MP-SPDZ Documentation

Marcel Keller

May 31, 2023

CONTENTS:

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

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

CHAPTER

ONE

REFERENCE

1.1 Getting started

This is a software to benchmark various secure multi-party computation (MPC) protocols in a variety of security models such as honest and dishonest majority, semi-honest/passive and malicious/active corruption. The underlying technologies span secret sharing, homomorphic encryption, and garbled circuits.

1.1.1 Contact

Filing an issue on GitHub is the preferred way of contacting us, but you can also write an email to mp-spdz@googlegroups.com (archive). Before reporting a problem, please check against the list of known issues and possible solutions.

1.1.1.1 Filing Issues

Please file complete code examples because it's usually not possible to reproduce problems from incomplete code.

1.1.2 Frequently Asked Questions

The documentation contains sections on a number of frequently asked topics as well as information on how to solve common issues.

1.1.3 TL;DR (Binary Distribution on Linux or Source Distribution on macOS)

This requires either a Linux distribution originally released 2014 or later (glibc 2.17) or macOS High Sierra or later as well as Python 3 and basic command-line utilities.

Download and unpack the distribution, then execute the following from the top folder:

```
Scripts/tldr.sh
./compile.py tutorial
echo 1 2 3 4 > Player-Data/Input-P0-0
echo 1 2 3 4 > Player-Data/Input-P1-0
Scripts/mascot.sh tutorial
```

This runs the tutorial with two parties and malicious security.

1.1.4 TL;DR (Source Distribution)

On Linux, this requires a working toolchain and all requirements. On Ubuntu, the following might suffice:

```
sudo apt-get install automake build-essential cmake git libboost-dev libboost-thread-dev␣
↪libntl-dev libsodium-dev libssl-dev libtool m4 python3 texinfo yasm
```

On MacOS, this requires brew to be installed, which will be used for all dependencies. It will execute the tutorial with two parties and malicious security.

Note that this only works with a git clone but not with a binary release.

```
make -j 8 tldr
./compile.py tutorial
echo 1 2 3 4 > Player-Data/Input-P0-0
echo 1 2 3 4 > Player-Data/Input-P1-0
Scripts/mascot.sh tutorial
```

1.1.5 TL;DR (Docker)

Build a docker image for `mascot-party.x`:

```
docker build --tag mpspdz:mascot-party --build-arg machine=mascot-party.x .
```

Run the the tutorial:

```
docker run --rm -it mpspdz:mascot-party ./Scripts/mascot.sh tutorial
```

See the `Dockerfile` for examples of how it can be used.

1.1.6 Preface

The primary aim of this software is to run the same computation in various protocols in order to compare the performance. All protocols in the matrix below are fully implemented. However, this does not mean that the software has undergone a security review as should be done with critical production code.

1.1.7 Protocols

The following table lists all protocols that are fully supported.

Security model	Mod prime / GF(2^n)	Mod 2^k	Bin. SS	Garbling
Malicious, dishonest majority	MASCOT / LowGear / HighGear	SPDZ2k	Tiny / Tinier	BMR
Covert, dishonest majority	CowGear / ChaiGear	N/A	N/A	N/A
Semi-honest, dishonest majority	Semi / Hemi / Temi / Soho	Semi2k	SemiBin	Yao's GC / BMR
Malicious, honest majority	Shamir / Rep3 / PS / SY	Brain / Rep3 / PS / SY	Rep3 / CCD / PS	BMR
Semi-honest, honest majority	Shamir / ATLAS / Rep3	Rep3	Rep3 / CCD	BMR
Malicious, honest super-majority	Rep4	Rep4	Rep4	N/A
Semi-honest, dealer	Dealer	Dealer	Dealer	N/A

Modulo prime and modulo 2^k are the two settings that allow integer-like computation. For k = 64, the latter corresponds to the computation available on the widely used 64-bit processors. GF(2^n) denotes Galois extension fields of order 2^n, which are different to computation modulo 2^n. In particular, every element has an inverse, which is not the case modulo 2^n. See this article for an introduction. Modulo prime and GF(2^n) are lumped together because the protocols are very similar due to the mathematical properties.

Bin. SS stands for binary secret sharing, that is secret sharing modulo two. In some settings, this requires specific protocols as some protocols require the domain size to be larger than two. In other settings, the protocol is the same mathematically speaking, but a specific implementation allows for optimizations such as using the inherent parallelism of bit-wise operations on machine words.

A security model specifies how many parties are "allowed" to misbehave in what sense. Malicious means that not following the protocol will at least be detected while semi-honest means that even corrupted parties are assumed to follow the protocol. See this paper for an explanation of the various security models and a high-level introduction to multi-party computation.

1.1.7.1 Finding the most efficient protocol

Lower security requirements generally allow for more efficient protocols. Within the same security model (line in the table above), there are a few things to consider:

- Computation domain: Arithmetic protocols (modulo prime or power of two) are preferable for many applications because they offer integer addition and multiplication at low cost. However, binary circuits might be a better option if there is very little integer computation. See below to find the most efficient mixed-circuit variant. Furthermore, local computation modulo a power of two is cheaper, but MP-SPDZ does not offer this domain with homomorphic encryption.
- Secret sharing vs garbled circuits: Computation using secret sharing requires a number of communication rounds that grows depending on the computation, which is not the case for garbled circuits. However, the cost of integer computation as a binary circuit often offset this. MP-SPDZ only offers garbled circuit with binary computation.
- Underlying technology for dishonest majority: While secret sharing alone suffice honest-majority computation, dishonest majority requires either homomorphic encryption (HE) or oblivious transfer (OT). The two options offer a computation-communication trade-off: While OT is easier to compute, HE requires less communication. Furthermore, the latter requires a certain of batching to be efficient, which makes OT preferable for smaller tasks.
- Malicious, honest-majority three-party computation: A number of protocols are available for this setting, but SY/SPDZ-wise is the most efficient one for a number of reasons: It requires the lowest communication, and it is the only one offering constant-communication dot products.

- Fixed-point multiplication: Three- and four-party replicated secret sharing as well semi-honest full-threshold protocols allow a special probabilistic truncation protocol (see Dalskov et al. and Dalskov et al.). You can activate it by adding `program.use_trunc_pr = True` at the beginning of your high-level program.
- Larger number of parties: ATLAS scales better than the plain Shamir protocol, and Temi scale better than Hemi or Semi.
- Minor variants: Some command-line options change aspects of the protocols such as:
  - `--bucket-size`: In some malicious binary computation and malicious edaBit generation, a smaller bucket size allows preprocessing in smaller batches at a higher asymptotic cost.
  - `--batch-size`: Preprocessing in smaller batches avoids generating too much but larger batches save communication rounds.
  - `--direct`: In dishonest-majority protocols, direct communication instead of star-shaped saves communication rounds at the expense of a quadratic amount. This might be beneficial with a small number of parties.
  - `--bits-from-squares`: In some protocols computing modulo a prime (Shamir, Rep3, SPDZ-wise), this switches from generating random bits via XOR of parties' inputs to generation using the root of a random square.

1.1.8 Paper and Citation

The design of MP-SPDZ is described in this paper. If you use it for an academic project, please cite:

```
@inproceedings{mp-spdz,
    author = {Marcel Keller},
    title = {{MP-SPDZ}: A Versatile Framework for Multi-Party Computation},
    booktitle = {Proceedings of the 2020 ACM SIGSAC Conference on
    Computer and Communications Security},
    year = {2020},
    doi = {10.1145/3372297.3417872},
    url = {https://doi.org/10.1145/3372297.3417872},
}
```

1.1.9 History

The software started out as an implementation of the improved SPDZ protocol. The name SPDZ is derived from the authors of the original protocol.

This repository combines the functionality previously published in the following repositories:

- https://github.com/bristolcrypto/SPDZ-2
- https://github.com/mkskeller/SPDZ-BMR-ORAM
- https://github.com/mkskeller/SPDZ-Yao

1.1.10 Alternatives

There is another fork of SPDZ-2 called SCALE-MAMBA. The main differences at the time of writing are as follows:

- It provides honest-majority computation for any Q2 structure.
- For dishonest majority computation, it provides integration of SPDZ/Overdrive offline and online phases but without secure key generation.
- It only provides computation modulo a prime.
- It only provides malicious security.

More information can be found here: https://homes.esat.kuleuven.be/~nsmart/SCALE

1.1.11 Overview

For the actual computation, the software implements a virtual machine that executes programs in a specific bytecode. Such code can be generated from high-level Python code using a compiler that optimizes the computation with a particular focus on minimizing the number of communication rounds (for protocol based on secret sharing) or on AES-NI pipelining (for garbled circuits).

The software uses two different bytecode sets, one for arithmetic circuits and one for boolean circuits. The high-level code slightly differs between the two variants, but we aim to keep these differences a at minimum.

In the section on computation we will explain how to compile a high-level program for the various computation domains and then how to run it with different protocols.

The section on offline phases will explain how to benchmark the offline phases required for the SPDZ protocol. Running the online phase outputs the amount of offline material required, which allows to compute the preprocessing time for a particular computation.

1.1.12 Requirements

- GCC 5 or later (tested with up to 11) or LLVM/clang 5 or later (tested with up to 12). We recommend clang because it performs better. Note that GCC 5/6 and clang 9 don't support libOTe, so you need to deactivate its use for these compilers (see the next section).
- For protocol using oblivious transfer, libOTe with the necessary patches but without SimplestOT. The easiest way is to run `make libote`, which will install it as needed in a subdirectory. libOTe requires CMake of version at least 3.15, which is not available by default on older systems such as Ubuntu 18.04. You can run `make cmake` to install it locally.
- MPIR library, compiled with C++ support (use flag `--enable-cxx` when running configure). You can use `make -j8 mpir` to install it locally.
- libsodium library, tested against 1.0.18
- OpenSSL, tested against 1.1.1
- Boost.Asio with SSL support (`libboost-dev` on Ubuntu), tested against 1.71
- Boost.Thread for BMR (`libboost-thread-dev` on Ubuntu), tested against 1.71
- x86 or ARM 64-bit CPU (the latter tested with AWS Gravitron and Apple Silicon)
- Python 3.5 or later
- NTL library for homomorphic encryption (optional; tested with NTL 10.5)
- If using macOS, Sierra or later

- Windows/VirtualBox: see this issue for a discussion

1.1.13 Compilation

1. Edit `CONFIG` or `CONFIG.mine` to your needs:
   - On x86, the binaries are optimized for the CPU you are compiling on. For all optimizations on x86, a CPU supporting AES-NI, PCLMUL, AVX2, BMI2, ADX is required. This includes mainstream processors released 2014 or later. If you intend to run on a different CPU than compiling, you might need to change the `ARCH` variable in `CONFIG` or `CONFIG.mine` to `-march=<cpu>`. See the GCC documentation for the possible options.
   - For optimal results on Linux on ARM, add `ARCH = -march=armv8.2-a+crypto` to `CONFIG.mine`. This enables the hardware support for AES. See the GCC documentation on available options.
   - To benchmark online-only protocols or Overdrive offline phases, add the following line at the top: `MY_CFLAGS = -DINSECURE`
   - `PREP_DIR` should point to a local, unversioned directory to store preprocessing data (the default is `Player-Data` in the current directory).
   - `SSL_DIR` should point to a local, unversioned directory to store ssl keys (the default is `Player-Data` in the current directory).
   - For homomorphic encryption with GF(2^40), set `USE_NTL = 1`.
   - To use KOS instead of SoftSpokenOT, add `USE_KOS = 1` and `SECURE = -DINSECURE` to `CONFIG.mine`. This is necessary with GCC 5 and 6 because these compilers don't support the C++ standard used by libOTe.
2. Run `make` to compile all the software (use the flag `-j` for faster compilation using multiple threads). See below on how to compile specific parts only. Remember to run `make clean` first after changing `CONFIG` or `CONFIG.mine`.

1.2 Running computation

See `Programs/Source/` for some example MPC programs, in particular `tutorial.mpc`. Furthermore, Read the Docs hosts a more detailed reference of the high-level functionality extracted from the Python code in the `Compiler` directory as well as a summary of relevant compiler options.

1.2.1 Compiling high-level programs

There are three computation domains, and the high-level programs have to be compiled accordingly.

1.2.1.1 Arithmetic modulo a prime

```
./compile.py [-F <integer bit length>] [-P <prime>] <program>
```

The integer bit length defaults to 64, and the prime defaults to none given. If a prime is given, it has to be at least two bits longer than the integer length.

Note that in this context integers do not wrap around according to the bit integer bit length but the length is used for non-linear computations such as comparison. Overflow in secret integers might have security implications if no concrete prime is given.

The parameters given together with the computation mandate some restriction on the prime modulus, either an exact value or a minimum length. The latter is roughly the integer length plus 40 (default security parameter). The restrictions

are communicated to the virtual machines, which will use an appropriate prime if they have been compiled accordingly. By default, they are compiled for prime bit lengths up to 256. For larger primes, you will have to compile with `MOD = -DGFP_MOD_SZ=<number of limbs>` in `CONFIG.mine` where the number of limbs is the the prime length divided by 64 rounded up.

The precision for fixed- and floating-point computation are not affected by the integer bit length but can be set in the code directly. For fixed-point computation this is done via `sfix.set_precision()`.

1.2.1.2 Arithmetic modulo 2^k

```
./compile.py -R <integer bit length> <program>
```

The length is communicated to the virtual machines and automatically used if supported. By default, they support bit lengths 64, 72, and 128. If another length is required, use `MOD = -DRING_SIZE=<bit length>` in `CONFIG.mine`.

1.2.1.3 Binary circuits

```
./compile.py -B <integer bit length> <program>
```

The integer length can be any number up to a maximum depending on the protocol. All protocols support at least 64-bit integers.

Fixed-point numbers (`sfix`) always use 16/16-bit precision by default in binary circuits. This can be changed with `sfix.set_precision`. See the tutorial.

If you would like to use integers of various precisions, you can use `sbitint.get_type(n)` to get a type for `n`-bit arithmetic.

1.2.1.4 Mixed circuits

MP-SPDZ allows to mix computation between arithmetic and binary secret sharing in the same security model. In the compiler, this is used to switch from arithmetic to binary computation for certain non-linear functions such as comparison, bit decomposition, truncation, and modulo power of two, which are use for fixed- and floating-point operations. There are several ways of achieving this as described below.

Classic daBits

You can activate this by adding `-X` when compiling arithmetic circuits, that is `./compile.py -X [-F <integer bit length>] <program>` for computation modulo a prime and `./compile.py -X -R <integer bit length> <program>` for computation modulo 2^k.

Internally, this uses daBits described by Rotaru and Wood, that is secret random bits shared in different domains. Some security models allow direct conversion of random bits from arithmetic to binary while others require inputs from several parties followed by computing XOR and checking for malicious security as described by Rotaru and Wood in Section 4.1.

Extended daBits

Extended daBits were introduced by Escudero et al.. You can activate them by using `-Y` instead of `-X`. Note that this also activates classic daBits when useful.

Local share conversion

This technique has been used by Mohassel and Rindal as well as Araki et al. for three parties and Demmler et al. for two parties. It involves locally converting an arithmetic share to a set of binary shares, from which the binary equivalent to the arithmetic share is reconstructed using a binary adder. This requires additive secret sharing over a ring without any MACs. You can activate it by using `-Z <n>` with the compiler where `n` is the number of parties for the standard variant and 2 for the special variant by Mohassel and Rindal (available in Rep3 only).

Finding the most efficient variant

Where available, local share conversion is likely the most efficient variant. Otherwise, edaBits likely offer an asymptotic benefit. When using edaBits with malicious protocols, there is a trade-off between cost per item and batch size. The lowest cost per item requires large batches of edaBits (more than one million at once), which is only worthwhile for accordingly large computation. This setting can be selected by running the virtual machine with `-B 3`. For smaller computation, try `-B 4` or `-B 5`, which set the batch size to ~10,000 and ~1,000, respectively, at a higher asymptotic cost. `-B 4` is the default.

1.2.1.5 Bristol Fashion circuits

Bristol Fashion is the name of a description format of binary circuits used by SCALE-MAMBA. You can access such circuits from the high-level language if they are present in `Programs/Circuits`. To run the AES-128 circuit provided with SCALE-MAMBA, you can run the following:

```
make Programs/Circuits
./compile.py aes_circuit
Scripts/semi.sh aes_circuit
```

This downloads the circuit, compiles it to MP-SPDZ bytecode, and runs it as semi-honest two-party computation 1000 times in parallel. It should then output the AES test vector `0x3ad77bb40d7a3660a89ecaf32466ef97`. You can run it with any other protocol as well.

See the documentation for further examples.

1.2.1.6 Compiling programs directly in Python

You may prefer to not have an entirely static `.mpc` file to compile, and may want to compile based on dynamic inputs. For example, you may want to be able to compile with different sizes of input data without making a code change to the `.mpc` file. To handle this, the compiler an also be directly imported, and a function can be compiled with the following interface:

```
# hello_world.mpc
from Compiler.library import print_ln
from Compiler.compilerLib import Compiler

compiler = Compiler()
```

(continues on next page)

(continued from previous page)

```
@compiler.register_function('helloworld')
def hello_world():
    print_ln('hello world')

if __name__ == "__main__":
    compiler.compile_func()
```

You could then run this with the same args as used with `compile.py`:

```
python hello_world.mpc <compile args>
```

This is particularly useful if want to add new command line arguments specifically for your `.mpc` file. See test_args.mpc for more details on this use case.

Note that when using this approach, all objects provided in the high level interface (e.g. sint, print_ln) need to be imported, because the `.mpc` file is interpreted directly by Python (instead of being read by `compile.py`.)

1.2.1.7 Compiling and running programs from external directories

Programs can also be edited, compiled and run from any directory with the above basic structure. So for a source file in `./Programs/Source/`, all MP-SPDZ scripts must be run from `./`. Any setup scripts such as `setup-ssl.sh` script must also be run from `./` to create the relevant data. For example:

```
MP-SPDZ$ cd ../
$ mkdir myprogs
$ cd myprogs
$ mkdir -p Programs/Source
$ vi Programs/Source/test.mpc
$ ../MP-SPDZ/compile.py test.mpc
$ ls Programs/
Bytecode  Public-Input  Schedules  Source
$ ../MP-SPDZ/Scripts/setup-ssl.sh
$ ls
Player-Data Programs
$ ../MP-SPDZ/Scripts/rep-field.sh test
```

1.2.2 TensorFlow inference

MP-SPDZ supports inference with selected TensorFlow graphs, in particular DenseNet, ResNet, and SqueezeNet as used in CrypTFlow. For example, you can run SqueezeNet inference for ImageNet as follows:

```
git clone https://github.com/mkskeller/EzPC
cd EzPC/Athos/Networks/SqueezeNetImgNet
axel -a -n 5 -c --output ./PreTrainedModel https://github.com/avoroshilov/tf-squeezenet/
↪raw/master/sqz_full.mat
pip3 install scipy==1.1.0
python3 squeezenet_main.py --in ./SampleImages/n02109961_36.JPEG --saveTFMetadata True
python3 squeezenet_main.py --in ./SampleImages/n02109961_36.JPEG --scalingFac 12 --
↪saveImgAndWtData True
cd ../../../..
```

(continues on next page)

(continued from previous page)

```
Scripts/fixed-rep-to-float.py EzPC/Athos/Networks/SqueezeNetImgNet/SqNetImgNet_img_input.
↪inp
./compile.py -R 64 tf EzPC/Athos/Networks/SqueezeNetImgNet/graphDef.bin 1 trunc_pr split
Scripts/ring.sh tf-EzPC_Athos_Networks_SqueezeNetImgNet_graphDef.bin-1-trunc_pr-split
```

This requires TensorFlow and the axel command-line utility to be installed. It runs inference with three-party semi-honest computation, similar to CrypTFlow's Porthos. Replace 1 by the desired number of thread in the last two lines. If you run with some other protocols, you will need to remove `trunc_pr` and/or `split`. Also note that you will need to use a CrypTFlow repository that includes the patch in https://github.com/mkskeller/EzPC/commit/2021be90d21dc26894be98f33cd10dd26769f479.

The reference contains further documentation on available layers.

1.2.3 Emulation

For arithmetic circuits modulo a power of two and binary circuits, you can emulate the computation as follows:

```
./emulate.x <program>
```

This runs the compiled bytecode in cleartext computation.

1.2.4 Dishonest majority

Some full implementations require oblivious transfer, which is implemented as OT extension based on https://github.com/mkskeller/SimpleOT or https://github.com/mkskeller/SimplestOT_C, depending on whether AVX is available.

1.2.4.1 Secret sharing

The following table shows all programs for dishonest-majority computation using secret sharing:

Program	Protocol	Domain	Security	Script
`mascot-party.x`	MASCOT	Mod prime	Malicious	`mascot.sh`
`mama-party.x`	MASCOT*	Mod prime	Malicious	`mama.sh`
`spdz2k-party.x`	SPDZ2k	Mod 2^k	Malicious	`spdz2k.sh`
`semi-party.x`	OT-based	Mod prime	Semi-honest	`semi.sh`
`semi2k-party.x`	OT-based	Mod 2^k	Semi-honest	`semi2k.sh`
`lowgear-party.x`	LowGear	Mod prime	Malicious	`lowgear.sh`
`highgear-party.x`	HighGear	Mod prime	Malicious	`highgear.sh`
`cowgear-party.x`	Adapted LowGear	Mod prime	Covert	`cowgear.sh`
`chaigear-party.x`	Adapted HighGear	Mod prime	Covert	`chaigear.sh`
`hemi-party.x`	Semi-homomorphic encryption	Mod prime	Semi-honest	`hemi.sh`
`temi-party.x`	Adapted CDN01	Mod prime	Semi-honest	`temi.sh`
`soho-party.x`	Somewhat homomorphic encryption	Mod prime	Semi-honest	`soho.sh`
`semi-bin-party.x`	OT-based	Binary	Semi-honest	`semi-bin.sh`
`tiny-party.x`	Adapted SPDZ2k	Binary	Malicious	`tiny.sh`
`tinier-party.x`	FKOS15	Binary	Malicious	`tinier.sh`

Mama denotes MASCOT with several MACs to increase the security parameter to a multiple of the prime length.

Semi and Semi2k denote the result of stripping MASCOT/SPDZ2k of all steps required for malicious security, namely amplifying, sacrificing, MAC generation, and OT correlation checks. What remains is the generation of additively shared Beaver triples using OT.

Similarly, SemiBin denotes a protocol that generates bit-wise multiplication triples using OT without any element of malicious security.

Tiny denotes the adaption of SPDZ2k to the binary setting. In particular, the SPDZ2k sacrifice does not work for bits, so we replace it by cut-and-choose according to Furukawa et al.

The virtual machines for LowGear and HighGear run a key generation similar to the one by Rotaru et al.. The main difference is using daBits to generate maBits. CowGear and ChaiGear denote covertly secure versions of LowGear and HighGear. In all relevant programs, option `-T` activates TopGear zero-knowledge proofs in both.

Hemi and Soho denote the stripped version of LowGear and HighGear, respectively, for semi-honest security similar to Semi, that is, generating additively shared Beaver triples using semi-homomorphic encryption. Temi in turn denotes the adaption of Cramer et al. to LWE-based semi-homomorphic encryption. Both Hemi and Temi use the diagonal packing by Halevi and Shoup for matrix multiplication.

We will use MASCOT to demonstrate the use, but the other protocols work similarly.

First compile the virtual machine:

```
make -j8 mascot-party.x
```

and a high-level program, for example the tutorial (use `-R 64` for SPDZ2k and Semi2k and `-B <precision>` for SemiBin):

```
./compile.py -F 64 tutorial
```

To run the tutorial with two parties on one machine, run:

```
./mascot-party.x -N 2 -I -p 0 tutorial
```

`./mascot-party.x -N 2 -I -p 1 tutorial` (in a separate terminal)

Using `-I` activates interactive mode, which means that inputs are solicited from standard input, and outputs are given to any party. Omitting `-I` leads to inputs being read from `Player-Data/Input-P<party number>-0` in text format.

Or, you can use a script to do run two parties in non-interactive mode automatically:

```
Scripts/mascot.sh tutorial
```

To run a program on two different machines, `mascot-party.x` needs to be passed the machine where the first party is running, e.g. if this machine is name `diffie` on the local network:

```
./mascot-party.x -N 2 -h diffie 0 tutorial
```

```
./mascot-party.x -N 2 -h diffie 1 tutorial
```

The software uses TCP ports around 5000 by default, use the `-pn` argument to change that.

1.2.4.2 Yao's garbled circuits

We use half-gate garbling as described by Zahur et al. and Guo et al.. Alternatively, you can activate the implementation optimized by Bellare et al. by adding `MY_CFLAGS += -DFULL_GATES` to `CONFIG.mine`.

Compile the virtual machine:

```
make -j 8 yao
```

and the high-level program:

```
./compile.py -B <integer bit length> <program>
```

Then run as follows:

- Garbler: `./yao-party.x [-I] -p 0 <program>`
- Evaluator: `./yao-party.x [-I] -p 1 -h <garbler host> <program>`

When running locally, you can omit the host argument. As above, `-I` activates interactive input, otherwise inputs are read from `Player-Data/Input-P<playerno>-0`.

By default, the circuit is garbled in chunks that are evaluated whenever received.You can activate garbling all at once by adding `-O` to the command line on both sides.

1.2.5 Honest majority

The following table shows all programs for honest-majority computation:

Program	Sharing	Domain	Malicious	# parties	Script
`replicated-ring-party.x`	Replicated	Mod 2^k	N	3	`ring.sh`
`brain-party.x`	Replicated	Mod 2^k	Y	3	`brain.sh`
`ps-rep-ring-party.x`	Replicated	Mod 2^k	Y	3	`ps-rep-ring.sh`
`malicious-rep-ring-party.x`	Replicated	Mod 2^k	Y	3	`mal-rep-ring.sh`
`sy-rep-ring-party.x`	SPDZ-wise replicated	Mod 2^k	Y	3	`sy-rep-ring.sh`
`rep4-ring-party.x`	Replicated	Mod 2^k	Y	4	`rep4-ring.sh`
`replicated-bin-party.x`	Replicated	Binary	N	3	`replicated.sh`
`malicious-rep-bin-party.x`	Replicated	Binary	Y	3	`mal-rep-bin.sh`
`ps-rep-bin-party.x`	Replicated	Binary	Y	3	`ps-rep-bin.sh`
`replicated-field-party.x`	Replicated	Mod prime	N	3	`rep-field.sh`
`ps-rep-field-party.x`	Replicated	Mod prime	Y	3	`ps-rep-field.sh`
`sy-rep-field-party.x`	SPDZ-wise replicated	Mod prime	Y	3	`sy-rep-field.sh`
`malicious-rep-field-party.x`	Replicated	Mod prime	Y	3	`mal-rep-field.sh`
`atlas-party.x`	ATLAS	Mod prime	N	3 or more	`atlas.sh`
`shamir-party.x`	Shamir	Mod prime	N	3 or more	`shamir.sh`
`malicious-shamir-party.x`	Shamir	Mod prime	Y	3 or more	`mal-shamir.sh`
`sy-shamir-party.x`	SPDZ-wise Shamir	Mod prime	Y	3 or more	`sy-shamir.sh`
`ccd-party.x`	CCD/Shamir	Binary	N	3 or more	`ccd.sh`
`malicious-cdd-party.x`	CCD/Shamir	Binary	Y	3 or more	`mal-ccd.sh`

We use the "generate random triple optimistically/sacrifice/Beaver" methodology described by Lindell and Nof to achieve malicious security with plain arithmetic replicated secret sharing, except for the "PS" (post-sacrifice) protocols where the actual multiplication is executed optimistically and checked later as also described by Lindell and Nof. The

implementations used by `brain-party.x`, `malicious-rep-ring-party.x -S`, `malicious-rep-ring-party.x`, and `ps-rep-ring-party.x` correspond to the protocols called DOS18 preprocessing (single), ABF+17 preprocessing, CDE+18 preprocessing, and postprocessing, respectively, by Eerikson et al. We use resharing by Cramer et al. for Shamir's secret sharing and the optimized approach by Araki et al. for replicated secret sharing. The CCD protocols are named after the historic paper by Chaum, Crépeau, and Damgård, which introduced binary computation using Shamir secret sharing over extension fields of characteristic two. SY/SPDZ-wise refers to the line of work started by Chida et al. for computation modulo a prime and furthered by Abspoel et al. for computation modulo a power of two. It involves sharing both a secret value and information-theoretic tag similar to SPDZ but not with additive secret sharing, hence the name. Rep4 refers to the four-party protocol by Dalskov et al.. `malicious-rep-bin-party.x` is based on cut-and-choose triple generation by Furukawa et al. but using Beaver multiplication instead of their post-sacrifice approach. `ps-rep-bin-party.x` is based on the post-sacrifice approach by Araki et al. but without using their cache optimization.

All protocols in this section require encrypted channels because the information received by the honest majority suffices the reconstruct all secrets. Therefore, an eavesdropper on the network could learn all information.

MP-SPDZ uses OpenSSL for secure channels. You can generate the necessary certificates and keys as follows:

```
Scripts/setup-ssl.sh [<number of parties> <ssl_dir>]
```

The programs expect the keys and certificates to be in `SSL_DIR/P<i>.key` and `SSL_DIR/P<i>.pem`, respectively, and the certificates to have the common name `P<i>` for player `<i>`. Furthermore, the relevant root certificates have to be in `SSL_DIR` such that OpenSSL can find them (run `c_rehash <ssl_dir>`). The script above takes care of all this by generating self-signed certificates. Therefore, if you are running the programs on different hosts you will need to copy the certificate files. Note that `<ssl_dir>` must match `SSL_DIR` set in `CONFIG` or `CONFIG.mine`. Just like `SSL_DIR`, `<ssl_dir>` defaults to `Player-Data`.

In the following, we will walk through running the tutorial modulo 2^k with three parties. The other programs work similarly.

First, compile the virtual machine:

```
make -j 8 replicated-ring-party.x
```

In order to compile a high-level program, use `./compile.py -R 64`:

```
./compile.py -R 64 tutorial
```

If using another computation domain, use `-F` or `-B` as described in the relevant section above.

Finally, run the three parties as follows:

```
./replicated-ring-party.x -I 0 tutorial
```

`./replicated-ring-party.x -I 1 tutorial` (in a separate terminal)

`./replicated-ring-party.x -I 2 tutorial` (in a separate terminal)

or

```
Scripts/ring.sh tutorial
```

The `-I` argument enables interactive inputs, and in the tutorial party 0 and 1 will be asked to provide three numbers. Otherwise, and when using the script, the inputs are read from `Player-Data/Input-P<playerno>-0`.

When using programs based on Shamir's secret sharing, you can specify the number of parties with `-N` and the maximum number of corrupted parties with `-T`. The latter can be at most half the number of parties.

1.2.6 Dealer model

This security model defines a special party that generates correlated randomness such as multiplication triples, which is then used by all other parties. MP-SPDZ implements the canonical protocol where the other parties run the online phase of the semi-honest protocol in Semi(2k/Bin) and the dealer provides all preprocessing. The security assumption is that dealer doesn't collude with any other party, but all but one of the other parties are allowed to collude. In our implementation, the dealer is the party with the highest number, so with three parties overall, Party 0 and 1 run the online phase.

Program	Sharing	Domain	Malicious	# parties	Script
`dealer-ring-party.x`	Additive	Mod 2^k	N	3+	`dealer-ring.sh`

1.2.7 BMR

BMR (Beaver-Micali-Rogaway) is a method of generating a garbled circuit using another secure computation protocol. We have implemented BMR based on all available implementations using GF(2^128) because the nature of this field particularly suits the Free-XOR optimization for garbled circuits. Our implementation is based on the SPDZ-BMR-ORAM construction. The following table lists the available schemes.

Program	Protocol	Dishonest Maj.	Malicious	# parties	Script
`real-bmr-party.x`	MASCOT	Y	Y	2 or more	`real-bmr.sh`
`semi-bmr-party.x`	Semi	Y	N	2 or more	`semi-bmr.sh`
`shamir-bmr-party.x`	Shamir	N	N	3 or more	`shamir-bmr.sh`
`mal-shamir-bmr-party.x`	Shamir	N	Y	3 or more	`mal-shamir-bmr.sh`
`rep-bmr-party.x`	Replicated	N	N	3	`rep-bmr.sh`
`mal-rep-bmr-party.x`	Replicated	N	Y	3	`mal-rep-bmr.sh`

In the following, we will walk through running the tutorial with BMR based on MASCOT and two parties. The other programs work similarly.

First, compile the virtual machine. In order to run with more than three parties, change the definition of `MAX_N_PARTIES` in `BMR/config.h` accordingly.

```
make -j 8 real-bmr-party.x
```

In order to compile a high-level program, use `./compile.py -B`:

```
./compile.py -B 32 tutorial
```

Finally, run the two parties as follows:

```
./real-bmr-party.x -I 0 tutorial
```

`./real-bmr-party.x -I 1 tutorial` (in a separate terminal)

or

```
Scripts/real-bmr.sh tutorial
```

The `-I` enable interactive inputs, and in the tutorial party 0 and 1 will be asked to provide three numbers. Otherwise, and when using the script, the inputs are read from `Player-Data/Input-P<playerno>-0`.

1.2.8 Online-only benchmarking

In this section we show how to benchmark purely the data-dependent (often called online) phase of some protocols. This requires to generate the output of a previous phase. There are two options to do that:

1. For select protocols, you can run preprocessing as required.
2. You can run insecure preprocessing. For this, you will have to (re)compile the software after adding `MY_CFLAGS = -DINSECURE` to `CONFIG.mine` in order to run this insecure generation. Make sure to run `make clean` before recompiling any binaries. Then, you need to run `make Fake-Offline.x <protocol>-party.x`.

1.2.8.1 SPDZ

The SPDZ protocol uses preprocessing, that is, in a first (sometimes called offline) phase correlated randomness is generated independent of the actual inputs of the computation. Only the second ("online") phase combines this randomness with the actual inputs in order to produce the desired results. The preprocessed data can only be used once, thus more computation requires more preprocessing. MASCOT and Overdrive are the names for two alternative preprocessing phases to go with the SPDZ online phase.

All programs required in this section can be compiled with the target `online`:

```
make -j 8 online
```

To setup for benchmarking the online phase

This requires the INSECURE flag to be set before compilation as explained above. For a secure offline phase, see the section on SPDZ-2 below.

Run the command below. **If you haven't added `MY_CFLAGS = -DINSECURE` to `CONFIG.mine` before compiling, it will fail.**

```
Scripts/setup-online.sh
```

This sets up parameters for the online phase for 2 parties with a 128-bit prime field and 128-bit binary field, and creates fake offline data (multiplication triples etc.) for these parameters.

Parameters can be customised by running

```
Scripts/setup-online.sh <nparties> <nbitsp> [<nbits2>]
```

To compile a program

To compile for example the program in `./Programs/Source/tutorial.mpc`, run:

```
./compile.py tutorial
```

This creates the bytecode and schedule files in Programs/Bytecode/ and Programs/Schedules/

To run a program

To run the above program with two parties on one machine, run:

`./Player-Online.x -N 2 0 tutorial`

`./Player-Online.x -N 2 1 tutorial` (in a separate terminal)

Or, you can use a script to do the above automatically:

`Scripts/run-online.sh tutorial`

To run a program on two different machines, firstly the preprocessing data must be copied across to the second machine (or shared using sshfs), and secondly, Player-Online.x needs to be passed the machine where the first party is running. e.g. if this machine is name `diffie` on the local network:

`./Player-Online.x -N 2 -h diffie 0 test_all`

`./Player-Online.x -N 2 -h diffie 1 test_all`

The software uses TCP ports around 5000 by default, use the `-pn` argument to change that.

1.2.8.2 SPDZ2k

Creating fake offline data for SPDZ2k requires to call `Fake-Offline.x` directly instead of via `setup-online.sh`:

`./Fake-Offline.x <nparties> -Z <bit length k for SPDZ2k> -S <security parameter>`

You will need to run `spdz2k-party.x -F` in order to use the data from storage.

1.2.8.3 Other protocols

Preprocessing data for the default parameters of most other protocols can be produced as follows:

`./Fake-Offline.x <nparties> -e <edaBit length,...>`

The `-e` command-line parameters accepts a list of integers separated by commas.

You can then run the protocol with argument `-F`. Note that when running on several hosts, you will need to distribute the data in `Player-Data`. The preprocessing files contain `-P<party number>` indicating which party will access it.

1.2.8.4 BMR

This part has been developed to benchmark ORAM for the Eurocrypt 2018 paper by Marcel Keller and Avishay Yanay. It only allows to benchmark the data-dependent phase. The data-independent and function-independent phases are emulated insecurely.

By default, the implementations is optimized for two parties. You can change this by defining `N_PARTIES` accordingly in `BMR/config.h`. If you entirely delete the definition, it will be able to run for any number of parties albeit slower.

Compile the virtual machine:

`make -j 8 bmr`

After compiling the mpc file:

- Run everything locally: `Scripts/bmr-program-run.sh <program> <number of parties>`.
- Run on different hosts: `Scripts/bmr-program-run-remote.sh <program> <host1> <host2> [...]`

Oblivious RAM

You can benchmark the ORAM implementation as follows:

1) Edit `Program/Source/gc_oram.mpc` to change size and to choose Circuit ORAM or linear scan without ORAM.
2) Run `./compile.py -D gc_oram`. The `-D` argument instructs the compiler to remove dead code. This is useful for more complex programs such as this one.
3) Run `gc_oram` in the virtual machines as explained above.

1.2.9 Preprocessing as required

For select protocols, you can run all required preprocessing but not the actual computation. First, compile the binary:

```
make <protocol>-offline.x
```

At the time of writing the supported protocols are `mascot`, `cowgear`, and `mal-shamir`.

If you have not done so already, then compile your high-level program:

```
./compile.py <program>
```

Finally, run the parties as follows:

```
./<protocol>-offline.x -p 0 & ./<protocol>-offline.x -p 1 & ...
```

The options for the network setup are the same as for the complete computation above.

If you run the preprocessing on different hosts, make sure to use the same player number in the preprocessing and the online phase.

1.2.10 Benchmarking offline phases

1.2.10.1 SPDZ-2 offline phase

This implementation is suitable to generate the preprocessed data used in the online phase. You need to compile with `USE_NTL = 1` in `CONFIG.mine` to run this.

For quick run on one machine, you can call the following:

```
./spdz2-offline.x -p 0 & ./spdz2-offline.x -p 1
```

More generally, run the following on every machine:

```
./spdz2-offline.x -p <number of party> -N <total number of parties> -h <hostname of party
0> -c <covert security parameter>
```

The number of parties are counted from 0. As seen in the quick example, you can omit the total number of parties if it is 2 and the hostname if all parties run on the same machine. Invoke `./spdz2-offline.x` for more explanation on the options.

`./spdz2-offline.x` provides covert security according to some parameter c (at least 2). A malicious adversary will get caught with probability 1-1/c. There is a linear correlation between c and the running time, that is, running with 2c takes twice as long as running with c. The default for c is 10.

The program will generate every kind of randomness required by the online phase except input tuples until you stop it. You can shut it down gracefully pressing Ctrl-c (or sending the interrupt signal `SIGINT`), but only after an initial phase, the end of which is marked by the output `Starting to produce gf2n`. Note that the initial phase has been reported to take up to an hour. Furthermore, 3 GB of RAM are required per party.

1.2.10.2 Benchmarking the MASCOT or SPDZ2k offline phase

These implementations are not suitable to generate the preprocessed data for the online phase because they can only generate either multiplication triples or bits.

MASCOT can be run as follows:

```
host1:$ ./ot-offline.x -p 0 -c
```

```
host2:$ ./ot-offline.x -p 1 -c
```

For SPDZ2k, use `-Z <k>` to set the computation domain to Z_{2^k}, and `-S` to set the security parameter. The latter defaults to k. At the time of writing, the following combinations are available: 32/32, 64/64, 64/48, and 66/48.

Running `./ot-offline.x` without parameters give the full menu of options such as how many items to generate in how many threads and loops.

1.2.10.3 Benchmarking Overdrive offline phases

We have implemented several protocols to measure the maximal throughput for the Overdrive paper. As for MASCOT, these implementations are not suited to generate data for the online phase because they only generate one type at a time.

Binary	Protocol
`simple-offline.x`	SPDZ-1 and High Gear (with command-line argument `-g`)
`pairwise-offline.x`	Low Gear
`cnc-offline.x`	SPDZ-2 with malicious security (covert security with command-line argument `-c`)

These programs can be run similarly to `spdz2-offline.x`, for example:

```
host1:$ ./simple-offline.x -p 0 -h host1
```

```
host2:$ ./simple-offline.x -p 1 -h host1
```

Running any program without arguments describes all command-line arguments.

Memory usage

Lattice-based ciphertexts are relatively large (in the order of megabytes), and the zero-knowledge proofs we use require storing some hundred of them. You must therefore expect to use at least some hundred megabytes of memory per thread. The memory usage is linear in `MAX_MOD_SZ` (determining the maximum integer size for computations in steps of 64 bits), so you can try to reduce it (see the compilation section for how set it). For some choices of parameters, 4 is enough while others require up to 8. The programs above indicate the minimum `MAX_MOD_SZ` required, and they fail during the parameter generation if it is too low.

1.3 Compilation process

The easiest way of using MP-SPDZ is using `compile.py` as described below. If you would like to run compilation directly from Python, see Direct Compilation in Python.

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

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

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

The following options influence the computation domain:

-F `<integer length>`

--field=`<integer length>`

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

-P `<prime>`

--prime=`<prime>`

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

-R `<ring size>`

--ring=`<ring size>`

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

-B `<integer length>`

--binary=`<integer length>`

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

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

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

-X

--mixed

Enables mixed computation using daBits.

-Y

--edabit

Enables mixed computation using edaBits.

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

-Z `<number of parties>`

--split=`<number of parties>`

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

The following options change less fundamental aspects of the computation:

-D

--dead-code-elimination

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

-b <budget>

--budget=<budget>

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

-C

--CISC

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

-l

--flow-optimization

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

1.3.1 Direct Compilation in Python

You may prefer to not have an entirely static *.mpc* file to compile, and may want to compile based on dynamic inputs. For example, you may want to be able to compile with different sizes of input data without making a code change to the *.mpc* file. To handle this, the compiler an also be directly imported, and a function can be compiled with the following interface:

You could then run this with the same args as used with *compile.py*:

This is particularly useful if want to add new command line arguments specifically for your *.mpc* file. See [test_args.mpc](Programs/Source/test_args.mpc) for more details on this use case.

Note that when using this approach, all objects provided in the high level interface (e.g. sint, print_ln) need to be imported, because the *.mpc* file is interpreted directly by Python (instead of being read by *compile.py*.)

1.3.2 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`.

1.4 High-Level Interface

1.4.1 Compiler.types module

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

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

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

1.4.1.1 Basic types

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

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

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

1.4.1.2 Container types

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

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

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

Parameters

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

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

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

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

Assignment.

Parameters

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

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

Assign the same value to all entries.

Parameters **value** – convertible to basic type

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

Assignment.

Parameters

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

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

Assignment.

Parameters

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

binary_output(*player=None*)

Binary output if supported by type.

Param player (default all)

classmethod create_from(*l*)

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

Parameters **l** – Python iterable or register vector

Returns *Array* of appropriate type containing the contents of l

expand_to_vector(*index*, *size*)

Create vector from single entry.

Parameters

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

get(*indices*)

Vector from arbitrary indices.

Parameters **indices** – regint vector or array

get_part(*base*, *size*)

Part array.

Parameters

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

Returns Array of same type

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

Return vector with content.

Parameters

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

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

Return vector with content.

Parameters

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

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

Fill with inputs from player if supported by type.

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

maybe_get(*condition*, *index*)

Return entry if condition is true.

Parameters

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

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

Change entry if condition is true.

Parameters

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

print_reveal_nested(*end='\n'*)

Reveal and print as list.

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

randomize(**args*)

Randomize according to data type.

read_from_file(*start*)

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

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

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

reveal()

Reveal the whole array.

Returns Array of relevant clear type.

reveal_list()

Reveal as list.

reveal_nested()

Reveal as list.

reveal_to(*player*)

Reveal secret array to `player`.

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

Returns *personal* containing an array

reveal_to_binary_output(*player=None*)

Reveal to binary output if supported by type.

Param player to reveal to (default all)

reveal_to_clients(*clients*)

Reveal contents to list of clients.

Parameters **clients** – list or array of client identifiers

same_shape()

Array of same length and type.

secure_permute(**args*, ***kwargs*)

Secure permutate in place according to the security model.

secure_shuffle()

Secure shuffle in place according to the security model.

shuffle()

Insecure shuffle in place.

sort(*n_threads=None*, *batcher=False*, *n_bits=None*)

Sort in place using radix sort with complexity $O(n \log n)$ for *sint* and *sfix*, and Batcher's odd-even mergesort with $O(n(\log n)^2)$ for *sfloat*.

Parameters

- **n_threads** – number of threads to use (single thread by default), need to use Batcher's algorithm for several threads
- **batcher** – use Batcher's odd-even mergesort in any case

- **n_bits** – number of bits in keys (default: global bit length)

write_to_file(*position=None*)

Write shares of integer representation to `Persistence/Transactions-P<playerno>.data`.

Parameters **position** – start position (int/regint/cint), defaults to end of file

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

Matrix.

Parameters

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

assign(*other*)

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

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

assign_all(*value*)

Assign the same value to all entries.

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

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

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

Parameters

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

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

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

Parameters

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

assign_vector_by_indices(*vector*, **indices*)

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

diag()

Matrix diagonal.

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

Matrix multiplication in the virtual machine.

Parameters

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

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

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

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

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

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

Parameters

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

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

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

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

Parameters

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

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

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

Matrix-matrix and matrix-vector multiplication.

Parameters

- **self** – two-dimensional
- **other** – Matrix or Array of matching size and type
- **n_threads** – number of threads (default: all in same thread)

get_column(*index*)

Get column as vector.

Parameters **index** – regint/cint/int

get_part(*start*, *size*)

Part multi-array.

Parameters

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

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

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

Parameters

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

get_slice_vector(*slice*)

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

Parameters **slice** – regint array

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

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

Parameters

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

get_vector_by_indices(**indices*)

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

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

iadd(*other*)

Element-wise addition in place.

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

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

Fill with inputs from player if supported by type.

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

mul_trans(*other*)

Matrix multiplication with transpose of `other`.

Parameters

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

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

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

Parameters

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

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

Alternative matrix multiplication.

Parameters

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

print_reveal_nested(*end='\n'*)

Reveal and print as nested list.

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

randomize(**args*)

Randomize according to data type.

read_from_file(*start*)

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

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

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

reveal_list()

Reveal as list.

reveal_nested()

Reveal as nested list.

reveal_to_binary_output(*player=None*)

Reveal to binary output if supported by type.

Param player to reveal to (default all)

reveal_to_clients(*clients*)

Reveal contents to list of clients.

Parameters **clients** – list or array of client identifiers

same_shape()

Returns new multidimensional array with same shape and basic type

schur(*other*)

Element-wise product.

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

Returns container of same shape and type as `self`

secure_permute(*permutation*, *reverse=False*)

Securely permute rows (first index).

secure_shuffle()

Securely shuffle rows (first index).

set_column(*index*, *vector*)

Change column.

Parameters

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

sort(*key_indices=None*, *n_bits=None*)

Sort sub-arrays (different first index) in place.

Parameters

- **key_indices** – indices to sorting keys, for example `(1, 2)` to sort three-dimensional array `a` by keys `a[*][1][2]`. Default is `(0, ..., 0)` of correct length.
- **n_bits** – number of bits in keys (default: global bit length)

trace()

Matrix trace.

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

Matrix multiplication with transpose of `self`

Parameters

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

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

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

Parameters

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

transpose()

Matrix transpose.

Parameters **self** – two-dimensional

write_to_file(*position=None*)

Write shares of integer representation to `Persistence/Transactions-P<playerno>.data`.

Parameters **position** – start position (int/regint/cint), defaults to end of file

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

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

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

max(*other*)

Maximum.

Parameters **other** – any compatible type

min(*other*)

Minimum.

Parameters **other** – any compatible type

read()

Read value.

Returns relevant basic type instance

reveal()

Reveal value.

Returns relevant clear type

square()

Square.

write(*value*)

Write value.

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

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

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

Parameters

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

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

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

assign(*other*)

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

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

assign_all(*value*)

Assign the same value to all entries.

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

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

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

Parameters

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

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

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

Parameters

- **vector** – vector of matching size convertible to relevant basic type

- **base** – compile-time (int)

assign_vector_by_indices(*vector*, **indices*)

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

diag()

Matrix diagonal.

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

Matrix multiplication in the virtual machine.

Parameters

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

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

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

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

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

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

Parameters

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

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

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

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

Parameters

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

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

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

Matrix-matrix and matrix-vector multiplication.

Parameters

- **self** – two-dimensional
- **other** – Matrix or Array of matching size and type
- **n_threads** – number of threads (default: all in same thread)

get_part(*start*, *size*)

Part multi-array.

Parameters

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

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

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

Parameters

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

get_slice_vector(*slice*)

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

Parameters **slice** – regint array

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

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

Parameters

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

get_vector_by_indices(**indices*)

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

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

iadd(*other*)

Element-wise addition in place.

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

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

Fill with inputs from player if supported by type.

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

mul_trans(*other*)

Matrix multiplication with transpose of `other`.

Parameters

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

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

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

Parameters

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

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

Alternative matrix multiplication.

Parameters

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

print_reveal_nested(*end='\n'*)

Reveal and print as nested list.

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

randomize(**args*)

Randomize according to data type.

read_from_file(*start*)

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

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

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

reveal_list()

Reveal as list.

reveal_nested()

Reveal as nested list.

reveal_to_binary_output(*player=None*)

Reveal to binary output if supported by type.

Param player to reveal to (default all)

reveal_to_clients(*clients*)

Reveal contents to list of clients.

Parameters **clients** – list or array of client identifiers

same_shape()

Returns new multidimensional array with same shape and basic type

schur(*other*)

Element-wise product.

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

Returns container of same shape and type as `self`

secure_permute(*permutation*, *reverse=False*)

Securely permute rows (first index).

secure_shuffle()

Securely shuffle rows (first index).

sort(*key_indices=None*, *n_bits=None*)

Sort sub-arrays (different first index) in place.

Parameters

- **key_indices** – indices to sorting keys, for example `(1, 2)` to sort three-dimensional array `a` by keys `a[*][1][2]`. Default is `(0, ..., 0)` of correct length.
- **n_bits** – number of bits in keys (default: global bit length)

trace()

Matrix trace.

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

Matrix multiplication with transpose of `self`

Parameters

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

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

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

Parameters

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

transpose()

Matrix transpose.

Parameters **self** – two-dimensional

write_to_file(*position=None*)

Write shares of integer representation to `Persistence/Transactions-P<playerno>.data`.

Parameters **position** – start position (int/regint/cint), defaults to end of file

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

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

Parameters **v** – cfix/float/int

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

Type-dependent array. Example:

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

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

Type-dependent matrix. Example:

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

classmethod Tensor(*shape*)

Type-dependent tensor of any dimension:

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

binary_output(*player=None*)

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

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

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

Load from memory by public address.

max(*other*)

Maximum.

Parameters **other** – any compatible type

min(*other*)

Minimum.

Parameters **other** – any compatible type

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

Clear fixed-point output.

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

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

Parameters

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

Param vector size (int)

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

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

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

Parameters

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

square()

Square.

store_in_mem(*address*)

Store in memory by public address.

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

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

Parameters

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

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

Helper class for printing revealed sfloats.

binary_output(*player=None*)

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

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

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

Output.

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

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

Parameters

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

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

Type-dependent array. Example:

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

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

Type-dependent matrix. Example:

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

classmethod Tensor(*shape*)

Type-dependent tensor of any dimension:

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

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

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

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

bit_and(*other*)

AND in binary circuits.

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

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

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

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

Parameters

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

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

Clear bit decomposition.

Parameters

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

bit_not()

NOT in binary circuits.

bit_or(*other*)

OR in binary circuits.

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

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

bit_xor(*other*)

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

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

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

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

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

half_adder(*other*)

Half adder in binary circuits.

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

Returns binary sum, carry

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

if_else(*a*, *b*)

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

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

Load from memory by public address.

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

Allocate memory (statically).

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

max(*other*)

Maximum.

Parameters **other** – any compatible type

min(*other*)

Minimum.

Parameters **other** – any compatible type

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

Output.

reveal()

Identity.

square()

Square.

store_in_mem(*address*)

Store in memory by public address.

update(*other*)

Update register. Useful in loops like *for_range()*.

Parameters **other** – any convertible type

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

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

Parameters

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

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

Type-dependent array. Example:

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

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

Type-dependent matrix. Example:

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

classmethod Tensor(*shape*)

Type-dependent tensor of any dimension:

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

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

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

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

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

Binary adder in arithmetic circuits.

Parameters

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

Returns list of 0/1 in relevant type

bit_and(*other*)

AND in arithmetic circuits.

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

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

classmethod bit_compose(*bits*)

Compose value from bits.

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

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

Clear bit decomposition.

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

Returns list of cint

bit_not()

NOT in arithmetic circuits.

bit_or(*other*)

OR in arithmetic circuits.

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

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

bit_xor(*other*)

XOR in arithmetic circuits.

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

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

cond_swap(*a*, *b*)

Swapping in arithmetic circuits.

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

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

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

digest(**args*, ***kwargs*)

Clear hashing (libsodium default).

half_adder(*other*)

Half adder in arithmetic circuits.

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

Returns binary sum, carry

Return type depending on inputs, secret if any is

if_else(*a*, *b*)

MUX on bit in arithmetic circuits.

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

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

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

legendre(**args*, ***kwargs*)

Clear Legendre symbol computation.

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

Clear comparison for particular bit length.

Parameters

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

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

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

Load from memory by public address.

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

Allocate memory (statically).

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

max(*other*)

Maximum.

Parameters **other** – any compatible type

min(*other*)

Minimum.

Parameters **other** – any compatible type

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

Clear modulo a power of two.

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

print_if(*string*)

Output if value is non-zero.

Parameters **string** – Python string

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

Output.

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

Receive clear value(s) from client.

Parameters

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

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

reveal()

Identity.

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

Clear shift.

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

square()

Square.

store_in_mem(*address*)

Store in memory by public address.

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

Convert to regint.

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

Returns regint

update(*other*)

Update register. Useful in loops like *for_range()*.

Parameters **other** – any convertible type

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

Send a list of clear values to a client.

Parameters

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

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

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

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

output()

Output.

class Compiler.types.**personal**(*player*, *value*)

Value known to one player. Supports operations with public values and personal values known to the same player. Can be used with *print_ln_to()*.

Parameters

- **player** – player (int)

- **value** – cleartext value (cint, cfix, cfloat) or array thereof

binary_output()

Write binary output to `Player-Data/Binary-Output-P<playerno>-<threadno>` if supported by underlying type. Player must be known at compile time.

bit_decompose(*length=None*)

Bit decomposition.

Parameters **length** – number of bits

reveal_to(*player*)

Pass personal value to another player.

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

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

Parameters

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

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

Type-dependent array. Example:

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

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

Type-dependent matrix. Example:

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

classmethod Tensor(*shape*)

Type-dependent tensor of any dimension:

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

binary_output(*player=None*)

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

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

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

Binary adder in arithmetic circuits.

Parameters

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

Returns list of 0/1 in relevant type

`bit_and`(*other*)

AND in arithmetic circuits.

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

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

`static bit_compose`(*bits*)

Clear bit composition.

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

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

Clear bit decomposition.

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

Returns list of regint

`bit_not`()

NOT in arithmetic circuits.

`bit_or`(*other*)

OR in arithmetic circuits.

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

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

`bit_xor`(*other*)

XOR in arithmetic circuits.

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

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

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

Swapping in arithmetic circuits.

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

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

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

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

Public insecure randomness.

Parameters

- `bit_length` – number of bits (int)
- `size` – vector size (int, default 1)

`half_adder`(*other*)

Half adder in arithmetic circuits.

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

Returns binary sum, carry

Return type depending on inputs, secret if any is

if_else(*a*, *b*)

MUX on bit in arithmetic circuits.

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

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

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

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

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

Parameters

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

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

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

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

Load from memory by public address.

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

Allocate memory (statically).

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

max(*other*)

Maximum.

Parameters **other** – any compatible type

min(*other*)

Minimum.

Parameters **other** – any compatible type

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

Clear modulo a power of two.

Return type *cint*

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

Pop from stack.

print_if(*string*)

Output string if value is non-zero.

Parameters **string** – Python string

print_reg_plain()

Output.

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

Push to stack.

Parameters **value** – any convertible type

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

Receive clear integer value(s) from client.

Parameters

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

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

reveal()

Identity.

shuffle()

Returns insecure shuffle of vector.

square()

Square.

store_in_mem(*address*)

Store in memory by public address.

update(*other*)

Update register. Useful in loops like *for_range()*.

Parameters **other** – any convertible type

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

Send a list of clear integers to a client.

Parameters

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

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

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

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

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

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

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

Type-dependent array. Example:

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

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

Type-dependent matrix. Example:

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

classmethod Tensor(*shape*)

Type-dependent tensor of any dimension:

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

bit_decompose(*n_bits=None*)

Bit decomposition.

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

Secret fixed-point reciprocal.

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

Secret dot product.

Parameters

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

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

Secret fixed-point input.

Parameters

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

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

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

Parameters

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

classmethod input_tensor_from(*player*, *shape*)

Input tensor secretly from player.

Parameters

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

classmethod input_tensor_from_client(*client_id*, *shape*)

Input tensor secretly from client.

Parameters

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

classmethod input_tensor_via(*player*, *content*)

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

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

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

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

Load from memory by public address.

max(*other*)

Maximum.

Parameters **other** – any compatible type

min(*other*)

Minimum.

Parameters **other** – any compatible type

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

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

Parameters

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

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

Returns list of shares

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

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

Parameters

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

reveal()

Reveal secret fixed-point number.

Returns relevant clear type

reveal_to(*player*)

Reveal secret value to `player`.

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

Returns `personal`

classmethod reveal_to_clients(*clients*, *values*)

Reveal securely to clients.

Parameters

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

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

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

Parameters

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

square()

Square.

store_in_mem(*address*)

Store in memory by public address.

update(*other*)

Update register. Useful in loops like *for_range()*.

Parameters **other** – any convertible type

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

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

Parameters

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

classmethod write_to_file(*shares*, *position=None*)

Write shares of integer representation to `Persistence/Transactions-P<playerno>.data`.

Parameters

- **shares** – (list or iterable of sfix)
- **position** – start position (int/regint/cint), defaults to end of file

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

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

v: significand

p: exponent

z: zero flag

s: sign bit

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

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

This data type only works with arithmetic computation.

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

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

Type-dependent array. Example:

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

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

Type-dependent matrix. Example:

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

classmethod Tensor(*shape*)

Type-dependent tensor of any dimension:

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

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

Secret floating-point input.

Parameters

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

classmethod input_tensor_from(*player*, *shape*)

Input tensor secretly from player.

Parameters

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

classmethod input_tensor_from_client(*client_id*, *shape*)

Input tensor secretly from client.

Parameters

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

classmethod input_tensor_via(*player*, *content*)

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

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

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

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

Load from memory by public address.

max(*other*)

Maximum.

Parameters **other** – any compatible type

min(*other*)

Minimum.

Parameters **other** – any compatible type

reveal()

Reveal secret floating-point number.

Returns cfloat

round_to_int()

Secret floating-point rounding to integer.

Returns sint

square()

Square.

store_in_mem(*address*)

Store in memory by public address.

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

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

Parameters

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

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

Type-dependent array. Example:

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

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

Type-dependent matrix. Example:

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

classmethod Tensor(*shape*)

Type-dependent tensor of any dimension:

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

bit_and(*other*)

AND in binary circuits.

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

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

classmethod bit_compose(*bits*)

Compose value from bits.

Parameters **bits** – iterable of any type convertible to sint

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

Secret bit decomposition.

Parameters

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

Returns list of sgf2n

bit_not()

NOT in binary circuits.

bit_or(*other*)

OR in binary circuits.

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

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

bit_xor(*other*)

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

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

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

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

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

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

Secret dot product.

Parameters

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

Return type same as inputs

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

Secret comparison.

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

Returns 0/1 (sgf2n)

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

Secret input from player.

Parameters

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

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

Secret random bit according to security model.

Returns 0/1 50-50

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

classmethod get_random_input_mask_for(**args*, ***kwargs*)

Secret random input mask according to security model.

Returns mask (sint), mask (personal cint)

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

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

Secret random inverse tuple according to security model.

Returns (a, a^{-1})

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

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

Secret random square according to security model.

Returns (a, a^2)

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

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

Secret random triple according to security model.

Returns (a, b, ab)

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

half_adder(*other*)

Half adder in binary circuits.

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

Returns binary sum, carry

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

if_else(a, b)

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

classmethod input_tensor_from(*player*, *shape*)

Input tensor secretly from player.

Parameters

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

classmethod input_tensor_from_client(*client_id*, *shape*)

Input tensor secretly from client.

Parameters

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

classmethod input_tensor_via(*player*, *content*)

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

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

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

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

Load from memory by public address.

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

Allocate memory (statically).

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

max(*other*)

Maximum.

Parameters **other** – any compatible type

min(*other*)

Minimum.

Parameters **other** – any compatible type

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

Secret comparison.

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

Returns 0/1 (sgf2n)

raw_right_shift(**args*, ***kwargs*)

Local right shift in supported protocols. In integer-like protocols, the output is potentially off by one.

Parameters **length** – number of bits

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

Reveal secret value publicly.

Return type relevant clear type

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

Reveal secret value to `player`.

Parameters **player** – int

Returns `personal`

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

Secret right shift by public value:

Parameters

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

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

Secret square.

store_in_mem(*address*)

Store in memory by public address.

update(*other*)

Update register. Useful in loops like *for_range()*.

Parameters **other** – any convertible type

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

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

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

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

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

Parameters

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

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

Initializing from a *personal* value implies the relevant party inputting their value securely.

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 Tensor(*shape*)

Type-dependent tensor of any dimension:

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

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

Binary adder in arithmetic circuits.

Parameters

- **a** – summand (list of 0/1 in compatible type)

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

Returns list of 0/1 in relevant type

bit_and(*other*)

AND in arithmetic circuits.

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

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

classmethod bit_compose(*bits*)

Compose value from bits.

Parameters **bits** – iterable of any type convertible to sint

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

Secret bit decomposition.

bit_not()

NOT in arithmetic circuits.

bit_or(*other*)

OR in arithmetic circuits.

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

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

bit_xor(*other*)

XOR in arithmetic circuits.

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

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

cond_swap(*a*, *b*)

Swapping in arithmetic circuits.

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

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

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

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

Secret dot product.

Parameters

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

Return type same as inputs

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

Secret comparison (signed).

Parameters

- **other** – sint/cint/regint/int

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

Returns 0/1 (sintbit)

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

Bit in arithmetic and binary circuit according to security model

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

Bits in arithmetic and binary circuit

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

Secret input.

Parameters

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

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

Secret random ring element according to security model.

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

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

Secret random bit according to security model.

Returns 0/1 50-50

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

classmethod get_random_input_mask_for(**args*, ***kwargs*)

Secret random input mask according to security model.

Returns mask (sint), mask (personal cint)

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

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

Secret random n-bit number according to security model.

Parameters

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

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

Secret random inverse tuple according to security model.

Returns (a, a^{-1})

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

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

Secret random square according to security model.

Returns (a, a^2)

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

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

Secret random triple according to security model.

Returns (a, b, ab)

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

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

Secret comparison (signed).

Parameters

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

Returns 0/1 (sintbit)

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

Secret comparison (signed).

Parameters

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

Returns 0/1 (sintbit)

half_adder(*other*)

Half adder in arithmetic circuits.

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

Returns binary sum, carry

Return type depending on inputs, secret if any is

if_else(*a*, *b*)

MUX on bit in arithmetic circuits.

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

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

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

classmethod input_tensor_from(*player*, *shape*)

Input tensor secretly from player.

Parameters

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

classmethod input_tensor_from_client(*client_id*, *shape*)

Input tensor secretly from client.

Parameters

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

classmethod input_tensor_via(*player*, *content*)

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

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

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

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

Secret integer division.

Parameters

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

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

Secret left shift.

Parameters

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

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

Secret comparison (signed).

Parameters

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

Returns 0/1 (sintbit)

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

Secret comparison (signed).

Parameters

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

Returns 0/1 (sintbit)

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

Load from memory by public address.

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

Allocate memory (statically).

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

max(*other*)

Maximum.

Parameters **other** – any compatible type

min(*other*)

Minimum.

Parameters **other** – any compatible type

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

Secret modulo power of two.

Parameters

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

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

Secret comparison (signed).

Parameters

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

Returns 0/1 (sintbit)

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

Secret power of two.

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

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

Private integer division as per Veugen and Abspoel

Parameters

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

raw_right_shift(**args*, ***kwargs*)

Local right shift in supported protocols. In integer-like protocols, the output is potentially off by one.

Parameters **length** – number of bits

classmethod read_from_file(*start*, *n_items*)

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

Parameters

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

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

Returns list of shares

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

Receive secret-shared value(s) from client.

Parameters

- **client_id** – Client id (regint)

- **n** – number of values (default 1)
- **size** – vector size of values (default 1)

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

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

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

Parameters

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

Returns list of sint

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

Reveal secret value publicly.

Return type relevant clear type

reveal_to(*player*)

Reveal secret value to `player`.

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

Returns *personal*

classmethod reveal_to_clients(*clients*, *values*)

Reveal securely to clients.

Parameters

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

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

Secret right shift.

Parameters

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

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

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

Parameters

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

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

Secret square.

store_in_mem(*address*)

Store in memory by public address.

update(*other*)

Update register. Useful in loops like `for_range()`.

Parameters **other** – any convertible type

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

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

Parameters

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

static write_to_file(*shares*, *position=None*)

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

Parameters

- **shares** – (list or iterable of sint)
- **position** – start position (int/regint/cint), defaults to end of file

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

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

Parameters

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

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

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

1.4.2 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.

Computation using these types will always be executed as a binary circuit. See *Protocol Pairs* for the exact protocols.

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

Clear bits register. Helper type with limited functionality.

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

Type-dependent array. Example:

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

bit_and(*other*)

AND in binary circuits.

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

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

bit_not()

NOT in binary circuits.

bit_or(*other*)

OR in binary circuits.

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

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

bit_xor(*other*)

XOR in binary circuits.

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

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

classmethod get_type(*length*)

Returns a fixed-length type.

half_adder(*other*)

Half adder in binary circuits.

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

Returns binary sum, carry

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

if_else(*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.

update(*other*)

Update register. Useful in loops like *for_range()*.

Parameters **other** – any convertible type

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

Single secret bit.

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

Type-dependent array. Example:

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

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

Binary adder in binary circuits.

Parameters

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

- **get_carry** – add final carry to output

Returns list of 0/1 in relevant type

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_or(*other*)

OR in binary circuits.

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

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

bit_xor(*other*)

XOR in binary circuits.

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

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

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

Secret input from `player`.

Param player (int)

classmethod get_type(*length*)

Returns a fixed-length type.

half_adder(*other*)

Half adder in binary circuits.

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

Returns binary sum, carry

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

if_else(*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.

popcnt()

Population count / Hamming weight.

Returns *sbits* of required length

update(*other*)

Update register. Useful in loops like *for_range()*.

Parameters **other** – any convertible type

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

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

Example:

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

will output roughly:

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

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

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

Type-dependent array. Example:

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

add(**args*, ***kwargs*)

Secret fixed-point addition.

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

bit_decompose(*n_bits=None*)

Bit decomposition.

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

Secret fixed-point reciprocal.

classmethod get_input_from(*player*)

Secret input from `player`.

Param player (int)

max(*other*)

Maximum.

Parameters **other** – any compatible type

min(*other*)

Minimum.

Parameters **other** – any compatible type

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)

square()

Square.

store_in_mem(*address*)

Store in memory by public address.

update(*other*)

Update register. Useful in loops like *for_range()*.

Parameters **other** – any convertible type

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

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

Type-dependent array. Example:

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

add(**args*, ***kwargs*)

Secret fixed-point addition.

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

bit_decompose(*n_bits=None*)

Bit decomposition.

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

Secret fixed-point reciprocal.

classmethod get_input_from(*player*)

Secret input from `player`.

Param player (int)

max(*other*)

Maximum.

Parameters **other** – any compatible type

min(*other*)

Minimum.

Parameters **other** – any compatible type

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)

square()

Square.

store_in_mem(*address*)

Store in memory by public address.

update(*other*)

Update register. Useful in loops like *for_range()*.

Parameters **other** – any convertible type

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

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

Type-dependent array. Example:

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

classmethod bit_adder(*a*, *b*, *carry_in=0*, *get_carry=False*)

Binary adder in binary circuits.

Parameters

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

Returns list of 0/1 in relevant type

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_or(*other*)

OR in binary circuits.

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

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

bit_xor(*other*)

XOR in binary circuits.

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

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

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

Secret input from `player`.

Param player (int)

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

Returns a signed integer type with fixed length.

Parameters **n** – length

static half_adder(*a*, *b*)

Half adder in binary circuits.

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

Returns binary sum, carry

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

if_else(*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.

max(*other*)

Maximum.

Parameters **other** – any compatible type

min(*other*)

Minimum.

Parameters **other** – any compatible type

popcnt()

Population count / Hamming weight.

Returns *sbits* of required length

pow2(*k*)

Computer integer power of two.

Parameters **k** – bit length of input

square()

Square.

update(*other*)

Update register. Useful in loops like *for_range()*.

Parameters **other** – any convertible type

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

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
```

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_or(*other*)

OR in binary circuits.

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

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

bit_xor(*other*)

XOR in binary circuits.

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

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

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`.

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)

max(*other*)

Maximum.

Parameters **other** – any compatible type

min(*other*)

Minimum.

Parameters **other** – any compatible type

popcnt()

Population count / Hamming weight.

Returns *sbitintvec* of required length

pow2(*k*)

Computer integer power of two.

Parameters **k** – bit length of input

square()

Square.

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

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

Type-dependent array. Example:

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

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

Binary adder in binary circuits.

Parameters

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

Returns list of 0/1 in relevant type

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_or(*other*)

OR in binary circuits.

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

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

bit_xor(*other*)

XOR in binary circuits.

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

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

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

Secret input from `player`.

Param player (int)

classmethod get_type(*length*)

Returns a fixed-length type.

half_adder(*other*)

Half adder in binary circuits.

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

Returns binary sum, carry

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

if_else(*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

update(*other*)

Update register. Useful in loops like *for_range()*.

Parameters **other** – any convertible type

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

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]
```

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_or(*other*)

OR in binary circuits.

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

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

bit_xor(*other*)

XOR in binary circuits.

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

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

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`.

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)

popcnt()

Population count / Hamming weight.

Returns *sbitintvec* of required length

1.4.3 Compiler.library module

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

Compiler.library.**accept_client_connection**(*port*)

Accept client connection on specific port base.

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

Returns client id

Compiler.library.**break_loop**()

Break out of loop.

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

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

Parameters **name** – Name for identification (optional)

Compiler.library.**check_point**()

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

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

Crash virtual machine.

Parameters **condition** – crash if true (default: true)

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

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

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

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

Decorator to execute loop bodies consecutively. Arguments work as in Python `range()`, but they can be any public integer. Information has to be passed out via container types such as *Array* or using `update()`. 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):
```

(continues on next page)

(continued from previous page)

```
    a[i] = i
    x.update(x + 1)
```

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

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

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

Parameters

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

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

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

Parameters

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

Example:

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

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

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

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

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

Parameters

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

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

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

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

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

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

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

Parameters

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

Example:

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

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

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

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

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

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

This will output:

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

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

Returns the thread argument.

Compiler.library.**get_number_of_players**()

Returns the number of players

Return type *regint*

Compiler.library.**get_player_id**()

Returns player number

Return type *localint* (cannot be used for computation)

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

Returns the thread number.

`Compiler.library.get_threshold()`

The threshold is the maximal number of corrupted players.

Return type *regint*

`Compiler.library.if_(condition)`

Conditional execution without else block.

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

Usage:

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

`Compiler.library.if_e(condition)`

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

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

Usage:

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

`Compiler.library.listen_for_clients(port)`

Listen for clients on specific port base.

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

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

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

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

result = summer()
```

Parameters

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

`Compiler.library.map_sum_simple(n_threads, n_loops, type, size)`

Vectorized multi-threaded sum reduction. The following computes a 100 sums of ten squares in three threads:

```
@map_sum_simple(3, 10, sint, 100)
def summer(i):
    return sint(regint.inc(100, i, 0)) ** 2

result = summer()
```

Parameters

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

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

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

Parameters

- **n_threads** – compile-time (int)
- **n_items** – regint/cint/int (default: `n_threads`)
- **max_size** – maximum size to be processed at once (default: no limit)

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

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

`Compiler.library.print_float_precision(n)`

Set the precision for floating-point printing.

Parameters **n** – number of digits (int)

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

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

Parameters

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

Example:

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

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

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

Parameters

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

Example:

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

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

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

Parameters

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

Example:

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

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

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

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

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

Compiler.library.**public_input**()

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

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

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

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

Conditionally print an error message and abort the runtime.

Parameters

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

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

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

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

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

Stop timer. Fails if not running.

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

Compiler.library.**tree_reduce**(*function*, *sequence*)

Round-efficient reduction. The following computes the maximum of the list `l`:

```
m = tree_reduce(lambda x, y: x.max(y), l)
```

Parameters

- **function** – reduction function taking two arguments
- **sequence** – list, vector, or array

`Compiler.library.tree_reduce_multithread(n_threads, function, vector)`

Round-efficient reduction in several threads. The following code computes the maximum of an array in 10 threads:

```
tree_reduce_multithread(10, lambda x, y: x.max(y), a)
```

Parameters

- **n_threads** – number of threads (int)
- **function** – reduction function taking exactly two arguments
- **vector** – register vector or array

`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
```

1.4.4 Compiler.mpc_math module

Module for math operations.

Implements trigonometric and logarithmic functions.

This has to imported explicitly.

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

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

Parameters **x** – fractional input (sfix).

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

`Compiler.mpc_math.acos(x)`

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

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

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

`Compiler.mpc_math.asin(x)`

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

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

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

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

Returns the cosine of any given fractional value.

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

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

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

Power of two for fixed-point numbers.

Parameters

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

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

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

Reciprocal square root approximation by Lu et al.

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

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

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

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

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

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

Parameters

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

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

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

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

Parameters

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

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

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

Returns the sine of any given fractional value.

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

Returns sin of `x` (sfix, sfloat)

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

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

Parameters `x` – fractional input (sfix).

Returns square root of `x` (sfix).

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

Returns the tangent of any given fractional value.

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

Returns tan of `x` (sfix, sfloat)

`Compiler.mpc_math.tanh(x)`

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

1.4.5 Compiler.ml module

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

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

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

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

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

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

Inference can be run as follows:

```
data = sfix.Matrix(n_test, n_features)
data.input_from(0)
```

(continues on next page)

(continued from previous page)

```
res = sgd.eval(data)
print_ln('Results: %s', [x.reveal() for x in res])
```

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

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

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

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

Bases: *Compiler.ml.Optimizer*

Adam/AMSgrad optimizer.

Parameters

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

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

Bases: Compiler.ml.NoVariableLayer

Fixed-point addition layer.

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

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

Bases: Compiler.ml.NoVariableLayer

Fixed-point Argmax layer.

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

class Compiler.ml.**BatchNorm**(*shape*, *approx=True*, *args=None*)

Bases: Compiler.ml.Layer

Fixed-point batch normalization layer.

Parameters

- **shape** – input/output shape (tuple/list of four int)
- **approx** – use approximate square root

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

Bases: Compiler.ml.NoVariableLayer

Fixed-point concatentation layer.

Parameters

- **inputs** – two input layers (tuple/list)

- **dimension** – dimension for concatenation (must be 3)

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

Bases: `Compiler.ml.DenseBase`

Fixed-point dense (matrix multiplication) layer.

Parameters

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

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

Bases: `Compiler.ml.NoVariableLayer`

Dropout layer.

Parameters

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

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

Bases: `Compiler.ml.FixBase`, `Compiler.ml.AveragePool2d`

Fixed-point 2D AvgPool layer.

Parameters

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

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

Bases: `Compiler.ml.Conv2d`, `Compiler.ml.FixBase`

Fixed-point 2D convolution layer.

Parameters

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

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

Bases: Compiler.ml.Layer

Fixed-point fused batch normalization layer (inference only).

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

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

Bases: Compiler.ml.NoVariableLayer

Fixed-point MaxPool layer.

Parameters

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

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

Bases: Compiler.ml.MultiOutputBase

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

Parameters

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

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

Bases: object

Base class for graphs of layers.

backward(***kwargs*)

Compute backward propagation.

eval(***kwargs*)

Compute evaluation after training.

Parameters

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

forward(***kwargs*)

Compute graph.

Parameters

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

property layers

Get all layers.

reset()

Initialize weights.

run(***kwargs*)

Run training.

Parameters

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

set_layers_with_inputs(*layers*)

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

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

Bases: `Compiler.ml.NoVariableLayer`

Fixed-point logistic regression output layer.

Parameters

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

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

Bases: `Compiler.ml.ElementWiseLayer`

Fixed-point ReLU layer.

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

static f(*x*)

ReLU function (maximum of input and zero).

static f_prime(*x*)

ReLU derivative.

prime_type

alias of *Compiler.types.sint*

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

Bases: `Compiler.ml.MultiOutputBase`

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

Parameters

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

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

Bases: *Compiler.ml.Optimizer*

Stochastic gradient descent.

Parameters

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

reset(***kwargs*)

Reset layer parameters.

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

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

Bases: Compiler.ml.ElementWiseLayer

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 or 0 if **x** has length 1

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

Iterative matrix inverse approximation.

Parameters

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

Compiler.ml.**relu**(*x*)

ReLU function (maximum of input and zero).

Compiler.ml.**relu_prime**(*x*)

ReLU derivative.

Compiler.ml.**sigmoid**(*x*)

Sigmoid function.

Parameters **x** – sfix

Compiler.ml.**sigmoid_prime**(*x*)

Sigmoid derivative.

Parameters **x** – sfix

Compiler.ml.**softmax**(*x*)

Softmax.

Parameters **x** – vector or list of sfix

Returns sfix vector

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

Iterative linear solution approximation for $Ax = b$.

Parameters

- **progress** – print some information on the progress (implies revealing)

- **n_threads** – number of threads to use
- **stop** – whether to stop when converged (implies revealing)

Compiler.ml.**var**(*x*)

Variance.

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

Piece-wise approximate sigmoid as in Hong et al.

Parameters

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

1.4.6 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):
```

(continues on next page)

(continued from previous page)

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

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

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

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

`Compiler.circuit.sha3_256(x)`

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

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

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

```
0x4a43f8804b0ad882fa493be44dff80f562d661a05647c15166d71ebff8c6ffa7
0xf0d7aa0ab2d92d580bb080e17cbb52627932ba37f085d3931270d31c39357067
```

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

1.4.7 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'>, name=None)`

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

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

join_tapes(*thread_numbers*)

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

Parameters **thread_numbers** – list of thread numbers

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

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

```
def f():
    ...
def g():
    ...
```

(continues on next page)

(continued from previous page)

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

Parameters

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

Returns tape handle

options_from_args()

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

public_input(*x*)

Append a value to the public input file.

run_tapes(*args*)

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

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

Returns list of thread numbers

property security

The statistical security parameter for non-linear functions.

set_bit_length(*bit_length*)

Change the integer bit length for non-linear functions.

use_dabit

Setting whether to use daBits for non-linear functionality.

use_edabit(*change=None*)

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

Parameters **change** – change setting if not `None`

Returns setting if `change` is `None`

use_invperm(*change=None*)

Set whether to use the low-level INVPERM instruction to inverse a permutation (see sint.inverse_permutation). The INVPERM instruction assumes a semi-honest two-party environment. If false, a general protocol implemented in the high-level language is used.

Parameters **change** – change setting if not `None`

Returns setting if `change` is `None`

use_split(*change=None*)

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

Parameters **change** – change setting if not `None`

Returns setting if `change` is `None`

use_square(*change=None*)

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

Parameters **change** – change setting if not None

Returns setting if change is None

property use_trunc_pr

Setting whether to use special probabilistic truncation.

1.4.8 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 / sfix

1.5 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)
```

1.5.1 Instructions

The following table list all instructions except the ones for $\mathrm{GF}(2^n)$ computation, untested ones, and those considered obsolete.

Name	Code	
CISC	0x0	Meta instruction for emulation
LDI	0x1	Assign (constant) immediate value to clear register (vector)
LDSI	0x2	Assign (constant) immediate value to secret register (vector)
LDMC	0x3	Assign clear memory value(s) to clear register (vector) by immediate address
LDMS	0x4	Assign secret memory value(s) to secret register (vector) by immediate address
STMC	0x5	Assign clear register (vector) to clear memory value(s) by immediate address
STMS	0x6	Assign secret register (vector) to secret memory value(s) by immediate address
LDMCI	0x7	Assign clear memory value(s) to clear register (vector) by register address
LDMSI	0x8	Assign secret memory value(s) to secret register (vector) by register address
STMCI	0x9	Assign clear register (vector) to clear memory value(s) by register address
STMSI	0xa	Assign secret register (vector) to secret memory value(s) by register address
MOVC	0xb	Copy clear register (vector)
MOVS	0xc	Copy secret register (vector)
LDTN	0x10	Store the number of the current thread in clear integer register
LDARG	0x11	Store the argument passed to the current thread in clear integer register
REQBL	0x12	Requirement on computation modulus
STARG	0x13	Copy clear integer register to the thread argument

continues on next page

Table 1 – continued from previous page

Name	Code	
TIME	0x14	Output time since start of computation
START	0x15	Start timer
STOP	0x16	Stop timer
USE	0x17	Offline data usage
USE_INP	0x18	Input usage
RUN_TAPE	0x19	Start tape/bytecode file in another thread
JOIN_TAPE	0x1a	Join thread
CRASH	0x1b	Crash runtime if the value in the register is not zero
USE_PREP	0x1c	Custom preprocessed data usage
USE_MATMUL	0x1f	Matrix multiplication usage
ADDC	0x20	Clear addition
ADDS	0x21	Secret addition
ADDM	0x22	Mixed addition
ADDCI	0x23	Addition of clear register (vector) and (constant) immediate value
ADDSI	0x24	Addition of secret register (vector) and (constant) immediate value
SUBC	0x25	Clear subtraction
SUBS	0x26	Secret subtraction
SUBML	0x27	Subtract clear from secret value
SUBMR	0x28	Subtract secret from clear value
SUBCI	0x29	Subtraction of (constant) immediate value from clear register (vector)
SUBSI	0x2a	Subtraction of (constant) immediate value from secret register (vector)
SUBCFI	0x2b	Subtraction of clear register (vector) from (constant) immediate value
SUBSFI	0x2c	Subtraction of secret register (vector) from (constant) immediate value
MULC	0x30	Clear multiplication
MULM	0x31	Multiply secret and clear value
MULCI	0x32	Multiplication of clear register (vector) and (constant) immediate value
MULSI	0x33	Multiplication of secret register (vector) and (constant) immediate value
DIVC	0x34	Clear division
DIVCI	0x35	Division of secret register (vector) and (constant) immediate value
MODC	0x36	Clear modular reduction
MODCI	0x37	Modular reduction of clear register (vector) and (constant) immediate value
LEGENDREC	0x38	Clear Legendre symbol computation (a/p) over prime p (the computation modulus)
DIGESTC	0x39	Clear truncated hash computation
INV2M	0x3a	Inverse of power of two modulo prime (the computation modulus)
FLOORDIVC	0x3b	Clear integer floor division
TRIPLE	0x50	Store fresh random triple(s) in secret register (vectors)
BIT	0x51	Store fresh random triple(s) in secret register (vectors)
SQUARE	0x52	Store fresh random square(s) in secret register (vectors)
INV	0x53	Store fresh random inverse(s) in secret register (vectors)
INPUTMASK	0x56	Store fresh random input mask(s) in secret register (vector) and clear register (vector) of the relevant player
PREP	0x57	Store custom preprocessed data in secret register (vectors)
DABIT	0x58	Store fresh random daBit(s) in secret register (vectors)

continues on next page

Table 1 – continued from previous page

Name	Code	
EDABIT	0x59	Store fresh random loose edaBit(s) in secret register (vectors)
SEDABIT	0x5a	Store fresh random strict edaBit(s) in secret register (vectors)
RANDOMS	0x5b	Store fresh length-restricted random shares(s) in secret register (vectors)
INPUTMASKREG	0x5c	Store fresh random input mask(s) in secret register (vector) and clear register (vector) of the relevant player
RANDOMFULLS	0x5d	Store share(s) of a fresh secret random element in secret register (vectors)
READSOCKETC	0x63	Read a variable number of clear values in internal representation from socket for a specified client id and store them in clear registers
READSOCKETS	0x64	Read a variable number of secret shares (potentially with MAC) from a socket for a client id and store them in registers
WRITESOCKETS	0x66	Write a variable number of secret shares (potentially with MAC) from registers into a socket for a specified client id
READSOCKETINT	0x69	Read a variable number of 32-bit integers from socket for a specified client id and store them in clear integer registers
WRITESOCKETSHARE	0x6b	Write a variable number of shares (without MACs) from secret registers into socket for a specified client id
LISTEN	0x6c	Open a server socket on a party-specific port number and listen for client connections (non-blocking)
ACCEPTCLIENTCONNECTION	0x6d	Wait for a connection at the given port and write socket handle to clear integer register
CLOSECLIENTCONNECTION	0x6e	Close connection to client
ANDC	0x70	Logical AND of clear (vector) registers
XORC	0x71	Logical XOR of clear (vector) registers
ORC	0x72	Logical OR of clear (vector) registers
ANDCI	0x73	Logical AND of clear register (vector) and (constant) immediate value
XORCI	0x74	Logical XOR of clear register (vector) and (constant) immediate value
ORCI	0x75	Logical OR of clear register (vector) and (constant) immediate value
NOTC	0x76	Clear logical NOT of a constant number of bits of clear (vector) register
SHLC	0x80	Bitwise left shift of clear register (vector)
SHRC	0x81	Bitwise right shift of clear register (vector)
SHLCI	0x82	Bitwise left shift of clear register (vector) by (constant) immediate value
SHRCI	0x83	Bitwise right shift of clear register (vector) by (constant) immediate value
SHRSI	0x84	Bitwise right shift of secret register (vector) by (constant) immediate value
JMP	0x90	Unconditional relative jump in the bytecode (compile-time parameter)
JMPNZ	0x91	Conditional relative jump in the bytecode
JMPEQZ	0x92	Conditional relative jump in the bytecode
EQZC	0x93	Clear integer zero test
LTZC	0x94	Clear integer less than zero test
LTC	0x95	Clear integer less-than comparison
GTC	0x96	Clear integer greater-than comparison
EQC	0x97	Clear integer equality test

continues on next page

Table 1 – continued from previous page

Name	Code	
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
CONV2DS	0xac	Secret 2D convolution
PRIVATEOUTPUT	0xad	Private input from cint
CHECK	0xaf	Force MAC check in current thread and all idle thread if current thread is the main thread
PRINTREG	0xb1	Debugging output of clear register (vector)
RAND	0xb2	Store insecure random value of specified length in clear integer register (vector)
PRINTREGPLAIN	0xb3	Output clear register
PRINTCHR	0xb4	Output a single byte
PRINTSTR	0xb5	Output four bytes
PUBINPUT	0xb6	Store public input in clear register (vector)
PRINTFLOATPLAIN	0xbc	Output floating-number from clear registers
WRITEFILESHARE	0xbd	Write shares to `Persistence/Transactions-P<playerno>.data` (appending at the end)
READFILESHARE	0xbe	Read shares from `Persistence/Transactions-P<playerno>.data`
CONDPRINTSTR	0xbf	Conditionally output four bytes
CONVINT	0xc0	Convert clear integer register (vector) to clear register (vector)
CONVMODP	0xc1	Convert clear integer register (vector) to clear register (vector)
LDMINT	0xca	Assign clear integer memory value(s) to clear integer register (vector) by immediate address
STMINT	0xcb	Assign clear integer register (vector) to clear integer memory value(s) by immediate address
LDMINTI	0xcc	Assign clear integer memory value(s) to clear integer register (vector) by register address
STMINTI	0xcd	Assign clear integer register (vector) to clear integer memory value(s) by register address
PUSHINT	0xce	Pushes clear integer register to the thread-local stack
POPINT	0xcf	Pops from the thread-local stack to clear integer register
MOVINT	0xd0	Copy clear integer register (vector)
INCINT	0xd1	Create incremental clear integer vector
SHUFFLE	0xd2	Randomly shuffles clear integer vector with public randomness
PRINTFLOATPREC	0xe0	Set number of digits after decimal point for *print_float_plain*
CONDPRINTPLAIN	0xe1	Conditionally output clear register (with precision)

continues on next page

Table 1 – continued from previous page

Name	Code	
NPLAYERS	0xe2	Store number of players in clear integer register
THRESHOLD	0xe3	Store maximal number of corrupt players in clear integer register
PLAYERID	0xe4	Store current player number in clear integer register
USE_EDABIT	0xe5	edaBit usage
INTOUTPUT	0xe6	Binary integer output
FLOATOUTPUT	0xe7	Binary floating-point output
INPUTMIXED	0xf2	Store private input in secret registers (vectors)
INPUTMIXEDREG	0xf3	Store private input in secret registers (vectors)
RAWINPUT	0xf4	Store private input in secret registers (vectors)
INPUTPERSONAL	0xf5	Private input from cint
SENDPERSONAL	0xf6	Private input from cint
SECSHUFFLE	0xfa	Secure shuffling
GENSECSHUFFLE	0xfb	Generate secure shuffle to bit used several times
APPLYSHUFFLE	0xfc	Generate secure shuffle to bit used several times
DELSHUFFLE	0xfd	Delete secure shuffle
INVPERM	0xfe	Calculate the inverse permutation of a secret permutation
GMULBITC	0x136	Clear modular reduction
GMULBITM	0x137	Modular reduction of clear register (vector) and (constant) immediate value
XORS	0x200	Bitwise XOR of secret bit register vectors
XORM	0x201	Bitwise XOR of single secret and clear bit registers
ANDRS	0x202	Constant-vector AND of secret bit registers
BITDECS	0x203	Secret bit register decomposition
BITCOMS	0x204	Secret bit register decomposition
CONVSINT	0x205	Copy clear integer register to secret bit register
LDBITS	0x20a	Store immediate in secret bit register
ANDS	0x20b	Bitwise AND of secret bit register vector
TRANS	0x20c	Secret bit register vector transpose
BITB	0x20d	Copy fresh secret random bit to secret bit register
ANDM	0x20e	Bitwise AND of single secret and clear bit registers
NOTS	0x20f	Bitwise NOT of secret register vector
XORCBI	0x210	Bitwise XOR of single clear bit register and immediate
BITDECC	0x211	Secret bit register decomposition
NOTCB	0x212	Bitwise NOT of secret register vector
CONVCINT	0x213	Copy clear integer register to clear bit register
REVEAL	0x214	Reveal secret bit register vectors and copy result to clear bit register vectors
LDMCB	0x217	Copy clear bit memory cell with compile-time address to clear bit register
STMCB	0x218	Copy clear bit register to clear bit memory cell with compile-time address
XORCB	0x219	Bitwise XOR of two single clear bit registers
ADDCB	0x21a	Integer addition two single clear bit registers
ADDCBI	0x21b	Integer addition single clear bit register and immediate
MULCBI	0x21c	Integer multiplication single clear bit register and immediate
SHRCBI	0x21d	Right shift of clear bit register by immediate
SHLCBI	0x21e	Left shift of clear bit register by immediate
CONVCINTVEC	0x21f	Copy clear register vector by bit to clear bit register vectors
PRINTREGSIGNED	0x220	Signed output of clear bit register

continues on next page

Table 1 – continued from previous page

Name	Code	
PRINTREGB	0x221	Debug output of clear bit register
PRINTREGPLAINB	0x222	Output clear bit register
PRINTFLOATPLAINB	0x223	Output floating-number from clear bit registers
CONDPRINTSTRB	0x224	Conditionally output four bytes
CONVCBIT	0x230	Copy clear bit register to clear integer register
CONVCBITVEC	0x231	Copy clear bit register vector to clear register by bit
LDMSB	0x240	Copy secret bit memory cell with compile-time address to secret bit register
STMSB	0x241	Copy secret bit register to secret bit memory cell with compile-time address
LDMSBI	0x242	Copy secret bit memory cell with run-time address to secret bit register
STMSBI	0x243	Copy secret bit register to secret bit memory cell with run-time address
MOVSB	0x244	Copy secret bit register
INPUTB	0x246	Copy private input to secret bit register vectors
INPUTBVEC	0x247	Copy private input to secret bit registers bit by bit
SPLIT	0x248	Local share conversion
CONVCBIT2S	0x249	Copy clear bit register vector to secret bit register vector
LDMCBI	0x258	Copy clear bit memory cell with run-time address to clear bit register
STMCBI	0x259	Copy clear bit register to clear bit memory cell with run-time address

1.5.2 Compiler.instructions module

This module contains all instruction types for arithmetic computation and general control of the virtual machine such as control flow.

The parameter descriptions refer to the instruction arguments in the right order.

class `Compiler.instructions.`**`acceptclientconnection`**`(`**args*, ***kwargs*`)`

Wait for a connection at the given port and write socket handle to clear integer register.

Param client id destination (regint)

Param port number (regint)

class `Compiler.instructions.`**`addc`**`(`**args*, ***kwargs*`)`

Clear addition.

Param result (cint)

Param summand (cint)

Param summand (cint)

class `Compiler.instructions.`**`addci`**`(`**args*, ***kwargs*`)`

Addition of clear register (vector) and (constant) immediate value.

Param result (cint)

Param summand (cint)

Param summand (int)

class Compiler.instructions.**addint**(**args*, ***kwargs*)

Clear integer register (vector) addition.

Param result (regint)

Param summand (regint)

Param summand (regint)

op(*b*, /)

Same as a + b.

class Compiler.instructions.**addm**(**args*, ***kwargs*)

Mixed addition.

Param result (sint)

Param summand (sint)

Param summand (cint)

class Compiler.instructions.**adds**(**args*, ***kwargs*)

Secret addition.

Param result (sint)

Param summand (sint)

Param summand (sint)

class Compiler.instructions.**addsi**(**args*, ***kwargs*)

Addition of secret register (vector) and (constant) immediate value.

Param result (cint)

Param summand (cint)

Param summand (int)

class Compiler.instructions.**andc**(**args*, ***kwargs*)

Logical AND of clear (vector) registers.

Param result (cint)

Param operand (cint)

Param operand (cint)

class Compiler.instructions.**andci**(**args*, ***kwargs*)

Logical AND of clear register (vector) and (constant) immediate value.

Param result (cint)

Param operand (cint)

Param operand (int)

class Compiler.instructions.**applyshuffle**(**args*, ***kwargs*)

Generate secure shuffle to bit used several times.

Param destination (sint)

Param source (sint)

Param number of elements to be treated as one (int)

Param handle (regint)

Param reverse (0/1)

class `Compiler.instructions.`**`asm_open`**(**args*, ***kwargs*)

Reveal secret registers (vectors) to clear registers (vectors).

Param number of argument to follow (odd number)

Param check after opening (0/1)

Param destination (cint)

Param source (sint)

Param (repeat the last two)…

class `Compiler.instructions.`**`bit`**(**args*, ***kwargs*)

Store fresh random triple(s) in secret register (vectors).

Param destination (sint)

class `Compiler.instructions.`**`bitdecint`**(**args*, ***kwargs*)

Clear integer bit decomposition.

Param number of arguments to follow / number of bits minus one (int)

Param source (regint)

Param destination for least significant bit (regint)

Param (destination for one bit higher)…

class `Compiler.instructions.`**`check`**(**args*, ***kwargs*)

Force MAC check in current thread and all idle thread if current thread is the main thread.

class `Compiler.instructions.`**`cisc`**

Meta instruction for emulation. This instruction is only generated when using `-K` with `compile.py`. The header looks as follows:

Param number of arguments after name plus one

Param name (16 bytes, zero-padded)

Currently, the following names are supported:

LTZ Less than zero.

param number of arguments in this unit (must be 6)

param vector size

param result (sint)

param input (sint)

param bit length

param (ignored)

param (repeat)…

Trunc Truncation.

param number of arguments in this unit (must be 8)

param vector size

param result (sint)

param input (sint)

param bit length

param number of bits to truncate

param (ignored)

param 0 for unsigned or 1 for signed

param (repeat)…

FPDiv Fixed-point division. Division by zero results in zero without error.

param number of arguments in this unit (must be at least 7)

param vector size

param result (sint)

param dividend (sint)

param divisor (sint)

param (ignored)

param fixed-point precision

param (repeat)…

exp2_fx Fixed-point power of two.

param number of arguments in this unit (must be at least 6)

param vector size

param result (sint)

param exponent (sint)

param (ignored)

param fixed-point precision

param (repeat)…

log2_fx Fixed-point logarithm with base 2.

param number of arguments in this unit (must be at least 6)

param vector size

param result (sint)

param input (sint)

param (ignored)

param fixed-point precision

param (repeat)…

class `Compiler.instructions.`**`closeclientconnection`**(**args*, ***kwargs*)

Close connection to client.

Param client id (regint)

class Compiler.instructions.**cond_print_plain**(**args*, ***kwargs*)

Conditionally output clear register (with precision). Outputs $x \cdot 2^p$ where p is the precision.

Param condition (cint, no output if zero)

Param source (cint)

Param precision (cint)

class Compiler.instructions.**cond_print_str**(*cond*, *val*)

Conditionally output four bytes.

Param condition (cint, no output if zero)

Param four bytes (int)

class Compiler.instructions.**conv2ds**(**args*, ***kwargs*)

Secret 2D convolution.

Param result (sint vector in row-first order)

Param inputs (sint vector in row-first order)

Param weights (sint vector in row-first order)

Param output height (int)

Param output width (int)

Param input height (int)

Param input width (int)

Param weight height (int)

Param weight width (int)

Param stride height (int)

Param stride width (int)

Param number of channels (int)

Param padding height (int)

Param padding width (int)

Param batch size (int)

class Compiler.instructions.**convint**(**args*, ***kwargs*)

Convert clear integer register (vector) to clear register (vector).

Param destination (cint)

Param source (regint)

class Compiler.instructions.**convmodp**(**args*, ***kwargs*)

Convert clear integer register (vector) to clear register (vector). If the bit length is zero, the unsigned conversion is used, otherwise signed conversion is used. This makes a difference when computing modulo a prime p. Signed conversion of $p - 1$ results in -1 while signed conversion results in $(p - 1) \mod 2^{64}$.

Param destination (regint)

Param source (cint)

Param bit length (int)

class Compiler.instructions.**crash**(**args*, ***kwargs*)

Crash runtime if the value in the register is not zero.

Param Crash condition (regint)

class Compiler.instructions.**dabit**(**args*, ***kwargs*)

Store fresh random daBit(s) in secret register (vectors).

Param arithmetic part (sint)

Param binary part (sbit)

class Compiler.instructions.**delshuffle**(**args*, ***kwargs*)

Delete secure shuffle.

Param handle (regint)

class Compiler.instructions.**digestc**(**args*, ***kwargs*)

Clear truncated hash computation.

Param result (cint)

Param input (cint)

Param byte length of hash value used (int)

class Compiler.instructions.**divc**(**args*, ***kwargs*)

Clear division.

Param result (cint)

Param dividend (cint)

Param divisor (cint)

class Compiler.instructions.**divci**(**args*, ***kwargs*)

Division of secret register (vector) and (constant) immediate value.

Param result (cint)

Param dividend (cint)

Param divisor (int)

class Compiler.instructions.**divint**(**args*, ***kwargs*)

Clear integer register (element-wise vector) division with floor rounding.

Param result (regint)

Param dividend (regint)

Param divisor (regint)

op(*b*, /)

Same as a // b.

class Compiler.instructions.**dotprods**(**args*)

Dot product of secret registers (vectors). Note that the vectorized version works element-wise.

Param number of arguments to follow (int)

Param twice the dot product length plus two (I know…)

Param result (sint)

Param first factor (sint)

Param first factor (sint)

Param second factor (sint)

Param second factor (sint)

Param (remaining factors)…

Param (repeat from dot product length)…

`get_def()`

Return the set of registers that are written to in this instruction.

`get_used()`

Return the set of registers that are read in this instruction.

`class Compiler.instructions.edabit(*args, **kwargs)`

Store fresh random loose edaBit(s) in secret register (vectors). The length is the first argument minus one.

Param number of arguments to follow / number of bits plus two (int)

Param arithmetic (sint)

Param binary (sbit)

Param (binary)…

`class Compiler.instructions.eqc(*args, **kwargs)`

Clear integer equality test. The result is 1 if the operands are equal and 0 otherwise.

Param destination (regint)

Param first operand (regint)

Param second operand (regint)

`op(b, /)`

Same as a == b.

`class Compiler.instructions.eqzc(*args, **kwargs)`

Clear integer zero test. The result is 1 for true and 0 for false.

Param destination (regint)

Param operand (regint)

`class Compiler.instructions.floatoutput(*args, **kwargs)`

Binary floating-point output.

Param player (int)

Param significand (cint)

Param exponent (cint)

Param zero bit (cint)

Param sign bit (cint)

`class Compiler.instructions.floordivc(*args, **kwargs)`

Clear integer floor division.

Param result (cint)

Param dividend (cint)

Param divisor (cint)

class Compiler.instructions.**gensecshuffle**(**args*, ***kwargs*)

Generate secure shuffle to bit used several times.

Param destination (regint)

Param size (int)

class Compiler.instructions.**gtc**(**args*, ***kwargs*)

Clear integer greater-than comparison. The result is 1 if the first operand is greater and 0 otherwise.

Param destination (regint)

Param first operand (regint)

Param second operand (regint)

op(*b*, /)

Same as a > b.

class Compiler.instructions.**incint**(**args*, ***kwargs*)

Create incremental clear integer vector. For example, vector size 10, base 1, increment 2, repeat 3, and wrap 2 produces the following:

```
(1, 1, 1, 3, 3, 3, 1, 1, 1, 3)
```

This is because the first number is always the `base`, every number is repeated `repeat` times, after which `increment` is added, and after `wrap` increments the number returns to `base`.

Param destination (regint)

Param base (non-vector regint)

Param increment (int)

Param repeat (int)

Param wrap (int)

class Compiler.instructions.**inputfix**(**args*, ***kwargs*)

class Compiler.instructions.**inputfloat**(**args*, ***kwargs*)

class Compiler.instructions.**inputmask**(**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 (int)

class Compiler.instructions.**inputmaskreg**(**args*, ***kwargs*)

Store fresh random input mask(s) in secret register (vector) and clear register (vector) of the relevant player.

Param mask (sint)

Param mask (cint, player only)

Param player (regint)

class Compiler.instructions.**inputmixed**(*name*, **args*)

Store private input in secret registers (vectors). The input is read as integer or floating-point number and the latter is then converted to the internal representation using the given precision. This instruction uses compile-time player numbers.

Param number of arguments to follow (int)

Param type (0: integer, 1: fixed-point, 2: floating-point)

Param destination (sint)

Param destination (sint, only for floating-point)

Param destination (sint, only for floating-point)

Param destination (sint, only for floating-point)

Param fixed-point precision or precision of floating-point significand (int, not with integer)

Param input player (int)

Param (repeat from type parameter)...

class `Compiler.instructions.`**`inputmixedreg`**`(`*name*, **args*`)`

Store private input in secret registers (vectors). The input is read as integer or floating-point number and the latter is then converted to the internal representation using the given precision. This instruction uses run-time player numbers.

Param number of arguments to follow (int)

Param type (0: integer, 1: fixed-point, 2: floating-point)

Param destination (sint)

Param destination (sint, only for floating-point)

Param destination (sint, only for floating-point)

Param destination (sint, only for floating-point)

Param fixed-point precision or precision of floating-point significand (int, not with integer)

Param input player (regint)

Param (repeat from type parameter)...

class `Compiler.instructions.`**`inputpersonal`**`(`**args*`)`

Private input from cint.

Param vector size (int)

Param player (int)

Param destination (sint)

Param source (cint)

Param (repeat from vector size)...

class `Compiler.instructions.`**`intoutput`**`(`**args*, ***kwargs*`)`

Binary integer output.

Param player (int)

Param regint

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

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

Param result (cint)

Param exponent (int)

class Compiler.instructions.**inverse**(**args*, ***kwargs*)

Store fresh random inverse(s) in secret register (vectors).

Param value (sint)

Param inverse (sint)

class Compiler.instructions.**inverse_permutation**(**args*, ***kwargs*)

Calculate the inverse permutation of a secret permutation.

Param destination (sint)

Param source (sint)

class Compiler.instructions.**jmp**(**args*, ***kwargs*)

Unconditional relative jump in the bytecode (compile-time parameter). The parameter is added to the regular jump of one after every instruction. This means that a jump of 0 results in a no-op while a jump of -1 results in an infinite loop.

Param number of instructions (int)

class Compiler.instructions.**jmpeqz**(**args*, ***kwargs*)

Conditional relative jump in the bytecode. The parameter is added to the regular jump of one after every instruction. This means that a jump of 0 results in a no-op while a jump of -1 results in an infinite loop.

Param condition (regint, only jump if zero)

Param number of instructions (int)

class Compiler.instructions.**jmpi**(**args*, ***kwargs*)

Unconditional relative jump in the bytecode (run-time parameter). The parameter is added to the regular jump of one after every instruction. This means that a jump of 0 results in a no-op while a jump of -1 results in an infinite loop.

Param number of instructions (regint)

class Compiler.instructions.**jmpnz**(**args*, ***kwargs*)

Conditional relative jump in the bytecode. The parameter is added to the regular jump of one after every instruction. This means that a jump of 0 results in a no-op while a jump of -1 results in an infinite loop.

Param condition (regint, only jump if not zero)

Param number of instructions (int)

class Compiler.instructions.**join_tape**(**args*, ***kwargs*)

Join thread.

Param virtual machine thread number (int)

class Compiler.instructions.**ldarg**(**args*, ***kwargs*)

Store the argument passed to the current thread in clear integer register.

Param destination (regint)

class Compiler.instructions.**ldi**(**args*, ***kwargs*)

Assign (constant) immediate value to clear register (vector).

Param destination (cint)

Param value (int)

class Compiler.instructions.**ldint**(**args*, ***kwargs*)

Store (constant) immediate value in clear integer register (vector).

Param destination (regint)

Param immediate (int)

class Compiler.instructions.**ldmc**(**args*, ***kwargs*)

Assign clear memory value(s) to clear register (vector) by immediate address. The vectorized version starts at the base address and then iterates the memory address.

Param destination (cint)

Param memory address base (int)

class Compiler.instructions.**ldmci**(**args*, ***kwargs*)

Assign clear memory value(s) to clear register (vector) by register address. The vectorized version starts at the base address and then iterates the memory address.

Param destination (cint)

Param memory address base (regint)

direct

alias of *Compiler.instructions.ldmc*

class Compiler.instructions.**ldmint**(**args*, ***kwargs*)

Assign clear integer memory value(s) to clear integer register (vector) by immediate address. The vectorized version starts at the base address and then iterates the memory address.

Param destination (regint)

Param memory address base (int)

class Compiler.instructions.**ldminti**(**args*, ***kwargs*)

Assign clear integer memory value(s) to clear integer register (vector) by register address. The vectorized version starts at the base address and then iterates the memory address.

Param destination (regint)

Param memory address base (regint)

direct

alias of *Compiler.instructions.ldmint*

class Compiler.instructions.**ldms**(**args*, ***kwargs*)

Assign secret memory value(s) to secret register (vector) by immediate address. The vectorized version starts at the base address and then iterates the memory address.

Param destination (sint)

Param memory address base (int)

class Compiler.instructions.**ldmsi**(**args*, ***kwargs*)

Assign secret memory value(s) to secret register (vector) by register address. The vectorized version starts at the base address and then iterates the memory address.

Param destination (sint)

Param memory address base (regint)

direct

alias of *Compiler.instructions.ldms*

class Compiler.instructions.**ldsi**(**args*, ***kwargs*)

Assign (constant) immediate value to secret register (vector).

Param destination (sint)

Param value (int)

class Compiler.instructions.**ldtn**(**args*, ***kwargs*)

Store the number of the current thread in clear integer register.

Param destination (regint)

class Compiler.instructions.**legendrec**(**args*, ***kwargs*)

Clear Legendre symbol computation (a/p) over prime p (the computation modulus).

Param result (cint)

Param a (int)

class Compiler.instructions.**listen**(**args*, ***kwargs*)

Open a server socket on a party-specific port number and listen for client connections (non-blocking).

Param port number (regint)

class Compiler.instructions.**ltc**(**args*, ***kwargs*)

Clear integer less-than comparison. The result is 1 if the first operand is less and 0 otherwise.

Param destination (regint)

Param first operand (regint)

Param second operand (regint)

op(*b*, /)

Same as a < b.

class Compiler.instructions.**ltzc**(**args*, ***kwargs*)

Clear integer less than zero test. The result is 1 for true and 0 for false.

Param destination (regint)

Param operand (regint)

class Compiler.instructions.**matmuls**(**args*, ***kwargs*)

Secret matrix multiplication from registers. All matrices are represented as vectors in row-first order.

Param result (sint vector)

Param first factor (sint vector)

Param second factor (sint vector)

Param number of rows in first factor and result (int)

Param number of columns in first factor and rows in second factor (int)

Param number of columns in second factor and result (int)

class Compiler.instructions.**matmulsm**(**args*, ***kwargs*)

Secret matrix multiplication reading directly from memory.

Param result (sint vector in row-first order)

Param base address of first factor (regint value)

Param base address of second factor (regint value)

Param number of rows in first factor and result (int)

Param number of columns in first factor and rows in second factor (int)

Param number of columns in second factor and result (int)

Param rows of first factor to use (regint vector, length as number of rows in first factor)

Param columns of first factor to use (regint vector, length below)

Param rows of second factor to use (regint vector, length below)

Param columns of second factor to use (regint vector, length below)

Param number of columns of first / rows of second factor to use (int)

Param number of columns of second factor to use (int)

class `Compiler.instructions.`**`modc`**`(`**args*, ***kwargs*`)`

Clear modular reduction.

Param result (cint)

Param dividend (cint)

Param divisor (cint)

class `Compiler.instructions.`**`modci`**`(`**args*, ***kwargs*`)`

Modular reduction of clear register (vector) and (constant) immediate value.

Param result (cint)

Param dividend (cint)

Param divisor (int)

class `Compiler.instructions.`**`movc`**`(`**args*, ***kwargs*`)`

Copy clear register (vector).

Param destination (cint)

Param source (cint)

class `Compiler.instructions.`**`movint`**`(`**args*, ***kwargs*`)`

Copy clear integer register (vector).

Param destination (regint)

Param source (regint)

class `Compiler.instructions.`**`movs`**`(`**args*, ***kwargs*`)`

Copy secret register (vector).

Param destination (cint)

Param source (cint)

class `Compiler.instructions.`**`mulc`**`(`**args*, ***kwargs*`)`

Clear multiplication.

Param result (cint)

Param factor (cint)

Param factor (cint)

class Compiler.instructions.**mulci**(**args*, ***kwargs*)

Multiplication of clear register (vector) and (constant) immediate value.

Param result (cint)

Param factor (cint)

Param factor (int)

class Compiler.instructions.**mulint**(**args*, ***kwargs*)

Clear integer register (element-wise vector) multiplication.

Param result (regint)

Param factor (regint)

Param factor (regint)

op(*b*, /)

Same as a * b.

class Compiler.instructions.**mulm**(**args*, ***kwargs*)

Multiply secret and clear value.

Param result (sint)

Param factor (sint)

Param factor (cint)

class Compiler.instructions.**mulrs**(*res*, *x*, *y*)

Constant-vector multiplication of secret registers.

Param number of arguments to follow (multiple of four)

Param vector size (int)

Param result (sint)

Param vector factor (sint)

Param constant factor (sint)

Param (repeat the last four)…

get_def()

Return the set of registers that are written to in this instruction.

get_used()

Return the set of registers that are read in this instruction.

class Compiler.instructions.**muls**(**args*, ***kwargs*)

(Element-wise) multiplication of secret registers (vectors).

Param number of arguments to follow (multiple of three)

Param result (sint)

Param factor (sint)

Param factor (sint)

Param (repeat the last three)…

class Compiler.instructions.**mulsi**(**args*, ***kwargs*)

Multiplication of secret register (vector) and (constant) immediate value.

Param result (sint)

Param factor (sint)

Param factor (int)

class Compiler.instructions.**notc**(**args*, ***kwargs*)

Clear logical NOT of a constant number of bits of clear (vector) register.

Param result (cint)

Param operand (cint)

Param bit length (int)

class Compiler.instructions.**nplayers**(**args*, ***kwargs*)

Store number of players in clear integer register.

Param destination (regint)

class Compiler.instructions.**orc**(**args*, ***kwargs*)

Logical OR of clear (vector) registers.

Param result (cint)

Param operand (cint)

Param operand (cint)

class Compiler.instructions.**orci**(**args*, ***kwargs*)

Logical OR of clear register (vector) and (constant) immediate value.

Param result (cint)

Param operand (cint)

Param operand (int)

class Compiler.instructions.**personal_base**(**args*)

class Compiler.instructions.**playerid**(**args*, ***kwargs*)

Store current player number in clear integer register.

Param destination (regint)

class Compiler.instructions.**popint**(**args*, ***kwargs*)

Pops from the thread-local stack to clear integer register.

Param destination (regint)

class Compiler.instructions.**prep**(**args*, ***kwargs*)

Store custom preprocessed data in secret register (vectors).

Param number of arguments to follow (int)

Param tag (16 bytes / 4 units, cut off at first zero byte)

Param destination (sint)

Param (repeat destination)…

class Compiler.instructions.**print_char**(*ch*)

Output a single byte.

Param byte (int)

class Compiler.instructions.**print_char4**(*val*)

Output four bytes.

Param four bytes (int)

class Compiler.instructions.**print_float_plain**(**args*, ***kwargs*)

Output floating-number from clear registers.

Param significand (cint)

Param exponent (cint)

Param zero bit (cint, zero output if bit is one)

Param sign bit (cint, negative output if bit is one)

Param NaN (cint, regular number if zero)

class Compiler.instructions.**print_float_prec**(**args*, ***kwargs*)

Set number of digits after decimal point for `print_float_plain`.

Param number of digits (int)

class Compiler.instructions.**print_int**(**args*, ***kwargs*)

Output clear integer register.

Param source (regint)

class Compiler.instructions.**print_reg**(*reg*, *comment=''*)

Debugging output of clear register (vector).

Param source (cint)

Param comment (4 bytes / 1 unit)

class Compiler.instructions.**print_reg_plain**(**args*, ***kwargs*)

Output clear register.

Param source (cint)

class Compiler.instructions.**privateoutput**(**args*)

Private input from cint.

Param vector size (int)

Param player (int)

Param destination (cint)

Param source (sint)

Param (repeat from vector size)…

class Compiler.instructions.**pubinput**(**args*, ***kwargs*)

Store public input in clear register (vector).

Param destination (cint)

class Compiler.instructions.**pushint**(**args*, ***kwargs*)

Pushes clear integer register to the thread-local stack.

Param source (regint)

class Compiler.instructions.**rand**(**args*, ***kwargs*)

Store insecure random value of specified length in clear integer register (vector).

Param destination (regint)

Param length (regint)

class Compiler.instructions.**randomfulls**(**args*, ***kwargs*)

Store share(s) of a fresh secret random element in secret register (vectors).

Param destination (sint)

class Compiler.instructions.**randoms**(**args*, ***kwargs*)

Store fresh length-restricted random shares(s) in secret register (vectors). This is only implemented for protocols that also implement local share conversion with `split`.

Param destination (sint)

Param length (int)

class Compiler.instructions.**rawinput**(**args*, ***kwargs*)

Store private input in secret registers (vectors). The input is read in the internal binary format according to the protocol.

Param number of arguments to follow (multiple of two)

Param player number (int)

Param destination (sint)

class Compiler.instructions.**readsharesfromfile**(**args*, ***kwargs*)

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

Param number of arguments to follow / number of shares plus two (int)

Param starting position in number of shares from beginning (regint)

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

Param destination for share (sint)

Param (repeat from destination for share)…

class Compiler.instructions.**readsocketc**(**args*, ***kwargs*)

Read a variable number of clear values in internal representation from socket for a specified client id and store them in clear registers.

Param number of arguments to follow / number of inputs minus one (int)

Param client id (regint)

Param vector size (int)

Param destination (cint)

Param (repeat destination)…

class `Compiler.instructions.`**`readsocketint`**`(`**args*, ***kwargs*`)`

Read a variable number of 32-bit integers from socket for a specified client id and store them in clear integer registers.

Param number of arguments to follow / number of inputs minus one (int)

Param client id (regint)

Param vector size (int)

Param destination (regint)

Param (repeat destination)…

class `Compiler.instructions.`**`readsockets`**`(`**args*, ***kwargs*`)`

Read a variable number of secret shares (potentially with MAC) from a socket for a client id and store them in registers. If the protocol uses MACs, the client should be different for every party.

Param client id (regint)

Param vector size (int)

Param source (sint)

Param (repeat source)…

class `Compiler.instructions.`**`reqbl`**`(`**args*, ***kwargs*`)`

Requirement on computation modulus. Minimal bit length of prime if positive, minus exact bit length of power of two if negative.

Param requirement (int)

class `Compiler.instructions.`**`run_tape`**`(`**args*, ***kwargs*`)`

Start tape/bytecode file in another thread.

Param number of arguments to follow (multiple of three)

Param virtual machine thread number (int)

Param tape number (int)

Param tape argument (int)

Param (repeat the last three)…

class `Compiler.instructions.`**`secshuffle`**`(`**args*, ***kwargs*`)`

Secure shuffling.

Param destination (sint)

Param source (sint)

class `Compiler.instructions.`**`sedabit`**`(`**args*, ***kwargs*`)`

Store fresh random strict edaBit(s) in secret register (vectors). The length is the first argument minus one.

Param number of arguments to follow / number of bits plus two (int)

Param arithmetic (sint)

Param binary (sbit)

Param (binary)…

class Compiler.instructions.**sendpersonal**(**args*)

Private input from cint.

Param vector size (int)

Param destination player (int)

Param destination (cint)

Param source player (int)

Param source (cint)

Param (repeat from vector size)…

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

Bitwise left shift of clear register (vector).

Param result (cint)

Param first operand (cint)

Param second operand (cint)

class Compiler.instructions.**shlci**(**args*, ***kwargs*)

Bitwise left shift of clear register (vector) by (constant) immediate value.

Param result (cint)

Param first operand (cint)

Param second operand (int)

class Compiler.instructions.**shrc**(**args*, ***kwargs*)

Bitwise right shift of clear register (vector).

Param result (cint)

Param first operand (cint)

Param second operand (cint)

class Compiler.instructions.**shrci**(**args*, ***kwargs*)

Bitwise right shift of clear register (vector) by (constant) immediate value.

Param result (cint)

Param first operand (cint)

Param second operand (int)

class Compiler.instructions.**shrsi**(**args*, ***kwargs*)

Bitwise right shift of secret register (vector) by (constant) immediate value. This only makes sense in connection with protocols allowing local share conversion (i.e., based on additive secret sharing modulo a power of two). Moreover, the result is not a secret sharing of the right shift of the secret value but needs to be corrected using the overflow. This is explained by Dalskov et al. in the appendix.

Param result (sint)

Param first operand (sint)

Param second operand (int)

class Compiler.instructions.**shuffle**(**args*, ***kwargs*)

Randomly shuffles clear integer vector with public randomness.

Param destination (regint)

Param source (regint)

class Compiler.instructions.**shuffle_base**(**args*, ***kwargs*)

class Compiler.instructions.**square**(**args*, ***kwargs*)

Store fresh random square(s) in secret register (vectors).

Param value (sint)

Param square (sint)

class Compiler.instructions.**starg**(**args*, ***kwargs*)

Copy clear integer register to the thread argument.

Param source (regint)

class Compiler.instructions.**start**(**args*, ***kwargs*)

Start timer.

Param timer number (int)

class Compiler.instructions.**stmc**(**args*, ***kwargs*)

Assign clear register (vector) to clear memory value(s) by immediate address. The vectorized version starts at the base address and then iterates the memory address.

Param source (cint)

Param memory address base (int)

class Compiler.instructions.**stmci**(**args*, ***kwargs*)

Assign clear register (vector) to clear memory value(s) by register address. The vectorized version starts at the base address and then iterates the memory address.

Param source (cint)

Param memory address base (regint)

direct

alias of *Compiler.instructions.stmc*

class Compiler.instructions.**stmint**(**args*, ***kwargs*)

Assign clear integer register (vector) to clear integer memory value(s) by immediate address. The vectorized version starts at the base address and then iterates the memory address.

Param source (regint)

Param memory address base (int)

class Compiler.instructions.**stminti**(**args*, ***kwargs*)

Assign clear integer register (vector) to clear integer memory value(s) by register address. The vectorized version starts at the base address and then iterates the memory address.

Param source (regint)

Param memory address base (regint)

direct

alias of *Compiler.instructions.stmint*

class Compiler.instructions.**stms**(**args*, ***kwargs*)

Assign secret register (vector) to secret memory value(s) by immediate address. The vectorized version starts at the base address and then iterates the memory address.

Param source (sint)

Param memory address base (int)

class Compiler.instructions.**stmsi**(**args*, ***kwargs*)

Assign secret register (vector) to secret memory value(s) by register address. The vectorized version starts at the base address and then iterates the memory address.

Param source (sint)

Param memory address base (regint)

direct

alias of *Compiler.instructions.stms*

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

Stop timer.

Param timer number (int)

class Compiler.instructions.**subc**(**args*, ***kwargs*)

Clear subtraction.

Param result (cint)

Param first operand (cint)

Param second operand (cint)

class Compiler.instructions.**subcfi**(**args*, ***kwargs*)

Subtraction of clear register (vector) from (constant) immediate value.

Param result (cint)

Param first operand (int)

Param second operand (cint)

class Compiler.instructions.**subci**(**args*, ***kwargs*)

Subtraction of (constant) immediate value from clear register (vector).

Param result (cint)

Param first operand (cint)

Param second operand (int)

class Compiler.instructions.**subint**(**args*, ***kwargs*)

Clear integer register (vector) subtraction.

Param result (regint)

Param first operand (regint)

Param second operand (regint)

op(*b*, /)

Same as a - b.

class Compiler.instructions.**subml**(**args*, ***kwargs*)

Subtract clear from secret value.

Param result (sint)

Param first operand (sint)

Param second operand (cint)

class Compiler.instructions.**submr**(**args*, ***kwargs*)

Subtract secret from clear value.

Param result (sint)

Param first operand (cint)

Param second operand (sint)

class Compiler.instructions.**subs**(**args*, ***kwargs*)

Secret subtraction.

Param result (sint)

Param first operand (sint)

Param second operand (sint)

class Compiler.instructions.**subsfi**(**args*, ***kwargs*)

Subtraction of secret register (vector) from (constant) immediate value.

Param result (sint)

Param first operand (int)

Param second operand (sint)

class Compiler.instructions.**subsi**(**args*, ***kwargs*)

Subtraction of (constant) immediate value from secret register (vector).

Param result (sint)

Param first operand (sint)

Param second operand (int)

class Compiler.instructions.**threshold**(**args*, ***kwargs*)

Store maximal number of corrupt players in clear integer register.

Param destination (regint)

class Compiler.instructions.**time**(**args*, ***kwargs*)

Output time since start of computation.

class Compiler.instructions.**triple**(**args*, ***kwargs*)

Store fresh random triple(s) in secret register (vectors).

Param factor (sint)

Param factor (sint)

Param product (sint)

class `Compiler.instructions.`**`trunc_pr`**(**args*, ***kwargs*)

Probabilistic truncation if supported by the protocol.

Param number of arguments to follow (multiple of four)

Param destination (sint)

Param source (sint)

Param bit length of source (int)

Param number of bits to truncate (int)

class `Compiler.instructions.`**`use`**(**args*, ***kwargs*)

Offline data usage. Necessary to avoid reusage while using preprocessing from files. Also used to multithreading for expensive preprocessing.

Param domain (0: integer, 1: $\mathrm{GF}(2^n)$, 2: bit)

Param type (0: triple, 1: square, 2: bit, 3: inverse, 6: daBit)

Param number (int, -1 for unknown)

class `Compiler.instructions.`**`use_edabit`**(**args*, ***kwargs*)

edaBit usage. Necessary to avoid reusage while using preprocessing from files. Also used to multithreading for expensive preprocessing.

Param loose/strict (0/1)

Param length (int)

Param number (int, -1 for unknown)

class `Compiler.instructions.`**`use_inp`**(**args*, ***kwargs*)

Input usage. Necessary to avoid reusage while using preprocessing from files.

Param domain (0: integer, 1: $\mathrm{GF}(2^n)$, 2: bit)

Param input player (int)

Param number (int, -1 for unknown)

class `Compiler.instructions.`**`use_matmul`**(**args*, ***kwargs*)

Matrix multiplication usage. Used for multithreading of preprocessing.

Param number of left-hand rows (int)

Param number of left-hand columns/right-hand rows (int)

Param number of right-hand columns (int)

Param number (int, -1 for unknown)

class `Compiler.instructions.`**`use_prep`**(**args*, ***kwargs*)

Custom preprocessed data usage.

Param tag (16 bytes / 4 units, cut off at first zero byte)

Param number of items to use (int, -1 for unknown)

class `Compiler.instructions.`**`writesharestofile`**(**args*, ***kwargs*)

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

Param number of arguments to follow / number of shares plus one (int)

Param position (regint, -1 for appending)

Param source (sint)

Param (repeat from source)…

class `Compiler.instructions.`**`writesockets`**`(`**args*, ***kwargs*`)`

Write a variable number of secret shares (potentially with MAC) from registers into a socket for a specified client id. If the protocol uses MACs, the client should be different for every party.

Param number of arguments to follow

Param client id (regint)

Param message type (must be 0)

Param vector size (int)

Param source (sint)

Param (repeat source)…

class `Compiler.instructions.`**`writesocketshare`**`(`**args*, ***kwargs*`)`

Write a variable number of shares (without MACs) from secret registers into socket for a specified client id.

Param client id (regint)

Param message type (must be 0)

Param vector size (int)

Param source (sint)

Param (repeat source)…

class `Compiler.instructions.`**`xorc`**`(`**args*, ***kwargs*`)`

Logical XOR of clear (vector) registers.

Param result (cint)

Param operand (cint)

Param operand (cint)

class `Compiler.instructions.`**`xorci`**`(`**args*, ***kwargs*`)`

Logical XOR of clear register (vector) and (constant) immediate value.

Param result (cint)

Param operand (cint)

Param operand (int)

1.5.3 Compiler.GC.instructions module

This module constrains instructions for binary circuits. Unlike arithmetic instructions, they generally do not use the vector size in the instruction code field. Instead the number of bits affected is given as an extra argument. Also note that a register holds 64 values instead of just one as is the case for arithmetic instructions. Therefore, an instruction for 65-128 bits will affect two registers etc. Similarly, a memory cell holds 64 bits.

class `Compiler.GC.instructions.`**`addcb`**`(`**args*, ***kwargs*`)`

Integer addition two single clear bit registers.

Param result (cbit)

Param summand (cbit)

Param summand (cbit)

class `Compiler.GC.instructions.`**`addcbi`**(**args*, ***kwargs*)

Integer addition single clear bit register and immediate.

Param result (cbit)

Param summand (cbit)

Param summand (int)

class `Compiler.GC.instructions.`**`andm`**(**args*, ***kwargs*)

Bitwise AND of single secret and clear bit registers.

Param number of bits (less or equal 64)

Param result (sbit)

Param operand (sbit)

Param operand (cbit)

class `Compiler.GC.instructions.`**`andrs`**(**args*, ***kwargs*)

Constant-vector AND of secret bit registers.

Param number of arguments to follow (multiple of four)

Param number of bits (int)

Param result vector (sbit)

Param vector operand (sbit)

Param single operand (sbit)

Param (repeat from number of bits)…

class `Compiler.GC.instructions.`**`ands`**(**args*, ***kwargs*)

Bitwise AND of secret bit register vector.

Param number of arguments to follow (multiple of four)

Param number of bits (int)

Param result (sbit)

Param operand (sbit)

Param operand (sbit)

Param (repeat from number of bits)…

class `Compiler.GC.instructions.`**`bitb`**(**args*, ***kwargs*)

Copy fresh secret random bit to secret bit register.

Param destination (sbit)

class `Compiler.GC.instructions.`**`bitcoms`**(**args*, ***kwargs*)

Secret bit register decomposition.

Param number of arguments to follow / number of bits plus one (int)

Param destination (sbit)

Param source for least significant bit (sbit)

Param (source for one bit higher)…

class Compiler.GC.instructions.**bitdecc**(**args*, ***kwargs*)

Secret bit register decomposition.

Param number of arguments to follow / number of bits plus one (int)

Param source (sbit)

Param destination for least significant bit (sbit)

Param (destination for one bit higher)…

class Compiler.GC.instructions.**bitdecs**(**args*, ***kwargs*)

Secret bit register decomposition.

Param number of arguments to follow / number of bits plus one (int)

Param source (sbit)

Param destination for least significant bit (sbit)

Param (destination for one bit higher)…

class Compiler.GC.instructions.**cond_print_strb**(*cond*, *val*)

Conditionally output four bytes.

Param condition (cbit, no output if zero)

Param four bytes (int)

class Compiler.GC.instructions.**convcbit**(**args*, ***kwargs*)

Copy clear bit register to clear integer register.

Param destination (regint)

Param source (cbit)

class Compiler.GC.instructions.**convcbit2s**(**args*, ***kwargs*)

Copy clear bit register vector to secret bit register vector.

Param number of bits (int)

Param destination (sbit)

Param source (cbit)

class Compiler.GC.instructions.**convcbitvec**(**args*)

Copy clear bit register vector to clear register by bit. This means that every element of the destination register vector will hold one bit.

Param number of bits / vector length (int)

Param destination (regint)

Param source (cbit)

class Compiler.GC.instructions.**convcint**(**args*, ***kwargs*)

Copy clear integer register to clear bit register.

Param number of bits (int)

Param destination (cbit)

Param source (regint)

class Compiler.GC.instructions.**convcintvec**(**args*, ***kwargs*)

Copy clear register vector by bit to clear bit register vectors. This means that the first destination will hold the least significant bits of all inputs etc.

Param number of arguments to follow / number of bits plus one (int)

Param source (cint)

Param destination for least significant bits (sbit)

Param (destination for bits one step higher)…

class Compiler.GC.instructions.**convsint**(**args*, ***kwargs*)

Copy clear integer register to secret bit register.

Param number of bits (int)

Param destination (sbit)

Param source (regint)

class Compiler.GC.instructions.**inputb**(**args*, ***kwargs*)

Copy private input to secret bit register vectors. The input is read as floating-point number, multiplied by a power of two, and then rounded to an integer.

Param number of arguments to follow (multiple of four)

Param player number (int)

Param number of bits in output (int)

Param exponent to power of two factor (int)

Param destination (sbit)

class Compiler.GC.instructions.**inputbvec**(**args*, ***kwargs*)

Copy private input to secret bit registers bit by bit. The input is read as floating-point number, multiplied by a power of two, rounded to an integer, and then decomposed into bits.

Param total number of arguments to follow (int)

Param number of arguments to follow for one input / number of bits plus three (int)

Param exponent to power of two factor (int)

Param player number (int)

Param destination for least significant bit (sbit)

Param (destination for one bit higher)…

Param (repeat from number of arguments to follow for one input)…

class Compiler.GC.instructions.**ldbits**(**args*, ***kwargs*)

Store immediate in secret bit register.

Param destination (sbit)

Param number of bits (int)

Param immediate (int)

class Compiler.GC.instructions.**ldmcb**(**args*, ***kwargs*)

Copy clear bit memory cell with compile-time address to clear bit register.

Param destination (cbit)

Param memory address (int)

class Compiler.GC.instructions.**ldmcbi**(**args*, ***kwargs*)

Copy clear bit memory cell with run-time address to clear bit register.

Param destination (cbit)

Param memory address (regint)

direct

alias of *Compiler.GC.instructions.ldmcb*

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)

direct

alias of *Compiler.GC.instructions.ldmsb*

class Compiler.GC.instructions.**movsb**(**args*, ***kwargs*)

Copy secret bit register.

Param destination (sbit)

Param source (sbit)

class Compiler.GC.instructions.**mulcbi**(**args*, ***kwargs*)

Integer multiplication single clear bit register and immediate.

Param result (cbit)

Param factor (cbit)

Param factor (int)

class Compiler.GC.instructions.**notcb**(**args*, ***kwargs*)

Bitwise NOT of secret register vector.

Param number of bits

Param result (cbit)

Param operand (cbit)

class Compiler.GC.instructions.**nots**(**args*, ***kwargs*)

Bitwise NOT of secret register vector.

Param number of bits (less or equal 64)

Param result (sbit)

Param operand (sbit)

class Compiler.GC.instructions.**print_float_plainb**(**args*, ***kwargs*)

Output floating-number from clear bit registers.

Param significand (cbit)

Param exponent (cbit)

Param zero bit (cbit, zero output if bit is one)

Param sign bit (cbit, negative output if bit is one)

Param NaN (cbit, regular number if zero)

class Compiler.GC.instructions.**print_reg_plainb**(**args*, ***kwargs*)

Output clear bit register.

Param source (cbit)

class Compiler.GC.instructions.**print_reg_signed**(**args*, ***kwargs*)

Signed output of clear bit register.

Param bit length (int)

Param source (cbit)

class Compiler.GC.instructions.**print_regb**(*reg*, *comment=''*)

Debug output of clear bit register.

Param source (cbit)

Param comment (4 bytes / 1 unit)

class Compiler.GC.instructions.**reveal**(**args*, ***kwargs*)

Reveal secret bit register vectors and copy result to clear bit register vectors.

Param number of arguments to follow (multiple of three)

Param number of bits (int)

Param destination (cbit)

Param source (sbit)

Param (repeat from number of bits)…

class Compiler.GC.instructions.**shlcbi**(**args*, ***kwargs*)

Left shift of clear bit register by immediate.

Param destination (cbit)

Param source (cbit)

Param number of bits to shift (int)

class Compiler.GC.instructions.**shrcbi**(**args*, ***kwargs*)

Right shift of clear bit register by immediate.

Param destination (cbit)

Param source (cbit)

Param number of bits to shift (int)

class Compiler.GC.instructions.**split**(**args*, ***kwargs*)

Local share conversion. This instruction use the vector length in the instruction code field.

Param number of arguments to follow (number of bits times number of additive shares plus one)

Param source (sint)

Param first share of least significant bit

Param second share of least significant bit

Param (remaining share of least significant bit)…

Param (repeat from first share for bit one step higher)…

class Compiler.GC.instructions.**stmcb**(**args*, ***kwargs*)

Copy clear bit register to clear bit memory cell with compile-time address.

Param source (cbit)

Param memory address (int)

class Compiler.GC.instructions.**stmcbi**(**args*, ***kwargs*)

Copy clear bit register to clear bit memory cell with run-time address.

Param source (cbit)

Param memory address (regint)

direct

alias of *Compiler.GC.instructions.stmcb*

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)

direct

alias of *Compiler.GC.instructions.stmsb*

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)...

1.6 Low-Level Interface

In the following we will explain the basic of the C++ interface by walking trough `Utils/paper-example.cpp`.

```
template<class T>
void run(char** argv, int prime_length);
```

MP-SPDZ heavily uses templating to allow to reuse code between different protocols. `run()` is a simple example of this. The entire virtual machine in the `Processor` directory is built on the same principle. The central type is a type representing a share in a particular type.

```
// bit length of prime
const int prime_length = 128;

// compute number of 64-bit words needed
const int n_limbs = (prime_length + 63) / 64;
```

Computation modulo a prime requires to fix the number of limbs (64-bit words) at compile time. This allows for optimal memory usage and computation.

```
if (protocol == "MASCOT")
    run<Share<gfp_<0, n_limbs>>>(argv, prime_length);
else if (protocol == "CowGear")
    run<CowGearShare<gfp_<0, n_limbs>>>(argv, prime_length);
```

Share types for computation module a prime (and in $\mathrm{GF}(2^n)$) generally take one parameter for the computation domain. *gfp_* in turn takes two parameters, a counter and the number of limbs. The counter allows to use several instances with different parameters. It can be chosen freely, but the convention is to use 0 for the online phase and 1 for the offline phase where required.

```
else if (protocol == "SPDZ2k")
    run<Spdz2kShare<64, 64>>(argv, 0);
```

Share types for computation modulo a power of two simply take the exponent as parameter, and some take an additional security parameter.

```
int my_number = atoi(argv[1]);
int n_parties = atoi(argv[2]);
int port_base = 9999;
Names N(my_number, n_parties, "localhost", port_base);
```

All implemented protocols require point-to-point connections between all parties. *Names* objects represent a setup of hostnames and IPs used to set up the actual connections. The chosen initialization provides a way where every party connects to party 0 on a specified location (localhost in this case), which then broadcasts the locations of all parties. The base port number is used to derive the port numbers for the parties to listen on (base + party number). See the the *Names* class for other possibilities such as a text file containing hostname and port number for each party.

```
CryptoPlayer P(N);
```

The networking setup is used to set up the actual connections. *CryptoPlayer* uses encrypted connection while *PlainPlayer* does not. If you use several instances (for several threads for example), you must use an integer identifier as the second parameter, which must differ from any other by at least the number of parties.

```
ProtocolSetup<T> setup(P, prime_length);
```

We have to use a specific prime for computation modulo a prime. This deterministically generates one of the desired length if necessary. For computation modulo a power of two, this does not do anything. Some protocols use an information-theoretic tag that is constant throughout the protocol. This code reads it from storage if available or generates a fresh one otherwise.

```
ProtocolSet<T> set(P, setup);
auto& input = set.input;
auto& protocol = set.protocol;
auto& output = set.output;
```

The `ProtocolSet` contains one instance for every essential protocol step.

```
int n = 1000;
vector<T> a(n), b(n);
T c;
typename T::clear result;
```

Remember that `T` stands for a share in the protocol. The derived type `T::clear` stands for the cleartext domain. Share types support linear operations such as addition, subtraction, and multiplication with a constant. Use `T::constant()` to convert a constant to a share type.

```
input.reset_all(P);
for (int i = 0; i < n; i++)
    input.add_from_all(i);
input.exchange();
for (int i = 0; i < n; i++)
{
    a[i] = input.finalize(0);
    b[i] = input.finalize(1);
}
```

The interface for all protocols proceeds in four stages:

1. Initialization. This is required to initialize and reset data structures in consecutive use.
2. Local data preparation
3. Communication
4. Output extraction

This blueprint allows for a minimal number of communication rounds.

```
protocol.init_dotprod(&processor);
for (int i = 0; i < n; i++)
    protocol.prepare_dotprod(a[i], b[i]);
protocol.next_dotprod();
protocol.exchange();
c = protocol.finalize_dotprod(n);
```

The initialization of the multiplication sets the preprocessing and output instances to use in Beaver multiplication. `next_dotprod()` separates dot products in the data preparation phase.

```
set.check();
```

Some protocols require a check of all multiplications up to a certain point. To guarantee that outputs do not reveal secret information, it has to be run before using the output protocol.

```
output.init_open(P);
output.prepare_open(c);
output.exchange(P);
result = output.finalize_open();

cout << "result: " << result << endl;
output.Check(P);
```

The output protocol follows the same blueprint as the multiplication protocol.

```
set.check();
```

Some output protocols require an additional to guarantee the correctness of outputs.

1.6.1 Thread Safety

The low-level interface generally isn't thread-safe. In particular, you should only use one instance of *ProtocolSetup* in the whole program, and you should use only one instance of *CryptoPlayer*/*PlainPlayer* and *ProtocolSet* per thread.

1.6.2 Domain Types

gfp_<X, L>	Computation modulo a prime. L is the number of 64-bit limbs, that is, it covers primes of bit length $64(L-1)+1$ to $64L$. The type has to be initialized using init_field() or init_default(). The latter picks a prime given a bit length.
SignedZ2<K> / Z2<K>	Computation modulo 2^K. This is not a field.
gf2n_short / gf2n_long / gf2n_<T>	$GF(2^n)$. T denotes a type that is used to store the values. It must support a variety of integer operations. The type has to be initialized using init_field(). The choice of degrees is limited. At the time of writing, 4, 8, 28, 40, 63, and 128 are supported if the storage type is large enough.

1.6.3 Share Types

Type	Protocol
AtlasShare<T>	Semi-honest version of ATLAS (Section 4.2). T must represent a field.
ChaiGearShare<T>	HighGear with covert key setup. T must be gfp_<X, L> or gf2n_short.
CowGearShare<T>	LowGear with covert key setup. T must be gfp_<X, L> or gf2n_short.
HemiShare<T>	Semi-honest protocol with Beaver multiplication based on semi-homomorphic encryption. T must be gfp_<X, L> or gf2n_short.
HighGearShare<T>	HighGear. T must be gfp_<X, L> or gf2n_short.
LowGearShare<T>	LowGear. T must be gfp_<X, L> or gf2n_short.
MaliciousShamirShare<T>	Shamir secret sharing with Beaver multiplication and sacrifice. T must represent a field.
MamaShare<T, N>	MASCOT with multiple MACs. T must represent a field, N is the number of MACs.
PostSacriRepFieldShare<T>	Post-sacrifice protocol using three-party replicated secret sharing with T representing a field.
PostSacriRepRingShare<K, S>	Post-sacrifice protocol using replicated three-party secret sharing modulo 2^K with security parameter S.
Rep3Share2<K>	Three-party semi-honest protocol using replicated secret sharing modulo 2^K.
Rep4Share<T>	Four-party malicious protocol using replicated secret sharing over a field.
Rep4Share2<K>	Four-party malicious protocol using replicated secret sharing modulo 2^K.
SemiShare2<K>	Semi-honest dishonest-majority protocol using Beaver multiplication based on oblivious transfer modulo 2^K.
SemiShare<T>	Semi-honest dishonest-majority protocol using Beaver multiplication based on oblivious transfer in a field.
ShamirShare<T>	Semi-honest protocol based on Shamir's secret sharing. T must represent a field.
Share<T>	MASCOT. T must represent a field.
SohoShare<T>	Semi-honest protocol with Beaver multiplication based on somewhat homomorphic encryption. T must be gfp_<X, L> or gf2n_short.
Spdz2kShare<K, S>	SPDZ2k computing modulo 2^K with security parameter S.
SpdzWiseShare<K, S>	SPDZ-wise computing modulo 2^K with security parameter S.
SpdzWiseShare<T>	SPDZ-wise. T must be MaliciousShamirShare or MaliciousRep3Share.
TemiShare<T>	Semi-honest protocol with Beaver multiplication based on threshold semi-homomorphic encryption. T must be gfp_<X, L> or gf2n_short.

1.6.4 Protocol Setup

template<class **T**>

class **ProtocolSetup**

Global setup for an arithmetic share type

Subclassed by *MixedProtocolSetup< T >*

Public Functions

inline **ProtocolSetup**(*Player* &P, int prime_length = 0, string directory = "")

Parameters

- **P** – communication instance (used for MAC generation if needed)
- **prime_length** – length of prime if computing modulo a prime
- **directory** – location to read MAC if needed

inline **ProtocolSetup**(bigint prime, *Player* &P, string directory = "")

Parameters

- **prime** – modulus for computation
- **P** – communication instance (used for MAC generation if needed)
- **directory** – location to read MAC if needed

template<class **T**>

class **ProtocolSet**

Input, multiplication, and output protocol instance for an arithmetic share type

Public Functions

inline **ProtocolSet**(*Player* &P, const *ProtocolSetup*<*T*> &setup)

Parameters

- **P** – communication instance
- **setup** – one-time setup instance

inline void **check**()

Run all protocol checks

template<class **T**>

class **BinaryProtocolSetup**

Global setup for a binary share type

Public Functions

inline **BinaryProtocolSetup**(*Player* &P, string directory = "")

Parameters

- **P** – communication instance (used for MAC generation if needed)
- **directory** – location to read MAC if needed

template<class **T**>

class **BinaryProtocolSet**

Input, multiplication, and output protocol instance for a binary share type

Public Functions

inline **BinaryProtocolSet**(*Player* &P, const *BinaryProtocolSetup*<*T*> &setup)

Parameters

- **P** – communication instance
- **setup** – one-time setup instance

inline void **check**()

Run all protocol checks

template<class **T**>

class **MixedProtocolSetup** : public *ProtocolSetup*<*T*>

Global setup for an arithmetic share type and the corresponding binary one

Public Functions

inline **MixedProtocolSetup**(*Player* &P, int prime_length = 0, string directory = "")

Parameters

- **P** – communication instance (used for MAC generation if needed)
- **prime_length** – length of prime if computing modulo a prime
- **directory** – location to read MAC if needed

template<class **T**>

class **MixedProtocolSet**

Input, multiplication, and output protocol instance for an arithmetic share type and the corresponding binary one

Public Functions

inline **MixedProtocolSet**(*Player* &P, const *MixedProtocolSetup*<*T*> &setup)

Parameters

- **P** – communication instance
- **setup** – one-time setup instance

inline void **check**()

Run all protocol checks

1.6.5 Protocol Interfaces

template<class **T**>

class **ProtocolBase**

Abstract base class for multiplication protocols

Subclassed by Replicated< T >

Public Functions

T **mul**(const *T* &x, const *T* &y)

Single multiplication.

inline virtual void **init**(*Preprocessing*<*T*>&, typename *T*::MAC_Check&)

Initialize protocol if needed (repeated call possible)

virtual void **init_mul**() = 0

Initialize multiplication round.

virtual void **prepare_mul**(const *T* &x, const *T* &y, int n = -1) = 0

Schedule multiplication of operand pair.

virtual void **exchange**() = 0

Run multiplication protocol.

virtual *T* **finalize_mul**(int n = -1) = 0

Get next multiplication result.

virtual void **finalize_mult**(*T* &res, int n = -1)

Store next multiplication result in `res`

inline void **init_dotprod**()

Initialize dot product round.

inline void **prepare_dotprod**(const *T* &x, const *T* &y)

Add operand pair to current dot product.

inline void **next_dotprod**()

Finish dot product.

T **finalize_dotprod**(int length)

Get next dot product result.

template<class **T**>

class **InputBase**

Abstract base for input protocols

Subclassed by Input< T >

Public Functions

virtual void **reset**(int player) = 0

Initialize input round for `player`

void **reset_all**(PlayerBase &P)

Initialize input round for all players.

virtual void **add_mine**(const typename *T*::open_type &input, int n_bits = -1) = 0

Schedule input from me.

virtual void **add_other**(int player, int n_bits = -1) = 0

Schedule input from other player.

void **add_from_all**(const typename *T*::open_type &input, int n_bits = -1)

Schedule input from all players.

virtual void **exchange**()

Run input protocol for all players.

virtual *T* **finalize**(int player, int n_bits = -1)

Get share for next input from `player`

template<class **T**>

class **MAC_Check_Base**

Abstract base class for opening protocols

Public Functions

inline virtual void **Check**(const *Player* &P)

Run checking protocol.

inline const *T*::mac_key_type::Scalar &**get_alphai**() const

Get MAC key.

virtual void **POpen**(vector<typename *T*::open_type> &values, const vector<*T*> &S, const *Player* &P)

Open values in `S` and store results in `values`

inline *T*::open_type **open**(const *T* &secret, const *Player* &P)

Open single value.

virtual void **init_open**(const *Player* &P, int n = 0)

Initialize opening round.

virtual void **prepare_open**(const *T* &secret)

Add value to be opened.

virtual void **exchange**(const *Player* &P) = 0

Run opening protocol.

virtual *T*::clear **finalize_open**()

Get next opened value.

virtual void **CheckFor**(const typename *T*::open_type &value, const vector<*T*> &shares, const *Player* &P)

Check whether all `shares` are `value`

template<class **T**>

class **Preprocessing** : public PrepBase

Abstract base class for preprocessing

Subclassed by *BufferPrep< T >*, Sub_Data_Files< T >

Public Functions

virtual array<*T*, 3> **get_triple**(int n_bits)

Get fresh random multiplication triple.

virtual *T* **get_bit**()

Get fresh random bit.

virtual *T* **get_random**()

Get fresh random value in domain.

virtual void **get_dabit**(*T* &a, typename *T*::bit_type &b)

Store fresh daBit in `a` (arithmetic part) and `b` (binary part)

template<int>
edabitvec<*T*> **get_edabitvec**(bool strict, int n_bits)

Get fresh edaBit chunk.

template<class **T**>

class **BufferPrep** : public *Preprocessing*<*T*>

Abstract base class for live preprocessing

Subclassed by BitPrep< T >

Public Functions

virtual *T* **get_random**()

Get fresh random value.

Public Static Functions

static inline void **basic_setup**(*Player* &P)

Key-independent setup if necessary (cryptosystem parameters)

static inline void **setup**(*Player* &P, typename *T*::mac_key_type alphai)

Generate keys if necessary.

static inline void **teardown**()

Free memory of global cryptosystem parameters.

1.6.6 Domain Reference

template<int **X**, int **L**>

class **gfp_** : public ValueInterface

Type for values in a field defined by integers modulo a prime in a specific range for fixed storage. It supports basic arithmetic operations and bit-wise operations. The latter use the canonical representation in the range `[0, p-1]`. `X` is a counter to allow several moduli being used at the same time. `L` is the number of 64-bit limbs, that is, the prime has to have bit length in `[64*L-63, 64*L]`. See *gfpvar_* for a more flexible alternative.

Public Functions

inline **gfp_**()

Initialize to zero.

inline **gfp_**(const mpz_class &x)

Convert from integer without range restrictions.

template<int **Y**>
gfp_(const *gfp_*<*Y*, *L*> &x)

Convert from different domain via canonical integer representation.

*gfp_* **sqrRoot**()

Deterministic square root.

inline void **randomize**(*PRNG* &G, int n = -1)

Sample with uniform distribution.

Parameters

- **G** – randomness generator
- **n** – (unused)

inline void **pack**(*octetStream* &o, int n = -1) const

Append to buffer in native format.

Parameters

- **o** – buffer
- **n** – (unused)

inline void **unpack**(*octetStream* &o, int n = -1)

Read from buffer in native format

Parameters

- **o** – buffer
- **n** – (unused)

Public Static Functions

static void **init_field**(const bigint &p, bool mont = true)

Initialize the field.

Parameters

- **p** – prime modulus
- **mont** – whether to use Montgomery representation

static void **init_default**(int lgp, bool mont = true)

Initialize the field to a prime of a given bit length.

Parameters

- **lgp** – bit length
- **mont** – whether to use Montgomery representation

static inline const bigint &**pr**()

Get the prime modulus

Friends

inline friend friend ostream & operator<< (ostream &s, const gfp_ &x)

Human-readable output in the range `[-p/2, p/2]`.

Parameters

- **s** – output stream
- **x** – value

inline friend friend istream & operator>> (istream &s, gfp_ &x)

Human-readable input without range restrictions

Parameters

- **s** – input stream
- **x** – value

template<int **X**, int **L**>

class **gfpvar_**

Type for values in a field defined by integers modulo a prime up to a certain length for fixed storage. `X` is a counter to allow several moduli being used at the same time. `L` is the maximum number of 64-bit limbs, that is, the prime has to have bit length at most `64*L`. The interface replicates *gfp_*.

template<int **K**>

class **Z2** : public ValueInterface

Type for values in the ring defined by the integers modulo `2^K` representing `[0, 2^K-1]`. It supports arithmetic, bit-wise, and output streaming operations. It does not need initialization because `K` completely defines the domain.

Subclassed by *SignedZ2< K >*

Public Functions

inline **Z2**()

Initialize to zero.

Z2(const bigint &x)

Convert from unrestricted integer.

template<int **L**>
inline **Z2**(const *Z2*<*L*> &x)

Convert from different domain via the canonical integer representation.

Z2 **sqrRoot**()

Deterministic square root for values with least significate bit 1. Raises an exception otherwise.

void **randomize**(*PRNG* &G, int n = -1)

Sample with uniform distribution.

Parameters

- **G** – randomness generator
- **n** – (unused)

void **pack**(*octetStream* &o, int = -1) const

Append to buffer in native format.

Parameters

- **o** – buffer
- **n** – (unused)

void **unpack**(*octetStream* &o, int n = -1)

Read from buffer in native format

Parameters

- **o** – buffer
- **n** – (unused)

template<int **K**>

class **SignedZ2** : public *Z2*<*K*>

Type for values in the ring defined by the integers modulo `2^K` representing `[-2^(K-1), 2^(K-1)-1]`. It supports arithmetic, bit-wise, comparison, and output streaming operations. It does not need initialization because `K` completely defines the domain.

Public Functions

inline **SignedZ2**()

Initialization to zero

template<int **L**>
inline **SignedZ2**(const *SignedZ2*<*L*> &other)

Conversion from another domain via the signed representation

1.7 Machine Learning

MP-SPDZ supports a limited subset of the Keras interface for machine learning. This includes the SGD and Adam optimizers and the following layer types: dense, 2D convolution, 2D max-pooling, and dropout.

In the following we will walk through the example code in `keras_mnist_dense.mpc`, which trains a dense neural network for MNIST. It starts by defining tensors to hold data:

```
training_samples = sfix.Tensor([60000, 28, 28])
training_labels = sint.Tensor([60000, 10])

test_samples = sfix.Tensor([10000, 28, 28])
test_labels = sint.Tensor([10000, 10])
```

The tensors are then filled with inputs from party 0 in the order that is used by `convert.sh` in the preparation code:

```
training_labels.input_from(0)
training_samples.input_from(0)

test_labels.input_from(0)
test_samples.input_from(0)
```

This is followed by Keras-like code setting up the model and training it:

```
from Compiler import ml
tf = ml

layers = [
  tf.keras.layers.Flatten(),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dense(10,  activation='softmax')
]

model = tf.keras.models.Sequential(layers)

optim = tf.keras.optimizers.SGD(momentum=0.9, learning_rate=0.01)

model.compile(optimizer=optim)

opt = model.fit(
  training_samples,
  training_labels,
  epochs=1,
  batch_size=128,
  validation_data=(test_samples, test_labels)
)
```

Lastly, the model is stored on disk in secret-shared form:

```
for var in model.trainable_variables:
  var.write_to_file()
```

1.7.1 Prediction

The example code in `keras_mnist_dense_predict.mpc` uses the model stored above for prediction. Much of the setup is the same, but instead of training it reads the model from disk:

```
model.build(test_samples.sizes)

start = 0
for var in model.trainable_variables:
  start = var.read_from_file(start)
```

Then it runs the prediction:

```
guesses = model.predict(test_samples)
```

Using `var.input_from(player)` instead the model would be input privately by a party.

1.8 Networking

All protocols in MP-SPDZ rely on point-to-point connections between all pairs of parties. This is realized using TCP, which means that every party must be reachable under at least one TCP port. The default is to set this port to a base plus the player number. This allows for easily running all parties on the same host. The base defaults to 5000, which can be changed with the command-line option `--portnumbase`. However, the scripts in `Scripts` use a random base port number, which can be changed using the same option.

There are two ways of communicating hosts and individually setting ports:

1. All parties first connect to a coordination server, which broadcasts the data for all parties. This is the default with the coordination server being run as a thread of party 0. The hostname of the coordination server has to be given with the command-line parameter `--hostname`, and the coordination server runs on the base port number, thus defaulting to 5000. Furthermore, you can specify a party's listening port using `--my-port`.
2. The parties read the information from a local file, which needs to be the same everywhere. The file can be specified using `--ip-file-name` and has the following format:

   ```
   <host0>[:<port0>]
   <host1>[:<port1>]
   ...
   ```

   The hosts can be both hostnames and IP addresses. If not given, the ports default to base plus party number.

Whether or not encrypted connections are used depends on the security model of the protocol. Honest-majority protocols default to encrypted whereas dishonest-majority protocols default to unencrypted. You change this by either using `--encrypted/-e` or `--unencrypted/-u`.

If using encryption, the certificates (`Player-Data/*.pem`) must be the same on all hosts, and you have to run `c_rehash Player-Data` on all of them. `Scripts/setup-ssl.sh` can be used to generate the necessary certificates. The common name has to be `P<player number>` for computing parties and `C<client number>` for clients.

1.8.1 Internal Infrastructure

The internal networking infrastructure of MP-SPDZ reflects the needs of the various multi-party computation. For example, some protocols require a simultaneous broadcast from all parties whereas other protocols require that every party sends different information to different parties (include none at all). The infrastructure makes sure to send and receive in parallel whenever possible.

All communication is handled through two subclasses of *`Player`* defined in `Networking/Player.h`. *`PlainPlayer`* communicates in cleartext while *`CryptoPlayer`* uses TLS encryption. The former uses the same BSD socket for sending and receiving but the latter uses two different connections for sending and receiving. This is because TLS communication is never truly one-way due key renewals etc., so the only way for simultaneous sending and receiving we found was to use two connections in two threads.

If you wish to use a different networking facility, we recommend to subclass *`Player`* and fill in the virtual-only functions required by the compiler (e.g., `send_to_no_stats()` for sending to one other party). Note that not all protocols require all functions, so you only need to properly implement those you need. You can then replace uses of *`PlainPlayer`* or *`CryptoPlayer`* by your own class. Furthermore, you might need to extend the `Names` class to suit your purpose. By default, *`Names`* manages one TCP port that a party is listening on for connections. If this suits you, you don't need to change anything

1.8.1.1 Reference

class **Names**

Network setup (hostnames and port numbers)

Public Functions

void **init**(int player, int pnb, int my_port, const char *servername, bool setup_socket = true)

Initialize with central server

Parameters

- **player** – my number
- **pnb** – base port number (server listens one below)
- **my_port** – my port number (DEFAULT_PORT for default, which is base port number plus player number)
- **servername** – location of server
- **setup_socket** – whether to start listening

inline **Names**(int player, int pnb, int my_port, const char *servername)

Names(int player, int nplayers, const string &servername, int pnb, int my_port = *DEFAULT_PORT*)

Initialize with central server running on player 0

Parameters

- **player** – my number
- **nplayers** – number of players
- **servername** – location of player 0
- **pnb** – base port number
- **my_port** – my port number (DEFAULT_PORT for default, which is base port number plus player number)

void **init**(int player, int pnb, vector<string> Nms)

Initialize without central server

Parameters

- **player** – my number
- **pnb** – base port number
- **Nms** – locations of all parties

inline **Names**(int player, int pnb, vector<string> Nms)

void **init**(int player, int pnb, const string &hostsfile, int players = 0)

Initialize from file. One party per line, format <hostname>[:<port>]

Parameters

- **player** – my number
- **pnb** – base port number

- **hostsfile** – filename
- **players** – number of players (0 to take from file)

inline **Names**(int player, int pnb, const string &hostsfile)

Names(ez::ezOptionParser &opt, int argc, const char **argv, int default_nplayers = 2)

Initialize from command-line options

Parameters

- **opt** – option parser instance
- **argc** – number of command-line arguments
- **argv** – command-line arguments
- **default_nplayers** – default number of players (used if not given in arguments)

Names(int my_num = 0, int num_players = 1)

Names(const *Names* &other)

~Names()

inline int **num_players**() const

inline int **my_num**() const

inline const string **get_name**(int i) const

inline int **get_portnum_base**() const

Public Static Attributes

static const int **DEFAULT_PORT** = -1

class **Player** : public PlayerBase

Abstract class for multi-player communication. `*_no_stats` functions are called by their equivalents after accounting for communications statistics.

Subclassed by AllButLastPlayer, *MultiPlayer< T >*, *MultiPlayer< int >*, *MultiPlayer< ssl_socket *>*

Public Functions

inline virtual int **num_players**() const

Get number of players

inline virtual int **my_num**() const

Get my player number

virtual void **send_all**(const *octetStream* &o) const

Send the same to all other players

void **send_to**(int player, const *octetStream* &o) const

Send to a specific player

void **receive_all**(vector<*octetStream*> &os) const

Receive from all other players. Information from player 0 at `os[0]` etc.

virtual void **receive_player**(int i, *octetStream* &o) const

Receive from a specific player

void **send_relative**(const vector<*octetStream*> &o) const

Send to all other players by offset. `o[0]` gets sent to the next player etc.

void **receive_relative**(vector<*octetStream*> &o) const

Receive from all other players by offset. `o[0]` will contain data from the next player etc.

void **receive_relative**(int offset, *octetStream* &o) const

Receive from other player specified by offset. 1 stands for the next player etc.

void **exchange**(int other, const *octetStream* &to_send, *octetStream* &ot_receive) const

Exchange information with one other party, reusing the buffer if possible.

void **exchange**(int other, *octetStream* &o) const

Exchange information with one other party, reusing the buffer.

void **exchange_relative**(int offset, *octetStream* &o) const

Exchange information with one other party specified by offset, reusing the buffer if possible.

inline virtual void **pass_around**(*octetStream* &o, int offset = 1) const

Send information to a party while receiving from another by offset, The default is to send to the next party while receiving from the previous. The buffer is reused.

void **pass_around**(*octetStream* &to_send, *octetStream* &to_receive, int offset) const

Send information to a party while receiving from another by offset. The default is to send to the next party while receiving from the previous.

virtual void **unchecked_broadcast**(vector<*octetStream*> &o) const

Broadcast and receive data to/from all players. Assumes o[player_no] contains the data to be broadcast by me.

virtual void **Broadcast_Receive**(vector<*octetStream*> &o) const

Broadcast and receive data to/from all players with eventual verification. Assumes o[player_no] contains the data to be broadcast by me.

virtual void **Check_Broadcast**() const

Run protocol to verify broadcast is correct

virtual void **send_receive_all**(const vector<*octetStream*> &to_send, vector<*octetStream*> &to_receive) const

Send something different to each player.

void **send_receive_all**(const vector<bool> &senders, const vector<*octetStream*> &to_send, vector<*octetStream*> &to_receive) const

Specified senders only send something different to each player.

Parameters

- **senders** – set whether a player sends or not, must be equal on all players
- **to_send** – data to send by player number
- **to_receive** – received data by player number

void **send_receive_all**(const vector<vector<bool>> &channels, const vector<*octetStream*> &to_send, vector<*octetStream*> &to_receive) const

Send something different only one specified channels.

Parameters

- **channels** – channel[i][j] indicates whether party i sends to party j
- **to_send** – data to send by player number
- **to_receive** – received data by player number

virtual void **partial_broadcast**(const vector<bool> &senders, const vector<bool> &receivers, vector<*octetStream*> &os) const

Specified senders broadcast information to specified receivers.

Parameters

- **senders** – specify which parties do send
- **receivers** – specify which parties do send
- **os** – data to send at os[my_number], received data elsewhere

template<class **T**>

class **MultiPlayer** : public *Player*

Multi-player communication helper class. T = int for unencrypted BSD sockets and T = ssl_socket* for Boost SSL streams.

class **PlainPlayer** : public *MultiPlayer*<int>

Plaintext multi-player communication

Subclassed by ThreadPlayer

Public Functions

PlainPlayer(const *Names* &Nms, const string &id)

Start a new set of unencrypted connections.

Parameters

- **Nms** – network setup
- **id** – unique identifier

class **CryptoPlayer** : public *MultiPlayer*<ssl_socket*>

Encrypted multi-party communication. Uses OpenSSL and certificates issued to "P<player_no>". Sending and receiving is done in separate threads to allow for bidirectional communication.

Public Functions

CryptoPlayer(const *Names* &Nms, const string &id)

Start a new set of encrypted connections.

Parameters

- **Nms** – network setup
- **id** – unique identifier

virtual void **partial_broadcast**(const vector<bool> &my_senders, const vector<bool> &my_receivers, vector<*octetStream*> &os) const

Specified senders broadcast information to specified receivers.

Parameters

- **senders** – specify which parties do send
- **receivers** – specify which parties do send
- **os** – data to send at `os[my_number]`, received data elsewhere

class **octetStream**

Buffer for network communication with a pointer for sequential reading. When sent over the network or stored in a file, the length is prefixed as eight bytes in little-endian order.

Public Functions

inline void **resize**(size_t l)

Increase allocation if needed.

void **clear**()

Free memory.

octetStream(size_t maxlen)

Initial allocation.

octetStream(size_t len, const octet *source)

Initial buffer.

octetStream(const string &other)

Initial buffer.

inline size_t **get_ptr**() const

Number of bytes already read.

inline size_t **get_length**() const

Length.

inline size_t **get_total_length**() const

Length including size tag.

inline size_t **get_max_length**() const

Allocation.

inline octet ***get_data**() const

Data pointer.

inline octet ***get_data_ptr()** const

Read pointer.

inline bool **done()** const

Whether done reading.

inline bool **empty()** const

Whether empty.

inline size_t **left()** const

Bytes left to read.

string **str()** const

Convert to string.

octetStream **hash()** const

Hash content.

void **concat**(const *octetStream* &os)

Append other buffer.

inline void **reset_read_head()**

Reset reading.

inline void **reset_write_head()**

Set length to zero but keep allocation.

inline bool **operator==**(const *octetStream* &a) const

Equality test.

void **append_random**(size_t num)

Append `num` random bytes.

inline void **append**(const octet *x, const size_t l)

Append `l` bytes from `x`

inline void **consume**(octet *x, const size_t l)

Read `l` bytes to `x`

inline void **store**(unsigned int a)

Append 4-byte integer.

void **store**(int a)

Append 4-byte integer.

inline void **get**(unsigned int &a)

Read 4-byte integer.

void **get**(int &a)

Read 4-byte integer.

inline void **store**(size_t a)

Append 8-byte integer.

inline void **get**(size_t &a)

Read 8-byte integer.

inline void **store_int**(size_t a, int n_bytes)

Append integer of `n_bytes` bytes.

inline size_t **get_int**(int n_bytes)

Read integer of `n_bytes` bytes.

template<int **N_BYTES**>
inline void **store_int**(size_t a)

Append integer of `N_BYTES` bytes.

template<int **N_BYTES**>
inline size_t **get_int**()

Read integer of `N_BYTES` bytes.

void **store**(const bigint &x)

Append big integer.

void **get**(bigint &ans)

Read big integer.

template<class **T**>
void **store**(const *T* &x)

Append instance of type implementing `pack`

template<class **T**>
T **get**()

Read instance of type implementing `unpack`

template<class **T**>
void **get**(*T* &ans)

Read instance of type implementing `unpack`

template<class **T**>
void **store**(const vector<*T*> &v)

Append vector of type implementing `pack`

template<class **T**>
void **get**(vector<*T*> &v, const *T* &init = {})

Read vector of type implementing `unpack`

Parameters

- **v** – results
- **init** – initialization if required

template<class **T**>
void **get_no_resize**(vector<*T*> &v)

Read vector of type implementing `unpack` if vector already has the right size

inline void **consume**(*octetStream* &s, size_t l)

Read `l` bytes into separate buffer.

void **store**(const string &str)

Append string.

void **get**(string &str)

Read string.

template<class **T**>

inline void **Send**(*T* socket_num) const

Send on `socket_num`

template<class **T**>
inline void **Receive**(*T* socket_num)

Receive on `socket_num`, overwriting current content.

void **input**(istream &s)

Input from stream, overwriting current content.

void **output**(ostream &s)

Output to stream.

template<class **T**>
inline void **exchange**(*T* send_socket, *T* receive_socket)

Send on `socket_num` while receiving on `receiving_socket`, overwriting current content

template<class **T**>
void **exchange**(*T* send_socket, *T* receive_socket, *octetStream* &receive_stream) const

Send this buffer on `socket_num` while receiving to `receive_stream` on `receiving_socket`

1.9 Input/Output

This section gives an overview over the input/output facilities.

1.9.1 Private Inputs from Computing Parties

All secret types have an input function (e.g. *Compiler.types.sint.get_input_from()* or *Compiler.types.sfix.get_input_from()*). Inputs are read as whitespace-separated text in order (independent of the data type) from `Player-Data/Input-P<player>-<thread>`, where `thread` is `0` for the main thread. You can change the prefix (`Player-Data/Input`) using the `-IF` option on the virtual machine binary. You can also use `-I` to read inputs from the command line. *Compiler.types.sint.input_tensor_from()* and *Compiler.types.sfix.input_tensor_from()* allow inputting a tensor.

1.9.2 Compile-Time Data via Private Input

input_tensor_via() is a convenience function that allows to use data available at compile-time via private input.

1.9.3 Public Inputs

All types can be assigned a hard-coded value at compile time, e.g. `sint(1)`. This is impractical for larger amounts of data. *foreach_enumerate()* provides a facility for this case. It uses *public_input* internally, which reads from `Programs/Public-Input/<progname>`.

1.9.4 Public Outputs

By default, *print_ln()* and related functions only output to the terminal on party 0. This allows to run several parties in one terminal without spoiling the output. You can use interactive mode with option `-I` in order to output on all parties. Note that this also to reading inputs from the command line unless you specify `-IF` as well. You can also specify a file prefix with `-OF`, so that outputs are written to `<prefix>-P<player>-<thread>`.

1.9.5 Private Outputs to Computing Parties

Some types provide a function to reveal a value only to a specific party (e.g., *Compiler.types.sint.reveal_to()*). It can be used conjunction with *print_ln_to()* in order to output it.

1.9.6 Binary Output

Most types returned by `reveal()` or `reveal_to()` feature a `binary_output()` method, which writes to `Player-Data/Binary-Output-P<playerno>-<threadno>`. The format is either signed 64-bit integer or double-precision floating-point in machine byte order (usually little endian).

1.9.7 Clients (Non-computing Parties)

Compiler.types.sint.receive_from_client() and *Compiler.types.sint.reveal_to_clients()* allow communicating securely with the clients. See this example covering both client code and server-side high-level code. *Compiler.types.sint.input_tensor_from_client()* and *Compiler.types.MultiArray.reveal_to_clients()*. The same functions are available for *sfix* and *Array*, respectively. See also *Reference* below.

1.9.8 Secret Shares

Compiler.types.sint.read_from_file() and *Compiler.types.sint.write_to_file()* allow reading and writing secret shares to and from files. These instructions use `Persistence/Transactions-P<playerno>.data`. The format depends on the protocol with the following principles.

- One share follows the other without metadata.
- If there is a MAC, it comes after the share.
- Numbers are stored in little-endian format.
- Numbers modulo a power of two are stored with the minimal number of bytes.
- Numbers modulo a prime are stored in Montgomery representation in blocks of eight bytes.

Another possibility for persistence between program runs is to use the fact that the memory is stored in `Player-Data/Memory-<protocol>-P<player>` at the end of a run. The best way to use this is via the memory access functions like *store_in_mem()* and *load_mem()*. Make sure to only use addresses below `USER_MEM` specified in `Compiler/config.py` to avoid conflicts with the automatic allocation used for arrays etc. Note also that all types based on *sint* (e.g., *sfix*) share the same memory, and that the address is only a base address. This means that vectors will be written to the memory starting at the given address.

1.9.9 Reference

class **Client**

Client-side interface

Public Functions

Client(const vector<string> &hostnames, int port_base, int my_client_id)

Start a new set of connections to computing parties.

Parameters

- **hostnames** – location of computing parties
- **port_base** – port base
- **my_client_id** – client identifier

template<class **T**>
void **send_private_inputs**(const vector<*T*> &values)

Securely input private values.

Parameters **values** – vector of integer-like values

template<class **T**, class **U** = *T*>
vector<*U*> **receive_outputs**(int n)

Securely receive output values.

Parameters **n** – number of values

Returns vector of integer-like values

Public Members

vector<client_socket*> **sockets**

Sockets for cleartext communication

octetStream **specification**

Specification of computation domain

1.10 Non-linear Computation

While the computation of addition and multiplication varies from protocol, non-linear computation such as comparison in arithmetic domains (modulus other than two) only comes in three flavors throughout MP-SPDZ:

Unknown prime modulus This approach goes back to Catrina and de Hoogh. It crucially relies on the use of secret random bits in the arithmetic domain. Enough such bits allow to mask a secret value so that it is secure to reveal the masked value. This can then be split in bits as it is public. The public bits and the secret mask bits are then used to compute a number of non-linear functions. The same idea has been used to implement fixed-point and floating-point computation. We call this method "unknown prime modulus" because it only mandates a minimum modulus size for a given cleartext range, which is roughly the cleartext bit length plus a statistical security parameter. It has the downside that there is implicit enforcement of the cleartext range.

Known prime modulus Damgård et al. have proposed non-linear computation that involves an exact prime modulus. We have implemented the refined bit decomposition by Nishide and Ohta, which enables further non-linear computation. Our assumption with this method is that the cleartext space is slightly smaller the full range modulo the prime. This allows for comparison by taking a difference and extracting the most significant bit, which is different than the above works that implement comparison between two positive numbers modulo the prime. We also used an idea by Makri et al., namely that a random k-bit number is indistinguishable from a random number modulo p if the latter is close enough to 2^k.

Power-of-two modulus In the context of non-linear computation, there are two important differences to prime modulus setting:

1. Multiplication with a power of two effectively erases some of the most significant bits.
2. There is no right shift using multiplication. Modulo a prime, multiplying with a power of the inverse of two allows to right-shift numbers with enough zeros as least significant bits.

Taking this differences into account, Dalskov et al. have adapted the mask-and-reveal approach above to the setting of computation modulo a power of two.

See also this slide deck for an introduction to non-linear computation in arithmetic MPC.

1.10.1 Mixed-Circuit Computation

Another approach to non-linear computation is switching to binary computation for parts of the computation. MP-SPDZ implements protocols proposed for particular security models by a number of works: Demmler et al., Mohassel and Rindal, and Dalskov et al. MP-SPDZ also implements more general methods such as daBits and edaBits.

See also this slide deck for an introduction to mixed-circuit computation.

1.10.1.1 Protocol Pairs

The following table lists the matching arithmetic and binary protocols.

Arithmetic	Binary
MASCOT, SPDZ2k, LowGear, HighGear, CowGear, ChaiGear	Tinier with improved cut-and-choose analysis by Furukawa et al.
Semi, Hemi, Temi, Soho, Semi2k	SemiBin (Beaver triples modulo 2 using OT)
Malicious Shamir	Malicious Shamir over $GF(2^{40})$ for secure sacrificing
Malicious Rep3, Post-Sacrifice, SPDZ-wise replicated	Malicious Rep3 modulo 2
Rep4	Rep4 modulo 2
Shamir	Shamir over $GF(2^8)$
ATLAS	ATLAS over $GF(2^8)$
Rep3	Rep3

1.11 Preprocessing

Many protocols in MP-SPDZ use preprocessing, that is, producing secret shares that are independent of the actual data but help with the computation. Due to the independence, this can be done in batches to save communication rounds and even communication when using homomorphic encryption that works with large vectors such as LWE-based encryption.

Generally, preprocessing is done on demand and per computation threads. On demand means that batches of preprocessing data are computed whenever there is none in storage, and a computation thread is a thread created by control flow instructions such as `for_range_multithread()`.

The exceptions to the general rule are edaBit generation with malicious security and AND triples with malicious security and honest majority, both when using bucket size three. Bucket size three implies batches of over a million to achieve 40-bit statistical security, and in honest-majority binary computation the item size is 64, which makes the actual batch size 64 million triples. In multithreaded programs, the preprocessing is run centrally using the threads as helpers.

The batching means that the cost in terms of time and communication jump whenever another batch is generated. Note that, while some protocols are flexible with the batch size and can thus be controlled using `-b`, others mandate a batch size, which can be as large as a million.

1.11.1 Separate preprocessing

It is possible to separate out the preprocessing from the input-dependent ("online") phase. This is done by either option `-F` or `-f` on the virtual machines. In both cases, the preprocessing data is read from files, either all data per type from a single file (`-F`) or one file per thread (`-f`). The latter allows to use named pipes.

The file name depends on the protocol and the computation domain. It is generally `<prefix>/<number of players>-<protocol shorthand>-<domain length>/<preprocessing type>-<protocol shorthand>-P<player number>[-T<thread number>]`. For example, the triples for party 1 in SPDZ modulo a 128-bit prime can be found in `Player-Data/2-p-128/Triples-p-P1`. The protocol shorthand can be found by calling `<share type>::type_short()`. See *Share Types* for a description of the share types.

Preprocessing files start with a header describing the protocol and computation domain to avoid errors due to mismatches. The header is as follows:

- Length to follow (little-endian 8-byte number)
- Protocol descriptor
- Domain descriptor

The protocol descriptor is defined by `<share type>::type_string()`. For SPDZ modulo a prime it is `SPDZ gfp`.

The domain descriptor depends on the kind of domain:

Modulo a prime Serialization of the prime

- Sign bit (0 as 1 byte)
- Length to follow (little-endian 4-byte number)
- Prime (big-endian)

Modulo a power of two: Exponent (little-endian 4-byte number)

$GF(2^n)$

- Storage size in bytes (little-endian 8-byte number). Default is 16.
- n (little-endian 4-byte number)

As an example, the following output of `hexdump -C` describes SPDZ modulo the default 128-bit prime (170141183460469231731687303715885907969):

```
00000000  1d 00 00 00 00 00 00 00  53 50 44 5a 20 67 66 70  |........SPDZ gfp|
00000010  00 10 00 00 00 80 00 00  00 00 00 00 00 00 00 00  |................|
00000020  00 00 1b 80 01                                    |.....|
00000025
```

The actual data is stored is by simple concatenation. For example, triples are stored as repetitions of `a, b, ab`, and daBits are stored as repetitions of `a, b` where `a` is the arithmetic share and `b` is the binary share.

For protocols with MAC, the value share is stored before the MAC share.

Values are generally stored in little-endian order. Note the following domain specifics:

Modulo a prime Values are stored in Montgomery representation with R being the smallest power of 2^{64} larger than the prime. For example, $R = 2^{128}$ for a 128-bit prime. Furthermore, the values are stored in the smallest number of 8-byte blocks necessary, all in little-endian order.

Modulo a power of two: Values are stored in the smallest number of 8-byte blocks necessary, all in little-endian order.

$GF(2^n)$ Values are stored in blocks according to the storage size above, all in little-endian order.

For further details, have a look at `Utils/Fake-Offline.cpp`, which contains code that generates preprocessing data insecurely for a range of protocols (underlying the binary `Fake-Offline.x`).

`{mascot,cowgear,mal-shamir}-offline.x` generate sufficient preprocessing data for a specific high-level program with MASCOT, CowGear, and malicious Shamir secret sharing, respectively.

1.12 Adding a Protocol

In order to illustrate how to create a virtual machine for a new protocol, we have created one with blanks to be filled in. It is defined in the following files:

`Machines/no-party.cpp` Contains the main function.

`Protocols/NoShare.h` Contains the `NoShare` class, which is supposed to hold one share. `NoShare` takes the clear-text type as a template parameter.

`Protocols/NoProtocol.h` Contains a number of classes representing instances of protocols:

- **`NoInput`** Private input.
- **`NoProtocol`** Multiplication protocol.
- **`NoOutput`** Public output.

`Protocols/NoLivePrep.h` Contains the `NoLivePrep` class, representing a preprocessing instance.

The number of blanks can be overwhelming. We therefore recommend the following approach to get started. If the desired protocol resembles one that is already implemented, you can check its code for inspiration. The main function of `<protocol>-party.x` can be found in `Machines/<protocol>-party.cpp`, which in turns contains the name of the share class. For example `replicated-ring-party.x` is implemented in `Machines/replicated-ring-party.cpp`, which refers to `Rep3Share2()` in `Protocols/Rep3Share2.h`. There you will find that it uses `Replicated()` for multiplication, which is found in `Protocols/Replicated.h`.

1. Fill in the `constant()` static member function of `NoShare` as well as the `exchange()` member function of `NoOutput`. Check out `DirectSemiMC<T>::exchange_()` in `Protocols/SemiMC.hpp` for a simple example. It opens an additive secret sharing by sending all shares to all other parties and then summing up the received. See *this reference* for documentation on the necessary infrastructure. Constant sharing and public output allows to execute the following program:

```
print_ln('%s', sint(123).reveal())
```

   This allows to check the correct execution of further functionality.

2. Fill in the operator functions in `NoShare` and check them:

```
print_ln('%s', (sint(2) + sint(3)).reveal())
print_ln('%s', (sint(2) - sint(3)).reveal())
print_ln('%s', (sint(2) * cint(3)).reveal())
```

   Many protocols use these basic operations, which makes it beneficial to check the correctness

3. Fill in `NoProtocol`. Alternatively, if the desired protocol is based on Beaver multiplication, you can specify the following in `NoShare`:

```
typedef Beaver<This> Protocol;
```

   Then add the desired triple generation to `NoLivePrep::buffer_triples()`. In any case you should then be able to execute:

```
print_ln('%s', (sint(2) * sint(3)).reveal())
```

4. In order to execute many kinds of non-linear computation, random bits are needed. After filling in `NoLivePrep::buffer_bits()`, you should be able to execute:

```
print_ln('%s', (sint(2) < sint(3)).reveal()
```

1.12.1 Reference

The following classes are fundamental building blocks in protocols. See also the *this reference* for networking-related classes.

class **PRNG**

Pseudo-random number generator. This uses counter-mode AES by default, which can be changed libsodium's expansion by undefining `USE_AES`.

Subclassed by ElementPRNG< T >, *GlobalPRNG*, *SeededPRNG*

Public Functions

PRNG()

Construction without initialization. Usage without initilization will fail.

PRNG(*octetStream* &seed)

Initialize with `SEED_SIZE` bytes from buffer.

void **ReSeed**()

Initialize from local randomness.

void **SeedGlobally**(const *Player* &P, bool secure = true)

Coordinate random seed

Parameters

- **P** – communication instances
- **secure** – seeding prevents tampering at higher cost

void **SetSeed**(const unsigned char*)

Initialize with `SEED_SIZE` bytes from pointer.

void **SetSeed**(*PRNG* &G)

Initialize with seed from another instance.

inline bool **get_bit**()

Random bit.

inline unsigned char **get_uchar**()

Random bytes.

unsigned int **get_uint**()

Random 32-bit integer.

unsigned int **get_uint**(int upper)

Random 32-bit integer between 0 and `upper`

template<class **T**>
void **randomBnd**(*T* &res, const bigint &B, bool positive = true)

Random integer in `[0, B-1]`

Parameters

- **res** – result
- **B** – bound
- **positive** – positive result (random sign otherwise)

inline word **get_word**()

Random 64-bit integer.

inline __m128i **get_doubleword**()

Random 128-bit integer.

inline void **get_octets**(octet *ans, int len)

Fill array with random data

Parameters

- **ans** – result
- **len** – byte length

template<int **L**>
inline void **get_octets**(octet *ans)

Fill array with random data (compile-time length)

Parameters **ans** – result

template<class **T**>
inline *T* **get**()

Random instance of any supported class.

class **SeededPRNG** : public *PRNG*

Randomly seeded pseudo-random number generator.

class **GlobalPRNG** : public *PRNG*

Coordinated pseudo-random number with secure seeding.

1.13 Homomorphic Encryption

MP-SPDZ uses BGV encryption for triple generation in a number of protocols. This involves zero-knowledge proofs in some protocols and considerations about function privacy in all of them. The interface described below allows directly accessing the basic cryptographic operations in contexts where these considerations are not relevant. See `Utils/he-example.cpp` for some example code.

1.13.1 Reference

class **FHE_Params**

Cryptosystem parameters

Public Functions

FHE_Params(int n_mults = 1, int drown_sec = DEFAULT_SECURITY)

Initialization.

Parameters

- **n_mults** – number of ciphertext multiplications (0/1)
- **drown_sec** – parameter for function privacy (default 40)

void **pack**(*octetStream* &o) const

Append to buffer.

void **unpack**(*octetStream* &o)

Read from buffer.

void **basic_generation_mod_prime**(int plaintext_length)

Generate parameter for computation modulo a prime

Parameters **plaintext_length** – bit length of prime

class **FHE_KeyPair**

BGV key pair

Public Functions

inline **FHE_KeyPair**(const *FHE_Params* &params)

Initialization.

inline void **generate**()

Generate fresh keys.

Public Members

FHE_PK **pk**

Public key.

FHE_SK **sk**

Secret key.

class **FHE_SK**

BGV secret key. The class allows addition.

Public Functions

inline void **pack**(*octetStream* &os) const

Append to buffer.

inline void **unpack**(*octetStream* &os)

Read from buffer. Assumes parameters are set correctly.

Plaintext_<FFT_Data> **decrypt**(const *Ciphertext* &c)

Decryption for cleartexts modulo prime.

class **FHE_PK**

BGV public key.

Public Functions

template<class **FD**>
Ciphertext **encrypt**(const *Plaintext*<typename *FD*::T, *FD*, typename *FD*::S> &mess) const

Encryption.

void **pack**(*octetStream* &o) const

Append to buffer.

void **unpack**(*octetStream* &o)

Read from buffer. Assumes parameters are set correctly.

template<class **T**, class **FD**, class **_**>

class **Plaintext**

BGV plaintext. Use `Plaintext_mod_prime` instead of filling in the templates. The plaintext is held in one of the two representations or both, polynomial and evaluation. The latter is the one allowing element-wise multiplication over a vector. Plaintexts can be added, subtracted, and multiplied via operator overloading.

Public Functions

unsigned int **num_slots**() const

Number of slots.

Plaintext(const *FHE_Params* &params)

Initialization.

inline *T* **element**(int i) const

Read slot.

Parameters **i** – slot number

Returns slot content

inline void **set_element**(int i, const *T* &e)

Write to slot

Parameters

- **i** – slot number
- **e** – new slot content

void **pack**(*octetStream* &o) const

Append to buffer.

void **unpack**(*octetStream* &o)

Read from buffer. Assumes parameters are set correctly.

class **Ciphertext**

BGV ciphertext. The class allows adding two ciphertexts as well as adding a plaintext and a ciphertext via operator overloading. The multiplication of two ciphertexts requires the public key and thus needs a separate function.

Public Functions

inline *Ciphertext* **mul**(const *FHE_PK* &pk, const *Ciphertext* &x) const

Ciphertext multiplication.

Parameters

- **pk** – public key
- **x** – second ciphertext

Returns product ciphertext

void **rerandomize**(const *FHE_PK* &pk)

Re-randomize for circuit privacy.

inline void **pack**(*octetStream* &o) const

Append to buffer.

inline void **unpack**(*octetStream* &o)

Read from buffer. Assumes parameters are set correctly.

1.14 Troubleshooting

This section shows how to solve some common issues.

1.14.1 Crash without error message or `bad_alloc`

Some protocols require several gigabytes of memory, and the virtual machine will crash if there is not enough RAM. You can reduce the memory usage for some malicious protocols with `-B 5`. Furthermore, every computation thread requires separate resources, so consider reducing the number of threads with `for_range_multithreads()` and similar.

1.14.2 List indices must be integers or slices

You cannot access Python lists with runtime variables because the lists only exists at compile time. Consider using *`Array`*.

1.14.3 `compile.py` takes too long or runs out of memory

If you use Python loops (`for`), they are unrolled at compile-time, resulting in potentially too much virtual machine code. Consider using *`for_range()`* or similar. You can also use `-l` when compiling, which will replace simple loops by an optimized version.

1.14.4 Order of memory instructions not preserved

By default, the compiler runs optimizations that in some corner case can introduce errors with memory accesses such as accessing an *`Array`*. If you encounter such errors, you can fix this either with `-M` when compiling or placing *break_point()* around memory accesses.

1.14.5 Odd timings

Many protocols use preprocessing, which means they execute expensive computation to generates batches of information that can be used for computation until the information is used up. An effect of this is that computation can seem oddly slow or fast. For example, one multiplication has a similar cost then some thousand multiplications when using homomorphic encryption because one batch contains information for more than than 10,000 multiplications. Only when a second batch is necessary the cost shoots up. Other preprocessing methods allow for a variable batch size, which can be changed using `-b`. Smaller batch sizes generally reduce the communication cost while potentially increasing the number of communication rounds. Try adding `-b 10` to the virtual machine (or script) arguments for very short computations.

1.14.6 Disparities in round figures

The number of virtual machine rounds given by the compiler are not an exact prediction of network rounds but the number of relevant protocol calls (such as multiplication, input, output etc) in the program. The actual number of network rounds is determined by the choice of protocol, which might use several rounds per protocol call. Furthermore, communication at the beginning and the end of a computation such as random key distribution and MAC checks further increase the number of network rounds.

1.14.7 Handshake failures

If you run on different hosts, the certificates (`Player-Data/*.pem`) must be the same on all of them. Furthermore, party <i> requires `Player-Data/P<i>.key` that must match `Player-Data/P<i>.pem`, that is, they have to be generated to together. The easiest way of setting this up is to run `Scripts/setup-ssl.sh` on one host and then copy all `Player-Data/*.{pem,key}` to all other hosts. This is *not* secure but it suffices for experiments. A secure setup would generate every key pair locally and then distributed only the public keys. Finally, run `c_rehash Player-Data` on all hosts. The certificates generated by `Scripts/setup-ssl.sh` expire after a month, so you need to regenerate them. The same holds for `Scripts/setup-client.sh` if you use the client facility.

1.14.8 Connection failures

MP-SPDZ requires one TCP port per party to be open to other parties. In the default setting, it's 5000 on party 0, and 5001 on party 1 etc. You change change the base port (5000) using `--portnumbase` and individual ports for parties using `--my-port`. The scripts use a random base port number, which you can also change with `--portnumbase`.

1.14.9 Internally called tape has unknown offline data usage

Certain computations are not compatible with reading preprocessing from disk. You can compile the binaries with `MY_CFLAGS += -DINSECURE` in `CONFIG.mine` in order to execute the computation in a way that reuses preprocessing.

1.14.10 Illegal instruction

By default, the binaries are optimized for the machine they are compiled on. If you try to run them an another one, make sure set `ARCH` in `CONFIG` accordingly. Furthermore, if you run on an x86 processor without AVX (produced before 2011), you need to set `AVX_OT = 0` to run dishonest-majority protocols.

1.14.11 Invalid instruction

The compiler code and the virtual machine binary have to be from the same version because most version slightly change the bytecode. This mean you can only use the precompiled binaries with the Python code in the same release.

1.14.12 Computation used more preprocessing than expected

This indicates an error in the internal accounting of preprocessing. Please file a bug report.

1.14.13 Windows/VirtualBox performance

Performance when using Windows/VirtualBox is by default abysmal, as AVX/AVX2 instructions are deactivated (see e.g. here), which causes a dramatic performance loss. Deactivate Hyper-V/Hypervisor using:

```
bcdedit /set hypervisorlaunchtype off
DISM /Online /Disable-Feature:Microsoft-Hyper-V
```

Performance can be further increased when compiling MP-SPDZ yourself:

```
sudo apt-get update
sudo apt-get install automake build-essential git libboost-dev libboost-thread-dev␣
↪libntl-dev libsodium-dev libssl-dev libtool m4 python3 texinfo yasm
git clone https://github.com/data61/MP-SPDZ.git
cd MP-SPDZ
make tldr
```

See also this issue for a discussion.

1.14.14 mac_fail

This is a catch-all failure in protocols with malicious protocols that can be caused by something being wrong at any level. Please file a bug report with the specifics of your case.

CHAPTER

TWO

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

V

W

X

Z