***kwargs*)
Logical OR of clear (vector) registers.

Param result (cint)

Param operand (cint)

Param operand (cint)

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)

Compiler.instructions.**popint**(**args*, ***kwargs*)
Pops from the thread-local stack to clear integer register.

Param destination (regint)

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)

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)

Compiler.instructions.**print_reg**(*reg*, *comment=''*)
Debugging output of clear register (vector).

Param source (cint)

Param comment (4 bytes / 1 unit)

Compiler.instructions.**print_reg_plain**(**args*, ***kwargs*)
Output clear register.

Param source (cint)

Compiler.instructions.**pubinput**(**args*, ***kwargs*)
Store public input in clear register (vector).

Param destination (cint)

Compiler.instructions.**pushint**(**args*, ***kwargs*)
Pushes clear integer register to the thread-local stack.

Param source (regint)

Compiler.instructions.**rand**(**args*, ***kwargs*)
Store insecure random value of specified length in clear integer register (vector).

Param destination (regint)

Param length (regint)

Compiler.instructions.**randomfulls**(**args*, ***kwargs*)
Store share(s) of a fresh secret random element in secret register (vectors).

Param destination (sint)

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)

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)...

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 destination (cint)

Param (repeat destination)...

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 destination (regint)

Param (repeat destination)...

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 tape number (int)

Param virtual machine thread number (int)

Param tape argument (int)

Param (repeat the last three)...

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)...

Compiler.instructions.**shlc**(**args*, ***kwargs*)

Bitwise left shift of clear register (vector).

Param result (cint)

Param first operand (cint)

Param second operand (cint)

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)

Compiler.instructions.**shrc**(**args*, ***kwargs*)
Bitwise right shift of clear register (vector).

Param result (cint)

Param first operand (cint)

Param second operand (cint)

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)

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)

Compiler.instructions.**square**(**args*, ***kwargs*)
Store fresh random square(s) in secret register (vectors).

Param value (sint)

Param square (sint)

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)

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)

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)

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)

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)

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)

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)

class Compiler.instructions.**stop**(**args*, ***kwargs*)
Stop timer.

Param timer number (int)

Compiler.instructions.**subc**(**args*, ***kwargs*)
Clear subtraction.

Param result (cint)

Param first operand (cint)

Param second operand (cint)

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)

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)

Compiler.instructions.**subint**(**args*, ***kwargs*)
Clear integer register (vector) subtraction.

Param result (regint)

Param first operand (regint)

Param second operand (regint)

Compiler.instructions.**subml**(**args*, ***kwargs*)
Subtract clear from secret value.

Param result (sint)

Param first operand (sint)

Param second operand (cint)

Compiler.instructions.**submr**(**args*, ***kwargs*)
Subtract secret from clear value.

Param result (sint)

Param first operand (cint)

Param second operand (sint)

Compiler.instructions.**subs**(**args*, ***kwargs*)
Secret subtraction.

Param result (sint)

Param first operand (sint)

Param second operand (sint)

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)

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.

Compiler.instructions.**triple**(**args*, ***kwargs*)
Store fresh random triple(s) in secret register (vectors).

Param factor (sint)

Param factor (sint)

Param product (sint)

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)

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)...

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 source (sint)

Param (repeat source)...

Compiler.instructions.**xorc**(**args*, ***kwargs*)
Logical XOR of clear (vector) registers.

Param result (cint)

Param operand (cint)

Param operand (cint)

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)

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.**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.**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)

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.**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.

```
Names N;
int my_number = atoi(argv[1]);
int n_parties = atoi(argv[2]);
int port_base = 9999;
Server::start_networking(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. `Server::start_networking()` 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.

```
// initialize fields
T::clear::init_default(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.

```
T::clear::next::init_default(prime_length, false);
```

For computation modulo a prime, it is more efficient to use Montgomery representation, which is not compatible with the MASCOT offline phase however. This line initializes another field instance for MASCOT without using Montgomery representation.

```
// must initialize MAC key for security of some protocols
typename T::mac_key_type mac_key;
T::read_or_generate_mac_key("", P, mac_key);
```

Some protocols use an information-theoretic tag that is constant throughout the protocol. This codes reads it from storage if available or generates a fresh one otherwise.

```
// global OT setup
BaseMachine machine;
if (T::needs_ot)
    machine.ot_setups.push_back({P});
```

Many protocols for a dishonest majority use oblivious transfer. This block runs a few instances to seed the oblivious transfer extension. The resulting setup only works for one thread. For several threads, you need to add sufficiently many instances to `ot_setups` and set `BaseMachine::thread_num` (thread-local) to a different consecutive number in every thread.

```
// keeps tracks of preprocessing usage (triples etc)
DataPositions usage;
usage.set_num_players(P.num_players());
```

To help keeping track of the required preprocessing, it is necessary to initialize preprocessing instances with a `DataPositions` variable that will store the usage.

```
// initialize binary computation
T::bit_type::mac_key_type::init_field();
typename T::bit_type::mac_key_type binary_mac_key;
T::bit_type::part_type::read_or_generate_mac_key("", P, binary_mac_key);
GC::ShareThread<typename T::bit_type> thread(N,
        OnlineOptions::singleton, P, binary_mac_key, usage);
```

While this example only uses arithmetic computation, you need to initialize binary computation as well unless you use the compile-time option `NO_MIXED_CIRCUITS`.

```
// output protocol
typename T::MAC_Check output(mac_key);
```

Some output protocols use the MAC key to check the correctness.

```
// various preprocessing
typename T::LivePrep preprocessing(0, usage);
SubProcessor<T> processor(output, preprocessing, P);
```

In this example we use live preprocessing, but it is also possible to read preprocessing data from disk by using `Sub_Data_Files` instead. You can use a live preprocessing instances to generate preprocessing data independently, but many protocols require that a `SubProcessor` instance has been created as well. The latter essentially glues an instance of the output and the preprocessing protocol together, which is necessary for Beaver-based multiplication protocols.

```
// input protocol
typename T::Input input(processor, output);
```

Some input protocols depend on preprocessing and an output protocol, which is reflect in the standard constructor. Other constructors are available depending on the protocol.

```
// multiplication protocol
typename T::Protocol protocol(P);
```

This instantiates a multiplication protocol. `P` is required because some protocols start by exchanging keys for pseudo-random secret sharing.

```
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);
```

(continues on next page)

(continued from previous page)

```
    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.

```
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 except that it is necessary to call the checking in order to verify the outputs.

```
T::LivePrep::teardown();
```

This frees the memory used for global key material when using homomorphic encryption. Otherwise, this does not do anything.

2.4 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 to easily run all parties on the same host. The base defaults to 5000, which can be changed with the command-line option `--portnumbase`. 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 minus one, thus defaulting to 4999. 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 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.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

C

Index

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

W

X