MP-SPDZ Documentation

Marcel Keller

Nov 11, 2026

CONTENTS:

1 Getting Started **3**
- 1.1 Frequently Asked Questions . . . 3
- 1.2 Contact . . . 3
  - 1.2.1 Filing Issues . . . 3
- 1.3 Paper and Citation . . . 3
- 1.4 TL;DR (Binary Distribution on Linux or Source Distribution on macOS) . . . 4
- 1.5 TL;DR (Source from GitHub) . . . 4
- 1.6 TL;DR (Docker) . . . 4
- 1.7 Relevant papers . . . 5
- 1.8 Preface . . . 5
- 1.9 Protocols . . . 5
  - 1.9.1 Finding the most efficient protocol . . . 6
- 1.10 History . . . 6
- 1.11 Overview . . . 7
- 1.12 Requirements . . . 7
- 1.13 Compilation . . . 7

2 Running Computation **9**
- 2.1 Compiling high-level programs . . . 10
  - 2.1.1 Arithmetic modulo a prime . . . 10
  - 2.1.2 Arithmetic modulo 2^k . . . 10
  - 2.1.3 Binary circuits . . . 10
  - 2.1.4 Mixed circuits . . . 10
    - 2.1.4.1 Classic daBits . . . 11
    - 2.1.4.2 Extended daBits . . . 11
    - 2.1.4.3 Local share conversion . . . 11
    - 2.1.4.4 Finding the most efficient variant . . . 11
    - 2.1.4.5 Manual conversion . . . 11
  - 2.1.5 Bristol Fashion circuits . . . 11
  - 2.1.6 Compiling programs directly in Python . . . 12
  - 2.1.7 Compiling and running programs from external directories . . . 12
- 2.2 TensorFlow inference . . . 13
- 2.3 Emulation . . . 13
- 2.4 Dishonest majority . . . 13
  - 2.4.1 Secret sharing . . . 13
  - 2.4.2 Yao's garbled circuits . . . 15
- 2.5 Honest majority . . . 15
  - 2.5.1 Protocols with function-dependent preprocessing . . . 17
- 2.6 Dealer model . . . 18
- 2.7 BMR . . . 18

2.8 Online-only benchmarking . . . 19
2.8.1 SPDZ . . . 19
2.8.1.1 To setup for benchmarking the online phase . . . 19
2.8.1.2 To compile a program . . . 19
2.8.1.3 To run a program . . . 20
2.8.2 SPDZ2k . . . 20
2.8.3 Other protocols . . . 20
2.8.4 BMR . . . 20
2.8.4.1 Oblivious RAM . . . 21
2.9 Preprocessing as required . . . 21
2.10 Benchmarking offline phases . . . 21
2.10.1 Benchmarking the MASCOT or SPDZ2k offline phase . . . 21
2.10.2 Benchmarking Overdrive offline phases . . . 22
2.10.2.1 Memory usage . . . 22

3 Recommended Reading **23**

4 Compilation Process **27**
4.1 Compiler options . . . 27
4.2 Direct Compilation in Python . . . 29
4.3 Compilation vs run time . . . 30

5 Run-Time Options **31**
5.1 Efficiency trade-offs . . . 31
5.2 Protocol options . . . 32
5.3 Network setup . . . 33
5.4 Local facilities . . . 34

6 High-Level Interface **35**
6.1 Compiler.types module . . . 35
6.1.1 Basic types . . . 35
6.1.2 Container types . . . 36
6.2 Compiler.GC.types module . . . 93
6.3 Compiler.library module . . . 105
6.4 Compiler.mpc_math module . . . 113
6.5 Compiler.ml module . . . 115
6.6 Compiler.decision_tree module . . . 128
6.7 Compiler.circuit module . . . 129
6.8 Compiler.program module . . . 131
6.9 Compiler.oram module . . . 133
6.10 Compiler.sqrt_oram module . . . 135
6.11 Compiler.path_oblivious_heap module . . . 135
6.12 Compiler.dijkstra module . . . 138
6.13 Compiler.sorting module . . . 139
6.14 Compiler.sparse module . . . 140

7 Bytecode Utilities **143**
7.1 Memory usage . . . 143
7.2 Preprocessing usage . . . 143
7.3 Human-readable bytecode/circuit representation . . . 143

8 The Journey of a Program **145**
8.1 Execution . . . 147

9 Compiler Optimizations **149**

9.1 Merging . . . 149
9.2 CISC . . . 149

10 Virtual Machine **153**
10.1 Schedule File . . . 153
10.2 Bytecode . . . 154
10.2.1 Memory size indication . . . 155
10.3 Instructions . . . 155
10.4 Compiler.instructions module . . . 160
10.5 Compiler.GC.instructions module . . . 197

11 Low-Level Interface **209**
11.1 Thread Safety . . . 211
11.2 Domain Types . . . 211
11.3 Share Types . . . 212
11.4 Protocol Setup . . . 212
11.5 Protocol Interfaces . . . 215
11.6 Domain Reference . . . 218

12 Using High-Level Functionality in C++ **221**
12.1 Vector arguments and return values . . . 223
12.2 Binary values . . . 223
12.3 C++ compilation . . . 225
12.4 Reference . . . 225

13 Navigating the C++ Code **227**
13.1 Mathematical Domains . . . 227
13.2 Communication . . . 228
13.3 Randomness . . . 228
13.4 Protocols . . . 229
13.5 Virtual Machines . . . 230

14 Machine Learning Quickstart **231**

15 Machine Learning **233**
15.1 Data Input . . . 233
15.1.1 Integrated Data Input . . . 233
15.1.1.1 Data preprocessing . . . 235
15.1.2 Independent Data Input . . . 235
15.2 Training . . . 235
15.2.1 Logistic regression with SGD . . . 235
15.2.2 Linear regression with SGD . . . 236
15.2.3 PyTorch interface . . . 236
15.2.4 Keras interface . . . 237
15.2.5 Decision trees . . . 237
15.2.5.1 Data preparation . . . 238
15.3 Loading pre-trained models . . . 238
15.4 Storing and loading models . . . 239
15.5 Exporting models . . . 240

16 Networking **243**
16.1 Internal Infrastructure . . . 243
16.1.1 Reference . . . 244

17 Input/Output **253**

17.1 Private Inputs from Computing Parties . . . 253
17.2 Compile-Time Data via Private Input . . . 253
17.3 Public Inputs . . . 253
17.4 Public Outputs . . . 253
17.5 Private Outputs to Computing Parties . . . 253
17.6 Binary Output . . . 254
17.7 Clients (Non-computing Parties) . . . 254
17.8 Secret Shares via Socket . . . 254
17.9 Secret Shares via Files . . . 254
17.10 Python Trusted Client Tutorial . . . 254
17.11 Python Reference . . . 255
17.12 C++ Reference . . . 256

18 Client Interface **259**
18.1 Working Examples . . . 259
18.2 I/O MPC Instructions . . . 259
18.2.1 Connection Setup . . . 259
18.2.2 Data Exchange . . . 260
18.3 Client-Side Interface . . . 260

19 Multinode Computation Example **261**

20 Non-linear Computation **263**
20.1 Mixed-Circuit Computation . . . 264
20.1.1 Protocol Pairs . . . 264

21 Preprocessing **265**
21.1 Separate preprocessing . . . 265

22 Lowest-Level Interface **267**

23 Adding a Protocol **271**
23.1 Reference . . . 272

24 Adding an Instruction **275**
24.1 Frontend . . . 275
24.2 Backend . . . 276
24.2.1 Parsing . . . 276
24.2.2 Resourcing . . . 277
24.2.3 Execution . . . 277

25 Homomorphic Encryption **279**
25.1 Reference . . . 279

26 Threshold ECDSA **283**
26.1 Compilation . . . 283
26.2 Running . . . 283

27 Security Fixes **285**
27.1 Insufficient check in SPDZ-wise replicated sharing over rings . . . 285
27.2 Insufficient checks in malicious replicated sharing with Beaver multiplication . . . 285
27.3 Malicious parties are not committed to MAC key across threads and executions . . . 285
27.4 Insufficient checks in Rep4 . . . 285
27.5 Co-ordinate MAC checks among threads and race condition in MAC checks . . . 285
27.6 Remove MAC key in case of failure . . . 286
27.7 Remotely caused buffer overflows . . . 286

27.8 Missing shuffling check in PS mod 2^k and Brain . . . 286
27.9 Insufficient drowning in pairwise protocols . . . 286
27.10 Proper accounting for random elements . . . 286
27.11 Insufficient security in non-interactive zero-knowledge proofs . . . 286
27.12 Security bug in SPDZ2k . . . 286
27.13 Skewed random bit generation . . . 286
27.14 All-zero secret keys in homomorphic encryption . . . 287
27.15 Insufficient randomness in SemiBin random bit generation . . . 287
27.16 Missing MAC key initialization . . . 287

28 Troubleshooting **289**
28.1 Crash without error message, `Killed`, or `bad_alloc` . . . 289
28.2 List indices must be integers or slices . . . 289
28.3 Local variable referenced before assignment . . . 289
28.4 `compile.py` takes too long or runs out of memory . . . 289
28.5 Cannot derive truth value from register . . . 289
28.6 Cannot branch on secret values . . . 290
28.7 Incorrect results when using `sfix` . . . 291
28.8 Variable results when using `sfix` . . . 291
28.9 Only party 0 produces outputs . . . 291
28.10 Order of memory instructions not preserved . . . 291
28.11 High number of rounds or slow WAN execution . . . 291
28.12 Odd timings . . . 291
28.13 Disparities in round figures . . . 291
28.14 Handshake failures . . . 292
28.15 Connection failures . . . 292
28.16 Internally called tape has unknown offline data usage . . . 292
28.17 Illegal instruction . . . 292
28.18 Invalid instruction . . . 292
28.19 Computation used more preprocessing than expected . . . 292
28.20 Required prime bit length is not the same as `-F` parameter during compilation . . . 292
28.21 Prime number not compatible with encryption scheme . . . 292
28.22 Windows/VirtualBox performance . . . 293
28.23 `mac_fail` . . . 293
28.24 Debugging errors in a virtual machine . . . 293

29 Indices and tables **295**

Python Module Index **297**

Index **299**

MP-SPDZ is a framework for multi-party computation, a privacy-enhancing technology focused on input privacy. It uses protocols running between several entities to create a conceptual black box in which data can operated on privately. Such a black box could be created between banks or healthcare providers to jointly operate on privacy-sensitive data. Please see this gentle introduction for more information on multi-party computation.

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

1. Quickstart tutorial
2. *Machine learning quickstart*
3. Implemented protocols
4. *Troubleshooting*
5. *Input/Output* lists all the ways of getting data in and out.

CHAPTER

ONE

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 Frequently Asked Questions

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

1.2 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.

1.2.1 Filing Issues

Please file complete code examples because it's usually not possible to reproduce problems from incomplete code, and please include which protocol you have used (if applicable) because there are considerable differences between the various protocols.

1.3 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},
}
```

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

This requires either a Linux distribution originally released 2019 or later (glibc 2.28) or macOS High Sierra or later as well as Python 3 and basic command-line utilities. On macOS, we recommend downloading directly from GitHub due to the use of git for dependencies (see next section).

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

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

This runs the tutorial with two parties and malicious security.

1.5 TL;DR (Source from GitHub)

You need to have Git in order to clone the repository.

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

```
sudo apt-get install automake build-essential clang cmake git libboost-dev libboost-
↪filesystem-dev libboost-iostreams-dev libboost-thread-dev libgmp-dev libntl-dev␣
↪libsodium-dev libssl-dev libtool python3
```

On MacOS, this requires High Sierra or later. It uses brew to install the required dependencies. You will be offered an automatic download if it is not found.

The following will execute the tutorial with two parties and malicious security.

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

On strong enough hardware setups (several cores and GB of RAM), you can speed up the last step by running `make -j8 mascot-party.x` beforehand.

1.6 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/compile-run.py mascot tutorial
```

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

1.7 Relevant papers

The protocols in MP-SPDZ are based on numerous papers, and many computations will rely on several, ranging from the most basic building blocks to higher-level protocols making use of the building blocks in a black-box manner. You can have the compiler output links to relevant reading with the option `--paper`. Use either of the following:

```
./compile.py --papers -E <protocol> <program>
Scripts/compile-run.py --papers <protocol> <program>
```

You can find a more systematic overview in the documentation.

1.8 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.

1.9 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 / Rep4*	*Brain / Rep3 / PS / SY / Rep4*	*Rep3 / CCD / PS / Rep4*	*BMR*
Semi-honest, honest majority	*Shamir / ATLAS / Rep3*	*Rep3* / *Astra / Trio*	*Rep3 / CCD*	*BMR*
Malicious, honest supermajority	*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.

1.9.1 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 protocols with any number of parties, 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.

1.10 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

1.11 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 protocols 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 differs between the two variants. Most computation functionality is available in both, but binary circuits are lacking some input-output functionality.

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.

1.12 Requirements

- GCC 11 or later (tested with up to 15) or LLVM/clang 11 or later (tested with up to 22). The default is to use clang because it performs better.
- For protocols 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.
- GMP library, compiled with C++ support (use flag `--enable-cxx` when running configure). Tested against 6.2.1 as supplied by Ubuntu.
- libsodium library, tested against 1.0.18
- OpenSSL, tested against 3.0.2
- Boost.Asio with SSL support (`libboost-dev` on Ubuntu), tested against 1.83
- Boost.Thread for BMR (`libboost-thread-dev` on Ubuntu), tested against 1.83
- 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 11.5.1)
- If using macOS, Sierra or later. Only the default Xcode version of clang is used for testing.
- Windows/VirtualBox: see this issue for a discussion

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. To run on CPUs without AVX2 (CPUs from before 2014), you should also add `AVX_OT = 0` to `CONFIG.mine`.
    - 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`.
- On macOS, there have been issues with non-system compilers. Add `CXX = /usr/bin/g++` to fix them.

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`.

CHAPTER

TWO

RUNNING COMPUTATION

See `Programs/Source/` for some example MPC programs, in particular `tutorial.mpc`. Furthermore, Read the Docs hosts a more detailed reference of all aspects of MP-SPDZ.

There are three ways of running computation:

1. Separate compilation and execution. This is the default in the further documentation. It allows to run the same program several times while only compiling once, for example:

```
./compile.py <program> <argument>
Scripts/mascot.sh <program>-<argument> [<runtime-arg>...]
Scripts/mascot.sh <program>-<argument> [<runtime-arg>...]
```

2. One-command local execution. This compiles the program and the virtual machine if necessary before executing it locally with the given protocol. The name of the protocols correspond to the script names below (without the `.sh`). Furthermore, some protocol-specific optimization options are automatically used as well as required options.

```
Scripts/compile-run.py -E mascot <program> <argument> -- [<runtime-arg>...]
```

3. One-command remote execution. This compiles the program and the virtual machine if necessary before uploading them together with all necessary input and certificate files via SSH.

```
Scripts/compile-run.py -H HOSTS -E mascot <program> <argument> -- [<runtime-arg>...]
```

   `HOSTS` has to be a text file in the following format:

```
[<user>@]<host0>[/<path>]
[<user>@]<host1>[/<path>]
...
```

   If does not start with `/` (only one `/` after the hostname), the path will be relative to the home directory of the user. Otherwise (`//` after the hostname it will be relative to the root directory.

   It is assumed that the SSH login is possible without password. This can be achieved using password-less SSH keys. See this tutorial for more information.

   Adding the compiler option `-t` (`--tidy_output`) groups the output prints by party; however, it delays the outputs until the execution is finished.

Even with the integrated execution it is important to keep in mind that there are two different phases, the compilation and the run-time phase. Any secret data is only available in the second phase, when the Python compilation has concluded. Therefore, the types like `sint` and `sfix` are mere placeholders for data to be used later, and they don't contain any shares. See also the documentation for what this means when using Python data structures and Python language features.

2.1 Compiling high-level programs

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

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 `-P` is optional, and it involves algorithms that are more expensive while allowing for a wider range of integer lengths.

The command-line options primarily affects non-linear computation such as comparisons. See the documentation on non-linear computation for more details and pointers to relevant papers.

Note that in this context integers do not wrap around according to the 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 computations 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()`.

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 (the latter except for SPDZ2k). If another length is required, use `MOD = -DRING_SIZE=<bit length>` in `CONFIG.mine`.

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 15/16-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 `sbitintvec.get_type(n)` to get a type for `n`-bit arithmetic.

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.

2.1.4.1 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.

2.1.4.2 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.

2.1.4.3 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).

2.1.4.4 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.

2.1.4.5 Manual conversion

You can switch between arithmetic and binary computation using constructors. For arithmetic to binary use `sbitvecint(x, n, m)` where `x` is of type `sint`, `n` is the number of bits in the result, and `m` is the bit length of the input, that is, `x` is in `[0,2**n-1]`. For binary to arithmetic use `sint(x)` where `x` is of type `sbitvec` or `sbitvecint`.

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.

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()

@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`.)

Furthermore, this only covers compilation, so you will need to run execution separately, for example:

```
Scripts/mascot.sh hello_world
```

Also note that programs in the above form are not compatible with `compile.py` and `compile-run.py`.

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
```

2.2 TensorFlow inference

Note: All networks mentioned below are now supported by the PyTorch interface, which is better integrated and thus easier to use. This section is merely kept to document the approach used for an earlier paper, but it is recommended to use the PyTorch interface.

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 numpy scipy pillow>=9.1 tensorflow
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 ../../../..
cp EzPC/Athos/Networks/SqueezeNetImgNet/SqNetImgNet_img_input.inp Player-Data/Input-
↪Binary-P0-0
./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 patches in https://github.com/mkskeller/EzPC.

The reference contains further documentation on available layers.

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, that is, *no* multi-party computation is performed.

2.4 Dishonest majority

All implementations require oblivious transfer at least for binary computation. The default is to use SoftSpokenOT, an OT extension. It features a parameter `k` determining the trade-off between computation and communication. The default choice 2, but you can use `-o softspoken=<k>` to change it. The OT extension is implemented based on https://github.com/mkskeller/SimpleOT or https://github.com/mkskeller/SimplestOT_C, depending on whether AVX is available.

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. Tinier on the other hand denotes the protocol by Frederiksen et al. also using the cut-and-choose sacrifice by 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 `--no-top-gear` disables the improved TopGear zero-knowledge proofs in favor of the original variants.

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 as described in Appendix B of this work. 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.

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 -G -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.

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/N	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. You can use it with the option `--semi-honest` to skip the checks needed for malicious security. `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.

2.5.1 Protocols with function-dependent preprocessing

MP-SPDZ implements two honest-majority protocols with function-dependent preprocessing, Astra and Trio. Both use computation modulo a power of two as the arithmetic domain, and they are closely related to three-party replicated secret sharing as shown in this paper.

There are two virtual machines for either protocol, one for preprocessing and one for the online phase: `{astra,trio}-{prep,}-party.x`. You can run them separately if your computation does not require communicating revealed values back from the online phase to the offline phase. Alternatively, you need to run a virtual machine for both phases simultaneously while facilitating the transmission of information between the phases through named pipes in `Player-Data`. `Scripts/{astra,trio}.sh` run both phases with the necessary setup. This only works only on Linux as macOS does not seem to offer the same functionality for named pipes.

Finally, the virtual machines don't implement mixed multiplications, so the compiler has to be configure to produced secret multiplications instead. This can be achieved be either running `./compile.py -E [astra|trio]` or by placing `program.use_mulm = False` at the beginning of the high-level program. You can also use the integrated execution, either

```
Scripts/compile-run.py [astra|trio] <progname> <args>...
```

or

```
Scripts/compile-run.py [astra|trio]-prep <progname> <args>...
Scripts/compile-run.py [astra|trio]-online <progname> <args>...
```

Alternatively, compile first and then run the phases separately:

```
./compile.py -E [astra|trio] <progname> <args>...
Scripts/[astra|trio]-prep.sh <progname-with-args>
Scripts/[astra|trio]-online.sh <progname-with-args>
```

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`

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 -G -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`.

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`.

Note that you can as well run the full protocol with option `-v` to see the cost split by preprocessing and online phase.

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
```

2.8.1.1 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>]`

2.8.1.2 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/

2.8.1.3 To run a program

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

`./mascot-party.x -F -N 2 0 tutorial`

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

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

`Scripts/mascot.sh -F tutorial`

MASCOT is one of the protocols that use SPDZ for the online phase, and `-F` causes the programs to read preprocessing material from files.

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, `mascot-party.x` needs to be passed the machine where the first party is running. E.g., if this machine is named `diffie` on the local network:

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

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

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

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.

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.

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. This is implementation is independent of the default design, so most of the previous documentation does not apply. In particular, you cannot use `compile-run.py`.

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 enabling insecure functionality:

```
echo MY_CFLAGS += -DINSECURE >> CONFIG.mine
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> [...]`

2.8.4.1 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 -G -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, for example: `Scripts/bmr-program-run.sh gc_oram 2`.

This only works with the partially insecure BMR implementation, so you cannot use any other protocol with it including Yao or any full BMR implementation. See the documentation on how to use ORAM in all other protocols.

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`, `lowgear`, `mal-shamir`, `semi`, `semi2k`, and `hemi`.

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.

After running the offline phase, you can use the online phase using the `-F` option as *above*.

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

2.10 Benchmarking offline phases

2.10.1 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.

2.10.2 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.

2.10.2.1 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.

CHAPTER

THREE

RECOMMENDED READING

The following table lists papers relevant for every protocol supported by MP-SPDZ. This is the same as the output when using `--papers` during compilation. If you prefer a more general yet comprehensive introduction to MPC, we recommend A Pragmatic Introduction to Secure Multi-Party Computation.

Protocol shorthand	Papers
astra	https://eprint.iacr.org/2025/919
atlas	https://eprint.iacr.org/2021/833
brain	https://eprint.iacr.org/2019/164
ccd	https://eprint.iacr.org/2000/037, https://doi.org/10.1145/62212.62214
chaigear	https://eprint.iacr.org/2017/1230, https://eprint.iacr.org/2019/035
cowgear	https://eprint.iacr.org/2017/1230, https://eprint.iacr.org/2019/035
dealer-ring	https://mp-spdz.readthedocs.io/en/latest/readme.html#dealer-model
hemi	https://eprint.iacr.org/2022/933, https://eprint.iacr.org/2017/1230, https://eprint.iacr.org/2014/106
highgear	https://eprint.iacr.org/2017/1230, https://eprint.iacr.org/2019/035, https://eprint.iacr.org/2019/1300
lowgear	https://eprint.iacr.org/2017/1230, https://eprint.iacr.org/2019/035, https://eprint.iacr.org/2019/1300
mal-rep-bmr	https://eprint.iacr.org/2017/981
mal-shamir-bmr	https://eprint.iacr.org/2017/981
mal-ccd	https://eprint.iacr.org/2017/816, https://doi.org/10.1145/62212.62214
mal-rep-bin	https://eprint.iacr.org/2016/944
mal-rep-field	https://eprint.iacr.org/2017/816
mal-rep-ring	https://eprint.iacr.org/2019/164
mal-shamir	https://eprint.iacr.org/2017/816
mama	https://eprint.iacr.org/2016/505
mascot	https://eprint.iacr.org/2016/505
ps-rep-bin	https://doi.org/10.1109/SP.2017.15
ps-rep-field	https://eprint.iacr.org/2017/816
ps-rep-ring	https://eprint.iacr.org/2019/164
real-bmr	https://eprint.iacr.org/2017/981
rep-bmr	https://eprint.iacr.org/2017/981
rep4-ring	https://eprint.iacr.org/2020/1330
replicated	https://eprint.iacr.org/2016/768
rep-field	https://eprint.iacr.org/2016/768
ring	https://eprint.iacr.org/2025/919, https://eprint.iacr.org/2016/768
semi	https://eprint.iacr.org/2016/505, https://eprint.iacr.org/2020/521
semi-bin	https://eprint.iacr.org/2016/505, https://eprint.iacr.org/2020/521

continues on next page

Table 1 – continued from previous page

Protocol shorthand	Papers
semi-bmr	https://eprint.iacr.org/2017/981
semi2k	https://eprint.iacr.org/2016/505, https://eprint.iacr.org/2020/521
shamir	https://eprint.iacr.org/2000/037
shamir-bmr	https://eprint.iacr.org/2017/981
soho	https://eprint.iacr.org/2017/1230, https://eprint.iacr.org/2020/521
spdz2k	https://eprint.iacr.org/2018/482
sy-rep-field	https://eprint.iacr.org/2018/570
sy-rep-ring	https://eprint.iacr.org/2019/1298
sy-shamir	https://eprint.iacr.org/2018/570
temi	https://eprint.iacr.org/2022/933, https://eprint.iacr.org/2017/1230, https://eprint.iacr.org/2014/106, https://eprint.iacr.org/2000/055
tinier	https://eprint.iacr.org/2015/901
tiny	https://eprint.iacr.org/2018/482, https://eprint.iacr.org/2016/944
trio	https://eprint.iacr.org/2025/919
yao	https://eprint.iacr.org/2014/756, https://eprint.iacr.org/2019/1168

Further recommended reading extracted from the code can be found in the following table. This mostly refers to higher-level protocols based on the arithmetic blackbox provided by the protocols above. You can use `--papers` when compiling a concrete program to find which papers are relevant for a particular computation.

Keywords	Papers
approximate inverse square root	https://doi.org/10.1145/3411501.3419427
bit decomposition, equality, truncation	https://eprint.iacr.org/2018/403
client inputs	https://eprint.iacr.org/2015/1006
comparison	https://eprint.iacr.org/2025/919
comparison, modulo, truncation	https://eprint.iacr.org/archive/2019/131/20191015:083051
decision tree learning	https://doi.org/10.56553/popets-2023-0021
deep learning	https://eprint.iacr.org/2022/933
Dijkstra's algorithm, heap queue, ORAM	https://eprint.iacr.org/2014/137
(e)daBits	https://eprint.iacr.org/2020/338
equality, modulo	https://www.researchgate.net/publication/225092133, https://doi.org/10.1007/978-3-642-15317-4_13 (paywall)
exponential	https://eprint.iacr.org/2026/944
exponential, logarithm, square root	https://eprint.iacr.org/2019/354
fixed-point numbers, probabilistic truncation	https://www.ifca.ai/pub/fc10/31_47.pdf
floating-point numbers, power of two, secret truncation	https://eprint.iacr.org/2012/405
full bit decomposition	https://doi.org/10.1007/978-3-540-71677-8_23
graph cycle detection, Quicksort, sorting, sparse matrix multiplication	https://eprint.iacr.org/2026/2457
loop optimization	https://eprint.iacr.org/2024/1990
path oblivious heap	https://eprint.iacr.org/2019/274
probabilistic truncation, secret truncation	https://eprint.iacr.org/2019/131
secure shuffling	https://eprint.iacr.org/2024/547
sorting	https://eprint.iacr.org/2014/121
sorting	https://eprint.iacr.org/2013/143
sparse matrix multiplication	https://doi.org/10.1145/3800506.3803493
square-root ORAM	https://doi.org/10.1109/SP.2016.21
two-party MUX	https://doi.org/10.1145/3372297.3417274

CHAPTER

FOUR

COMPILATION PROCESS

The easiest way of using MP-SPDZ is using `compile.py` or `Scripts/compile-run.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|py]`, 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|py]`, with `<progname>` as `program.args[0]`. The resulting program for the virtual machine will be called `<progname>[-<arg0>[-<arg1>...]`.

A common use case for an argument would be setting a loop size:

```
@for_range_opt(int(program.args[1]))
def _(i):
  ...
```

This can be compiled and run using

```
./compile.py <progname> <n_loops>
Scripts/<protocol>.sh <progname>-<n_loops>
```

or

```
Scripts/compile-run.py <protocol> <progname> <n_loops>
```

4.1 Compiler options

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>

Use bit decomposition by Nishide and Ohta with a concrete prime modulus for non-linear 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. You can use prime order domains without specifying this option. Using this option involves algorithms for non-linear computation which are generally more expensive but allow for integer lengths that are close to the bit length of the prime. See *Non-linear Computation* for more details

-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.

-G

--garbled-circuit

Compile for garbled circuits (does not replace *-B*).

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.

You can also tell the compiler which protocol you intend to run the computation with:

-E <protocol>

--execute <protocol>

Enable all suitable optimizations and restrictions for a particular protocol. This is the same as in `compile-run.py`. It will also let the compiler estimate the total communication cost for many arithmetic protocols.

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 1000 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.

4.2 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:

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

compiler = Compiler()

@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*.)

4.3 Compilation vs run time

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

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

CHAPTER

FIVE

RUN-TIME OPTIONS

This section explains the command-line arguments with the virtual machines (`<protocol>-party.x` or `Scripts/<protocol>.sh`). You can also use them with `Scripts/compile-run.py` by specifying them after an additional `--`:

```
Scripts/compile-run.py <protocol> <compile-time-args> -- <run-time-args>
```

5.1 Efficiency trade-offs

-B <bucket size>

--bucket-size=<bucket size>

Some protocols use a shuffle-and-sacrifice approach for malicious security, for example Araki et al.. The bucket size offers a trade-off between efficiency and batch size, that is, a larger bucket size allows for smaller batch size and less unused preprocessing and memory usage for smaller computations but a smaller bucket size offers better efficiency for large enough computations. The options are 3 (batch size 2^{20}), 4 (batch size 10386), and 5 (batch size 1024).

-b <bucket size>

--batch-size=<bucket size>

Many protocols allow freely choosing a batch size for preprocessing (for example, Beaver triples or random bits). The trade-off here is that smaller batch sizes require more rounds but might generate less unused data and require less memory while larger batch sizes reduce the number of rounds. The default is 1000 for dishonest-majority protocols and 10,000 for honest-majority protocols as the latter are usually more efficient. The virtual machines also use information about the program provided by the compiler to lower the preprocessing amount if appropriate.

-d

--direct

Some protocols allow a choice between direct communication (every party communicates with every other party or a constant fraction of the parties) and star-shaped communication (every party sends information to a chosen one, which then collates and sends the results to all parties). For example, the opening of an additive secret sharing could happen with all parties sending their shares to all others or a single party could receive all shares and then send the sum to all parties. In the first case, the communication is quadratic in one round whereas, in the second case, the communication is linear in two rounds. The default is star-shaped communication due to the asymptotic difference but you can select direct communication using this option.

-Q

--bits-from-squares

There are several ways to generate random secret bits such as XORing bits input by several parties or using the square root of squares of random numbers, which will be the original number with probability one half. This has been used by Damgård et al. for example. The virtual machines chose the random bit generation method

depending on the number of players as the first method does not scale well. However, you can force the square method using this option.

5.2 Protocol options

-E <error>

--trunc-error <error>

Probabilistic truncation as introduced by Mohassel and Zhang requires the truncated value to be in a negligibly small range of the secret-sharing domain. Otherwise, the result will be wrong with non-negligible probability. This parameter controls if this kind of cheap truncation is used depending on the range determined by the compiler. The default is 40, which means that if the range is larger than 2^-40 of the secret-sharing domain, another truncation is used if possible.

-F

--file-preprocessing

The default is to generate preprocessing data like Beaver triples on demand. With this option, it is instead read from files in `Player-Data`. You need to run `Fake-Offline.x` or the relevant `<protocol>-offline-party.x` beforehand to makes this work.

-f

--file-prep-per-thread

This is similar to `--file-preprocessing` but uses a separate file per thread instead of global files. MP-SPDZ does not provide the relevant functionality to match this. Instead, it is meant for external software to provide preprocessing in a streaming manner via a named pipe. See `../Utils/stream-fake-mascot-triples.cpp` for an example and this GitHub issue for further discussion.

-lg2 <bit length>

--lg2 <bit length>

This allows specifying n for $GF(2^n)$. The default is either 40, 64, or 128 depending on the protocol and the choice of options when compiling the binaries.

-lgp <bit length>

--lgp <bit length>

This allows specifying the bit length n of the prime modulus if applicable. The prime is chosen as the smallest prime of the form $2^{n-1} + x \cdot 2^{16} + 1$ for a positive integer x. This form makes it compatible with the LWE-based homomorphic encryption used for some protocols.

-P <modulus>

--prime <modulus>

This allows specifying the prime modulus if applicable.

-R <bit length>

--ring <bit length>

This allows specifying n for computation modulo 2^n if applicable. It defaults to 64. Note that this choice has to be fixed during compilation for any non-linear computation, which means you cannot change it here anymore.

-S <security parameter>

--security <security parameter>

This allows specifying the statistical security parameter if applicable. The default is 40, which means that the adversary can get away with cheating with probability 2^{-40}. This parameter does not apply to SPDZ2k, see below.

-SP <security parameter>

--spdz2k-security <security parameters>

This allows specifying the statistical security parameter for SPDZ2k. The default is 64, and it means that the adversary can get away with cheating with probability roughly $2^{-64+\log_2(64)} = 2^{-58}$.

5.3 Network setup

-e

--encrypted

-u

--unencrypted

The default is to use SSL-encrypted connections with honest-majority protocols and unencrypted connections with dishonest-majority protocols. The reasoning is that a network adversary can reconstruct all secrets with the secret sharing used for honest-majority protocol but not with the additive secret sharing used in dishonest-majority protocols. You can use the relevant option to choose the non-default option. However, you will need to compile the binaries with `SECURE = -DINSECURE` in `CONFIG.mine` to use unencrypted channels with honest-majority protocols.

-ext-server

--external-server

At the beginning of a computation, all parties have to connect each other. One way of facilitating is running `Server.x`, which waits for all parties to connect and broadcasts all hostnames. The default is to run this functionality within party 0, but this option makes party 0 receive the list of parties from elsewhere like every other party.

-h <hostname>

--hostname <hostname>

This is to specify the hostname where party 0 or the setup server (using `-ext-server`) is found.

-ip <filename>

--ip-file-name <filename>

Instead of having of party 0 or an external server collecting hostnames, you can put all information in a file. See *Input/Output* for the format.

-mp <port number>

--my-port <port number>

This allows specifying the port where a party listens for connections from other parties. The default is the base number (see below) plus the party number.

-pn <base number>

--portnumbase <base number>

This allows specifying the base for computing the party-specific port number by adding the party number. The logic to have this comes from running all parties on the same host for testing purposes. The default is 5000.

-p

--player

This allows specifying the party number. Alternatively, you can give it as the first argument.

-N

--nparties

This allows specifying the number of parties if applicable. The default is two for dishonest-majority protocols and three for honest-majority protocols. If an external server is used, you can specify the number of parties there instead.

5.4 Local facilities

--code-locations

This activates the output of the most important locations in the C++ code that are active for a particular computation.

-D <path>

--disk-memory <path>

By default, the virtual machines store everything in memory, including all data structures like (multi-)arrays. With this option, they instead use a memory-mapped file in the given path.

-I

--interactive

By default, inputs to the computation are read from files in `Player-Data` (see *Input/Output* for more details). This changes to reading from stdin in the main thread. Furthermore, this enables output on all parties rather than just party 0.

-IF <prefix>

--input-file <prefix>

By default, inputs read from `Player-Data/Input-[Binary-]-P<partyno>-<threadno>`. This changes the prefix from `Player-Data/Input`.

-m <old|empty>

--memory <old|empty>

By default, the memory (where data structures such as `Array` are stored) is initialized to zero at the beginning of a computation as it is when using `empty`. With `old`, it is instead loaded from `Player-Data/Memory-<type>-P<playerno>` where it is stored at the end of computation (curtailed to 2^{20} for efficiency).

-OF <prefix>

--output-file <prefix>

By default, `print_ln()` and similar output to stdout on party 0 but not on other parties. This changes the output to `<prefix>-P<partyno>-<threadno>`. You can also use `.` to output to stdout on all parties.

-o <option1[,option2...]>

--options <option1[,option2...]>

This provides a flexible way to add run-time options. Any mention of `has_option()` works with this. One example is `-o throw_exceptions`, which prevents exceptions from being caught, thus allowing debugging with GDB.

-v

--verbose

Use this option to active verbose benchmarking output like the number of preprocessing items and the cost of different phases.

CHAPTER

SIX

HIGH-LEVEL INTERFACE

6.1 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.

6.1.1 Basic types

All basic types 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)
```

The following types are available in arithmetic circuits and with reduced functionality in binary circuits.

sint	Secret integer in the protocol-specific domain.
sfix	Secret fixed-point number represented as secret integer, by multiplying with `2^f` and then rounding.
regint	Clear 64-bit integer.

The following types are only available in arithmetic circuits.

cint	Clear integer in same domain as secure computation (depends on protocol).
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.
personal	Value known to one player.

6.1.2 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[:]
```

Arrays aren't initialized on creation, you need to call *assign_all()* to initialize them to a constant value.

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*, *n_threads=None*, *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)

concat(*other*)

Concatenate two arrays.

classmethod create_from(*l*)

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

Parameters
l – Python iterable, register vector, or array

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

dot(*other*)

Dot product with another array.

expand_to_vector(*index*, *size*)

Create vector from single entry.

Parameters

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

from_binary(*other*, *n_threads=None*, ***kwargs*)

Convert secret binary container to arithmetic.

Parameters

- **other** – sbitvec (multi-)array
- **n_threads** – number of threads (default is to run in same thread)

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*, *skip=1*)

Return vector with content.

Parameters

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

get_reverse_vector()

Return vector with content in reverse order.

get_vector(*base=0*, *size=None*, *skip=1*)

Return vector with content.

Parameters

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

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

Fill with inputs from player if supported by type.

Parameters

- **player** – public (regint/cint/int)
- **binary** – whether to use binary (`Input-Binary-P<playerno>-<threadno>`) instead of cleartext (`Input-P<playerno>-<threadno>`)

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

permute(*permutation*, *reverse=False*, *n_threads=None*)

Public permutation.

Parameters

- **permutation** – cleartext :py:class`Array` containing number in $[0, n-1]$ where n is the length of this array
- **reverse** – whether to apply the inverse of the permutation

print_reveal_nested(*end='\n'*)

Reveal and print as list.

Parameters

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

randomize(**args*)

Randomize array according to data type. If it is *sfix*, the following will sample an individual uniformly random entry of the array `M` roughly in the range $[a, b]$:

```
M.randomize(a, b)
```

read_from_file(*start*, **args*, ***kwargs*)

Read content from `Persistence/Transactions-P<playerno>.data`. Precision must be the same as when storing if applicable. See *this section* for details on the data format.

Parameters

- **start** – starting position in number of shares from beginning (int/regint/cint)
- **crash_if_missing** – crash if file not found (default)

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

read_from_socket(*socket*, *debug=False*)

Read content from socket.

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

reveal_to_socket_by_party(*client_id*, *n_parties=None*)

Reveal i-th part to a specific client socket on party i.

Parameters

- **client_id** – regint

- **n_parties** – number of parties (default: first dimension length)

same_shape(*value_type=None*, ***kwargs*)

Array of same length and type.

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

Secure permute in place according to the security model. See *MultiArray.secure_shuffle()* for references.

Parameters

- **permutation** – output of `sint.get_secure_shuffle()`
- **reverse** – whether to apply inverse (default: False)

secure_shuffle(*n_threads=None*, *get_shuffle=False*)

Secure shuffle in place according to the security model. See *MultiArray.secure_shuffle()* for references.

Parameters

- **n_threads** – number of threads (default is to run in same thread)
- **get_shuffle** – return oblivious shuffle

shuffle()

Insecure shuffle in place.

sort(*n_threads=None*, *batcher=False*, *n_bits=None*, *signed=True*, *time=False*, *debug=False*, *quicksort=False*, ***kwargs*)

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)
- **signed** – most significant bit is sign
- **time** – granular timings
- **debug** – debugging output (insecure)
- **quicksort** – use Quicksort

sum()

Sum of elements.

to_binary_with_right_length(*bit_length*, *n_threads=None*, ***kwargs*)

Convert sint container to binary.

Parameters

- **bit_length** – bit length of input and output (potentially insecure if input exceeds the bit length)
- **n_threads** – number of threads (default is to run in same thread)

`to_column_matrix`()

Returns the array as Nx1 matrix.

Warning: This operation is in-place (without copying data), i.e., all changes to the values of the matrix will also affect the original array. :return: Matrix

`to_row_matrix`()

Returns the array as 1xN matrix.

Warning: This operation is in-place (without copying data), i.e., all changes to the values of the matrix will also affect the original array. :return: Matrix

`write_to_file`(*position=None*)

Write shares of integer representation to `Persistence/Transactions-P<playerno>.data`. See *this section* for details on the data format.

Parameters

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

`write_to_socket`(*socket*, *debug=False*)

Write content to socket.

`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

Matrices aren't initialized on creation, you need to call *assign_all()* to initialize them to a constant value.

`assign`(*other*, *base=0*, *n_threads=None*)

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

Parameters

other – container of matching size and type

`assign_all`(*value*, ***kwargs*)

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.

concat(*other*)

Concatenate two multi-arrays of matching dimension.

concat_columns(*other*)

Concatenate two matrices by columns.

static create_from(*rows*, *n_threads=None*)

Create or copy matrix.

Parameters

- **rows** – iterable, *Array*, or *Matrix*
- **n_threads** – multithreaded operation, requires input to be a container and not an iterable

diag()

Matrix diagonal.

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

Matrix multiplication in the virtual machine. Unlike *dot()*, this only works for sint and sfix, and it returns a vector instead of a data structure.

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)

Return type
Matrix or *Array* of appropriate size and type

from_binary(*other*, *n_threads=None*, ***kwargs*)

Convert secret binary container to arithmetic.

Parameters

- **other** – sbitvec (multi-)array
- **n_threads** – number of threads (default is to run in same thread)

get_column(*index*, *base=0*, *size=None*)

Get (part of) matrix column as vector.

Parameters

- **index** – regint/cint/int
- **base** – starting row (default 0)
- **size** – number of rows (default all remaining)

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 entries 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*, ***kwargs*)

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

Returns
Matrix 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)

`permute`(*permutation*, *reverse=False*, *n_threads=None*)

Public permutation along first dimension.

Parameters

- **permutation** – cleartext :py:class`Array` containing number in $[0, n-1]$ where n is the length of this array
- **reverse** – whether to apply the inverse of the permutation

print_reveal_nested(*end='\n'*)

Reveal and print as nested list.

Parameters

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

randomize(**args*, *n_threads=None*)

Randomize according to data type. If it is *sfix*, the following will sample an individual uniformly random entry of the multi-array `M` roughly in the range $[a, b]$:

```
M.randomize(a, b)
```

read_from_file(*start*, **args*, ***kwargs*)

Read content from `Persistence/Transactions-P<playerno>.data`. Precision must be the same as when storing if applicable. See *this section* for details on the data format.

Parameters

- **start** – starting position in number of shares from beginning (int/regint/cint)
- **crash_if_missing** – crash if file not found (default)

Returns

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

read_from_socket(*socket*, *debug=False*)

Read content from socket.

reveal()

Reveal to *MultiArray* of same shape.

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

reveal_to_socket_by_party(*client_id*, *n_parties=None*)

Reveal i-th part to a specific client socket on party i.

Parameters

- **client_id** – regint
- **n_parties** – number of parties (default: first dimension length)

same_shape(*value_type=None*, ***kwargs*)

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*, *n_threads=None*, *n_parallel=None*)

Securely permute rows (first index). See *secure_shuffle()* for references.

Parameters

- **permutation** – output of `sint.get_secure_shuffle()`
- **reverse** – whether to apply inverse (default: False)
- **n_threads** – How many threads should be used. Will not multithread when set to None (default: None)
- **n_parallel** – How many columns should be permuted in parallel. Will use the compiler's optimization budget is set to None. (default: None).

secure_shuffle()

Securely shuffle rows (first index). This uses the algorithm in Section 4.3 of Keller and Scholl or Section 3.2 of Asharov et al. if applicable.

set_column(*index*, *vector*, *base=0*)

Change column.

Parameters

- **index** – regint/cint/int
- **vector** – short enought vector of compatible type
- **base** – starting row (default 0)

sort(*key_indices=None*, *n_bits=None*, *batcher=False*, *n_threads=None*, *signed=True*, *value_n_bits=None*, *key_prep=None*, *time=False*, *debug=False*, *quicksort=False*, ***kwargs*)

Sort sub-arrays (different first index) in place. This uses radix sort.

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)
- **batcher** – whether to use Batcher's odd-even merge sorting
- **n_threads** – number of threads to use (single thread by default), only works with Batcher's algorithm
- **signed** – most significant bit is sign
- **value_n_bits** – number of bits in values (default: global bit length)
- **key_prep** – modifier function for keys

- **time** – granular timings
- **debug** – debugging output (insecure)
- **quicksort** – use Quicksort

to_binary_with_right_length(*bit_length*, *n_threads=None*, ***kwargs*)

Convert sint container to binary.

Parameters

- **bit_length** – bit length of input and output (potentially insecure if input exceeds the bit length)
- **n_threads** – number of threads (default is to run in same thread)

trace()

Matrix trace.

trans_mul(*other*)

Matrix multiplication with transpose of `self`

Parameters

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

Returns

Matrix 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(*n_threads=None*)

Matrix transpose.

Parameters

self – two-dimensional

write_to_file(*position=None*)

Write shares of integer representation to `Persistence/Transactions-P<playerno>.data`. See *this section* for details on the data format.

Parameters

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

write_to_socket(*socket*, *debug=False*)

Write content to socket.

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

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)

from_binary(*other*, *n_threads=None*, ***kwargs*)

Convert secret binary container to arithmetic.

Parameters

- **other** – sbitvec (multi-)array
- **n_threads** – number of threads (default is to run in same thread)

iadd(*other*)

Addition assignment.

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

reveal_to_clients(*clients*)

Reveal contents to list of clients.

Parameters
clients – list or array of client identifiers

reveal_to_socket_by_party(*client_id*, *n_parties=None*)

Reveal i-th part to a specific client socket on party i.

Parameters

- **client_id** – regint
- **n_parties** – number of parties (default: first dimension length)

square()

Square.

to_binary_with_right_length(*bit_length*, *n_threads=None*, ***kwargs*)

Convert sint container to binary.

Parameters

- **bit_length** – bit length of input and output (potentially insecure if input exceeds the bit length)
- **n_threads** – number of threads (default is to run in same thread)

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][:]
```

Arrays aren't initialized on creation, you need to call *assign_all()* to initialize them to a constant value.

assign(*other*, *base=0*, *n_threads=None*)

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

Parameters
other – container of matching size and type

assign_all(*value*, ***kwargs*)

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.

concat(*other*)

Concatenate two multi-arrays of matching dimension.

diag()

Matrix diagonal.

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

Matrix multiplication in the virtual machine. Unlike *dot()*, this only works for sint and sfix, and it returns a vector instead of a data structure.

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)

Return type
Matrix or *Array* of appropriate size and type

`from_binary`(*other*, *n_threads=None*, ***kwargs*)

Convert secret binary container to arithmetic.

Parameters

- **other** – sbitvec (multi-)array
- **n_threads** – number of threads (default is to run in same thread)

`get_column`(*index*, *base=0*, *size=None*)

Get (part of) matrix column as vector.

Parameters

- **index** – regint/cint/int
- **base** – starting row (default 0)
- **size** – number of rows (default all remaining)

`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 entries 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*, ***kwargs*)

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

Returns
Matrix 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)

permute(*permutation*, *reverse=False*, *n_threads=None*)

Public permutation along first dimension.

Parameters

- **permutation** – cleartext :py:class`Array` containing number in $[0, n-1]$ where n is the length of this array
- **reverse** – whether to apply the inverse of the permutation

print_reveal_nested(*end='\n'*)

Reveal and print as nested list.

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

randomize(**args*, *n_threads=None*)

Randomize according to data type. If it is *sfix*, the following will sample an individual uniformly random entry of the multi-array `M` roughly in the range $[a, b]$:

```
M.randomize(a, b)
```

read_from_file(*start*, **args*, ***kwargs*)

Read content from `Persistence/Transactions-P<playerno>.data`. Precision must be the same as when storing if applicable. See *this section* for details on the data format.

Parameters

- **start** – starting position in number of shares from beginning (int/regint/cint)
- **crash_if_missing** – crash if file not found (default)

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

read_from_socket(*socket*, *debug=False*)

Read content from socket.

reveal()

Reveal to *MultiArray* of same shape.

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

reveal_to_socket_by_party(*client_id*, *n_parties=None*)

Reveal i-th part to a specific client socket on party i.

Parameters

- **client_id** – regint
- **n_parties** – number of parties (default: first dimension length)

same_shape(*value_type=None*, ***kwargs*)

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*, *n_threads=None*, *n_parallel=None*)

Securely permute rows (first index). See *secure_shuffle()* for references.

Parameters

- **permutation** – output of `sint.get_secure_shuffle()`
- **reverse** – whether to apply inverse (default: False)
- **n_threads** – How many threads should be used. Will not multithread when set to None (default: None)
- **n_parallel** – How many columns should be permuted in parallel. Will use the compiler's optimization budget is set to None. (default: None).

`secure_shuffle`()

Securely shuffle rows (first index). This uses the algorithm in Section 4.3 of Keller and Scholl or Section 3.2 of Asharov et al. if applicable.

`set_column`(*index*, *vector*, *base=0*)

Change column.

Parameters

- **index** – regint/cint/int
- **vector** – short enought vector of compatible type
- **base** – starting row (default 0)

`sort`(*key_indices=None*, *n_bits=None*, *batcher=False*, *n_threads=None*, *signed=True*, *value_n_bits=None*, *key_prep=None*, *time=False*, *debug=False*, *quicksort=False*, ***kwargs*)

Sort sub-arrays (different first index) in place. This uses radix sort.

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)
- **batcher** – whether to use Batcher's odd-even merge sorting
- **n_threads** – number of threads to use (single thread by default), only works with Batcher's algorithm
- **signed** – most significant bit is sign
- **value_n_bits** – number of bits in values (default: global bit length)
- **key_prep** – modifier function for keys
- **time** – granular timings
- **debug** – debugging output (insecure)
- **quicksort** – use Quicksort

`to_binary_with_right_length`(*bit_length*, *n_threads=None*, ***kwargs*)

Convert sint container to binary.

Parameters

- **bit_length** – bit length of input and output (potentially insecure if input exceeds the bit length)
- **n_threads** – number of threads (default is to run in same thread)

trace()

Matrix trace.

trans_mul(*other*)

Matrix multiplication with transpose of `self`

Parameters

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

Returns

Matrix 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(*n_threads=None*)

Matrix transpose.

Parameters

self – two-dimensional

write_to_file(*position=None*)

Write shares of integer representation to `Persistence/Transactions-P<playerno>.data`. See *this section* for details on the data format.

Parameters

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

write_to_socket(*socket*, *debug=False*)

Write content to socket.

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 *sintbit*.

Similarly to *Compiler.types.cint*, this type is restricted to arithmetic circuits due to the fact that only arithmetic protocols offer communication-less public-private integer operations.

Parameters

v – cfix/float/int

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

Type-dependent array. Example:

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

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

Type-dependent matrix. Example:

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

classmethod MemValue(*value*)

classmethod Tensor(*shape*, ***kwargs*)

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)

iadd(*other*)

Addition assignment. This uses `update()` internally.

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

Clear fixed-point comparison.

Parameters
other – cfix/cint/regint/int

Returns
0/1

Return type
regint

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. The initial defaults are chosen to allow the best optimization of probabilistic truncation in computation modulo 2^64 (2*k < 64). Generally, 2*k must be at most the integer length for rings and at most m-s-1 for computation modulo an m-bit prime and statistical security s (default 40).

Parameters

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

square()

Square.

store_in_mem(*address*)

Store in memory by public address.

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

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

Parameters

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

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

Helper class for printing revealed sfloats.

binary_output(*player=None*)

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

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

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

Output.

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

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

Parameters

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

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

Type-dependent array. Example:

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

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

Type-dependent matrix. Example:

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

classmethod MemValue(*value*)

classmethod Tensor(*shape*, ***kwargs*)

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()`.

field_div(*other*)

Field division of public values. Not available for computation modulo a power of two.

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

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)

iadd(*other*)

Addition assignment. This uses *update()* internally.

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*, ***kwargs*)

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.

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

Read shares from `Persistence/Transactions-[gf2n-]P<playerno>.data`. See *this section* for details on the data format.

Parameters

- **start** – starting position in number of shares from beginning (int/regint/cint)
- **n_items** – number of items (int)
- **crash_if_missing** – crash if file not found (default)
- **size** – vector size (int)

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

Returns
list of shares

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

classmethod write_to_file(**args*, ***kwargs*)

Write shares to `Persistence/Transactions-[gf2n-]P<playerno>.data` (appending at the end). See *this section* for details on the data format.

Parameters

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

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

Send a list of values to a client by converting to 64-bit regint.

Parameters

- **client_id** – Client id (regint)
- **values** – list of cgf2n

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.

This type is restricted to arithmetic circuits due to the fact that only arithmetic protocols offer communication-less public-private integer operations.

Parameters

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

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

Type-dependent array. Example:

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

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

Type-dependent matrix. Example:

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

classmethod MemValue(*value*)

classmethod Tensor(*shape*, ***kwargs*)

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*)

Single-bit 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*, *signed=False*)

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`()

Single-bit NOT in arithmetic circuits.

`bit_or`(*other*)

Single-bit 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*)

Single-bit 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).

`field_div`(*other*)

Field division of public values. Not available for computation modulo a power of two.

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

`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

iadd(*other*)

Addition assignment. This uses *update()* internally.

if_else(*a*, *b*)

MUX on bit in arithmetic circuits:

```
print_ln('%s', sint(0).if_else(sint(2), sint(3)).reveal())
print_ln('%s', sint(1).if_else(sint(4), sint(5)).reveal())
```

This will output:

```
3
4
```

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*, ***kwargs*)

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 – bytearray

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

Output.

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

Read shares from `Persistence/Transactions-[gf2n-]P<playerno>.data`. See *this section* for details on the data format.

Parameters

- **start** – starting position in number of shares from beginning (int/regint/cint)
- **n_items** – number of items (int)
- **crash_if_missing** – crash if file not found (default)
- **size** – vector size (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 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

to_regint_by_bit(**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_file(**args*, ***kwargs*)

Write shares to `Persistence/Transactions-[gf2n-]P<playerno>.data` (appending at the end). See *this section* for details on the data format.

Parameters

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

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()*. It is possible to convert to secret types like *sint*.

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

classmethod read_fix(*player*, *f*, *k*, *precision*)

Read fixed-point value from Player-Data/Input-Binary-P<player>-<threadnum> only on party player.

Parameters

- **player** – player (int)
- **f** – fixed-point precision (int)
- **k** – fixed-point length (int)
- **precision** – input precision (1: single, 2: double)

Returns
personal cfix

classmethod read_int(*player*, *n_bytes=None*)

Read integer from Player-Data/Input-Binary-P<player>-<threadnum> only on party player.

Parameters
player – player (int)

Returns
personal cint

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 MemValue(*value*)

classmethod Tensor(*shape*, ***kwargs*)

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*)

Single-bit 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()

Single-bit NOT in arithmetic circuits.

bit_or(*other*)

Single-bit 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*)

Single-bit 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

`iadd`(*other*)

Addition assignment. This uses *update()* internally.

`if_else`(*a*, *b*)

MUX on bit in arithmetic circuits:

```
print_ln('%s', sint(0).if_else(sint(2), sint(3)).reveal())
print_ln('%s', sint(1).if_else(sint(4), sint(5)).reveal())
```

This will output:

```
3
4
```

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*, ***kwargs*)

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. Made obsolete by *update()*.

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. Made obsolete by *update()*.

Parameters
value – any convertible type

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

Read shares from `Persistence/Transactions-[gf2n-]P<playerno>.data`. See *this section* for details on the data format.

Parameters

- **start** – starting position in number of shares from beginning (int/regint/cint)
- **n_items** – number of items (int)
- **crash_if_missing** – crash if file not found (default)
- **size** – vector size (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 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_file(**args*, ***kwargs*)

Write shares to `Persistence/Transactions-[gf2n-]P<playerno>.data` (appending at the end). See *this section* for details on the data format.

Parameters

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

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 *sintbit*. The other operand can be any of sfix/sint/cfix/regint/cint/int/float. It also supports `abs()` and `**`.

Note that the default precision (16 bits after the dot, 31 bits in total) only allows numbers up to $2^{31-16-1} \approx 16000$ with the smallest non-zero number being $2^{-16} \approx 0.000015$. You can change this using *set_precision()*.

Fixed-point multiplication is not linear in the sense of the computation domain. Therefore, techniques from *Non-linear Computation* have to be used.

Many operations (including multiplication and division) use probabilistic trunctation by default. This means that the results are not deterministc but random within a small range around the deterministic result. You can switch to (more expensive) deterministic computation by setting `sfix.round_nearest` to true. See Catrina and de Hoogh for an introduction to probabilistic truncation.

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

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

Type-dependent array. Example:

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

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

Type-dependent matrix. Example:

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

classmethod MemValue(*value*)

classmethod Tensor(*shape*, ***kwargs*)

Type-dependent tensor of any dimension:

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

bit_decompose(*n_bits=None*, ***kwargs*)

Bit decomposition.

compute_reciprocal(*args, **kwargs)

Secret fixed-point reciprocal.

dot(other)

Dot product with any vector or iterable.

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

equal(*args, **kwargs)

Comparison.

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

Returns
0/1

Return type
same as internal representation

classmethod get_input_from(*args, **kwargs)

Secret fixed-point input.

Parameters

- **player** – public (regint/cint/int)
- **binary** – whether to use binary (Input-Binary-P<playerno>-<threadno>) instead of cleartext (Input-P<playerno>-<threadno>)
- **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
- **symmetric** – symmetric distribution at higher cost
- **public_randomness** – use public randomness (avoids preprocessing)
- **size** – vector size (int, default 1)

greater_equal(*args, **kwargs)

Comparison.

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

Returns
0/1

Return type
same as internal representation

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

Comparison.

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

Returns
0/1

Return type
same as internal representation

iadd(*other*)

Addition assignment. This uses *update()* internally.

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=None*, *shape=None*, *binary=True*, *one_hot=False*, *skip_input=False*, *n_bytes=None*)

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` or `Player-Data/Input-Binary-P<player>-0` if running on another host.

Parameters

- **player** – player to input via (int)
- **content** – nested Python list or numpy array (binary mode only) or left out if not available
- **shape** – shape if content not given
- **binary** – binary mode (bool)
- **one_hot** – one-hot encoding (bool)

less_equal(**args*, ***kwargs*)

Comparison.

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

Returns
0/1

Return type
same as internal representation

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

Comparison.

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

Returns
0/1

Return type
same as internal representation

`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

`not_equal`(**args*, ***kwargs*)

Comparison.

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

Returns
0/1

Return type
same as internal representation

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

Read shares from `Persistence/Transactions-P<playerno>.data`. Precision must be the same as when storing. See *this section* for details on the data format.

Parameters

- **start** – starting position in number of shares from beginning (int/regint/cint)
- **n_items** – number of items (int)
- **crash_if_missing** – crash if file not found (default)

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 via *sint.receive_from_client()*. Assumes client has already converted values to integer representation.

Parameters

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

Returns
list of length n

rep_type
alias of *cint*

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 via *sint.reveal_to_clients()*.

Parameters

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

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

classmethod set_precision(*f*, *k=None*)
Set the precision of the integer representation. 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`. See *this section* for details on the data format.

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. See Aliasgari et al. for details.

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 MemValue(*value*)

classmethod Tensor(*shape*, ***kwargs*)

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)

iadd(*other*)

Addition assignment. This uses *update()* internally.

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=None*, *shape=None*, *binary=True*, *one_hot=False*, *skip_input=False*, *n_bytes=None*)

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` or `Player-Data/Input-Binary-P<player>-0` if running on another host.

Parameters

- **player** – player to input via (int)
- **content** – nested Python list or numpy array (binary mode only) or left out if not available
- **shape** – shape if content not given
- **binary** – binary mode (bool)
- **one_hot** – one-hot encoding (bool)

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.

update(*other*)

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

Parameters
other – any convertible type

class Compiler.types.**sgf2n**(***kwargs*)

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

Parameters

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

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

Type-dependent array. Example:

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

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

Type-dependent matrix. Example:

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

classmethod MemValue(*value*)

classmethod Tensor(*shape*, ***kwargs*)

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*, *signed=False*)

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)

field_div(*other*)

Secret field division.

Parameters
other – any compatible type

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)

iadd(*other*)

Addition assignment. This uses *update()* internally.

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=None*, *shape=None*, *binary=True*, *one_hot=False*, *skip_input=False*, *n_bytes=None*)

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` or `Player-Data/Input-Binary-P<player>-0` if running on another host.

Parameters

- **player** – player to input via (int)
- **content** – nested Python list or numpy array (binary mode only) or left out if not available
- **shape** – shape if content not given

- **binary** – binary mode (bool)
- **one_hot** – one-hot encoding (bool)

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

Load from memory by public address.

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

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

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

Read shares from `Persistence/Transactions-[gf2n-]P<playerno>.data`. See *this section* for details on the data format.

Parameters

- **start** – starting position in number of shares from beginning (int/regint/cint)
- **n_items** – number of items (int)
- **crash_if_missing** – crash if file not found (default)
- **size** – vector size (int)

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

Returns
list of shares

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

classmethod write_to_file(**args*, ***kwargs*)

Write shares to `Persistence/Transactions-[gf2n-]P<playerno>.data` (appending at the end). See *this section* for details on the data format.

Parameters

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

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 a fixed-point division. 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. It defaults to the global parameters set by `program.set_bit_length()`, and its default 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. Modulo prime, the behaviour is undefined and potentially insecure if the operands are longer than the bit length.

Instances of sint are understood to be signed. This means that, for modulo N, numbers in $[0, N/2)$ are understood as positive numbers whereas numbers in $[N/2, N)$ are understood to be negative, namely $x - N$. This ensures expected arithmetic such as $-1 + 1 = (N - 1) + 1 = N = 0 \mod N$.

See *Non-linear Computation* for an overview of how non-linear computation is implemented.

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 MemValue(*value*)

classmethod Tensor(*shape*, ***kwargs*)

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*)

Single-bit 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*, *signed=False*)

Compose value from bits.

Parameters
bits – iterable of any type convertible to sint

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

Secret bit decomposition.

bit_not()

Single-bit NOT in arithmetic circuits.

bit_or(*other*)

Single-bit 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*)

Single-bit 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)

field_div(*other*)

Secret field division.

Parameters
other – any compatible type

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)
- **binary** – whether to use binary (`Input-Binary-P<playerno>-<threadno>`) instead of cleartext (`Input-P<playerno>-<threadno>`)
- **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(**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)

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

iadd(*other*)

Addition assignment. This uses *update()* internally.

if_else(*a*, *b*)

MUX on bit in arithmetic circuits:

```
print_ln('%s', sint(0).if_else(sint(2), sint(3)).reveal())
print_ln('%s', sint(1).if_else(sint(4), sint(5)).reveal())
```

This will output:

```
3
4
```

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=None*, *shape=None*, *binary=True*, *one_hot=False*, *skip_input=False*, *n_bytes=None*)

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` or `Player-Data/Input-Binary-P<player>-0` if running on another host.

Parameters

- **player** – player to input via (int)
- **content** – nested Python list or numpy array (binary mode only) or left out if not available
- **shape** – shape if content not given
- **binary** – binary mode (bool)
- **one_hot** – one-hot encoding (bool)

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

Secret integer division. Note that the domain bit length needs to be about four times the bit length.

Parameters

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

int_mod(*args, **kwargs)

Secret integer modulo. Note that the domain bit length needs to be about four times the bit length.

Parameters

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

left_shift(*other*, *bit_length=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(*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)

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*, **kwargs)

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(**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)

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

Secret power of two.

Parameters

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

prefix_sum(**args*, ***kwargs*)

Prefix sum.

private_division(*divisor*, *active=None*, *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(**args*, ***kwargs*)

Read shares from `Persistence/Transactions-[gf2n-]P<playerno>.data`. See *this section* for details on the data format.

Parameters

- **start** – starting position in number of shares from beginning (int/regint/cint)
- **n_items** – number of items (int)
- **crash_if_missing** – crash if file not found (default)
- **size** – vector size (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. unless `program.active` is set to false, in which case it uses random values to mask the clients' input.

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. Uses `program.active` to determine whether to use triples for active security.

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)
- **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

write_fully_to_socket(**args*, ***kwargs*)

Send full secret to socket

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

classmethod write_to_file(**args*, ***kwargs*)

Write shares to `Persistence/Transactions-[gf2n-]P<playerno>.data` (appending at the end). See *this section* for details on the data format.

Parameters

- **shares** – (list or iterable of shares)
- **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 (`&`, `|`, `^`).

6.2 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)
```

classmethod MemValue(*value*)

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*)

Bit-wise oblivious selection:

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

This will output 1 because it selects the two least significant bits from 5 and the rest of the bits from 2.

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)
```

classmethod MemValue(*value*)

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.**sbitfixvec**(*value=None*, *k=None*, **args*, ***kwargs*)

Values and vectors of fixed-point numbers for parallel binary computation:

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

Refer to the following example for the vector functionality:

```
a = sbitfixvec([sbitfixvec(0.3), sbitfixvec(0.5)])
b = sbitfixvec([sbitfixvec(0.4), sbitfixvec(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).elements()
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)
```

classmethod MemValue(*value*)

add(**args*, ***kwargs*)

Secret fixed-point addition.

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

bit_decompose(*n_bits=None*, ***kwargs*)

Bit decomposition.

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

Secret fixed-point reciprocal.

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

Comparison.

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

Returns
0/1

Return type
same as internal representation

classmethod get_input_from(*player*, *size=1*)

Secret input from `player`.

Param
player (int)

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

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

Parameters

- **lower** – float
- **upper** – float
- **symmetric** – symmetric distribution at higher cost
- **public_randomness** – use public randomness (avoids preprocessing)
- **size** – vector size (int, default 1)

greater_equal(**args*, ***kwargs*)

Comparison.

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

Returns
0/1

Return type
same as internal representation

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

Comparison.

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

Returns
0/1

Return type
same as internal representation

iadd(*other*)

Addition assignment. This uses *update()* internally.

less_equal(**args*, ***kwargs*)

Comparison.

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

Returns
0/1

Return type
same as internal representation

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

Comparison.

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

Returns
0/1

Return type
same as internal representation

`max`(*other*)

Maximum.

Parameters
`other` – any compatible type

`min`(*other*)

Minimum.

Parameters
`other` – any compatible type

`not_equal`(**args*, ***kwargs*)

Comparison.

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

Returns
0/1

Return type
same as internal representation

`reveal`()

Reveal secret fixed-point number.

Returns
relevant clear type

`round_nearest = False`

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

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

Set the precision of the integer representation. 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.**sbitintvec**(*elements=None*, *length=None*, *input_length=None*, *signed=False*)

Values and vectors of signed integers for parallel binary computation:

```
si32 = sbitintvec.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
```

The following example uses vectors of size two:

```
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).elements()
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)

`equal`(*other*, *bit_length=None*)

Return self==value.

`classmethod from_hex`(*string*, *reverse=True*)

Create from hexadecimal string (little-endian).

`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)

`iadd`(*other*)

Addition assignment. This uses `update()` internally.

`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

reveal_print_hex()

Reveal and print in hexademical (one line per element).

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 initialized from *regint* and *sint*.

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

Type-dependent array. Example:

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

classmethod MemValue(*value*)

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*)

Bit-wise oblivious selection:

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

This will output 1 because it selects the two least significant bits from 5 and the rest of the bits from 2.

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*, *signed=False*)

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 four 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]
```

4. Private input:

```
x = sbitvec.get_type(32).get_input_from(player)
```

`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)

`equal`(*other*, *bit_length=None*)

Return self==value.

`classmethod from_hex`(*string*, *reverse=True*)

Create from hexadecimal string (little-endian).

`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

`reveal_print_hex`()

Reveal and print in hexademical (one line per element).

6.3 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*, *players=None*)

Accept client connection on specific port base.

Parameters

- **port** – port base (int/regint/cint)
- **players** – subset of players (default: all)

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*, ***kwargs*)

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

The following should output 10:

```
n = 10
a = sint.Array(n)
x = sint(0)
```

(continues on next page)

(continued from previous page)

```
@for_range(n)
def _(i):
    a[i] = i
    x.update(x + 1)
print_ln('%s', x.reveal())
```

Compiler.library.**for_range_multithread**(*n_threads*, *n_parallel*, *n_loops*, *thread_mem_req={}*, *budget=None*, ***kwargs*)

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**(*start*, *stop=None*, *step=None*, *budget=None*, ***kwargs*)

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

- **`start/stop/step`** – int/regint/cint (used as in `range()`) or `start` only as list/tuple of int (see below). The optimization works best if all arguments are int or :py:obj:None. Otherwise, the budget has to be smaller than the number of loops for any optimization to take place.
- **`budget`** – number of instructions after which to start optimization (default is 1000 or as given with `--budget`)

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*, *budget=None*, ***kwargs*)

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):
    ...
```

Note that you cannot use registers across threads. Use *MemValue* instead:

```
a = MemValue(sint(0))
@for_range_opt_multithread(8, 80)
def _(i):
    b = a + 1
```

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

Decorator to execute a loop `n_loops` up to `n_parallel` loop bodies with optimized communication in a single thread. In most cases, it is easier to use *for_range_opt()*. Using any other control flow instruction inside the loop breaks the optimization.

Increasing the optimization parameter increases the the compilation time and memory usage (both during compilation and execution), but it potentially decreases the number of networking rounds.

Parameters

- **n_parallel** – optimization parameter (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.**function**(*function*)

Create a run-time function. The arguments can be memory or basic types, and return values can be basic types:

```
@function
def f(x, y, z):
    y.write(1)
    z[0] = 2
    return x + 3

a = MemValue(sint(0))
b = sint.Array(10)
c = f(sint(4), a, b)

print_ln('%s %s %s', a.reveal(), b[0].reveal(), c.reveal())
```

This should output:

```
1 2 7
```

You can use run-time functions recursively but without return values in this case.

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

Returns the thread argument.

Compiler.library.**get_cmdline_arg**(*idx*)

Return run-time command-line 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*, ***kwargs*)

Conditional execution without else block. Note that this does not work compile-time (Python) data structures, so if you change lists, dictionaries, or objects, you might not get the desired result. You can avoid problems by creating container types like *Array* and *MemValue* *before* the condition and only using assignment operations inside.

Parameters
condition – regint/cint/int

Usage:

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

Compiler.library.**if_e**(*condition*, ***kwargs*)

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.**init_client_connection**(*host*, *port*, *my_id*, *relative_port=True*)

Initiate connection to another party as client.

Parameters

- **host** – hostname
- **port** – port base (int/regint/cint)
- **my_id** – client id to use
- **relative_port** – whether to add party number to port number

Returns
connection id

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
```

(continues on next page)

(continued from previous page)

```
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*, ***kwargs*)

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*, *get_counters=False*)

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(3, 25)
def f(base, size):
    ...
```

This can be used for example to distribute multiplications across several threads:

```
a = sint.Array(n)
b = sint.Array(n)
c = sint.Array(n)

@multithread(n_threads, n, max_size=max_batch_size)
def f(base, size):
    c.assign_vector(a.get_vector(base=base, size=size) *
                    b.get_vector(base=base, size=size), base=base)
```

This multiplies a and b across n_threads threads in batches of size at most max_batch_size.

Compiler.library.**print_both**(*s*, **args*, *end='\n'*)

Print line during compilation and execution.

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*, ***kwargs*)

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)
- **print_secrets** – whether to output secret shares

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* or use *-OF .* 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_secrets=False*)

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

Parameters

- **s** – format string

- **args** – arguments (any type)
- **print_secrets** – whether to output secret shares

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*, ***kwargs*)

While-do loop.

Parameters
condition – function returning public integer (regint/cint/int)

The following executes an ten-fold loop:

```
i = regint(0)
@while_do(lambda: i < 10)
def f():
    ...
    i.update(i + 1)
    ...
```

6.4 Compiler.mpc_math module

Module for math operations.

Most of the functionality is due to Aly and Smart with some optimizations by Keller and Sun.

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*, *zero_output=False*)

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*)

Square root.

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.

6.5 Compiler.ml module

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

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

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

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

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

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

Inference can be run as follows:

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

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

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

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

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

Bases: *Optimizer*

Adam/AMSgrad optimizer.

Parameters

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

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

Returns

sfix/sint Array (depening on `top`)

fit(*X*, *Y*, *epochs=1*, *batch_size=128*, *validation_data=(None, None)*, *program=None*, *reset=True*, *print_accuracy=False*, *print_loss=False*, *sample_mask=None*)

Train model.

Parameters

- **X** – training sample data (sfix tensor)
- **Y** – training labels (sint/sfix tensor)
- **epochs** – number of epochs (int)
- **batch_size** – batch size (int)
- **validation_data** – tuple of test sample data and labels for accuracy testing (optional; reveals labels)

- **program** – Program instance to use command-line parameters (optional)
- **reset** – whether to initialize model
- **print_accuracy** – print accuracy on training data (reveals labels)
- **print_loss** – reveal and print training loss after every batch
- **sample_mask** – 0/1 vector or Array to mask samples (experimental, only for 0/1 labels, 0 means ignore sample)

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.

reveal_correctness(*data*, *truth*, *batch_size=128*, *running=False*)

Test correctness by revealing results.

Parameters

- **data** – test sample data
- **truth** – test labels
- **batch_size** – batch size
- **running** – output after every batch

reveal_model_to_binary()

Reveal model and store it in the binary output file, see *Exporting models* for details.

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.

property trainable_variables

List of all trainable variables.

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

Bases: `NoVariableLayer`

Fixed-point addition layer.

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

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

Bases: NoVariableLayer

Fixed-point Argmax layer.

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

class Compiler.ml.**BatchNorm**(*shape*, *approx=True*, *args=None*)

Bases: 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: NoVariableLayer

Fixed-point concatentation layer.

Parameters

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

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

Bases: DenseBase

Fixed-point dense (matrix multiplication) layer. Supports inputs of size [N, d, d_in] to map to [N, d, d_out]. If d > 1, the layer behaves like torch's nn.Linear which loops over the additional d

Parameters

- **N** – number of examples
- **d_in** – input dimension
- **d_out** – output dimension
- **d** – (optional) extra dimension

class Compiler.ml.**Dropout**(*N*, *d1=None*, *d2=1*, *alpha=0.5*)

Bases: NoVariableLayer

Dropout layer.

Parameters

- **N** – number of examples
- **shape** – list [N, . . .] where N is the number of examples and an arbitrary amount of further dimensions
- **alpha** – probability (power of two)

class Compiler.ml.**FixAveragePool2d**(*input_shape*, *output_shape*, *filter_size*, *strides=(1, 1)*, *padding=0*)

Bases: PoolBase, FixBase

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 (int or tuple/list of two int)
- **strides** – strides (int or tuple/list of two int)
- **padding** – 'SAME', 'VALID', int, or 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*, *weight_type=None*, *bias=True*)

Bases: Conv2d, FixBase

Fixed-point 2D convolution layer.

Parameters

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

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

Bases: Layer

Fixed-point fused batch normalization layer (inference only).

Parameters

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

class Compiler.ml.**Gelu**(*shape*, *inputs=None*, *approx=True*)

Bases: ElementWiseLayer

Fixed-point GeLU layer. Based on Dong et al., "PUMA: SECURE INFERENCE OF LLAMA-7B IN FIVE MINUTES"

Parameters

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

prime_type

alias of *sfix*

class Compiler.ml.**LayerNorm**(*shape*, *approx=True*, *layernorm_eps=None*, *args=None*)

Bases: Layer

Fixed-point layer normalization layer. For now assumes we only want to normalize over the last dimension

Parameters

- **shape** – input/output shape (tuple/list of any number of int)
- **approx** – use approximate square root

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

Bases: PoolBase

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), 'SAME', integer, or list/tuple of integers

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

Bases: MultiOutputBase

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

Parameters

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

class Compiler.ml.**Optimizer**(*layers=[]*, *report_loss=None*, *time_layers=False*, *program=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

Returns

sfix/sint Array (depening on top)

fit(*X*, *Y*, *epochs=1*, *batch_size=128*, *validation_data=(None, None)*, *program=None*, *reset=True*, *print_accuracy=False*, *print_loss=False*, *sample_mask=None*)

Train model.

Parameters

- **X** – training sample data (sfix tensor)
- **Y** – training labels (sint/sfix tensor)
- **epochs** – number of epochs (int)
- **batch_size** – batch size (int)

- **validation_data** – tuple of test sample data and labels for accuracy testing (optional; reveals labels)
- **program** – `Program` instance to use command-line parameters (optional)
- **reset** – whether to initialize model
- **print_accuracy** – print accuracy on training data (reveals labels)
- **print_loss** – reveal and print training loss after every batch
- **sample_mask** – 0/1 vector or Array to mask samples (experimental, only for 0/1 labels, 0 means ignore sample)

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.

reveal_correctness(*data*, *truth*, *batch_size=128*, *running=False*)

Test correctness by revealing results.

Parameters

- **data** – test sample data
- **truth** – test labels
- **batch_size** – batch size
- **running** – output after every batch

reveal_model_to_binary()

Reveal model and store it in the binary output file, see *Exporting models* for details.

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.

property trainable_variables

List of all trainable variables.

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

Bases: OutputBase

Fixed-point logistic regression output layer.

Parameters

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

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

Bases: ElementWiseLayer

Fixed-point ReLU layer.

Parameters

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

prime_type

alias of *sint*

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

Bases: MultiOutputBase

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

Parameters

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

class Compiler.ml.**SGD**(*layers*, *n_epochs=1*, *debug=False*, *report_loss=None*)

Bases: *Optimizer*

Stochastic gradient descent.

Parameters

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

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

Returns

sfix/sint Array (depening on top)

fit(*X*, *Y*, *epochs=1*, *batch_size=128*, *validation_data=(None, None)*, *program=None*, *reset=True*, *print_accuracy=False*, *print_loss=False*, *sample_mask=None*)

Train model.

Parameters

- **X** – training sample data (sfix tensor)
- **Y** – training labels (sint/sfix tensor)
- **epochs** – number of epochs (int)
- **batch_size** – batch size (int)
- **validation_data** – tuple of test sample data and labels for accuracy testing (optional; reveals labels)
- **program** – `Program` instance to use command-line parameters (optional)
- **reset** – whether to initialize model
- **print_accuracy** – print accuracy on training data (reveals labels)
- **print_loss** – reveal and print training loss after every batch
- **sample_mask** – 0/1 vector or Array to mask samples (experimental, only for 0/1 labels, 0 means ignore sample)

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(***kwargs*)

Reset layer parameters.

Parameters

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

reveal_correctness(*data*, *truth*, *batch_size=128*, *running=False*)

Test correctness by revealing results.

Parameters

- **data** – test sample data
- **truth** – test labels
- **batch_size** – batch size
- **running** – output after every batch

reveal_model_to_binary()

Reveal model and store it in the binary output file, see *Exporting models* for details.

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.

`property trainable_variables`

List of all trainable variables.

`class Compiler.ml.SGDLinear`(*n_epochs=1*, *batch_size=1*, *program=None*)

Bases: `OneLayerSGD`

Linear regression using SGD.

Parameters

- **n_epochs** – number of epochs
- **batch_size** – batch size
- **program** – program object to use command-line options from (default is not to use any)

`fit`(*X_train*, *y_train*, ***kwargs*)

Train classifier.

Parameters

- **X_train** – training data (sfix matrix)
- **y_train** – training binary labels (sint/sfix array)
- **sample_mask** – sample masking (see *Optimizer.fit()*)

`fit_with_testing`(*X_train*, *y_train*, *X_test*, *y_test*, ***kwargs*)

Train classifier with accuracy output after every epoch. This reveals all labels to simplify the accuracy computation.

Parameters

- **X_train** – training data (sfix matrix)
- **y_train** – training labels (sint/sfix array)
- **X_test** – testing data (sfix matrix)
- **y_test** – testing labels (sint/sfix array)
- **sample_mask** – sample masking (see *Optimizer.fit()*)

`predict`(*X*)

Use model for prediction.

Parameters
X – sample data with row-wise samples (sfix matrix)

Returns
sfix array

`class Compiler.ml.SGDLogistic`(*n_epochs=1*, *batch_size=1*, *program=None*)

Bases: `OneLayerSGD`

Logistic regression using SGD. The member `opt` refers to the internal instance of *Optimizer*, which allows to use the funcionality therein.

Parameters

- **n_epochs** – number of epochs

- **batch_size** – batch size
- **program** – program object to use command-line options from (default is not to use any)

fit(*X_train*, *y_train*, ***kwargs*)

Train classifier.

Parameters

- **X_train** – training data (sfix matrix)
- **y_train** – training binary labels (sint/sfix array)
- **sample_mask** – sample masking (see *Optimizer.fit()*)

fit_with_testing(*X_train*, *y_train*, *X_test*, *y_test*, ***kwargs*)

Train classifier with accuracy output after every epoch. This reveals all labels to simplify the accuracy computation.

Parameters

- **X_train** – training data (sfix matrix)
- **y_train** – training labels (sint/sfix array)
- **X_test** – testing data (sfix matrix)
- **y_test** – testing labels (sint/sfix array)
- **sample_mask** – sample masking (see *Optimizer.fit()*)

predict(*X*)

Use model to predict labels.

Parameters
X – sample data with row-wise samples (sfix matrix)

Returns
sint array

predict_proba(*X*)

Use model for probility estimates.

Parameters
X – sample data with row-wise samples (sfix matrix)

Returns
sfix array

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

Bases: ElementWiseLayer

Fixed-point square layer.

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

prime_type

alias of *sfix*

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

Bases: *MultiOutput*

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

Compute index of maximum element.

Parameters
x – iterable

Returns
sint or 0 if x has length 1

Compiler.ml.**cholesky**(*A*, *reveal_diagonal=False*)

Cholesky decomposition.

Returns
lower triangular matrix

Compiler.ml.**easyConv2d**(*input_shape*, *batch_size*, *out_channels*, *kernel_size*, *stride=1*, *padding=0*, *bias=True*, ***kwargs*)

More convenient interface to *FixConv2d*.

Parameters

- **input_shape** – input shape (tuple/list of four int)
- **out_channels** – output channels (int)
- **kernel_size** – kernel size (int or tuple/list of two int)
- **stride** – stride (int or tuple/list of two int)
- **padding** – 'SAME', 'VALID', int, or tuple/list of two int
- **bias** – whether layer has bias (bool)

Compiler.ml.**easyMaxPool**(*input_shape*, *kernel_size*, *stride=None*, *padding=0*)

More convenient interface to *MaxPool*.

Parameters

- **input_shape** – input shape (tuple/list of four int)
- **kernel_size** – kernel size (int or tuple/list of two int)
- **stride** – stride (int or tuple/list of two int)
- **padding** – 'SAME', 'VALID', int, or tuple/list of two int

Compiler.ml.**layers_from_torch**(*model*, *data_input_shape*, *batch_size*, *input_via=None*, *regression=False*, *layer_args={}*, *program=None*)

Convert a PyTorch Module object to MP-SPDZ layers.

Parameters

- **model** – PyTorch Module object
- **data_input_shape** – input shape (list of four int)
- **batch_size** – batch size (int)
- **input_via** – player to input model data via (default: don't)
- **regression** – regression (default: classification)

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

Iterative matrix inverse approximation. This is based on the conjugate gradients algorithm in Section 10.2.4 of these lecture notes.

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_cholesky**(*A*, *b*, *debug=False*)

Linear solver using Cholesky decomposition.

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.**solve_lower**(*A*, *b*)

Linear solver where A is lower triangular quadratic.

Compiler.ml.**solve_upper**(*A*, *b*)

Linear solver where A is upper triangular quadratic.

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

6.6 Compiler.decision_tree module

class Compiler.decision_tree.**TreeClassifier**(*max_depth*, *n_threads=None*)

Tree classification with convenient interface. Uses *TreeTrainer* internally.

Parameters

- **max_depth** – the depth of the decision tree
- **n_threads** – number of threads used in training

fit(*X*, *y*, *attr_types=None*)

Train tree.

Parameters

- **X** – sample data with row-wise samples (sint/sfix matrix)
- **y** – binary labels (sint list/array)

fit_with_testing(*X_train*, *y_train*, *X_test*, *y_test*, *attr_types=None*, *output_trees=False*, *debug=False*)

Train tree with accuracy output after every level.

Parameters

- **X_train** – training data with row-wise samples (sint/sfix matrix)
- **y_train** – training binary labels (sint list/array)
- **X_test** – testing data with row-wise samples (sint/sfix matrix)
- **y_test** – testing binary labels (sint list/array)
- **attr_types** – attributes types (list of 'b'/'c' for binary/continuous; default is all continuous)
- **output_trees** – output tree after every level
- **debug** – output debugging information

output()

Output decision tree.

predict(*X*)

Use tree for prediction.

Parameters

X – sample data with row-wise samples (sint/sfix matrix)

Returns

sint array

class Compiler.decision_tree.**TreeTrainer**(*x*, *y*, *h*, *binary=False*, *attr_lengths=None*, *n_threads=None*)

Decision tree training by Hamada et al.

Parameters

- **x** – sample data (by attribute, list or *Matrix*)
- **y** – binary labels (list or sint vector)

- **h** – height (int)
- **binary** – binary attributes instead of continuous
- **attr_lengths** – attribute description for mixed data (list of 0/1 for continuous/binary)
- **n_threads** – number of threads (default: single thread)

train()

Train and return decision tree.

train_with_testing(**test_set*, *output=False*)

Train decision tree and test against test data.

Parameters

- **y** – binary labels (list or sint vector)
- **x** – sample data (by attribute, list or *Matrix*)
- **output** – output tree after every level

Returns

tree

Compiler.decision_tree.**output_decision_tree**(*layers*)

Print decision tree output by *TreeTrainer*.

Compiler.decision_tree.**preprocess_pandas**(*data*)

Preprocess pandas data frame to suit *TreeClassifier* by expanding non-continuous attributes to several binary attributes as a unary encoding.

Returns

a tuple of the processed data and a type list for the `attr_types` argument.

Compiler.decision_tree.**run_decision_tree**(*layers*, *data*)

Run decision tree against sample data.

Parameters

- **layers** – tree output by *TreeTrainer*
- **data** – sample data (*Array*)

Returns

binary label

6.7 Compiler.circuit module

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

```
make Programs/Circuits
```

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

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

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

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

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

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

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

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

```
from circuit import ieee_float

values = []

for i in range(2):
    for j in range(10):
        values.append(sbitintvec.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.**sha256**(*x*)

This function implements SHA2-256:

```
from circuit import sha256
a = sbitvec.from_vec([])
b = sbitvec.from_hex('d3', reverse=False)
c = sbitvec.from_hex(

↪'3ebfb06db8c38d5ba037f1363e118550aad94606e26835a01af05078533cc25f2f39573c04b632f62f68c294ab31f2a3
↪',
  reverse=False)
```

(continues on next page)

(continued from previous page)

```
d = sbitvec.from_hex(
↪'5a86b737eaea8ee976a0a24da63e7ed7eefad18a101c1211e2b3650c5187c2a8a650547208251f6d4237e661c7bf4c77
↪',
  reverse=False)

for x in a, b, c:
  sha256(x).reveal().print_reg()
```

This should output the hashes of the above inputs, namely the test vectors of SHA2-256 for 0, 8, 440, 512 bits:

```
Reg[0] = 0xe3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855 #
Reg[0] = 0x28969cdfa74a12c82f3bad960b0b000aca2ac329deea5c2328ebc6f2ba9802c1 #
Reg[0] = 0x6595a2ef537a69ba8583dfbf7f5bec0ab1f93ce4c8ee1916eff44a93af5749c4 #
Reg[0] = 0x42e61e174fbb3897d6dd6cef3dd2802fe67b331953b06114a65c772859dfc1aa #
```

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

This function implements SHA3-256 for inputs of any length:

```
from circuit import sha3_256
a = sbitvec.from_vec([])
b = sbitvec.from_hex('cc')
c = sbitvec.from_hex('41fb')
d = sbitvec.from_hex('1f877c')
e = sbitvec.from_vec([sbit(0)] * 8)
f = sbitvec.from_hex(
↪'41fb683492842387483289298398472828923894982792928374385838282837 2f17188141fb683492842387483289
↪')
g = sbitvec.from_hex(
↪'41fb683492842387483289298398472828923894982792928374385838282837 2f17188141fb683492842387483289
↪')
h = sbitvec.from_vec([sbit(0)] * 3000)
for x in a, b, c, d, e, f, g, h:
    sha3_256(x).reveal_print_hex()
```

This should output the hashes of the above inputs, beginning with the test vectors of SHA3-256 for 0, 8, 16, and 24 bits as well as the hash of the 0 byte:

```
Reg[0] = 0xa7ffc6f8bf1ed76651c14756a061d662f580ff4de43b49fa82d80a4b80f8434a #
Reg[0] = 0x677035391cd3701293d385f037ba32796252bb7ce180b00b582dd9b20aaad7f0 #
Reg[0] = 0x39f31b6e653dfcd9caed2602fd87f61b6254f581312fb6eeec4d7148fa2e72aa #
Reg[0] = 0xbc22345e4bd3f792a341cf18ac0789f1c9c966712a501b19d1b6632ccd408ec5 #
Reg[0] = 0x5d53469f20fef4f8eab52b88044ede69c77a6a68a60728609fc4a65ff531e7d0 #
Reg[0] = 0xf5f673ec50d662039871fd53fae3ced069baf09030132d6d60d2ba7040b02b18 #
Reg[0] = 0xa8a42e808f9dc0f43366d5de91511f42e9c3f8f37de0307f010bf629401edd2a #
Reg[0] = 0xf722631013ecacd42b4c7259e9fe22b8c81a86e9fe0d4a626800e7f50c5a8978 #
```

6.8 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 instance is available as `program`. When compiling directly from Python code, an instance has to be created before running any instructions.

property active

Whether to use actively secure protocols.

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*, *finalize=True*, ***kwargs*)

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

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

Parameters

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

Returns
tape handle

options_from_args()

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

protect_memory(*status*)

Enable or disable memory protection.

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.

6.9 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)
```

The introductory book by Evans et al. contains a chapter dedicated to oblivious RAM.

class Compiler.oram.**LinearORAM**(**args*, ***kwargs*)

Contiguous ORAM that stores entries in order and accesses the entire array for reading and writing in order to hide the address.

Parameters

- **size** – number of entries
- **value_type** – `sint` (default) / `sg2fn` / `sfix`
- **value_length** – number of values per entry (default: 1)

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

Create an ORAM instance suitable for the size based on experiments. This uses *LinearORAM* for sizes up to a few thousand and *RecursiveORAM* above that.

Parameters

- **size** – number of entries
- **value_type** – `sint` (default) / `sg2fn` / `sfix`
- **value_length** – number of values per entry (default: 1)

class Compiler.oram.**RecursiveORAM**(*size*, *value_type=None*, *value_length=1*, *entry_size=None*, *bucket_oram=<class 'Compiler.oram.TrivialORAM'>*, *init_rounds=-1*)

Secure tree ORAM using secure index. This uses the approach by Keller and Scholl.

Parameters

- **size** – number of entries
- **value_type** – `sint` (default) / `sg2fn` / `sfix`
- **value_length** – number of values per entry (default: 1)

index_structure

alias of `RecursiveIndexStructure`

Compiler.oram.**decompose_demux**(*x*, *n_bits*, *n_threads=None*)

Bit decomposition followed by *demux_matrix()*.

Parameters

- **x** – secret inputs (vector, `Array`, or list)
- **n_bits** – number of bits in entries
- **n_threads** – number of threads (default is running in same thread)

Returns

$2^n \times m$-matrix of secret bits

Compiler.oram.**demux_matrix**(*x*, *n_threads=None*)

Demuxing (converting from binary to unary represention).

Parameters

- **x** – $n \times m$-matrix of secret bits (m entries of n bits)
- **res** – array for result if given (default is to allocate)
- **n_threads** – number of threads (default is running in same thread)

Returns
$2^n \times m$-matrix of secret bits

6.10 Compiler.sqrt_oram module

class Compiler.sqrt_oram.**SqrtOram**(*data: ~Compiler.sqrt_oram.T | ~Compiler.types.MultiArray*, *entry_length: int = 1*, *value_type: ~typing.Type[~Compiler.sqrt_oram.T] = <class 'Compiler.types.sint'>*, *k: int = 0*, *period: int | None = None*, *initialize: bool = True*, *empty_data=False*)

Oblivious RAM using the "Square-Root" algorithm.

Parameters

- **data** (MultiArray) – The data with which to initialize the ORAM. One may provide a MultiArray such that every "block" can hold multiple elements (an Array).
- **value_type** (sint) – The secret type to use, defaults to sint.
- **k** (*int*) – Leave at 0, this parameter is used to recursively pass down the depth of this ORAM.
- **period** (*int*) – Leave at None, this parameter is used to recursively pass down the top-level period.

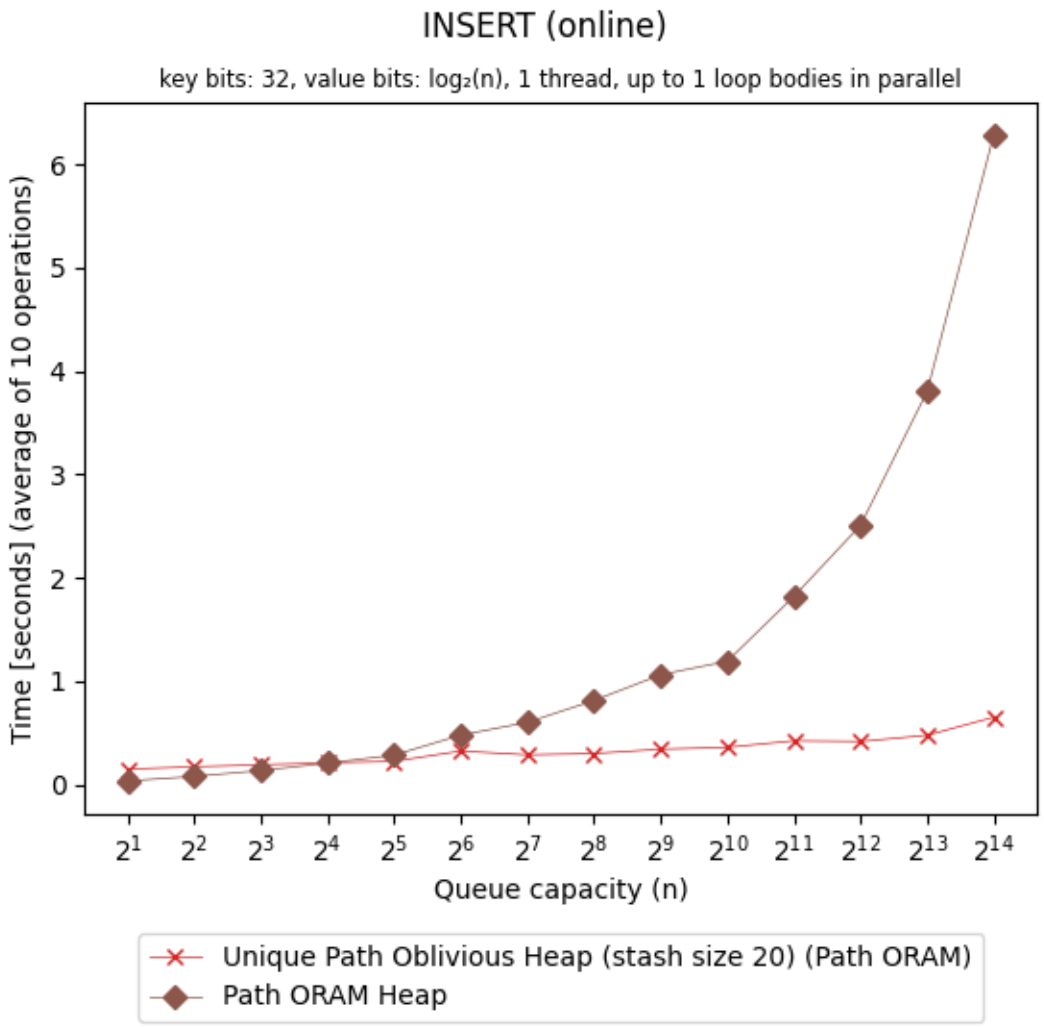
6.11 Compiler.path_oblivious_heap module

This module contains an implementation of the "Path Oblivious Heap" oblivious priority queue as proposed by Shi.

Path Oblivious Heap comes in two variants that build on either Path ORAM or Circuit ORAM. Both variants support inserting an element and extracting the element with the highest priority in time $O(\max(\log(n) + s, e))$ where n is the queue capacity, s is the ORAM stash size, and e is the ORAM eviction complexity. Assuming $s = O(1)$ and $e = O(\log(n))$, the operations are in $O(\log n)$. Currently, only the Path ORAM variant is implemented and tested (the *PathObliviousHeap*).

Furthermore, the *UniquePathObliviousHeap* class implements an *update()* operation that is comparable to that of HeapQ from dijkstra, in that it ensures that every value inserted in the queue is unique, and if *update()* is called with a value that is already in the queue, the priority of that value is updated to be equal to the new priority.

The following benchmark compares the online time of updating an element in HeapQ on top of Path ORAM and updating an element in *UniquePathObliviousHeap* on top of Path ORAM. *PathObliviousHeap* indeed seems to outperform HeapQ from around $n = 2^4$.



class Compiler.path_oblivious_heap.**POHToHeapQAdapter**(*max_size*, **args*, *int_type=<class 'Compiler.types.sint'>*, *variant=Path*, *bucket_size=None*, *stash_size=None*, *init_rounds=-1*, *entry_size=None*, ***kwargs*)

Adapts Path Oblivious Heap to the HeapQ interface, allowing plug-and-play replacement in the Dijkstra implementation.

extract_min(*fake: bool = False*) → _secret | None

Extract the element with the smallest (ie. highest) priority from the queue.

find_min(*fake: bool = False*) → _secret | None

Find the element with the smallest (ie. highest) priority in the queue and return its value and priority. Returns -1 if empty.

insert(*value*, *priority*, *fake: bool = False*) → None

Insert an element with a priority into the queue.

pop(*for_real=True*)

Renaming of pop to *extract_min()*.

update(*value*, *priority*, *for_real=True*)

Call *insert()* instead of update. Warning: When using this adapter, duplicate values are allowed to be inserted, and no values are ever updated.

class Compiler.path_oblivious_heap.**PathObliviousHeap**(*capacity: int*, *security: int | None = None*, *type_hiding_security: bool = False*, *int_type: ~typing.Type[~Compiler.types._secret] = <class 'Compiler.types.sint'>*, *entry_size: ~typing.Tuple[int] | None = None*, *variant: ~Compiler.path_oblivious_heap.POHVariant = Path*, *bucket_oram: ~Compiler.oram.AbstractORAM = <class 'Compiler.oram.TrivialORAM'>*, *bucket_size: int | None = None*, *stash_size: int | None = None*, *init_rounds: int = -1*)

A basic Path Oblivious Heap implementation supporting insert, extract_min, and find_min.

The queue is guaranteed to have at least the specified capacity with negligible error probability.

If inserting more entries than there is capacity for, the behavior depends on the value of the flag `oram.crash_on_overflow`. If the flag is set, the program crashes. Otherwise, the entry is simply not inserted.

Variables

- **capacity** – The capacity of the queue.
- **type_hiding_security** – A boolean indicating whether type hiding security is enabled. Enabling this makes the cost of every operation equal to the sum of the costs of all operations. This is initially set by passing an argument to the class constructor.
- **int_type** – The secret integer type of entry members.
- **entry_size** – A tuple specifying the bit lengths of the entries in the order (priority, value).

Iver tree
The MinTree data structure storing subtree-mins

extract_min(*fake: bool = False*) → _secret | None

Extract the element with the smallest (ie. highest) priority from the queue.

find_min(*fake: bool = False*) → _secret | None

Find the element with the smallest (ie. highest) priority in the queue and return its value and priority. Returns -1 if empty.

insert(*value*, *priority*, *fake: bool = False*) → None

Insert an element with a priority into the queue.

class Compiler.path_oblivious_heap.**UniquePOHToHeapQAdapter**(*max_size*, **args*, *int_type=<class 'Compiler.types.sint'>*, *variant=Path*, *oram_type=<function OptimalORAM>*, *bucket_size=None*, *stash_size=None*, *init_rounds=-1*, *entry_size=None*, ***kwargs*)

Adapts Unique Path Oblivious Heap to the HeapQ interface, allowing plug-and-play replacement in the Dijkstra implementation.

extract_min(*fake: bool = False*) → _secret | None

Extract the element with the smallest (ie. highest) priority from the queue.

find_min(*fake: bool = False*) → _secret | None

Find the element with the smallest (ie. highest) priority in the queue and return its value and priority. Returns -1 if empty.

insert(*value*, *priority*, *for_real=True*) → None

Insert an element with a priority into the queue.

pop(*for_real=True*)

Renaming of pop to *extract_min()*.

update(*value*, *priority*, *for_real=True*)

Update the priority of an entry with a given value. If such an entry does not already exist, it is inserted.

class Compiler.path_oblivious_heap.**UniquePathObliviousHeap**(**args*, *oram_type=<function OptimalORAM>*, *init_rounds=-1*, ***kwargs*)

A Path Oblivious Heap that ensures that all values in the queue are unique and supports updating a value with a new priority by maintaining a value to leaf index map using ORAM.

extract_min(*fake: bool = False*) → _secret | None

Extract the element with the smallest (ie. highest) priority from the queue.

find_min(*fake: bool = False*) → _secret | None

Find the element with the smallest (ie. highest) priority in the queue and return its value and priority. Returns -1 if empty.

insert(*value*, *priority*, *fake: bool = False*) → None

Insert an element with a priority into the queue.

update(*value*, *priority*, *empty=0*, *fake=False*)

Update the priority of an entry with a given value. If such an entry does not already exist, it is inserted.

Compiler.path_oblivious_heap.**path_oblivious_sort**(*keys: Array*, *values: Array*, *key_length: int*, *value_length: int | None = None*, ***kwargs*)

Sort values in place according to keys using Path Oblivious Heap by calling insert followed by extract min.

6.12 Compiler.dijkstra module

This module implements Dijkstra's algorithm based on oblivious RAM.

Compiler.dijkstra.**convert_graph**(*G*)

Convert a NetworkX directed graph to the cleartext representation of what *dijkstra()* expects.

Compiler.dijkstra.**dijkstra**(*source*, *edges*, *e_index*, *oram_type*, *n_loops=None*, *int_type=None*, *debug=False*)

Securely compute Dijstra's algorithm on a secret graph. See `../Programs/Source/dijkstra_example.mpc` for an explanation of the required inputs.

Parameters

- **source** – source node (secret or clear-text integer)
- **edges** – ORAM representation of edges
- **e_index** – ORAM representation of vertices
- **oram_type** – ORAM type to use internally (default: *OptimalORAM()*)
- **n_loops** – when to stop (default: number of edges)

- **int_type** – secret integer type (default: sint)

Compiler.dijkstra.**test_dijkstra**(*G*, *source*, *oram_type=<function OptimalORAM>*, *n_loops=None*, *int_type=<class 'Compiler.types.sint'>*)

Securely compute Dijstra's algorithm on a cleartext graph.

Parameters

- **G** – directed graph with NetworkX interface
- **source** – source node (secret or clear-text integer)
- **n_loops** – when to stop (default: number of edges)
- **int_type** – secret integer type (default: sint)

6.13 Compiler.sorting module

Compiler.sorting.**gen_multi_bit_perm**(*x*, *l*, *time=None*, *maybe_mixed=True*, *n_threads=None*)

GenMultiBitSort by Asharov et al.

Parameters

- **x** – entries of at most `l` bits (vector or array)
- **l** – number of bits to consider (int)
- **maybe_mixed** – whether to use mixed circuits if inputs are binary rather than arithmetic
- **n_threads** – number of threads (default is to run in the same thread)

Compiler.sorting.**quicksort**(*k*, *n_bits*, *debug=False*, *verbose=False*, *time=False*, *n_threads=None*, *return_permutation=False*)

Quicksort by Araki et al..

Parameters

- **k** – array to be sorted
- **n_bits** – bit size of entries
- **debug** – debugging
- **verbose** – output round numbers
- **time** – granular timings
- **n_threads** – number of threads (default is to run in the same thread)
- **return_permutation** – whether to return the permutation instead of applying it

Returns

if requested, tuple of oblivious shuffle and cleartext permutation

Compiler.sorting.**radix_sort**(*k*, *D=None*, *n_bits=None*, *signed=True*, *n_threads=None*, *l=2*, ***kwargs*)

Sort in place according to key.

Parameters

- **k** – keys (vector or Array of sint or sfix)
- **D** – Array or MultiArray to sort
- **n_bits** – number of bits in keys (int)
- **signed** – whether keys are signed (bool)

- **n_threads** – number of threads (default is to run in the same thread)
- **l** – number of bits to process at once

Compiler.sorting.**reveal_sort**(*k*, *D*, *reverse=False*, *n_threads=None*, *time=False*, *return_shuffle=False*, *shuffle=None*, *debug=False*)

Sort in place according to "perfect" key. The name hints at the fact that a random order of the keys is revealed.

Parameters

- **k** – vector or Array of sint containing exactly $0, \ldots, n-1$ in any order
- **D** – Array or MultiArray to sort
- **reverse** – whether key is a permutation in forward or backward order
- **n_threads** – number of threads (default is to run in the same thread)
- **time** – use timers for granular benchmarking
- **return_shuffle** – whether to return the secure shuffle
- **shuffle** – previously generated secure shuffle (default generate new one)
- **debug** – debugging output (insecure)

6.14 Compiler.sparse module

This module contains code for the sparse matrix multiplication by Keller as well as the graph cycle detection built on top.

class Compiler.sparse.**CellSparseMatrix**(*content*, *n_rows*, *n_cols*)

Cell-wise sparse matrix contain tuples of row index, column index, content, and zero indicator.

get_tight_row_matrix(*sort=False*, *debug=False*, *fail=False*, *time=False*, *n_threads=None*, *max_non_zero=None*)

Convert back to row-wise *SparseMatrix*.

Parameters

- **sort** – sort row entries putting non-empty first
- **debug** – debugging output (insecure)
- **fail** – check sub-algorithms (insecure)
- **time** – granular timings
- **n_threads** – number of threads (default is to run in same thread)
- **max_non_zero** – maximum non-zero entries per row in result (default: determine from input)

Compiler.sparse.**GroupOr**(*g*, *x*, *max_group_size*, *n_threads=None*)

Group-wise OR.

Parameters

- **g** – group indicators (vector or array)
- **x** – inputs (vector or array)
- **max_group_size** – maximum size of a group
- **n_threads** – number of threads (default is to run in the same thread)

class Compiler.sparse.**SparseMatrix**(*n_rows=None*, *n_cols=None*, *max_non_zero=None*)

Sparse matrix in row-wise and/or column-wise representation. Both are three-dimensional tensors. The first index represents rows or columns, and the second the sparse entries thereof. The third dimension are tuples consisting of index (column for row-wise and vice versa), content, and non-zero indicator. The constructs constructs an empty matrix, and *from_graph()* constructs an adjacency matrix for a graph.

find_cycles(*max_cycle*, *debug=False*, *fail=False*, *avg_output_limit=None*, *output=False*, *follow=None*, *n_threads=None*, *time=False*, *n_heavy_threads=None*, *reduced=None*, ***kwargs*)

Find cycles in adjacency matrix.

Parameters

- **max_cycles** – maximum cycle length
- **debug** – print debugging information (insecure)
- **fail** – check sub-algorithms (insecure)
- **avg_output_limit** – maximum number of non-zero entries in intermediate matrices
- **output** – output results
- **follow** – indices for debugging outputs
- **n_threads** – number of threads (default is to run in same thread)
- **time** – granular timings
- **n_heavy_threads** – number of threads for first step (default is same as n_threads)
- **reduced** – pre-allocated matrix for intermediate results (default: fresh allocation)

Returns

matrix of results by cycle and entry

classmethod from_graph(*graph*, *randomize=False*, *dry=False*)

Construct sparse matrix from graph.

Parameters

- **graph** – graph with NetworkX interface
- **randomize** – use random indices for empty entries
- **dry** – don't process data, only use sizes

mul_without_removal(*other*, *avg_output_limit=None*, *balance_rows=False*, *time=False*, *debug=False*, *fail=False*, *n_threads=None*, *follow=None*, *n_heavy_threads=None*, ***kwargs*)

Multiplication by Damie et al.

Parameters

- **other** – multiplicand sparse matrix
- **avg_output_limit** – limit for average non-zero entries per row in product
- **balance_rows** – make sure every row appears once in output
- **time** – granular timings
- **debug** – output debugging information (insecure)
- **fail** – check correctness of sub-algorithms (insecure)
- **n_threads** – number of threads (default is to run in same thread)
- **follow** – list of indices to follow for debugging

- **n_heavy_threads** – number of threads for first part (default: same as `n_threads`)

Returns

CellSparseMatrix object

CHAPTER

SEVEN

BYTECODE UTILITIES

7.1 Memory usage

`Scripts/memory-usage.py <program-with-args>` gives you an estimate of the minimum RAM usage per party. The range is relatively large due to fact the bytecode is independent of the secret sharing.

7.2 Preprocessing usage

`Scripts/prep-usage.py <program-with-args>` gives you an upper limit for the usage of preprocessing data such as triples. Note that the exact number depends on the protocol in various ways, and that the usage is sometimes unpredictable resulting in `inf` given. For an exact number, you have to run the virtual machine or script using the `--verbose` argument.

7.3 Human-readable bytecode/circuit representation

`Scripts/decompile.py <program-with-args>` produces human-readable version of the bytecode in `Programs/Bytecode`. The filename format is `Programs/Bytecode/<program-with-args>-<tapeid>.asm`. For example, after compiling and decompiling the tutorial, you will find `Programs/Bytecode/tutorial-0.asm`. You can find the full list of tape names in the third line of `Programs/Schedule/tutorial.sch`. See *this section* for an explanation of instruction names.

CHAPTER

EIGHT

THE JOURNEY OF A PROGRAM

In this section, we will demonstrate the life cycle of a high-level program in MP-SPDZ.

Consider the following program:

```
print_ln('%s', sint(123).reveal())
```

It entails three steps: creating a constant secret sharing, revealing it to cleartext, and outputting it. The compilation will not execute any of this. Instead, it will create a description in a format specific to MP-SPDZ. For example, *reveal()* triggers a call to the constructor of *asm_open*, which will add an object thereof to a list of instructions.

Run the following to retrieve the human-readable representation of the computation:

```
echo print_ln('%s', sint(123).reveal()) > Programs/Source/journey.py
./compile.py -a debug journey
```

This will create `debug-journey-0` with the following content:

```
# journey-0--0
ldsi s0, 123 # 0
asm_open 3, True, c0, s0 # 1
print_reg_plain c0 # 2
print_char 10 # 3
use 0, 7, 1 # 4
# journey-0-memory-usage-1
ldmc c0, 8191 # 5
gldmc cg0, 8191 # 6
ldmint ci0, 8191 # 7
ldms s0, 8191 # 8
gldms sg0, 8191 # 9
active True # 10
```

The first block corresponds mostly to the program whereas the second block is more generic. More specifically:

```
ldsi s0, 123 # 0
```

ldsi loads constant values to secret registers, in this case 123 to the register `s0`.

```
asm_open 3, True, c0, s0 # 1
```

asm_open reveals values in secret registers to be stored in cleartext registers, in this case the content of `s0` to `c0`. The `True` argument triggers a correctness check in protocols where it is available, and the 3 indicates the number of arguments to follow as the instruction is batchable, that is, it can execute any number of

```
print_reg_plain c0 # 2
```

print_reg_plain outputs constant values to the console or a file, in this case the register `c0`.

```
print_char 10 # 3
```

print_char outputs a character to the console or a file, in this case the ASCII code for a new line.

```
use 0, 7, 1 # 4
```

use indicates the usage of preprocessing information or similar. This allows the virtual machine to account for resources before actually executing the program. This particular call indicates 1 opening (7) of sint (0). You can see the codes in `data_type` and `field_types` at the beginning of `Compiler/program.py`.

```
ldmc c0, 8191 # 5
gldmc cg0, 8191 # 6
ldmint ci0, 8191 # 7
ldms s0, 8191 # 8
gldms sg0, 8191 # 9
```

These instructions read memory cells to registers, for example *ldms*. In this context, the purpose is to indicate the memory usage. The addresses are all 8191 because 8192 is the default size for user memory given in `Compiler/config.py`. If you use *Array* or similar data-structures, these numbers will increase accordingly.

```
active True # 10
```

active indicates whether the program is compatible with active security.

The compilation above also creates `Programs/Bytecode/journey-0.bc`, the hexdump output of which looks as follows:

```
00000000  00 00 00 00 00 00 00 02  00 00 00 00 00 00 00 7b  |...............{|
00000010  00 00 00 00 00 00 00 a5  00 00 00 03 00 00 00 01  |................|
00000020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 b3  |................|
00000030  00 00 00 00 00 00 00 00  00 00 00 b4 00 00 00 0a  |................|
00000040  00 00 00 00 00 00 00 17  00 00 00 00 00 00 00 07  |................|
00000050  00 00 00 00 00 00 00 01  00 00 00 00 00 00 00 03  |................|
00000060  00 00 00 00 00 00 00 00  00 00 1f ff 00 00 00 00  |................|
00000070  00 00 01 03 00 00 00 00  00 00 00 00 00 00 1f ff  |................|
00000080  00 00 00 00 00 00 00 ca  00 00 00 00 00 00 00 00  |................|
00000090  00 00 1f ff 00 00 00 00  00 00 00 04 00 00 00 00  |................|
000000a0  00 00 00 00 00 00 1f ff  00 00 00 00 00 00 01 04  |................|
000000b0  00 00 00 00 00 00 00 00  00 00 1f ff 00 00 00 00  |................|
000000c0  00 00 00 e9 00 00 00 01                           |........|
000000c8
```

It consist of the instructions codes and the arguments in big-endian order. For example, 0x2 is the code for *ldsi*, 0xa5 is the code for *asm_open*, 0xb3 is the code for *print_reg_plain*, etc. You can also spot repeated occurrences of `1f ff`, which is the hexadecimal representation of 8191.

Finally, the compilation creates `Programs/Schedules/journey.sch`, which is a text file:

```
1
1
journey-0:11
```

(continues on next page)

(continued from previous page)

```
1 0
0
./compile.py journey
lgp:0
opts:
sec:40
```

The first two lines indicate the number of threads and bytecode files, followed by the names of bytecode files (and the number of instructions in each one). The fourth and fifth line are legacy, and the sixth indicates the compilation command line. The remaining lines indicate further options used during compilation.

8.1 Execution

We will now walk through what happens when executing the program above with Rep3 modulo 2^{64}. The main function in `Machines/replicated-ring-party.cpp` indirectly calls `Machine<sint, sgf2n>::run()` in `Processor/Machine.hpp` with `sint` being `Rep3Share2<64>`. Then, the following happens:

1. `Programs/Schedules/journey.sch` is parsed in `load_schedule()`.
2. `Programs/Bytecode/journey-0.bc` is parsed in `Machine<sint, sgf2n>::load_program()` where `Program::parse()`. This creates an internal representation of the code in `Program::p` where an `Instruction` object describes every instruction.
3. `Machine<sint, sgf2n>::prepare()` creates a computation thread using `pthread_create()`, which runs `thread_info<sint, sgf2n>::Main_Func()` in `Processor/Online-Thread.hpp`.
4. `Machine<sint, sgf2n>::run()` calls `Machine<sint, sgf2n>::run_tape()`, which signals the thread which code to run.
5. The computation thread waits for a signal in `thread_info<sint, sgf2n>::Sub_Main_Func()`. Once received, it calls `Program::execute()` in `Processor/Instruction.hpp`.
6. `Program::execute()` runs the main loop over the instructions. There is a switch statement acting on the instruction codes.
7. `LDSI` is defined in `ARITHMETIC_INSTRUCTIONS` in `Processor/instructions.h`. It calls `sint::constant()`, which is defined in `Protocols/Rep3Share.h` for `Rep3Share2<64>`. This is in turn calls `Replicated::assign()` in `Protocols/Replicated.h`, which creates a constant replicated secret sharing of 123, that is (123, 0) for party 0, (0, 123) for party 1, and (0, 0) for party 2.
8. `OPEN` is defined in another switch statement in `Instruction::execute()` in `Processor/Instruction.hpp`, where `SubProcessor::POpen()` in `Processor/Processor.hpp` is called. This is turn uses the four-step interface of *MAC_Check_Base* with an instance of `ReplicatedMC`. The communication happens in `ReplicatedMC::exchange()`, and the reconstruction (summation) happens `ReplicatedMC::finalize()`, both in `Protocols/ReplicatedMC.hpp`. The remaining functions mainly handle copying data and serialization.
9. `PRINTREGPLAIN` is also defined in the second switch statement, where `Instruction::print()` in `Processor/Instruction.hpp` is called. This function uses `SwitchableOutput`, which is used to output to console, to file, or not at all depending on the settings.
10. `PRINTCHR` is defined in `REGINT_INSTRUCTIONS` in `Processor/instructions.h`, which means that it's called via a switch statement in `Instruction::execute_regint()` in `Processor/Instruction.cpp`. It also uses `SwitchableOutput`.
11. The remaining instructions are executed similarly but not do have a relevant effect.

12. When `Program::execute()` is done, control returns to `thread_info<sint, sgf2n>::Sub_Main_Func()`, which signals completion to the main thread.
13. After receiving the signal, `Machine<sint, sgf2n>::run()` completes and outputs the various statistics and exits

CHAPTER

NINE

COMPILER OPTIMIZATIONS

A core tenet of MP-SPDZ is to reduce the number of communication rounds by parallelizing operations. The are two mechanisms for this, called merging and CISC. Merging is the more basic mechanism, doing just round reduction and nothing else. CISC in addition uses vectorization, which reduces the cost during compilation and the footprint. In the following, we will explain the two mechanisms using examples.

9.1 Merging

Merging is the original mechanism forming the basis for Keller et al. It is based on extending the common three-operand instruction design to any number of argument tuples. Consider the following high-level code:

```
sint(1) * sint(2)
sint(3) * sint(4)
```

Putting this into `Programs/Source/2mul.py` and running `./compile.py 2mul -a debug` results in the following content in `debug-2mul-0`:

```
# 2mul-0--0
ldsi s2, 1 # 0
ldsi s3, 2 # 1
ldsi s4, 3 # 2
ldsi s5, 4 # 3
muls 8, 1, s0, s2, s3, 1, s1, s4, s5 # 4
```

You can see all the inputs loaded in the first few lines appear in the last one. The arguments are follows (see also *`Compiler.instructions.muls`*): 8 indicates the number of arguments to follow, and every four arguments correspond to one multiplication. The first number withing every four arguments is the vector size, following by the result register and the input registers.

On the backend side, the multiplication is implemented in the `muls()` member function in `../Processor/Processor.hpp`. Notice that `protocol.exchange()` (where the communication happens) is only called once. See *Low-Level Interface* for more information.

Within the compiler, the optimization is executed in the `Merger` class in `../Compiler/allocator.py`. The `dependency_graph()` member function builds a dependency graph for all instructions, and instructions are merged in `longest_path_merge()`.

9.2 CISC

This mechanism takes its name from Complex Instruction Set Computer, which refers to the fact that more complex operations are treated as instructions internally before being merged and spelled in vectorized instructions. For a simple example, consider the following high-level code:

```
program.use_trunc_pr = True
sfix(1) * sfix(2)
sfix(3) * sfix(4)
```

The resulting file `debug-2fmul-0` starts similarly to the one above:

```
# 2fmul-0-cisc-1
ldsi s8, 65536 # 0
ldsi s9, 131072 # 1
ldsi s10, 196608 # 2
ldsi s11, 262144 # 3
muls 8, 1, s0, s8, s9, 1, s7, s10, s11 # 4
```

This corresponds to the fact that fixed-point multiplication starts with an integer multiplication before truncation. The file continues as follows:

```
# 2fmul-0--2
concats 5, s3(2), 1, s0, 1, s7 # 5
vmovs 2, s3(2), s3(2) # 6
# 2fmul-0-update-3
jmp 11 # 7
```

The first instruction creates a vector of size from the multiplication results, and the last instruction jumps over the next code block just to end up here:

```
# 2fmul-0-end-TruncPr(2)_47_16-5
ldint ci0, 3 # 19
stmint ci0, 8192 # 20
jmp -14 # 21
```

This code prepares for a function call. Functions are used for code reusability, i.e., the same code only has to be compiled once per tape and vector size. The last instruction jumps to the start of the function here (the code block jumped over above):

```
# 2fmul-0-begin-TruncPr(2)_47_16-4
ldarg ci1 # 8
vldi 2, c0(2), 32768 # 9
vmulci 2, c2(2), c0(2), 2147483647 # 10
vaddci 2, c0(2), c2(2), 32768 # 11
vaddm 2, s5(2), s3(2), c0(2) # 12
vtrunc_pr 2, 4, s3(2), s5(2), 47, 16 # 13
vsubsi 2, s5(2), s3(2), 1073741824 # 14
vmovs 2, s3(2), s5(2) # 15
vmovs 2, s1(2), s3(2) # 16
ldmint ci1, 8192 # 17
jmpi ci1 # 18
```

The actual truncation happens in the vectorized instructions starting with v. The only communication-relevant instruction is (v)trunc_pr, where truncation by 16 bits is done via a protocol defined in the virtual machine. For example, the implementation for Rep3 is found in `../Protocols/Replicated.hpp`. The other vectorized instructions are required to turn negative values into positive ones, which is a precondition for the protocol. See Protocol in 3.1 in Catrina and Saxena for an explanation. Lastly, the last two instructions load where to jump back to, which is here:

```
# 2fmul-0-call-TruncPr(2)_47_16-6
picks s0, s1(2), 0, 1 # 22
picks s0, s1(2), 1, 1 # 23
```

The two instructions extract the results from the vector, which makes them available individually for further computation.

While this example saves relatively little by using the CISC functionality, this isn't the case for other usages. For example, removing `program.use_trunc_pr` results in several hundred instructions, and more involved mathematical functions corresponds to thousands of instructions.

Internally, the CISC functionality is in implemented in functions decorators in `../Compiler/instructions_base.py`. Our example uses `ret_cisc()` on `TruncPr()` in `../Compiler/floatingpoint.py`. This is because `TruncPr()` returns an `sint`. Other decorators are `cisc()` (the result is stored in the first argument, which must be an `sint`) and `sfix_cisc()` (the result and all arguments are instances of `sfix`).

CHAPTER

TEN

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.

10.1 Schedule File

The schedule file is as follows in ASCII text:

```
<maximum number of threads>
<number of bytecode files>
<bytecode name>:<no of instructions>[ <bytecode name>:<no of instructions>...]
1 0
0
<compilation command line>
<domain requirements>
opts: <potential optimizations>
sec:<security parameter>
```

Domain requirements and potential optimizations are related to *Non-linear Computation*. Domain requirements is one of the following:

lgp:<length>
: minimum prime length

p:<prime>
: exact prime

R:<length>
: exact power of two

Potential optimizations is a any combination of `trunc_pr` (probabilistic truncation), `edabit` (edaBits), and `split` (share splitting). Presence indicates that they would change the compiled bytecode if used. This is used to indicate available optimizations when running a virtual machine.

For example, `./compile.py tutorial` generates the following schedule file:

```
1
1
tutorial-0:19444
1 0
```

(continues on next page)

(continued from previous page)

```
0
./compile.py tutorial
lgp:106
opts: edabit trunc_pr split
sec:40
```

This says that program has only one thread running one bytecode file, which is stored in `tutorial-0.bc` and has 19444 instructions. It requires a prime of length 106, and all protocol optimizations could potentially be used. The length 106 is composed as follows: assuming 64-bit integers, the difference used for comparison is a 65-bit integer, to which 40 bits are added for statistical masking, resulting in a 105 bits, and it takes a 106-bit prime to able to contain all 105-bit numbers. Finally, the last line indicates which compile-time options would change the program. This supports the virtual machine in suggesting options that are compatible with the protocol implementation.

10.2 Bytecode

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)
```

10.2.1 Memory size indication

By default, the compiler adds memory read instructions such as `ldms` at the end of the main bytecode file to indicate the memory size. This is to make sure that even when using memory with run-time addresses, the virtual machine is aware of the memory sizes.

10.3 Instructions

The following table list all instructions except the ones for $\mathrm{GF}(2^n)$ computation, untested ones, and those considered obsolete.

Name	Code	
CISC	0x0	Meta instruction for emulation
LDI	0x1	Assign (constant) immediate value to clear register (vector)
LDSI	0x2	Assign (constant) immediate value to secret register (vector)
LDMC	0x3	Assign clear memory value(s) to clear register (vector) by immediate address
LDMS	0x4	Assign secret memory value(s) to secret register (vector) by immediate address
STMC	0x5	Assign clear register (vector) to clear memory value(s) by immediate address
STMS	0x6	Assign secret register (vector) to secret memory value(s) by immediate address
LDMCI	0x7	Assign clear memory value(s) to clear register (vector) by register address
LDMSI	0x8	Assign secret memory value(s) to secret register (vector) by register address
STMCI	0x9	Assign clear register (vector) to clear memory value(s) by register address
STMSI	0xa	Assign secret register (vector) to secret memory value(s) by register address
MOVC	0xb	Copy clear register (vector)
MOVS	0xc	Copy secret register (vector)
LDTN	0x10	Store the number of the current thread in clear integer register
LDARG	0x11	Store the argument passed to the current thread in clear integer register
REQBL	0x12	Requirement on computation modulus
STARG	0x13	Copy clear integer register to the thread argument
TIME	0x14	Output time since start of computation
START	0x15	Start timer
STOP	0x16	Stop timer
USE	0x17	Offline data usage
USE_INP	0x18	Input usage
RUN_TAPE	0x19	Start tape/bytecode file in another thread
JOIN_TAPE	0x1a	Join thread
CRASH	0x1b	Crash runtime if the value in the register is not zero
USE_PREP	0x1c	Custom preprocessed data usage
USE_MATMUL	0x1f	Matrix multiplication usage

continues on next page

Table 1 – continued from previous page

Name	Code	
ADDC	0x20	Clear addition
ADDS	0x21	Secret addition
ADDM	0x22	Mixed addition
ADDCI	0x23	Addition of clear register (vector) and (constant) immediate value
ADDSI	0x24	Addition of secret register (vector) and (constant) immediate value
SUBC	0x25	Clear subtraction
SUBS	0x26	Secret subtraction
SUBML	0x27	Subtract clear from secret value
SUBMR	0x28	Subtract secret from clear value
SUBCI	0x29	Subtraction of (constant) immediate value from clear register (vector)
SUBSI	0x2a	Subtraction of (constant) immediate value from secret register (vector)
SUBCFI	0x2b	Subtraction of clear register (vector) from (constant) immediate value
SUBSFI	0x2c	Subtraction of secret register (vector) from (constant) immediate value
PREFIXSUMS	0x2d	Prefix sum
PICKS	0x2e	Extract part of vector
CONCATS	0x2f	Concatenate vectors
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
ZIPS	0x3f	Zip vectors
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)
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

continues on next page

Table 1 – continued from previous page

Name	Code	
RANDOMFULLS	0x5d	Store share(s) of a fresh secret random element in secret register (vectors)
UNSPLIT	0x5e	Bit injection (conversion from binary to arithmetic)
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
INITCLIENTCONNECTION	0x6f	Initialize connection
ANDC	0x70	Logical AND of clear (vector) registers
XORC	0x71	Logical XOR of clear (vector) registers
ORC	0x72	Logical OR of clear (vector) registers
ANDCI	0x73	Logical AND of clear register (vector) and (constant) immediate value
XORCI	0x74	Logical XOR of clear register (vector) and (constant) immediate value
ORCI	0x75	Logical OR of clear register (vector) and (constant) immediate value
NOTC	0x76	Clear logical NOT of a constant number of bits of clear (vector) register
SHLC	0x80	Bitwise left shift of clear register (vector)
SHRC	0x81	Bitwise right shift of clear register (vector)
SHLCI	0x82	Bitwise left shift of clear register (vector) by (constant) immediate value
SHRCI	0x83	Bitwise right shift of clear register (vector) by (constant) immediate value
SHRSI	0x84	Bitwise right shift of secret register (vector) by (constant) immediate value
JMP	0x90	Unconditional relative jump in the bytecode (compile-time parameter)
JMPNZ	0x91	Conditional relative jump in the bytecode
JMPEQZ	0x92	Conditional relative jump in the bytecode
EQZC	0x93	Clear integer zero test
LTZC	0x94	Clear integer less than zero test
LTC	0x95	Clear integer less-than comparison
GTC	0x96	Clear integer greater-than comparison
EQC	0x97	Clear integer equality test
JMPI	0x98	Unconditional relative jump in the bytecode (run-time parameter)
BITDECINT	0x99	Clear integer bit decomposition
LDINT	0x9a	Store (constant) immediate value in clear integer register (vector)
ADDINT	0x9b	Clear integer register (vector) addition

continues on next page

Table 1 – continued from previous page

Name	Code	
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
MULBITS	0xa4	(Element-wise) multiplication of secret registers (vectors)
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 output to 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)
WRITEFILECLEAR	0xc3	Write to `Persistence/Transactions-clear-P<playerno>.data` (appending at the end)
READFILECLEAR	0xc4	Read from `Persistence/Transactions-data-P<playerno>.data`
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
GROUPSUMINT	0xd3	Group-sum

continues on next page

Table 1 – continued from previous page

Name	Code	
QS_PERM	0xd4	Quicksort permutation as in Keller :param: permutation (regint) :param: output groups (regint) :param: input groups (regint) :param: comparison results (regint)
PRINTFLOATPREC	0xe0	Set number of digits after decimal point for *print_float_plain*
CONDPRINTPLAIN	0xe1	Conditionally output clear register (with precision)
NPLAYERS	0xe2	Store number of players in clear integer register
THRESHOLD	0xe3	Store maximal number of corrupt players in clear integer register
PLAYERID	0xe4	Store current player number in clear integer register
USE_EDABIT	0xe5	edaBit usage
INTOUTPUT	0xe6	Binary integer output
FLOATOUTPUT	0xe7	Binary floating-point output
FIXINPUT	0xe8	Binary fixed-point input
ACTIVE	0xe9	Indicate whether program is compatible with malicious-security protocols
PRINTREGPLAINS	0xea	Output secret register
CMDLINEARG	0xeb	Load command-line argument
CALL_TAPE	0xec	Start tape/bytecode file in same thread
CALL_ARG	0xed	Pseudo instruction for arguments in connection with `call_tape`
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	Clear 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

continues on next page

Table 1 – continued from previous page

Name	Code	
LDMCB	0x217	Copy clear bit memory cell with compile-time address to clear bit register
STMCB	0x218	Copy clear bit register to clear bit memory cell with compile-time address
XORCB	0x219	Bitwise XOR of two single clear bit registers
ADDCB	0x21a	Integer addition two single clear bit registers
ADDCBI	0x21b	Integer addition single clear bit register and immediate
MULCBI	0x21c	Integer multiplication single clear bit register and immediate
SHRCBI	0x21d	Right shift of clear bit register by immediate
SHLCBI	0x21e	Left shift of clear bit register by immediate
CONVCINTVEC	0x21f	Copy clear register vector by bit to clear bit register vectors
PRINTREGSIGNED	0x220	Signed output of clear bit register
PRINTREGB	0x221	Debug output of clear bit register
PRINTREGPLAINB	0x222	Output clear bit register
PRINTFLOATPLAINB	0x223	Output floating-number from clear bit registers
CONDPRINTSTRB	0x224	Conditionally output four bytes
PRINTREGPLAINSB	0x225	Output secret bit register
CONVCBIT	0x230	Copy clear bit vector to clear integer vector
CONVCBITVEC	0x231	Copy clear bit register vector to clear register by bit
GENSECSHUFFLEB	0x232	Generate secure random binary shuffle
DELSHUFFLEB	0x233	Delete secure random binary shuffle
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
ANDRSVEC	0x24a	Constant-vector AND of secret bit registers (vectorized version)
APPLYSHUFFLEB	0x24b	Apply secure random binary shuffle
CONCATSB	0x24c	Secret bit concatentation
TRANSVEC	0x24e	Secret bit register vector transpose
PICKSB	0x24f	Secret bit part selection
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
ZIPSB	0x280	Zip vectors

10.4 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.**active**(**args*, ***kwargs*)

Indicate whether program is compatible with malicious-security protocols.

Param
0 for no, 1 for yes

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
vector size (int)

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.**call_arg**(**args*, ***kwargs*)

Pseudo instruction for arguments in connection with *call_tape*.

Param
destination (register)

Param
register type (see `vm_types`)

class Compiler.instructions.**call_tape**(**args*, ***kwargs*)

Start tape/bytecode file in same thread. Arguments/return values starting from `direction` are optional.

Param
tape number (int)

Param
arg (regint)

Param
direction (0 for argument, 1 for return value)

Param
register type (see `vm_types`)

Param
register size (int)

Param
destination register

Param
source register

Param
(repeat from direction)

`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.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.cmdlinearg(*args, **kwargs)`
Load command-line argument.

Param
dest (regint)

Param
index (regint)

`class Compiler.instructions.concats(*args)`
Concatenate vectors.

Param
result (sint)

Param
start offset (int)

Param
input (sint)

Param
(repeat from offset)...

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
number of arguments to follow (int)

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)

Param
repeat from result...

`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
synchronize (int)

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.**file_base**(**args*, ***kwargs*)

class Compiler.instructions.**fixinput**(**args*, ***kwargs*)

Binary fixed-point input.

Param
player (int)

Param
destination (cint)

Param
exponent (int, for float/double) / byte length (1/8, for integer)

Param
input type (0: 64-bit integer, 1: float, 2: double)

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.**groupsumint**(**args*, ***kwargs*)

Group-sum. Every output element is the sum of all in input elements in the group.

Param
destination (regint)

Param
group markers (regint; 1 for the start of the group, 0 otherwise)

Param
input elements (regint)

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.**initclientconnection**(**args*, ***kwargs*)

Initialize connection.

Param
client id destination (regint)

Param
port number (regint)

Param
my client id (regint)

Param
hostname (variable string)

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**(**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 *ldmc*

class Compiler.instructions.**ldmint**(**args*, ***kwargs*)

Assign clear integer memory value(s) to clear integer register (vector) by immediate address. The vectorized version starts at the base address and then iterates the memory address.

Param
destination (regint)

Param
memory address base (int)

class Compiler.instructions.**ldminti**(**args*, ***kwargs*)

Assign clear integer memory value(s) to clear integer register (vector) by register address. The vectorized version starts at the base address and then iterates the memory address.

Param
destination (regint)

Param
memory address base (regint)

direct

alias of *ldmint*

class Compiler.instructions.**ldms**(**args*, ***kwargs*)

Assign secret memory value(s) to secret register (vector) by immediate address. The vectorized version starts at the base address and then iterates the memory address.

Param
destination (sint)

Param
memory address base (int)

class Compiler.instructions.**ldmsi**(**args*, ***kwargs*)

Assign secret memory value(s) to secret register (vector) by register address. The vectorized version starts at the base address and then iterates the memory address.

Param
destination (sint)

Param
memory address base (regint)

direct

alias of *ldms*

class Compiler.instructions.**ldsi**(**args*, ***kwargs*)

Assign (constant) immediate value to secret register (vector).

Param
destination (sint)

Param
value (int)

class Compiler.instructions.**ldtn**(**args*, ***kwargs*)

Store the number of the current thread in clear integer register.

Param
destination (regint)

class Compiler.instructions.**legendrec**(**args*, ***kwargs*)

Clear Legendre symbol computation (a/p) over prime p (the computation modulus).

Param
result (cint)

Param
a (int)

class Compiler.instructions.**listen**(**args*, ***kwargs*)

Open a server socket on a party-specific port number and listen for client connections (non-blocking).

Param
port number (regint)

class Compiler.instructions.**ltc**(**args*, ***kwargs*)

Clear integer less-than comparison. The result is 1 if the first operand is less and 0 otherwise.

Param
destination (regint)

Param
first operand (regint)

Param
second operand (regint)

op(*b*, */*)

Same as a < b.

class Compiler.instructions.**ltzc**(**args*, ***kwargs*)

Clear integer less than zero test. The result is 1 for true and 0 for false.

Param
destination (regint)

Param
operand (regint)

class Compiler.instructions.**matmuls**(**args*, ***kwargs*)

Secret matrix multiplication from registers. All matrices are represented as vectors in row-first order.

Param
result (sint vector)

Param
first factor (sint vector)

Param
second factor (sint vector)

Param
number of rows in first factor and result (int)

Param
number of columns in first factor and rows in second factor (int)

Param
number of columns in second factor and result (int)

class Compiler.instructions.**matmulsm**(**args*, *first_factor_base_addresses=None*, *second_factor_base_addresses=None*, *indices_values=None*, ***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 as number of columns in the first factor)

Param
rows of second factor to use (regint vector, length as number of columns in the first factor)

Param
columns of second factor to use (regint vector, length as number of columns in the second factor)

Param
total number of columns in the first factor, equal to used number of columns when all columns are used (int)

Param
total number of columns in the second factor, equal to used number of columns when all columns are used (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.**mul_base**(**args*, ***kwargs*)

class Compiler.instructions.**mulbits**(**args*, ***kwargs*)

(Element-wise) multiplication of secret registers (vectors).

Param
number of arguments to follow (multiple of four)

Param
vector size (int)

Param
result (sint)

Param
factor (sint)

Param
bit factor (sint)

Param
(repeat the last four)...

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 four)

Param
vector size (int)

Param
result (sint)

Param
factor (sint)

Param
factor (sint)

Param
(repeat the last four)…

class Compiler.instructions.**muls_base**(**args*, ***kwargs*)

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.**picks**(**args*)

Extract part of vector.

Param
result (sint)

Param
input (sint)

Param
start offset (int)

Param
step

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. Considered obsolete.

Param
destination (regint)

class Compiler.instructions.**prefixsums**(**args*, ***kwargs*)

Prefix sum.

Param
result (sint)

Param
input (sint)

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.**print_reg_plains**(**args*, ***kwargs*)

Output secret register.

Param
source (sint)

class Compiler.instructions.**privateoutput**(**args*)

Private output to 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. Considered obsolete.

Param
source (regint)

class Compiler.instructions.**qs_perm**(**args*, ***kwargs*)

Quicksort permutation as in Keller

Param
permutation (regint)

Param
output groups (regint)

Param
input groups (regint)

Param
comparison results (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.**readfileclear**(**args*, ***kwargs*)

Read from Persistence/Transactions-data-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.**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 *stmc*

class Compiler.instructions.**stmint**(**args*, ***kwargs*)

Assign clear integer register (vector) to clear integer memory value(s) by immediate address. The vectorized version starts at the base address and then iterates the memory address.

Param
source (regint)

Param
memory address base (int)

class Compiler.instructions.**stminti**(**args*, ***kwargs*)

Assign clear integer register (vector) to clear integer memory value(s) by register address. The vectorized version starts at the base address and then iterates the memory address.

Param
source (regint)

Param
memory address base (regint)

direct

alias of *stmint*

class Compiler.instructions.**stms**(**args*, ***kwargs*)

Assign secret register (vector) to secret memory value(s) by immediate address. The vectorized version starts at the base address and then iterates the memory address.

Param
source (sint)

Param
memory address base (int)

class Compiler.instructions.**stmsi**(**args*, ***kwargs*)

Assign secret register (vector) to secret memory value(s) by register address. The vectorized version starts at the base address and then iterates the memory address.

Param
source (sint)

Param
memory address base (regint)

direct

alias of *stms*

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

Stop timer.

Param
timer number (int)

class Compiler.instructions.**subc**(**args*, ***kwargs*)

Clear subtraction.

Param
result (cint)

Param
first operand (cint)

Param
second operand (cint)

class Compiler.instructions.**subcfi**(**args*, ***kwargs*)

Subtraction of clear register (vector) from (constant) immediate value.

Param
result (cint)

Param
first operand (int)

Param
second operand (cint)

class Compiler.instructions.**subci**(**args*, ***kwargs*)

Subtraction of (constant) immediate value from clear register (vector).

Param
result (cint)

Param
first operand (cint)

Param
second operand (int)

class Compiler.instructions.**subint**(**args*, ***kwargs*)

Clear integer register (vector) subtraction.

Param
result (regint)

Param
first operand (regint)

Param
second operand (regint)

op(*b*, */*)

Same as a - b.

class Compiler.instructions.**subml**(**args*, ***kwargs*)

Subtract clear from secret value.

Param
result (sint)

Param
first operand (sint)

Param
second operand (cint)

class Compiler.instructions.**submr**(**args*, ***kwargs*)

Subtract secret from clear value.

Param
result (sint)

Param
first operand (cint)

Param
second operand (sint)

class Compiler.instructions.**subs**(**args*, ***kwargs*)

Secret subtraction.

Param
result (sint)

Param
first operand (sint)

Param
second operand (sint)

class Compiler.instructions.**subsfi**(**args*, ***kwargs*)

Subtraction of secret register (vector) from (constant) immediate value.

Param
result (sint)

Param
first operand (int)

Param
second operand (sint)

class Compiler.instructions.**subsi**(**args*, ***kwargs*)

Subtraction of (constant) immediate value from secret register (vector).

Param
result (sint)

Param
first operand (sint)

Param
second operand (int)

class Compiler.instructions.**threshold**(**args*, ***kwargs*)

Store maximal number of corrupt players in clear integer register.

Param
destination (regint)

class Compiler.instructions.**time**(**args*, ***kwargs*)

Output time since start of computation.

class Compiler.instructions.**triple**(**args*, ***kwargs*)

Store fresh random triple(s) in secret register (vectors).

Param
factor (sint)

Param
factor (sint)

Param
product (sint)

class Compiler.instructions.**trunc_pr**(**args*, ***kwargs*)

Probabilistic truncation if supported by the protocol.

Param
number of arguments to follow (multiple of four)

Param
destination (sint)

Param
source (sint)

Param
bit length of source (int)

Param
number of bits to truncate (int)

class Compiler.instructions.**unsplit**(**args*, ***kwargs*)

Bit injection (conversion from binary to arithmetic).

Param
destination (sint)

Param
source (sbits)

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.**writefileclear**(**args*, ***kwargs*)

Write to `Persistence/Transactions-clear-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.**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)

class Compiler.instructions.**zips**(**args*)

Zip vectors.

Param
result (sint)

Param
operand (sint)

Param
operand (sint)

10.5 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.**BinaryCiscable**(**args*, ***kwargs*)

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.**andrsvec**(**args*, ***kwargs*)

Constant-vector AND of secret bit registers (vectorized version).

Param
total number of arguments to follow (int)

Param
number of arguments to follow for one operation / operation vector size plus three (int)

Param
vector size (int)

Param
result vector (sbit)

Param
(repeat)…

Param
constant operand (sbits)

Param
vector operand

Param
(repeat)…

Param
(repeat from number of arguments to follow for one operation)…

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.**applyshuffleb**(**args*, ***kwargs*)

Apply secure random binary shuffle.

Param
vector size (int)

Param
destination (sbit)

Param
source (sbit)

Param
number of bits to be treated as one (int)

Param
handle (regint)

Param
reverse (0/1)

`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)`

Clear 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.**concatsb**(**args*, ***kwargs*)

Secret bit concatentation.

Param
destination (sbit)

Param
number of source bits (int)

Param
source (sbit)

Param
(repeat from number of source bits)

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 vector to clear integer vector.

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*, ***kwargs*)

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
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.**delshuffleb**(**args*, ***kwargs*)

Delete secure random binary shuffle.

Param
handle (regint)

class Compiler.GC.instructions.**gensecshuffleb**(**args*, ***kwargs*)

Generate secure random binary shuffle.

Param
handle (regint)

Param
size (int)

Param
generate material immediately (int)

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 *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 *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.**picksb**(**args*, ***kwargs*)

Secret bit part selection.

Param
destination (sbit)

Param
source (sbit)

Param
base (int)

Param
skip (int)

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_plainsb**(**args*, ***kwargs*)

Output secret bit register.

Param
source (sbit)

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 (sbit)

Param
second share of least significant bit (sbit)

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 *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 *stmsb*

class Compiler.GC.instructions.**trans**(**args*, ***kwargs*)

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.**transvec**(**args*, ***kwargs*)

Secret bit register vector transpose.

Param
number of outputs (int)

Param
destination (sbit)

Param
number of inputs (int)

Param
source (sbit)

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)...

`class Compiler.GC.instructions.zipsb(*args)`

Zip vectors.

Param
result (sbit)

Param
operand (sbit)

Param
operand (sbit)

CHAPTER

ELEVEN

LOW-LEVEL INTERFACE

In the following we will explain the basics 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]);
```

(continues on next page)

(continued from previous page)

```
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.

11.1 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.

11.2 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.

11.3 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.
MaliciousShamirSh	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.
PostSacriRepField	Post-sacrifice protocol using three-party replicated secret sharing with T representing a field.
PostSacriRepRingS 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.

11.4 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

Public Static Functions

static inline void **set_batch_size**(size_t batch_size)

Set how much preprocessing is produced at once.

template<class **T**>

class **ProtocolSet**

Input, multiplication, and output protocol instance for an arithmetic share type

Public Functions

inline **ProtocolSet**(*Player* &P, const *ProtocolSetup*<*T*> &setup)

Parameters

- **P** – communication instance
- **setup** – one-time setup instance

template<class **sgf2n**>
inline **ProtocolSet**(*Player* &P, const Machine<*T*, *sgf2n*> &machine)

Parameters

- **P** – communication instance
- **machine** – virtual machine 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

template<class **sint**, class **sgf2n**>
inline **BinaryProtocolSet**(*Player* &P, const Machine<*sint*, *sgf2n*> &machine)

Parameters

- **P** – communication instance
- **machine** – virtual machine 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

template<class **sgf2n**>
inline **MixedProtocolSet**(*Player* &P, const Machine<*T*, *sgf2n*> &machine)

Parameters

- **P** – communication instance
- **machine** – virtual machine instance

inline void **check**()
Run all protocol checks

11.5 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, int n_bits = -1)

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)

inline virtual edabitvec<*T*> **get_edabitvec**(bool, int)

Get fresh edaBit chunk.

Public Static Functions

static inline void **basic_setup**(*Player*&)

Key-independent setup if necessary (cryptosystem parameters)

static inline void **setup**(*Player*&, typename *T*::mac_key_type)

Generate keys if necessary.

static inline void **teardown**()

Free memory of global cryptosystem parameters.

template<class **T**>

class **BufferPrep** : public *Preprocessing*<*T*>

Abstract base class for live preprocessing

Subclassed by BitPrep< T >

Public Functions

virtual edabitvec<*T*> **get_edabitvec**(bool strict, int n_bits)

Get fresh edaBit chunk.

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.

11.6 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. Convert to `bigint` to access the canonical integer representation.

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 **output**(ostream &s, bool human, bool signed_ = false) const

Output.

Parameters

- **s** – output stream
- **human** – human-readable or binary
- **signed_** – signed representation (range `[-p/2,p/2]` instead of `[0,p]`)

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 const *bigint* &**pr**(bool allow_zero = false)

Get the prime modulus

Friends

inline friend ostream &**operator<<**(ostream &s, const *gfp_* &x)

Human-readable output in the range `[0, p]`.

Parameters

- **s** – output stream
- **x** – value

inline 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_** : public ValueInterface

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 >*

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

The following is not used as a domain, but it helps using the above types, in particular `gfp_<X, L>` and `gfpvar_<X, L>`.

class **bigint** : public mpz_class

Type for arbitrarily large integers. This is a sub-class of `mpz_class` from GMP. As such, it implements all integers operations and input/output via C++ streams. In addition, the `get_ui()` member function allows retrieving the least significant 64 bits.

Public Functions

inline **bigint**()

Initialize to zero.

template<int **X**, int **L**>
bigint(const *gfp_*<*X*, *L*> &x)

Convert to canonical representation as non-negative number.

template<int **X**, int **L**>
bigint(const *gfpvar_*<*X*, *L*> &x)

Convert to canonical representation as non-negative number.

template<int **K**>
bigint(const *Z2*<*K*> &x)

Convert to canonical representation as non-negative number.

template<int **K**>
bigint(const *SignedZ2*<*K*> &x)

Convert to canonical representation as non-negative number.

template<int **X**, int **L**>
bigint &**from_signed**(const *gfp_*<*X*, *L*> &other)

Convert to signed representation in :math:`[-p/2,p/2]`.

CHAPTER

TWELVE

USING HIGH-LEVEL FUNCTIONALITY IN C++

The fact that most functionality is implemented on the high level (in the compiler) but the virtual machine running high-level code is somewhat limited makes it desirable to call high-level functionality from C++. MP-SPDZ supports defining functions on the high level and calling them from C++. Functions can have integer secrets (*sint*) and (multi-)arrays thereof as inputs and return values as well as (multi-)arrays of binary secrets (types created using *sbitvec* and *sbitintvec*).

As a simple example, consider `../Programs/Source/export-sort.py` and `../Utils/export-sort.cpp`. The Python part looks as follows:

```
@export
def sort(x):
  print_ln('x=%s', x.reveal())
  res = x.sort()
  print_ln('res=%s', x.reveal())

sort(sint.Array(1000))
```

This makes the sorting of integer arrays of length 1000 accessible to C++. The corresponding C++ code starts similarly to the *low-level code example*:

```
#include "Machines/maximal.hpp"

int main(int argc, const char** argv)
{
  assert(argc > 1);
  int my_number = atoi(argv[1]);
  int port_base = 9999;
  Names N(my_number, 3, "localhost", port_base);
```

This includes all necessary headers and makes contact with the other parties on the same machine. The next step is to set up an instance of the virtual machine:

```
typedef Rep3Share2<64> share_type;
Machine<share_type> machine(N);
```

In this example, we use replicated secret sharing modulo 2^{64}. Next, we prepare the inputs:

```
int n = 1000;

ProtocolSet<share_type> set(machine.get_player(), machine);
set.input.reset(0);
for (int i = 0; i < n; i++)
```

(continues on next page)

(continued from previous page)

```
{
    if (my_number == 0)
        set.input.add_mine(n - i);
    else
        set.input.add_other(0);
}
set.input.exchange();

vector<share_type> inputs;
for (int i = 0; i < n; i++)
    inputs.push_back(set.input.finalize(0));
```

This initializes a *ProtocolSet* using the virtual machine instead of a *ProtocolSetup*. This is necessary to avoid differing MAC keys and other setup variables. Then, party 0 inputs the numbers 1 to 1000 in reverse other, and the resulting secret shares are stored in `inputs`. Now we're ready to call the function:

```
vector<FunctionArgument> args = {{inputs, true}};
FunctionArgument res;

machine.run_function("sort", res, args);
```

This indicates that the function takes one argument, which is an array (as opposed to a vector, see below) and that we don't expect a return value. Lastly, we open and check the array:

```
Opener<share_type> MC(machine.get_player(), machine.get_sint_mac_key());
MC.init_open();
for (auto& x : inputs)
    MC.prepare_open(x);
MC.exchange();

if (my_number == 0)
{
    cout << "res: ";
    for (int i = 0; i < 10; i++)
        cout << MC.finalize_open() << " ";
    cout << endl;
}
else
{
    for (int i = 0; i < n; i++)
    {
        auto x = MC.finalize_open();
        if (x != i + 1)
        {
            cerr << "error at " << i << ": " << x << endl;
            exit(1);
        }
    }
}
```

The `Opener` class is convenience that is bound to a communication instance (unlike *MAC_Check_Base* instances, which require the communication instance in several function calls).

You can run the example as follows:

```
./compile.py -E ring export-sort
make export-sort.x
for i in 0 1 2; do ./export-sort.x $i & true; done
```

This makes sure that all the optimizations of the protocol are used.

12.1 Vector arguments and return values

Instead of arrays, it is also possible to use *sint* vectors as demonstrated in `../Programs/Source/export-trunc.py`:

```
@export
def trunc_pr(x):
  print_ln('x=%s', x.reveal())
  res = x.round(32, 2)
  print_ln('res=%s', res.reveal())
  return res

trunc_pr(sint(0, size=1000))
```

The calling C++ code in `../Utils/export-trunc.cpp` looks as follows:

```
int n = 1000;
vector<share_type> inputs;
for (int i = 0; i < n; i++)
    inputs.push_back(share_type::constant(i, my_number));

vector<FunctionArgument> args = {inputs};
vector<share_type> results(n);
FunctionArgument res(results);

machine.run_function("trunc_pr", res, args);
```

This creates integer shares using public constants instead of the input protocol as above. The *FunctionArgument* instance for both input and output are created using the vector of secret shares without the extra `true` argument.

12.2 Binary values

It is possible to input and output binary secrets with an array. Consider `../Programs/Source/export-b2a.py`, which converts arithmetic to binary shares:

```
@export
def b2a(res, x):
  print_ln('x=%s', x.reveal())
  res[:] = sint(x[:])
  print_ln('res=%s', x.reveal())

b2a(sint.Array(size=10), sbitvec.get_type(16).Array(10))
```

This demonstrates the requirement of using an array of an `sbitvec` type with a defined number of bits (16 in this case). `sbitintvec` is a sub-class and also permissible.

The C++ calling code looks as follows:

```
int n = 10;
vector<share_type> outputs(n);
vector<vector<share_type::bit_type>> inputs(n);

auto& inputter = set.binary.input;
inputter.reset(0);
for (int i = 0; i < n; i++)
    if (my_number == 0)
        inputter.add_mine(i + 1, 16);
    else
        inputter.add_other(0);
inputter.exchange();
for (int i = 0; i < n; i++)
    inputs.at(i).push_back(inputter.finalize(0, 16));

vector<FunctionArgument> args = {{outputs, true}, {16, inputs}};
FunctionArgument res;

machine.run_function("b2a", res, args);
```

This inputs the values 1 to 10 as 16-bit numbers. Note the nested vectors for the inputs. This is due to the fact `share_type::bit_type` can only hold up to 64 bits, so for longer bit lengths several entries have to be used.

Lastly, `../Programs/Source/export-a2b.py` covers the other direction:

```
@export
def a2b(x, res):
  print_ln('x=%s', x.reveal())
  res[:] = sbitvec(x, length=16)
  print_ln('res=%s', x.reveal())

a2b(sint(size=10), sbitvec.get_type(16).Array(10))
```

The calling C++ code in `../Utils/export-a2b.cpp` has to initialize the binary shares even when they are only used for output:

```
int n = 10;
vector<share_type> inputs;
for (int i = 0; i < n; i++)
    inputs.push_back(
            share_type::constant(i + 1, my_number,
                    machine.get_sint_mac_key()));

vector<vector<bit_share_type>> outputs(n,
        vector<bit_share_type>(1,
                bit_share_type::constant(0, my_number,
                        machine.get_bit_mac_key())));

vector<FunctionArgument> args = {{inputs}, {16, outputs}};
FunctionArgument res;

machine.run_function("a2b", res, args);
```

12.3 C++ compilation

The easiest way is to include `Machines/maximal.hpp` as in the first example and put the C++ in code `Utils/<name>.cpp` and calling `make <name>.x` in the main directory. If using oblivious transfer or homomorphic encryption, add the following line to `Makefile`:

```
<name>.x: $(FHEOFFLINE) $(OT)
```

Most of the examples work slightly differently, however, in order to distribute the compilation load. Most notably, `../Utils/export-a2b.cpp`, which supports several protocols, only includes `Machines/minimal.hpp` and "outsources" the virtual machine for the various protocols to `Machines/export-*.cpp`, which are all compiled separately.

12.4 Reference

class **FunctionArgument**

Inputs and outputs for functions exported in high-level code.

Public Functions

template<class **T**>
inline **FunctionArgument**(vector<*T*> &values, bool memory = false)

Argument with integer secret shares.

Parameters

- **values** – shares
- **memory** – whether shares are in a (multi-)array (true) or vector (false)

inline **FunctionArgument**()

Void argument.

template<class **T**>
inline **FunctionArgument**(size_t n_bits, vector<vector<*T*>> &values)

Argument with binary secret shares (always in array).

Parameters

- **n_bits** – number of bits
- **values** – shares (vector of vectors of bit_type of length ceil(n_bits/64))

inline **FunctionArgument**(vector<long> &values)

Argument with integer array.

CHAPTER

THIRTEEN

NAVIGATING THE C++ CODE

In this section, we will explain how the most important aspects of the C++ codebase fit together and explain them with brief examples.

MP-SPDZ heavily relies on templates (also called generic programming). This is to achieve high efficiency while retaining modularity as many MPC building blocks work in a number of contexts. A notable example of this is Beaver multiplication. The same code is used in more than 20 contexts without loss of efficiency. See this introduction if you're new to C++ templates.

Due to the size of the codebase (more than 100,000 lines of code in hundreds of files), we recommend using an integrated development environment (IDE) to navigate it. Eclipse has very useful features such as jumping to the definition of function or variable using F3 or find all references to a function or variable with Ctrl-Shift-g. The latter doesn't work well with templates, however, so global text search is necessary for a more comprehensive view.

The virtual machines offer the option `--code-locations` to output the most relevant locations in the code that are used by computation. For example:

```
Scripts/compile-run.py ring tutorial -- --code-locations
```

will output:

```
first call to ./Protocols/Replicated.hpp:<lineno>, virtual void Replicated<Rep3Share2<64>
↪>::exchange() [T = Rep3Share2<64>]
```

This indicates the invocation of the `exchange()` function of the Rep3 multiplication protocol. Note that, for efficiency reasons, this is the only call of the relevant class that will be mentioned. Consult *`ProtocolBase`* to see how protocol classes operate.

13.1 Mathematical Domains

All protocols in MP-SPDZ are based on finite domains for secret sharing, most importantly integers modulo a number. While basic CPU arithmetic and existing libraries provide all required functionality (e.g., modern CPUs generally work with integers modulo 2^{64} and the GNU Multiple Precision Arithmetic Library has provisions for a plethora of integer operations), we have found these to be inefficient in the context of MPC. The main reason is that an MPC protocol uses the same modulus throughout but the variable-length nature of GMP incurs a considerable cost for every operation, which is not necessary. MP-SPDZ therefore provides a tailored data type for every mathematical domain. These data types use operator overloading for easy use. For example,

```
cout << (Z2<5>(20) + Z2<5>(30)) << endl;
```

should output:

```
18
```

because Z2<5> represents computation modulo $2^5 = 32$. See the reference of *Z2* and *SignedZ2* for further details.

For computation modulo a prime on the other hand, the data type fixes only a range at compile time, so the exact modulus has be given before usage:

```
gfp_<0, 1>::init_field(13);
cout << (gfp_<0, 1>(8) + gfp_<0, 1>(7)) << endl;
```

should output:

```
2
```

The first parameter to *gfp_* is a counter that allows several moduli to be used at once, for example:

```
gfp_<0, 1>::init_field(13);
gfp_<1, 1>::init_field(17);
```

The second parameter denotes the number of 64-bit limbs, that is, it should be 1 for primes in $[0, 2^{64}]$, 2 for prime in $[2^{64}, 2^{128}]$ etc.

In addition to the fixed domains, *bigint* is a sub-class of `mpz_class` type in GMP. It used for conversions amongst other things.

13.2 Communication

MP-SPDZ provides a communication interface that is more involved than sending bytes via a socket. There are two reasons for doing this. First, the structure of MPC goes far beyond the query-reply pattern often found in online communication. For example, two parties might need to exchange a large quantity of information simultaneously. Second, the atomic quantity communicated in MPC (i.e., the numbers) are usually so small that it is preferential to send them in batches. The following example demonstrates the exchange of a vector of 64-bit numbers in the two-party setting:

```
Player& P = ...;
vector<Z2<64>> numbers;
// populate vector
...
octetStream os;
os.store(numbers);
P.pass_around(1, os);
os.get(numbers);
// numbers now contains the ones from the other side
```

No matter how many numbers there are, the framework makes sure to send and receive them at the same time. The number given to *Player::pass_around()* denotes an offset, that is, the numbers are sent to "next" party and received from the "previous" one (regarding player number with wrap-around). See *this section* for more details.

13.3 Randomness

Randomness is a crucial component of MPC (as for cryptography in general). Random number generation in MP-SPDZ centers on the *PRNG* class. It implements optimized random number generation based on hardware AES if available. This allows for local as well as coordinated randomness generation. An exampled for the first is as follows:

```
SeededPRNG G;
auto res = G.get<Z2<64>>();
```

This initializes the PRNG with secure randomness from libsodium and then generates a random 64-bit element.

On the other hand, the following initializes a global PRNG securely, that is, with a seed that cannot be influenced by any party, before generating a random element modulo a prime:

```
// initialize at some point
gfp_<0, 1>::init_field(prime);
Player& P = ...;
...
GlobalPRNG G(P);
auto res = G.get<gfp_<0, 1>>();
```

13.4 Protocols

The implementation of protocols is centered on the share types. They not only hold all values necessary to represent a secret value for one party, they also provide local operations, refer to other classes implementing protocols, and contain variables and static functions to describes protocols.

As an example, consider `Rep3Share<T>` in `../Protocols/Rep3Share.h`. It implements a share for three-party replicated secret sharing. It takes one template parameter for the mathematical domain because the secret sharing and the multiplication protocol work for any finite domain. The following typedef makes the cleartext domain generally accessible:

```
typedef T clear;
```

Further typedefs are used to indicate which class to use for inputs, multiplications, and outputs:

```
typedef ReplicatedInput<Rep3Share> Input;
typedef Replicated<Rep3Share> Protocol;
typedef ReplicatedMC<Rep3Share> MAC_Check;
```

The latter usually contains the name MAC_Check or MC because MAC checking is a core function of the output protocol in SPDZ.

These typedefs follow the general pattern that the *share type* is a template argument to the *protocol type*. This makes everything contained defined by the share type accessible to the protocol type. As an example of this, `ReplicatedMC<Rep3Share>` is a sub-class of *`MAC_Check_Base<Rep3Share>`*, which implements the general interface for opening shares. On the functions there is defined as follows:

```
virtual typename T::clear finalize_open();
```

Another important typedef in `Rep3Share` is the preprocessing type:

```
typedef typename conditional<T::characteristic_two,
        ReplicatedPrep<Rep3Share>, SemiRep3Prep<Rep3Share>>::type LivePrep;
```

It is more complicated because it uses meta-programming to assign different types depending on whether mathematical domain has characteristic two (i.e., it's $\mathrm{GF}(2^n)$). This is to avoid compiling code for a specific daBit generation that doesn't make sense in said domain. The preprocessing classes use polymorphism to mix and match the possible protocols. For example, `BitPrep<T>::buffer_squares()` to implements a generic protocol to generate square tuples from multiplication triples, but this isn't the most efficient way with replicated secret sharing, which is why `ReplicatedRingPrep<T>::buffer_squares()` overrides this with a more specific protocol in our example.

The four protocol types above are contained in an instance *`ProtocolSet<T>`* as documented in *Low-Level Interface* where `T` is a share type.

`Rep3Share<T>` is a sub-class of `FixedVec<T,2>`. The latter contains a pair of values in the cleartext domain as one would expect with this kind of secret sharing, and it implements element-wise addition, subtraction, and multiplication via operator overloading, which makes it straight-forward to run local operations with share types.

Lastly, `Rep3Share` defines a few variables that describe the protocols, for example:

```
const static bool dishonest_majority = false;
const static bool variable_players = false;
```

These indicate that replicated secret sharing requires and honest majority and fixed number of players. First is used to the set default number of parties and the second to decide whether to offer the `--nparties` command-line option.

13.5 Virtual Machines

The main function for the protocol-specific virtual machines is defined in the file of the appropriate name in the `Machines` directory. For example, the virtual machine for three-party replicated secret sharing over prime fields is defined in `../Machines/replicated-field-party.cpp`, and the main function looks as follows:

```
int main(int argc, const char** argv)
{
    HonestMajorityFieldMachine<Rep3Share>(argc, argv);
}
```

Indirectly, this calls an instance of `Machine<sint, sgf2n>` where `sint` and `sgf2n` denote the complete share type for integer and $\mathrm{GF}(2^n)$, respectively. The defaults are `Rep3Share<gfp_<0, 2>>` and `Rep3Share<gf2n_long>` in the example. To choose that, the constructor of `FieldMachine` (in `../Processor/FieldMachine.hpp`) contains code to the length for *gfp_* (the second parameter, the first is always 0). For protocols modulo a power of two other than SPDZ2k, this happens in the constructor of `RingMachine` or `HonestMajorityRingMachineWithSecurity` in `../Processor/RingMachine.hpp`. The purpose of all this is to fix the mathematical domains throughout for maximum performance.

The includes are structured in a way that all relevant templated code is included in these files, so compiling it makes sure that the object file contains most protocol-specific code. The main exceptions from this are code related to homomorphic encryption (in `libFHE.so`), oblivious transfer (included via object files), and Tinier (in `Machines/Tinier.o`). Furthermore, all general code is put in `libSPDZ.so`. All this is to reduce the compilation time and/or the binary size.

CHAPTER

FOURTEEN

MACHINE LEARNING QUICKSTART

This document is a short introduction to running privacy-preserving logistic regression in MP-SPDZ. It assumes that you have the framework already installed as explained in the installation instructions. For more information on how to run machine learning algorithms in MP-SPDZ, see the full machine learning section.

The easiest way to use is to put Python code in an `.mpc` in `Programs/Source`, for example `Programs/Source/foo.mpc`. Put the following code there to use the breast cancer dataset:

```
X = sfix.input_tensor_via(0, [[1, 2, 3], # 2 samples
                              [11, 12, 13]])
y = sint.input_tensor_via(0, [0, 1]) # 2 labels

from Compiler import ml
log = ml.SGDLogistic(100)
log.fit(X, y)

print_ln('%s', log.predict(X).reveal())
```

The first two lines make the data available to the secure computation. The next lines create a logistic regression instance and train it (for one hundred epochs). Finally, the last line uses the instances for predictions and outputs the results.

After adding all the above code to `Programs/Source/foo.mpc`, you can run it either insecurely:

```
Scripts/compile-emulate.py foo
```

or securely with three parties on the same machine:

```
Scripts/compile-run.py -E ring foo
```

The first call should give the following output:

```
$ Scripts/compile-emulate.py foo
Default bit length for compilation: 63
Default security parameter for compilation: 40
Compiling file Programs/Source/foo.mpc
Writing binary data to Player-Data/Input-Binary-P0-0
Setting learning rate to 0.01
Using SGD
Initializing dense weights in [-1.224745,1.224745]
Writing to Programs/Bytecode/foo-TruncPr(3)_47_16-2.bc
Writing to Programs/Bytecode/foo-multithread-1.bc
2 runs per epoch
Writing to Programs/Bytecode/foo-TruncPr(1)_47_16-5.bc
```

(continues on next page)

(continued from previous page)

```
Writing to Programs/Bytecode/foo-Dense-forward-4.bc
Writing to Programs/Bytecode/foo-TruncPr(1)_45_14-7.bc
Writing to Programs/Bytecode/foo-exp2_fx(1)_31_16_False-9.bc
Writing to Programs/Bytecode/foo-log2_fx(1)_31_16-11.bc
Writing to Programs/Bytecode/foo-TruncPr(1)_46_15-13.bc
Writing to Programs/Bytecode/foo-Output-forward-6.bc
Writing to Programs/Bytecode/foo-multithread-15.bc
Writing to Programs/Bytecode/foo-multithread-16.bc
Writing to Programs/Bytecode/foo-TruncPr(3)_46_15-18.bc
Writing to Programs/Bytecode/foo-multithread-17.bc
Initializing dense weights in [-1.224745,1.224745]
Writing to Programs/Bytecode/foo-multithread-19.bc
Writing to Programs/Bytecode/foo-TruncPr(2)_47_16-22.bc
Writing to Programs/Bytecode/foo-multithread-21.bc
Writing to Programs/Bytecode/foo-multithread-23.bc
Writing to Programs/Bytecode/foo-Dense-forward-20.bc
Writing to Programs/Bytecode/foo-FPDiv(1)_31_16-24.bc
Writing to Programs/Schedules/foo.sch
Writing to Programs/Bytecode/foo-0.bc
Hash: 8227349c6796977e0035cd9e92558560 3531eb9aa98ac586440c1abd360ae712
Program requires at most:
8 integer inputs from player 0
2402 integer opens
67654 integer bits
204509 integer triples
200 matrix multiplications (1x3 * 3x1)
200 matrix multiplications (3x1 * 1x1)
1 matrix multiplications (2x3 * 3x1)
37109 virtual machine rounds
Compilation finished, running program...
Using statistical security parameter 40
Trying to run 64-bit computation
Using SGD
done with epoch 99
[0, 1]
The following benchmarks are including preprocessing (offline phase).
Time = 0.0390132 seconds
```

See the documentation for further options such as different protocols or running remotely and the machine learning section for other machine learning methods.

CHAPTER

FIFTEEN

MACHINE LEARNING

The purpose of this document is to demonstrate the machine learning functionality of MP-SPDZ, a software implementing multi-party computation, one of the most important privacy-enhancing techniques. Please see this gentle introduction for more information on multi-party computation and the installation instructions on how to install the software.

MP-SPDZ supports a number of machine learning algorithms such as logistic and linear regression, decision trees, and some common deep learning functionality. The latter includes the SGD and Adam optimizers and the following layer types: dense, 2D convolution, 2D max-pooling, and dropout.

The machine learning code only works in with arithmetic machines, that is, you cannot compile it with `-B`.

This document explains how to input data, how to train a model, and how to use an existing model for prediction.

15.1 Data Input

It's easiest to input data if it's available during compilation, either centrally or per party. Another way is to only define the data size in the high-level code and put the data independently into the right files used by the virtual machine.

15.1.1 Integrated Data Input

If the data is available during compilation, for example as a PyTorch or numpy tensor, you can use *Compiler.types.sfix.input_tensor_via()* and *Compiler.types.sint.input_tensor_via()*. Consider the following code from `breast_logistic.mpc` (requiring scikit-learn):

```
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split

X, y = load_breast_cancer(return_X_y=True)

# normalize column-wise
X /= X.max(axis=0)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)

X_train = sfix.input_tensor_via(0, X_train)
y_train = sint.input_tensor_via(0, y_train)
```

This downloads the Wisconsin Breast Cancer dataset, normalizes the sample data, splits it into a training and a test set, and then converts it to an the relevant MP-SPDZ data structures. Under the hood, the data is stored in `Player-Data/Input-Binary-P0-0`, which is where binary-encoded inputs for player 0 are read from. You therefore have to copy said file if you execute it in another place than where you compiled it.

MP-SPDZ also allows splitting the data input between parties, for example horizontally:

```
a = sfix.input_tensor_via(0, X_train[len(X_train) // 2:])
b = sfix.input_tensor_via(1, X_train[:len(X_train) // 2])
X_train = a.concat(b)

a = sint.input_tensor_via(0, y_train[len(y_train) // 2:])
b = sint.input_tensor_via(1, y_train[:len(y_train) // 2])
y_train = a.concat(b)
```

The concatenation creates a unified secret tensor that can be used for training over the whole dataset. Similarly, you can split a dataset vertically:

```
a = sfix.input_tensor_via(0, X_train[:,:X_train.shape[1] // 2])
b = sfix.input_tensor_via(1, X_train[:,X_train.shape[1] // 2:])
X_train = a.concat_columns(b)
```

The three approaches in this section can be run as follows:

```
Scripts/compile-run.py -E ring breast_logistic
Scripts/compile-run.py -E ring breast_logistic horizontal
Scripts/compile-run.py -E ring breast_logistic vertical
```

In the last variants, the labels are all input via party 0.

Finally, MP-SPDZ also facilitates inputting data that is also available party by party. Party 0 can run:

```
a = sfix.input_tensor_via(0, X_train[:,:X_train.shape[1] // 2])
b = sfix.input_tensor_via(1, shape=X_train[:,X_train.shape[1] // 2:].shape)
X_train = a.concat_columns(b)
y_train = sint.input_tensor_via(0, y_train)
```

while party 1 runs:

```
a = sfix.input_tensor_via(0, shape=X_train[:,:X_train.shape[1] // 2].shape)
b = sfix.input_tensor_via(1, X_train[:,X_train.shape[1] // 2:])
X_train = a.concat_columns(b)
y_train = sint.input_tensor_via(0, shape=y_train.shape)
```

Note that that the respective party only accesses the shape of data they don't input.

You can run this case by running on one hand:

```
./compile.py breast_logistic party0
./semi-party.x 0 breast_logistic-party0
```

and on the other (but on the same host):

```
./compile.py breast_logistic party1
./semi-party.x 1 breast_logistic-party1
```

The compilation will output a hash at the end, which has to agree between the parties. Otherwise the virtual machine will abort with an error message. To run the two parties on different hosts, use the *networking options*.

15.1.1.1 Data preprocessing

Sometimes it's necessary to preprocess data. We're using the following code from `torch_mnist_dense.mpc` to demonstrate this:

```
ds = torchvision.datasets.MNIST(root='/tmp', train=train, download=True)
# normalize to [0,1] before input
samples = sfix.input_tensor_via(0, ds.data / 255)
labels = sint.input_tensor_via(0, ds.targets, one_hot=True)
```

This downloads the default training or the test set of MNIST (depending on `train`) and then processes it to make it usable. The sample data is normalized from an 8-bit integer to the interval $[0, 1]$ by dividing by 255. This is done within PyTorch for efficiency. Then, the labels are encoded as one-hot vectors because this is necessary for multi-label training in MP-SPDZ.

15.1.2 Independent Data Input

The example code in `keras_mnist_dense.mpc` 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)
```

The virtual machine then expect the data as whitespace-separated text in `Player-Data/Input-P0-0`. If you use `binary=True` with `input_from()`, the input is expected in `Player-Data/Input-Binary-P0-0`, value by value as single-precision float or 64-bit integer in the machine byte order (most likely little-endian these days).

15.2 Training

There are a number of interfaces for different algorithms.

15.2.1 Logistic regression with SGD

This is available via *SGDLogistic*. We will use `breast_logistic.mpc` as an example.

After inputting the data as above, you can call the following:

```
log = ml.SGDLogistic(20, 2, program)
log.fit(X_train, y_train)
```

This trains a logistic regression model in secret for 20 epochs with mini-batches of size 2. Adding the `program` object as a parameter uses further command-line parameters. Most notably, you can add `approx` to use a three-piece approximate sigmoid function:

```
Scripts/compile-emulate.py breast_logistic approx
```

Omitting it invokes the default sigmoid function.

To check accuracy during training, you can call the following instead of *fit()*:

```
log.fit_with_testing(X_train, y_train, X_test, y_test)
```

This outputs losses and accuracy for both the training and test set after every epoch.

You can use *predict()* to predict labels and *predict_proba()* to predict probabilities. The following outputs the correctness (0 for correct, ± 1 for incorrect) and a measure of how much off the probability estimate is:

```
print_ln('%s', (log.predict(X_test) - y_test.get_vector()).reveal())
print_ln('%s', (log.predict_proba(X_test) - y_test.get_vector()).reveal())
```

15.2.2 Linear regression with SGD

This is available via *SGDLinear*. It implements an interface similar to logistic regression. The main difference is that there is only *predict()* for prediction as there is no notion of labels in this case. See `diabetes.mpc` for an example of linear regression.

15.2.3 PyTorch interface

MP-SPDZ supports importing sequential models from PyTorch using *layers_from_torch()* as shown in this code snippet in `torch_mnist_dense.mpc`:

```
import torch.nn as nn

net = nn.Sequential(
  nn.Flatten(),
  nn.Linear(28 * 28, 128),
  nn.ReLU(),
  nn.Linear(128, 128),
  nn.ReLU(),
  nn.Linear(128, 10)
)

from Compiler import ml

ml.set_n_threads(int(program.args[2]))

layers = ml.layers_from_torch(net, training_samples.shape, 128)

optimizer = ml.SGD(layers)
optimizer.fit(
  training_samples,
  training_labels,
  epochs=int(program.args[1]),
  batch_size=128,
  validation_data=(test_samples, test_labels),
  program=program
)
```

This trains a network with three dense layers on MNIST using SGD, softmax, and cross-entropy loss. The number of epochs and threads is taken from the command line. For example, the following trains the network for 10 epochs using 4 threads:

```
Scripts/compile-emulate.py torch_mnist_dense 10 4
```

See `Programs/Source/torch_*.mpc` for further examples of the PyTorch functionality, *fit()* for further training options, and *Adam* for an alternative Optimizer.

15.2.4 Keras interface

The following Keras-like code sets up a model with three dense layers and then trains 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)
)
```

See `Programs/Source/keras_*.mpc` for further examples using the Keras interface.

15.2.5 Decision trees

MP-SPDZ can train decision trees for binary labels by using the algorithm by Hamada et al. The following example in `breast_tree.mpc` trains a tree of height five before outputting the difference between the prediction on a test set and the ground truth:

```
from Compiler.decision_tree import TreeClassifier
tree = TreeClassifier(max_depth=5)
tree.fit(X_train, y_train)
print_ln('%s', (tree.predict(X_test) - y_test.get_vector()).reveal())
```

You can run the example as follows:

```
Scripts/compile-emulate.py breast_tree
```

It is also possible to output the accuracy after every level:

```
tree.fit_with_testing(X_train, y_train, X_test, y_test)
```

You can output the trained tree as follows:

```
tree.output()
```

The format of the output follows the description of Hamada et al.

MP-SPDZ by default uses probabilistic rounding for fixed-point division, which is used to compute Gini coefficients in decision tree training. This has the effect that the tree isn't deterministic. You can switch to deterministic rounding as follows:

```
sfix.round_nearest = True
```

The `breast_tree.mpc` uses the following code to allow switching on the command line:

```
sfix.set_precision_from_args(program)
```

Nearest rounding can then be activated as follows:

```
Scripts/compile-emulate.py breast_tree nearest
```

15.2.5.1 Data preparation

MP-SPDZ currently support continuous and binary attributes but not discrete non-binary attributes. However, such attributes can be converted as follows using the pandas library:

```
import pandas
from sklearn.model_selection import train_test_split
from Compiler import decision_tree

data = pandas.read_csv(
  'https://raw.githubusercontent.com/jbrownlee/Datasets/master/adult-all.csv',␣
↪header=None)

data, attr_types = decision_tree.preprocess_pandas(data)

# label is last column
X = data[:,:-1]
y = data[:,-1]

X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
```

This downloads the adult dataset and convert discrete attributes to binary using one-hot encoding. See `easy_adult` for the full example. `attr_types` has to be used to indicates the attribute types during training:

```
tree.fit(X_train, y_train, attr_types=attr_types)
```

15.3 Loading pre-trained models

It is possible to import pre-trained from PyTorch as shown in `torch_mnist_lenet_predict.mpc`:

```
net = nn.Sequential(
  nn.Conv2d(1, 20, 5),
  nn.ReLU(),
  nn.MaxPool2d(2),
  nn.Conv2d(20, 50, 5),
  nn.ReLU(),
  nn.MaxPool2d(2),
  nn.Flatten(),
  nn.ReLU(),
  nn.Linear(800, 500),
  nn.ReLU(),
  nn.Linear(500, 10)
)

# train for a bit
transform = torchvision.transforms.Compose(
  [torchvision.transforms.ToTensor()])
ds = torchvision.datasets.MNIST(root='/tmp', transform=transform, train=True)
optimizer = torch.optim.Adam(net.parameters(), amsgrad=True)
criterion = nn.CrossEntropyLoss()

for i, data in enumerate(torch.utils.data.DataLoader(ds, batch_size=128)):
  inputs, labels = data
  optimizer.zero_grad()
  outputs = net(inputs)
  loss = criterion(outputs, labels)
  loss.backward()
  optimizer.step()
```

This trains LeNet on MNIST for one epoch. The model can then be input and used in MP-SPDZ:

```
from Compiler import ml
layers = ml.layers_from_torch(net, training_samples.shape, 128, input_via=0)
optimizer = ml.Optimizer(layers)
n_correct, loss = optimizer.reveal_correctness(test_samples, test_labels, 128,␣
↪running=True)
print_ln('Secure accuracy: %s/%s', n_correct, len(test_samples))
```

This outputs the accuracy of the network. You can use *eval()* instead of *reveal_correctness()* to retrieve probability distributions or top guesses (the latter with `top=True`) for any sample data.

You can also use some networks provided within PyTorch as demonstrated by `../Programs/Source/torch_squeeze.py`:

```
model = torchvision.models.get_model('SqueezeNet1_1', weights='DEFAULT')
layers = ml.layers_from_torch(model, ...)
```

15.4 Storing and loading models

Both the Keras interface and the native *Optimizer* class support an interface to iterate through all model parameters. The following code from `torch_mnist_dense.mpc` uses it to store the model on disk in secret-shared form:

```
for var in optimizer.trainable_variables:
  var.write_to_file()
```

The example code in `torch_mnist_dense_predict.mpc` then uses the model stored above for prediction. Much of the setup is the same, but instead of training it reads the model from disk:

```
optimizer = ml.Optimizer(layers)

start = 0
for var in optimizer.trainable_variables:
  start = var.read_from_file(start)
```

Then it runs the accuracy test:

```
n_correct, loss = optimizer.reveal_correctness(test_samples, test_labels, 128)
print_ln('Accuracy: %s/%s', n_correct, len(test_samples))
```

Using `var.input_from(player)` instead the model would be input privately by a party.

15.5 Exporting models

Models can be exported as follows:

```
optimizer.reveal_model_to_binary()
```

if `optimizer` is an instance of *Compiler.ml.Optimizer*. The model parameters are then stored in `Player-Data/Binary-Output-P<playerno>-0`. They can be imported for use in PyTorch:

```
f = open('Player-Data/Binary-Output-P0-0')

state = net.state_dict()

for name in state:
    shape = state[name].shape
    size = numpy.prod(shape)
    var = numpy.fromfile(f, 'double', count=size)
    var = var.reshape(shape)
    state[name] = torch.Tensor(var)

net.load_state_dict(state)
```

if `net` is a PyTorch module with the correct meta-parameters. This demonstrates that the parameters are stored with double precision in the canonical order.

There are a number of scripts in `Scripts`, namely `torch_cifar_alex_import.py`, `torch_mnist_dense_import.py`, and `torch_mnist_lenet_import.py`, which import the models output by `torch_alex_test.mpc`, `torch_mnist_dense.mpc`, and `torch_mnist_lenet_predict.mpc`. For example you can run:

```
$ Scripts/compile-emulate.py torch_mnist_lenet_predict
...
Secure accuracy: 9822/10000
...
$ Scripts/torch_mnist_lenet_import.py
Test accuracy of the network: 98.22 %
```

The accuracy values might vary as the model is freshly trained, but they should match.

CHAPTER

SIXTEEN

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.

16.1 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

16.1.1 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 *MultiPlayer< int >*, *MultiPlayer< ssl_socket * >*, AllButLastPlayer, *MultiPlayer< T >*

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

virtual void **send_all**(const vector<*octetStream*> &os) const

Send something different to each of the other players

virtual 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

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 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, long n_bytes = -1)

Append big integer.

void **get**(*bigint* &ans, long n_bytes = -1)

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**(const string &filename)

Input from file, overwriting current content.

void **input**(istream &s)

Input from stream, overwriting current content.

void **output**(ostream &s) const

Output to stream.

void **output**(const string &filename)

Output to file.

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`

CHAPTER

SEVENTEEN

INPUT/OUTPUT

This section gives an overview over the input/output facilities.

17.1 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.

17.2 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.

17.3 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>`.

17.4 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 or `-OF .` to activate the output without interactive mode. Note that the former also causes to 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>`.

17.5 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.

17.6 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).

17.7 Clients (Non-computing Parties)

`Compiler.types.sint.receive_from_client()` and *`Compiler.types.sint.reveal_to_clients()`* allow communicating securely with the clients. See the relevant section 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 *C++ Reference* below.

17.8 Secret Shares via Socket

Secret can be sent and received via socket by using *`write_to_socket()`* and *`read_from_socket()`* (and the same functions in *`sfix`*). The connections are set up in the same way as in the previous section. See *Multinode Computation Example* for an example how this is used to distribute every party among multiple nodes. If you use the client interface, you should use the *`octetStream`* class for serialization. The format is the same as in the following section.

17.9 Secret Shares via Files

`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`. This files use the same header as *preprocessing files*. The format for the shares data depends on the protocol and is created by the `output` member function of the relevant *share type*. It follows 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.

17.10 Python Trusted Client Tutorial

In this section, we will illustrate how to use the client interface to supplement individual parties in the secure computation. This example consists of `../Programs/Source/personal_client_example.py` for the server side and `../ExternalIO/personal-client-example.py` for the client side.

The servers start by listening for and accepting one connection:

```
listen_for_clients(15000)
socket = accept_client_connection(15000)
```

The clients in turn connect to the server that is assigned to them:

```
party = int(sys.argv[1])
client = Client(['localhost'], 15000 + party, 0)
```

`party` stands for the number of the relevant server. Then, the clients of the of the first two servers sample 1000 random values and send them to their assigned server:

```
n = 1000
if party < 2:
  client.send_public_inputs(random.gauss(0, 1) * 2 ** 16 for i in range(n))
```

Note that the values are multiplied by 2^{16} to match the default fixed-point precision.

The first two servers then receive these values, convert them to shares, and then send the *shares* to their personal client:

```
n = 1000
for i in range(2):
  x = personal.read_fix_from_socket(i, socket, n)
  sfix(x).write_fully_to_socket(socket)
```

Note that all servers run this code because they are all involved in the secret-sharing process. If you're aiming for the secret sharing to happen on the client side, see this section.

The clients receive the shares and sum them pair-wise before sending them back:

```
x = [client.receive_plain_values() for i in range(2)]
client.send_public_inputs(a + b for a, b in zip(*x))
```

Note that this works whether the shares have MACs or not because adding shares with MACs amounts to simply adding both.

The servers finally receive the summed values, perform another sum, and output the result:

```
res = sum(sfix.read_from_socket(socket, n))
print_ln('%s', res.reveal())
```

17.11 Python Reference

class ExternalIO.client.**Client**(*hostnames*, *port_base*, *my_client_id*)

Client to servers running secure computation. Works both as a client to all parties or a trusted client to a single party.

Parameters

- **hostnames** – hostnames or IP addresses to connect to
- **port_base** – port number for first hostname, increases by one for every additional hostname
- **my_client_id** – number to identify client

receive_outputs(*n*)

Receive outputs privately from the computation servers. This assumes that the client is connected to all servers.

Parameters

n – number of outputs

receive_plain_values(*socket=None*)

Receive values in the clear. This works for public inputs to all servers or to send shares to a single server.

Parameters
socket – socket to use (need to specify it there is more than one)

send_private_inputs(*values*)

Send inputs privately to the computation servers. This assumes that the client is connected to all servers.

Parameters
values – list of input values

send_public_inputs(*values*)

Send values in the clear. This works for public inputs to all servers or to send shares to a single server.

Parameters
values – list of values

17.12 C++ 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

CHAPTER

EIGHTEEN

CLIENT INTERFACE

The ExternalIO directory contains an example of managing I/O between external client processes and parties running MP-SPDZ engines. These instructions assume that MP-SPDZ has been built as per the project readme.

18.1 Working Examples

bankers-bonus-client.cpp and bankers-bonus-client.py act as a client to bankers_bonus.mpc and demonstrates sending input and receiving output as described by Damgård et al. The computation allows up to eight clients to input a number and computes the client with the largest input. You can run the C++ code as follows from the main directory:

```
make bankers-bonus-client.x
./compile.py bankers_bonus 1
Scripts/setup-ssl.sh <nparties>
Scripts/setup-clients.sh 3
PLAYERS=<nparties> Scripts/<protocol>.sh bankers_bonus-1 &
./bankers-bonus-client.x 0 <nparties> 100 0 &
./bankers-bonus-client.x 1 <nparties> 200 0 &
./bankers-bonus-client.x 2 <nparties> 50 1
```

<protocol> can be any arithmetic protocol (e.g., `mascot`) but not a binary protocol (e.g., `yao`). This should output that the winning id is 1. Note that the ids have to be incremental, and the client with the highest id has to input 1 as the last argument while the others have to input 0 there. Furthermore, <nparties> refers to the number of parties running the computation not the number of clients, and <protocol> can be the name of protocol script. The setup scripts generate the necessary SSL certificates and keys. Therefore, if you run the computation on different hosts, you will have to distribute the `*.pem` files.

For the Python client, make sure to install gmpy2, and run `ExternalIO/bankers-bonus-client.py` instead of `bankers-bonus-client.x`.

18.2 I/O MPC Instructions

18.2.1 Connection Setup

1. Listen for clients
2. Accept client connections
3. Close client connections

18.2.2 Data Exchange

Only the `sint` methods used in the example are documented here, equivalent methods are available for other data types. See the reference.

1. Public value from client
2. Secret value from client
3. Reveal secret value to clients

18.3 Client-Side Interface

The example uses the `Client` class implemented in `ExternalIO/Client.hpp` to handle the communication, see this reference for documentation.

CHAPTER

NINETEEN

MULTINODE COMPUTATION EXAMPLE

Multinode computation refers to the possibility of distributing a every party across several nodes. MP-SPDZ uses the client interface for communication between nodes. This means that you have to run `Scripts/setup-clients.sh` and distribute the certificates to run it across several machines.

In the following, we will explain the example in `../Programs/Source/multinode_example_main.py` and `../Programs/Source/multinode_example_worker.py`.

First, the one main node per party, listens and accepts connections from the worker nodes in the same logical party:

```
listen_for_clients(15000)

ready = regint.Array(n_nodes_per_party)

@for_range(n_nodes)
def _(i):
  ready[accept_client_connection(15000)] = 1

runtime_error_if(sum(ready) != n_nodes_party, 'connection problems')
```

Maintaining `ready` helps spot errors but isn't strictly necessary. Meanwhile, the workers connect to main node:

```
main = init_client_connection(host, 15000, worker_id)
```

Once the connection is established, the main node distributes the data among the workers:

```
@for_range(n_nodes_per_party)
def _(i):
  data.get_vector(base=i * n_ops_per_node,
    size=n_ops_per_node).write_fully_to_socket(i)
```

This sends a different chunk `data` to every node. The workers then receive it and execute the computation (squaring every number in the example), and send the result back:

```
@for_range_opt_multithread(n_threads, n_ops)
def _(i):
  data[i] = data[i] ** 2

data.write_to_socket(main)
```

Finally, the main node receives the result:

```
@for_range(n_nodes_per_party)
def _(i):
  data.assign_vector(sint.read_from_socket(i, size=n_ops_per_node),
    base=i * n_ops_per_node)
```

You can execute example with three parties, four worker nodes per party, five threads per worker node, and 1000 operations per thread as follows:

```
for i in $(seq 0 3); do
  Scripts/compile-run.py ring multinode_example_worker 5 1000 $i localhost & true
done

Scripts/compile-run.py ring multinode_example_main 4 5 1000
```

CHAPTER

TWENTY

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.

If you want to use this approach with a given prime, do *not* specify the prime during compilation but during execution, that is:

```
<protocol>-party.x -P <prime> ...
```

or:

```
Scripts/<protocol>.sh -P <prime> ...
```

If using `Scripts/compile-run.py`, put it after a double dash:

```
Scripts/compile-run.py <compiler-args> -- -P <prime> <execution-options>
```

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.

This approach is used if you specify a prime during compilation.

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.

20.1 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.

20.1.1 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

CHAPTER

TWENTYONE

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.

21.1 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
- MAC if applicable

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  31 00 00 00 00 00 00 00  53 50 44 5a 20 67 66 70  |1.......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 01 00 00  00 78 03 11 c6 61 5a 68  |.........x...aZh|
00000030  5a                                                |Z|
```

The last 128 bits denote the MAC and will differ from instance to instance. The MAC is stored to avoid errors that are hard to track otherwise.

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. As an example, consider the default 128-bit prime $p = 170141183460469231731687303715885907969$. The Montgomery representation of x is $xR \bmod p$. For $x = 1$, this is 170141183460469231731687303715882303487 or 0x7fffffffffffffffffffffffe47fff in hexadecimal. Using to little-endian, `hexdump -C` would output the following:

```
ff 7f e4 ff ff ff ff ff ff ff ff ff ff ff ff 7f
```

Modulo a power of two:
Values are stored in the smallest number of 8-byte blocks necessary, all in little-endian order, so 1 with a modulus of 2^{64} would result in the following `hexdump -C` output:

```
01 00 00 00 00 00 00 00
```

$GF(2^n)$
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.

CHAPTER

TWENTYTWO

LOWEST-LEVEL INTERFACE

In the following, we will introduce the most protocol-independent interfaces by walking through Utils/protocol-tutorial. It implements the Rep3 multiplication protocol independently of the usual protocol interface for illustration purposes.

```
// set up networking on localhost
int my_number = atoi(argv[1]);
int port_base = 9999;
Names N(my_number, 3, "localhost", port_base);
CryptoPlayer P(N);
```

This sets up pairwise encrypted connections as in *the low-level example*.

```
// correlated randomness for resharing
SeededPRNG G[2];
```

The protocol requires every pair of parties to have a common PRNG, so we need two instances. We use *SeededPRNG* to make sure to never use an uninitialized one.

```
// synchronize with other parties
octetStream os;
os.append(G[0].get_seed(), SEED_SIZE);
```

octetStream is generally used to serialize and aggregate network communication. In this case, we use it to store the seed of one of the PRNGs.

```
P.pass_around(os, os, 1);
```

Player::pass_around() allows simultaneous sending to the "next" party and receiving from the "previous" party. We use this with the buffer holding the seed. As we don't need the send buffer afterwards, we can use the same buffer for receiving.

```
G[1].SetSeed(os.consume(SEED_SIZE));
```

We seed the second PRNG using the received data. *PRNG::SetSeed()* implicitly uses the required number of bits.

```
// simplify code
typedef Z2<64> Z;
```

In this example, we use integers modulo 2^{64}, but the protocol also works for any modulus, so we could also use *gfp_*.

```
// start with same shares on all parties for simplicity
// replicated secret sharing of 3
Z a[2] = {1, 1};
```

(continues on next page)

(continued from previous page)

```
// and 6
Z b[2] = {2, 2};
```

For every secret number in Rep3, every party holds a pair of numbers in the domain such that every pair of parties has the same number. The sum of the unique numbers is the secret.

```
// compute an additive sharing of the product
Z cc = a[0] * (b[0] + b[1]) + a[1] * b[0];
```

In a first step, every party computes an additive share of the product. See Araki et al. for details. All domain classes support the standard operators.

```
// result shares
Z c[2];

// re-randomize
c[0] = cc + G[0].get<Z>() - G[1].get<Z>();
```

Sending the computed additive secret sharing directly to another party to get back to a replicative secret sharing would be insecure. Therefore, we randomize it using random numbers from the two PRNGs.

```
// send and receive share
os.reset_write_head();
c[0].pack(os);
P.pass_around(os, os, 1);
c[1].unpack(os);
```

We clear the buffer, serialize our share, send it to the "next" party, and receive one from the "previous" party. This concludes the multiplication protocol. `Z2::pack()` and `Z2::unpack()` are main methods for (de-)serialization. All domain classes support this. You can use *octetStream::output()* to write the buffer to a C++ output stream.

```
// open value to party 0
if (P.my_num() == 1)
{
    os.reset_write_head();
    c[0].pack(os);
    P.send_to(0, os);
}
```

To allow party 0 to output the result, party 1 serializes one of their shares and sends it to party 0.

```
// output result on party 0, which should be 18
if (P.my_num() == 0)
{
    P.receive_player(1, os);
    cout << "My shares: " << c[0] << ", " << c[1] << endl;
    cout << "Result: " << (os.get<Z>() + c[0] + c[1]) << endl;
}
```

Party 0 receives the missing share from party 1 and reconstructs the secret by summing up.

You can run the example as follows in the main directory:

```
make protocol-tutorial.x
for i in 0 1 2; do ./protocol-tutorial.x $i & true; done
```

CHAPTER

TWENTYTHREE

ADDING A PROTOCOL

In order to illustrate how to create a virtual machine for a new protocol, we have created one with blanks to be filled in. It is defined in the following files:

Machines/no-party.cpp
: Contains the main function.

Protocols/NoShare.h
: Contains the `NoShare` class, which is supposed to hold one share. `NoShare` takes the cleartext type as a template parameter.

Protocols/NoProtocol.h
: Contains a number of classes representing instances of protocols:

  NoInput
  : Private input.

  NoProtocol
  : Multiplication protocol.

  NoOutput
  : Public output.

Protocols/NoLivePrep.h
: Contains the `NoLivePrep` class, representing a preprocessing instance.

The number of blanks can be overwhelming. We therefore recommend the following approach to get started. If the desired protocol resembles one that is already implemented, you can check its code for inspiration. The main function of `<protocol>-party.x` can be found in `Machines/<protocol>-party.cpp`, which in turns contains the name of the share class. For example `replicated-ring-party.x` is implemented in `Machines/replicated-ring-party.cpp`, which refers to `Rep3Share2()` in `Protocols/Rep3Share2.h`. There you will find that it uses `Replicated()` for multiplication, which is found in `Protocols/Replicated.h`. You can also consult *the tutorial for the lowest-level interface*.

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('result: %s', sint(123).reveal())
   ```

   This allows to check the correct execution of further functionality.

   Put the above code in `Programs/Source/test.mpc` and run the following if your protocol works for two parties (otherwise add more parties and change the `-N` argument accordingly):

```
make no-party.x
./compile.py test
./no-party.x 0 test -N 2 & ./no-party.1 test -N 2
```

This should output `result:  123`.

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()
```

23.1 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 initialization will fail.

PRNG(*octetStream* &seed)

Initialize with `SEED_SIZE` bytes from buffer.

PRNG(const string &seed)

Initialize with `SEED_SIZE` bytes from buffer.

void **ReSeed**()

Initialize from local randomness.

void **SeedGlobally**(const PlayerBase &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.

template<class **T**>
void **SetSeedFrom**(const *T* &seed)

Initialize with seed from any serializable.

inline bool **get_bit**()

Random bit.

inline unsigned char **get_uchar**()

Random bytes.

inline 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.

CHAPTER

TWENTYFOUR

ADDING AN INSTRUCTION

If you want to add functionality that isn't captured by the current virtual machine design, you might need to add further instructions. This section explains how the virtual machine is built on both the frontend (Python) and the backend (C++).

24.1 Frontend

The instructions are defined as classes in `../Compiler/instructions.py` and `../Compiler/GC/instructions.py`. Every class requires the attributes `opcode` and `arg_format` to be set. Consider the example of *prefixsums*:

```
@base.vectorize
class prefixsums(base.Instruction):
  """ Prefix sum.

  :param: result (sint)
  :param: input (sint)

  """
  __slots__ = []
  code = base.opcodes['PREFIXSUMS']
  arg_format = ['sw','s']
```

`opcode` is set from `opcodes` in `../Compiler/instructions_base.py`. This is simply for convenience as it allows copying from the C++ code (see below). The only requirement for opcodes is that they are unique 10-bit integers.

`arg_format` has to be iterable over strings indicating the nature of arguments. In the example, `sw` indicates that a secret integer register is written to, and `s` indicates that a secret integer register is read from. `ArgFormats` defines all possible argument types:

```
ArgFormats = {
  'c': ClearModpAF,
  's': SecretModpAF,
  'cw': ClearModpAF,
  'sw': SecretModpAF,
  'cg': ClearGF2NAF,
  'sg': SecretGF2NAF,
  'cgw': ClearGF2NAF,
  'sgw': SecretGF2NAF,
  'ci': ClearIntAF,
  'ciw': ClearIntAF,
  '*': AnyRegAF,
```

(continues on next page)

(continued from previous page)

```
  '*w': AnyRegAF,
  'i': ImmediateModpAF,
  'ig': ImmediateGF2NAF,
  'int': IntArgFormat,
  'long': LongArgFormat,
  'p': PlayerNoAF,
  'str': String,
  'varstr': VarString,
}
```

The values of this dictionary are classes defined in `../Compiler/instructions_base.py` which can encode them into bytes to written to the bytecode files. Most types are encoded as four-byte values except `long`, which uses eight bytes, and `varstr`, which has a variable length. The type classes also define functionality to check arguments for correctness such as Python type.

By default, register arguments are understood as single registers. The `vectorize` decorator is an easy way to allow vector arguments if all register arguments have the same length. The vector size is stored independently of the arguments. The decorator creates two instructions, a base version for single registers and a vectorized version, which is called as follows in the example for length `n`:

```
vprefixsums(n, result, operand)
```

At the higher level, the vector length is usually derived from the input using the `vectorize` decorator as in *prefix_sum()*:

```
@vectorize
def prefix_sum(self):
    """ Prefix sum. """
    res = sint()
    prefixsums(res, self)
    return res
```

All instruction classes should inherit from `Instruction` in `../Compiler/instructions_base.py`.

24.2 Backend

The backend functionality has three aspects:

1. Parsing the bytecode and creating an internal representation
2. Figuring out the resource requirements of the instruction (registers and memory)
3. Execution

24.2.1 Parsing

The internal representation is done via the `Instruction` class defined in `../Processor/Instruction.h`. The arguments are parsed in `parse_operands()` defined in `../Processor/Instruction.hpp`. It contains a large switch statement covering all opcodes. Sticking to the example of *prefixsums*, the relevant code there is as follows:

```
case PREFIXSUMS:
  ...
  get_ints(r, s, 2);
  break;
```

This puts the two integer values corresponding to the two arguments into `r[0]` and `r[1]` within the `Instruction` object. `r` is an array of four 32-bit integers, which is enough for many simple instructions. More complex instruction use `start`, which is a variable-length C++ vector of 32-bit integers.

24.2.2 Resourcing

Because the number of registers depends on the programs, the virtual machine has to find out the requirements for every single instruction. The main function for this is `get_max_reg()` in `../Processor/Instruction.hpp`, which returns the maximum register that is written to for a particular register type. It contains two switch statements. The first one contains special treatment for instructions that write to more than one register type such as *`dabit`*. However, for most instruction including *`prefixsums`*, it checks the type currently queried against the type defined by `get_reg_type()` and returns 0 if there is a mismatch. `get_reg_type()` makes use of the fact that the opcodes are grouped. For `prefixsums`, it returns `SINT`, which is the default.

The second switch statement then treats further special cases where `start` is used or *r* contains registers of different types. None of this applies for `prefixsums`, so the return value is simply the maximum over `r` and `start` plus the vector size:

```
unsigned res = 0;
for (auto x : start)
  res = max(res, (unsigned)x);
for (auto x : r)
      res = max(res, (unsigned)x);
return res + size;
```

24.2.3 Execution

Execution is again defined by several switch statements over the opcode, the outermost of which is in `Program::execute()` defined in `../Processor/Instruction.hpp`. It uses the X macro pattern for a compact representation. *`prefixsums`* is implemented in `../Processor/instructions.h` as follows:

```
X(PREFIXSUMS, auto dest = &Procp.get_S()[r[0]]; auto op1 = &Procp.get_S()[r[1]]; \
  sint s, \
  s += *op1++; *dest++ = s) \
```

This macro has three arguments: the opcode, the setup step, and vector loop step. The setup step involves getting pointers according to the register addresses `r[0]` and `r[1]` as well as initializing a running variable. The loop step then adds the next input element to the running variable and stores in the destination.

Another important switch statement is in `Instruction::execute`. See *Execution* for further examples.

CHAPTER

TWENTYFIVE

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.

25.1 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, int = -1) const

Append to buffer.

inline void **unpack**(*octetStream* &os, int = -1)

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, int = -1) const

Append to buffer.

inline void **unpack**(*octetStream* &o, int = -1)

Read from buffer. Assumes parameters are set correctly.

CHAPTER

TWENTYSIX

THRESHOLD ECDSA

The ECDSA directory contains the code used for the benchmarks by Dalskov et al. `*-ecdsa-party.cpp` contains the high-level programs while the two phases are implemented in `preprocessing.hpp` and `sign.hpp`, respectively.

26.1 Compilation

- Compile the binaries: `make -j8 ecdsa`
- Or compile the static binaries: `make -j8 ecdsa-static`

26.2 Running

The following binaries are available, of which the first few have been used for the paper:

Protocol	Binary
MASCOT	`mascot-ecdsa-party.x`
Semi-honest OT	`semi-ecdsa-party.x`
Malicious Shamir	`mal-shamir-ecdsa-party.x`
Semi-honest Shamir	`shamir-ecdsa-party.x`
Malicious replicated	`mal-rep-ecdsa-party.x`
Semi-honest replicated	`rep-ecdsa-party.x`
—	—
ATLAS	`atlas-ecdsa-party.x`
SPDZ-wise replicated	`sy-rep-ecdsa-party.x`
Rep4	`rep4-ecdsa-party.x`

All binaries offer the same interface. With MASCOT for example, run the following:

```
./mascot-ecdsa-party.x -p 0 [-N <number of parties>] [-h <host of party 0>] [-D] [
↪<number of prep tuples>]
./mascot-ecdsa-party.x -p 1 [-N <number of parties>] [-h <host of party 0>] [-D] [
↪<number of prep tuples>]
...
```

`-D` activates delayed multiplication, deferring usage of the secret key until signing.

The number of parties defaults to 2 for OT-based protocols and to 3 for honest-majority protocols.

In addition, there is `fake-spdz-ecsda-party.x`, which runs only the online phase of SPDZ. You will need to run `Fake-ECDSA.x` beforehands and then distribute `Player-Data/ECSDA` to all parties.

CHAPTER

TWENTYSEVEN

SECURITY FIXES

This section explains select security fixes and credits people who have reported them.

27.1 Insufficient check in SPDZ-wise replicated sharing over rings

Nergiz Yuca discovered that the implementation of the zero check by Abspoel et al. would ignore the actual bits, which would make the check always pass together with another bug.

27.2 Insufficient checks in malicious replicated sharing with Beaver multiplication

Yufei Duan discovered a protocol bug in malicious honest-majority 3PC with Beaver multiplication where a delayed check could be used by a malicious party to guess an input to an earlier multiplication. The fixed protocol uses the replication to check every opening immediately without additional rounds. Every party receives the necessary values from one party and a hash thereof from another. One of the parties is honest, so any deviation will be immediately discovered.

27.3 Malicious parties are not committed to MAC key across threads and executions

MP-SPDZ used to read MAC key shares from a file if available and regenerate the relevant oblivious transfer instances or homomorphic encryptions at every run. For oblivious transfer, it even did so for every computation thread. Sela Navot discovered that this opens the possibility for malicious parties to generate valid shares with MAC under different MAC keys between different threads or runs, opening the possibility of selective failure attacks. The remedy is to only generate OT instances or encryptions once for a fresh MAC key and then store them on disk for future runs. This is done in `Player-Data/<protocol>-Secrets-<parameters>-P<player>-<n_players>`.

27.4 Insufficient checks in Rep4

Brüggemann and Schneider discovered selective failure attacks in the MP-SPDZ implementation of Rep4, which delayed consistency checks too much.

27.5 Co-ordinate MAC checks among threads and race condition in MAC checks

Kyster et al. found selective failure attacks if the result of computation in one thread would be used in another before checking. See also "SPDZ Multi-Threaded MAC Check" on mpcsec.org. A further related race condition has been

reported by Tadas Majorovas.

27.6 Remove MAC key in case of failure

If a MAC check fails, the MAC key should not be used again.

27.7 Remotely caused buffer overflows

MP-SPDZ generally sends length fields before data to allow the receiver to allocate a buffer of the correct size. However, Guopeng Lin found instances where and adversary could send data that causes a buffer overflow on the receiving side.

27.8 Missing shuffling check in PS mod 2^k and Brain

MP-SPDZ uses a class for every type of share and static variables within these classes to indicate a protocol provides malicious security. This variable is then used when there is a small difference between semi-honest and malicious protocols, such as in the generic secure shuffling protocol. The fact that this variable was incorrect for a few share classes lead to a necessary check being skipped.

27.9 Insufficient drowning in pairwise protocols

MP-SPDZ generates homomorphic encryption system parameters depending on variable parameters such as how much randomness is used noise drowning/flooding. An accidental reset to the default meant that any larger parameter given would be ignored.

27.10 Proper accounting for random elements

When using preprocessing from files, MP-SPDZ uses the predicted number of items to avoid using the item several times. There was an omission in the account of random elements used in some opening protocols, hence the potential for reuse.

27.11 Insufficient security in non-interactive zero-knowledge proofs

Sebastian Hasler pointed out that the default statistical security parameter does not suffice in the non-interactive setting where the adversary can try forging a proof without involving any other party.

27.12 Security bug in SPDZ2k

Daniel Escudero has found a bug in the original SPDZ2k where only the "payload bits" would be checked. This would lead to a high attack probability. Sections 3.2-3.4 of the updated paper describe the issue.

27.13 Skewed random bit generation

There was a bug in caching random bits such that every 64-th bit was guaranteed to be zero. This was only discovered when it caused issues with the initialization of machine learning models.

27.14 All-zero secret keys in homomorphic encryption

An insufficient initialization and checking of homomorphic encryption parameters lead to the generation of keys with Hamming weight 0.

27.15 Insufficient randomness in SemiBin random bit generation

The usage of an intricate inheritance structure for convenience lead to an assignment operation that would always assign zero.

27.16 Missing MAC key initialization

A missing initialization would lead to all MAC key shares for binary computation being zero.

CHAPTER

TWENTYEIGHT

TROUBLESHOOTING

This section shows how to solve some common issues.

28.1 Crash without error message, `Killed`, 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 many protocols with `--batch-size` (try 1 to confirm the issue and then increment to test the limits). Furthermore, the batch size for some malicious protocols can be reduced with `--bucket-size 5`. Every computation thread requires separate resources, so consider reducing the number of threads with `for_range_multithreads()` and similar. Lastly, you can use `--disk-memory <path>` to use disk space instead of RAM for large programs. Use `Scripts/memory-usage.py <program-with-args>` to get an estimate of the memory usage of a specific program.

28.2 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*.

28.3 Local variable referenced before assignment

This error can occur if you try to reassign a variable in a run-time loop like *for_range()*. Use `update()` instead of assignment. See *for_range()* for an example. You can also use *iadd()* instead of +=.

28.4 `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_opt()* or similar. You can also use `-l` when compiling, which will replace simple loops by an optimized version.

28.5 Cannot derive truth value from register

This message appears when you try to use branching on run-time data types, for example:

```
x = cint(0)
y = 0
if x == 0:
  y = 1
  print_ln('x is zero')
```

There a number of ways to solve this:

1. Use the `--flow-optimization` argument during compilation.
2. Use run-time branching:

   ```
   x = cint(0)
   y = cint(0)
   @if_(x == 0)
   def _():
     y.update(1)
     print_ln('x is zero')
   ```

   See *if_e()* for the equivalent to if/else.

3. Use conditional statements:

   ```
   check = x == 0
   y = check.if_else(1, y)
   print_ln_if(check, 'x is zero')
   ```

Use `bit_and` etc. for more elaborate conditions:

```
@if_(a.bit_and(b.bit_or(c)))
def _():
  ...
```

The underlying reason for this is that registers are only a placeholder during the execution in Python, the actual value of which is only defined in the virtual machine at a later time. See *The Journey of a Program* to get an understanding of the overall design.

28.6 Cannot branch on secret values

This message appears when you try to use branching on secret data types, for example:

```
x = sint(0)
if x:
  y = 1
else:
  y = 2
```

Deciding whether to execute `y = 1` or `y = 2` would reveal `x`, which contradicts the secrecy guarantee of *sint*. However, you can use the following to achieve the desired `y` without revealing `x`:

```
y = (x != 0).if_else(1, 2)
```

If `x` is guaranteed to be 0 or 1, you can also use:

```
y = x.if_else(1, 2)
```

If your use case permits revealing `x`, see the previous section for considerations on branching with run-time values.

28.7 Incorrect results when using `sfix`

This is most likely caused by an overflow of the precision parameters because the default choice unlike accommodates numbers up to around 16,000. See *sfix* for an introduction and *set_precision()* for how to change the precision.

28.8 Variable results when using `sfix`

This is caused the usage of probabilistic rounding, which is used to restore the representation after a multiplication. See Catrina and Saxena for details. You can switch to deterministic rounding by calling `sfix.round_nearest = True`.

28.9 Only party 0 produces outputs

This is to improve readability when running all parties in the same terminal. You can activate outputs on other parties using `-OF .` as an argument to a virtual machine (`*-party.x`).

28.10 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*. The error message does not necessarily mean there will be errors, but the compiler cannot guarantee that there will not. If you encounter such errors, you can fix this either with `-M` when compiling or enable memory protection (*protect_memory()*) around specific memory accesses.

28.11 High number of rounds or slow WAN execution

You can increase the optimization budget using `--budget` during compilation. The budget controls the trade-off between compilation speed/memory usage and communication rounds during execution. The default is 1000, but 100,000 might give better results while still keeping compilation manageable.

28.12 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.

28.13 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.

28.14 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.

28.15 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`.

28.16 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.

28.17 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.

28.18 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.

28.19 Computation used more preprocessing than expected

This indicates an error in the internal accounting of preprocessing. Please file a bug report.

28.20 Required prime bit length is not the same as `-F` parameter during compilation

This is related to statistical masking that requires the prime to be a fair bit larger than the actual "payload" (40 by default). The technique goes to back to Catrina and de Hoogh. See also the paragraph on unknown prime moduli in *Non-linear Computation*.

28.21 Prime number not compatible with encryption scheme

MP-SPDZ only supports homomorphic encryption based on the number-theoretic transform, without it operations would expected to be considerably. The requirement is that the prime number equals one modulo a certain power of two. The exact power of two varies due to a number of parameters, but for the standard choice it's usually 2^{14} or 2^{15}. See Gentry et al. for more details on the underlying mathematics.

28.22 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.

28.23 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.

28.24 Debugging errors in a virtual machine

Unlike Python or Java, C++ gives limited information when something goes wrong. On Linux, the GNU Debugger (GDB) aims to mitigate this by providing more introspection into where exactly something went wrong. MP-SPDZ comes with a few scripts that facilitate its use. First, you need to make sure gdb and screen are installed. On Ubuntu, you can run the following:

```
sudo apt-get install gdb screen
```

You can then run the following script call:

```
prefix=gdb_screen Scripts/<protocol>.sh ... -o throw_exceptions
```

This runs every party in the background using the screen utility. You can get a party to the foreground using:

```
screen -r :<partyno>
```

This will show the relevant running inside GDB. You can use the sequence "Ctrl-a d" to return to your usual terminal.

If running the different parties separately, you can also use:

```
. Scripts/run-common.sh
gdb_front ./<protocol>-party.x ... -o throw_exceptions
```

If the virtual machine aborts due to an error, GDB will indicate where in the code this happened. For example, deactivating all range checks on memory accesses and then running an illegal memory access triggers a segfault and the following output:

```
Thread 13 "shamir-party.x" received signal SIGSEGV, Segmentation fault.
[Switching to Thread 0x7fffdffff640 (LWP 246396)]
0x0000000000434c57 in MemoryPart<ShamirShare<gfp_<0, 2> > >::indirect_read<StackedVector
↪<Integer> > (this=<optimised out>, inst=..., regs=..., indices=...) at ./Processor/
↪Memory.hpp:26
26            *dest++ = data[it->get()];
```

Entering `bt` (for backtrace) gives even more information as to where the error happened:

```
(gdb) bt
#0  0x0000000000434c57 in MemoryPart<ShamirShare<gfp_<0, 2> > >::indirect_read
↪<StackedVector<Integer> > (this=<optimised out>, inst=..., regs=..., indices=...) at ./
↪Processor/Memory.hpp:26
#1  Program::execute<ShamirShare<gfp_<0, 2> >, ShamirShare<gf2n_long> > (this=0x620cc0,␣
↪Proc=...) at ./Processor/Instruction.hpp:1486
#2  0x0000000000428fd1 in thread_info<ShamirShare<gfp_<0, 2> >, ShamirShare<gf2n_long> >
↪::Sub_Main_Func (this=<optimised out>, this@entry=0x656900) at ./Processor/Online-
↪Thread.hpp:280
#3  0x0000000000426e45 in thread_info<ShamirShare<gfp_<0, 2> >, ShamirShare<gf2n_long> >
↪::Main_Func_With_Purge (this=0x656900) at ./Processor/Online-Thread.hpp:431
#4  thread_info<ShamirShare<gfp_<0, 2> >, ShamirShare<gf2n_long> >::Main_Func␣
↪(ptr=0x656900) at ./Processor/Online-Thread.hpp:410
#5  0x00007ffff6bbaac3 in start_thread (arg=<optimised out>) at ./nptl/pthread_create.
↪c:442
#6  0x00007ffff6c4c850 in clone3 () at ../sysdeps/unix/sysv/linux/x86_64/clone3.S:81
```

This information can be very useful to find the error and fix bugs, so make sure to include it in GitHub issues etc.

CHAPTER

TWENTYNINE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

C

Compiler.circuit, 129
Compiler.decision_tree, 128
Compiler.dijkstra, 138
Compiler.GC.instructions, 197
Compiler.GC.types, 93
Compiler.instructions, 160
Compiler.library, 105
Compiler.ml, 115
Compiler.mpc_math, 113
Compiler.oram, 133
Compiler.path_oblivious_heap, 135
Compiler.program, 131
Compiler.sorting, 139
Compiler.sparse, 140
Compiler.sqrt_oram, 135
Compiler.types, 35

INDEX

Symbols

-B
    command line option, 28, 31
-C
    command line option, 29
-D
    command line option, 28, 34
-E
    command line option, 28, 32
-F
    command line option, 27, 32
-G
    command line option, 28
-I
    command line option, 34
-IF
    command line option, 34
-N
    command line option, 33
-OF
    command line option, 34
-P
    command line option, 27, 32
-Q
    command line option, 31
-R
    command line option, 28, 32
-S
    command line option, 32
-SP
    command line option, 32
-X
    command line option, 28
-Y
    command line option, 28
-Z
    command line option, 28
--CISC
    command line option, 29
--batch-size
    command line option, 31
--binary
    command line option, 28
--bits-from-squares
    command line option, 31
--bucket-size
    command line option, 31
--budget
    command line option, 29
--code-locations
    command line option, 34
--dead-code-elimination
    command line option, 28
--direct
    command line option, 31
--disk-memory
    command line option, 34
--edabit
    command line option, 28
--encrypted
    command line option, 33
--execute
    command line option, 28
--external-server
    command line option, 33
--field
    command line option, 27
--file-prep-per-thread
    command line option, 32
--file-preprocessing
    command line option, 32
--flow-optimization
    command line option, 29
--garbled-circuit
    command line option, 28
--hostname
    command line option, 33
--input-file
    command line option, 34
--interactive
    command line option, 34
--ip-file-name
    command line option, 33
--lg2

command line option, 32
--lgp
command line option, 32
--memory
command line option, 34
--mixed
command line option, 28
--my-port
command line option, 33
--nparties
command line option, 33
--options
command line option, 34
--output-file
command line option, 34
--player
command line option, 33
--portnumbase
command line option, 33
--prime
command line option, 27, 32
--ring
command line option, 28, 32
--security
command line option, 32
--spdz2k-security
command line option, 32
--split
command line option, 28
--trunc-error
command line option, 32
--unencrypted
command line option, 33
--verbose
command line option, 34
-b
command line option, 29, 31
-d
command line option, 31
-e
command line option, 33
-ext-server
command line option, 33
-f
command line option, 32
-h
command line option, 33
-ip
command line option, 33
-l
command line option, 29
-lg2
command line option, 32
-lgp
command line option, 32
-m
command line option, 34
-mp
command line option, 33
-o
command line option, 34
-p
command line option, 33
-pn
command line option, 33
-u
command line option, 33
-v
command line option, 34

A

accept_client_connection() (*in module Compiler.library*), 105
acceptclientconnection (*class in Compiler.instructions*), 160
acos() (*in module Compiler.mpc_math*), 113
active (*class in Compiler.instructions*), 161
active (*Compiler.program.Program property*), 132
Adam (*class in Compiler.ml*), 116
Add (*class in Compiler.ml*), 117
add() (*Compiler.GC.types.sbitfixvec method*), 96
addc (*class in Compiler.instructions*), 161
addcb (*class in Compiler.GC.instructions*), 197
addcbi (*class in Compiler.GC.instructions*), 197
addci (*class in Compiler.instructions*), 161
addint (*class in Compiler.instructions*), 161
addm (*class in Compiler.instructions*), 161
adds (*class in Compiler.instructions*), 162
addsi (*class in Compiler.instructions*), 162
andc (*class in Compiler.instructions*), 162
andci (*class in Compiler.instructions*), 162
andm (*class in Compiler.GC.instructions*), 197
andrs (*class in Compiler.GC.instructions*), 197
andrsvec (*class in Compiler.GC.instructions*), 198
ands (*class in Compiler.GC.instructions*), 198
applyshuffle (*class in Compiler.instructions*), 162
applyshuffleb (*class in Compiler.GC.instructions*), 198
approx_sigmoid() (*in module Compiler.ml*), 127
Argmax (*class in Compiler.ml*), 118
argmax() (*in module Compiler.ml*), 125
Array (*class in Compiler.types*), 36
Array() (*Compiler.GC.types.cbits class method*), 93
Array() (*Compiler.GC.types.sbit class method*), 94
Array() (*Compiler.GC.types.sbitfixvec class method*), 96
Array() (*Compiler.GC.types.sbits class method*), 101
Array() (*Compiler.types.cfix class method*), 55
Array() (*Compiler.types.cgf2n class method*), 57

Array() (*Compiler.types.cint class method*), 61
Array() (*Compiler.types.regint class method*), 66
Array() (*Compiler.types.sfix class method*), 71
Array() (*Compiler.types.sfloat class method*), 76
Array() (*Compiler.types.sgf2n class method*), 78
Array() (*Compiler.types.sint class method*), 84
asin() (*in module Compiler.mpc_math*), 113
asm_open (*class in Compiler.instructions*), 163
assign() (*Compiler.types.Array method*), 36
assign() (*Compiler.types.Matrix method*), 41
assign() (*Compiler.types.MultiArray method*), 49
assign_all() (*Compiler.types.Array method*), 36
assign_all() (*Compiler.types.Matrix method*), 41
assign_all() (*Compiler.types.MultiArray method*), 49
assign_part_vector() (*Compiler.types.Array method*), 36
assign_part_vector() (*Compiler.types.Matrix method*), 41
assign_part_vector() (*Compiler.types.MultiArray method*), 49
assign_vector() (*Compiler.types.Array method*), 36
assign_vector() (*Compiler.types.Matrix method*), 41
assign_vector() (*Compiler.types.MultiArray method*), 49
assign_vector_by_indices() (*Compiler.types.Matrix method*), 41
assign_vector_by_indices() (*Compiler.types.MultiArray method*), 49
atan() (*in module Compiler.mpc_math*), 113

B

backward() (*Compiler.ml.Adam method*), 116
backward() (*Compiler.ml.Optimizer method*), 120
backward() (*Compiler.ml.SGD method*), 122
BatchNorm (*class in Compiler.ml*), 118
bigint (*C++ class*), 220
bigint::bigint (*C++ function*), 220
bigint::from_signed (*C++ function*), 220
binary_output() (*Compiler.types.Array method*), 37
binary_output() (*Compiler.types.cfix method*), 56
binary_output() (*Compiler.types.cfloat method*), 57
binary_output() (*Compiler.types.cgf2n method*), 58
binary_output() (*Compiler.types.cint method*), 61
binary_output() (*Compiler.types.personal method*), 65
binary_output() (*Compiler.types.regint method*), 67
BinaryCiscable (*class in Compiler.GC.instructions*), 197
BinaryProtocolSet (*C++ class*), 213
BinaryProtocolSet::BinaryProtocolSet (*C++ function*), 214
BinaryProtocolSet::check (*C++ function*), 214
BinaryProtocolSetup (*C++ class*), 213
BinaryProtocolSetup::BinaryProtocolSetup (*C++ function*), 213
bit (*class in Compiler.instructions*), 163
bit_adder() (*Compiler.GC.types.sbit static method*), 94
bit_adder() (*Compiler.GC.types.sbits static method*), 101
bit_adder() (*Compiler.types.cint static method*), 61
bit_adder() (*Compiler.types.regint static method*), 67
bit_adder() (*Compiler.types.sint static method*), 84
bit_and() (*Compiler.GC.types.cbits method*), 93
bit_and() (*Compiler.GC.types.sbit method*), 94
bit_and() (*Compiler.GC.types.sbitintvec method*), 99
bit_and() (*Compiler.GC.types.sbits method*), 101
bit_and() (*Compiler.GC.types.sbitvec method*), 103
bit_and() (*Compiler.types.cgf2n method*), 58
bit_and() (*Compiler.types.cint method*), 61
bit_and() (*Compiler.types.regint method*), 67
bit_and() (*Compiler.types.sgf2n method*), 78
bit_and() (*Compiler.types.sint method*), 84
bit_compose() (*Compiler.types.cgf2n class method*), 58
bit_compose() (*Compiler.types.cint class method*), 61
bit_compose() (*Compiler.types.regint static method*), 67
bit_compose() (*Compiler.types.sgf2n class method*), 79
bit_compose() (*Compiler.types.sint class method*), 84
bit_decompose() (*Compiler.GC.types.sbitfixvec method*), 96
bit_decompose() (*Compiler.types.cgf2n method*), 58
bit_decompose() (*Compiler.types.cint method*), 62
bit_decompose() (*Compiler.types.personal method*), 66
bit_decompose() (*Compiler.types.regint method*), 67
bit_decompose() (*Compiler.types.sfix method*), 71
bit_decompose() (*Compiler.types.sgf2n method*), 79
bit_decompose() (*Compiler.types.sint method*), 84
bit_not() (*Compiler.GC.types.cbits method*), 93
bit_not() (*Compiler.GC.types.sbit method*), 94
bit_not() (*Compiler.GC.types.sbitintvec method*), 99
bit_not() (*Compiler.GC.types.sbits method*), 102
bit_not() (*Compiler.GC.types.sbitvec method*), 104
bit_not() (*Compiler.types.cgf2n method*), 58
bit_not() (*Compiler.types.cint method*), 62
bit_not() (*Compiler.types.regint method*), 67
bit_not() (*Compiler.types.sgf2n method*), 79
bit_not() (*Compiler.types.sint method*), 85
bit_or() (*Compiler.GC.types.cbits method*), 93
bit_or() (*Compiler.GC.types.sbit method*), 94
bit_or() (*Compiler.GC.types.sbitintvec method*), 100
bit_or() (*Compiler.GC.types.sbits method*), 102
bit_or() (*Compiler.GC.types.sbitvec method*), 104
bit_or() (*Compiler.types.cgf2n method*), 58
bit_or() (*Compiler.types.cint method*), 62
bit_or() (*Compiler.types.regint method*), 67
bit_or() (*Compiler.types.sgf2n method*), 79

bit_or() (*Compiler.types.sint method*), 85
bit_xor() (*Compiler.GC.types.cbits method*), 93
bit_xor() (*Compiler.GC.types.sbit method*), 94
bit_xor() (*Compiler.GC.types.sbitintvec method*), 100
bit_xor() (*Compiler.GC.types.sbits method*), 102
bit_xor() (*Compiler.GC.types.sbitvec method*), 104
bit_xor() (*Compiler.types.cgf2n method*), 58
bit_xor() (*Compiler.types.cint method*), 62
bit_xor() (*Compiler.types.regint method*), 68
bit_xor() (*Compiler.types.sgf2n method*), 79
bit_xor() (*Compiler.types.sint method*), 85
bitb (*class in Compiler.GC.instructions*), 199
bitcoms (*class in Compiler.GC.instructions*), 199
bitdecc (*class in Compiler.GC.instructions*), 199
bitdecint (*class in Compiler.instructions*), 163
bitdecs (*class in Compiler.GC.instructions*), 199
break_loop() (*in module Compiler.library*), 105
break_point() (*in module Compiler.library*), 105
BufferPrep (*C++ class*), 217
BufferPrep::basic_setup (*C++ function*), 217
BufferPrep::get_edabitvec (*C++ function*), 217
BufferPrep::get_random (*C++ function*), 217
BufferPrep::setup (*C++ function*), 217
BufferPrep::teardown (*C++ function*), 217

C

call_arg (*class in Compiler.instructions*), 163
call_tape (*class in Compiler.instructions*), 163
cbits (*class in Compiler.GC.types*), 93
CellSparseMatrix (*class in Compiler.sparse*), 140
cfix (*class in Compiler.types*), 55
cfloat (*class in Compiler.types*), 57
cgf2n (*class in Compiler.types*), 57
check (*class in Compiler.instructions*), 164
check_point() (*in module Compiler.library*), 105
cholesky() (*in module Compiler.ml*), 126
cint (*class in Compiler.types*), 60
Ciphertext (*C++ class*), 281
Ciphertext::mul (*C++ function*), 281
Ciphertext::pack (*C++ function*), 281
Ciphertext::rerandomize (*C++ function*), 281
Ciphertext::unpack (*C++ function*), 281
Circuit (*class in Compiler.circuit*), 129
cisc (*class in Compiler.instructions*), 164
Client (*C++ class*), 256
Client (*class in ExternalIO.client*), 255
Client::Client (*C++ function*), 256
Client::receive_outputs (*C++ function*), 256
Client::send_private_inputs (*C++ function*), 256
Client::sockets (*C++ member*), 256
Client::specification (*C++ member*), 256
closeclientconnection (*class in Compiler.instructions*), 166
cmdlinearg (*class in Compiler.instructions*), 166
command line option
    -B, 28, 31
    -C, 29
    -D, 28, 34
    -E, 28, 32
    -F, 27, 32
    -G, 28
    -I, 34
    -IF, 34
    -N, 33
    -OF, 34
    -P, 27, 32
    -Q, 31
    -R, 28, 32
    -S, 32
    -SP, 32
    -X, 28
    -Y, 28
    -Z, 28
    --CISC, 29
    --batch-size, 31
    --binary, 28
    --bits-from-squares, 31
    --bucket-size, 31
    --budget, 29
    --code-locations, 34
    --dead-code-elimination, 28
    --direct, 31
    --disk-memory, 34
    --edabit, 28
    --encrypted, 33
    --execute, 28
    --external-server, 33
    --field, 27
    --file-prep-per-thread, 32
    --file-preprocessing, 32
    --flow-optimization, 29
    --garbled-circuit, 28
    --hostname, 33
    --input-file, 34
    --interactive, 34
    --ip-file-name, 33
    --lg2, 32
    --lgp, 32
    --memory, 34
    --mixed, 28
    --my-port, 33
    --nparties, 33
    --options, 34
    --output-file, 34
    --player, 33
    --portnumbase, 33
    --prime, 27, 32
    --ring, 28, 32

--security, 32
--spdz2k-security, 32
--split, 28
--trunc-error, 32
--unencrypted, 33
--verbose, 34
-b, 29, 31
-d, 31
-e, 33
-ext-server, 33
-f, 32
-h, 33
-ip, 33
-l, 29
-lg2, 32
-lgp, 32
-m, 34
-mp, 33
-o, 34
-p, 33
-pn, 33
-u, 33
-v, 34
Compiler.circuit
module, 129
Compiler.decision_tree
module, 128
Compiler.dijkstra
module, 138
Compiler.GC.instructions
module, 197
Compiler.GC.types
module, 93
Compiler.instructions
module, 160
Compiler.library
module, 105
Compiler.ml
module, 115
Compiler.mpc_math
module, 113
Compiler.oram
module, 133
Compiler.path_oblivious_heap
module, 135
Compiler.program
module, 131
Compiler.sorting
module, 139
Compiler.sparse
module, 140
Compiler.sqrt_oram
module, 135
Compiler.types
module, 35
compute_reciprocal() (*Compiler.GC.types.sbitfixvec method*), 96
compute_reciprocal() (*Compiler.types.sfix method*), 71
Concat (*class in Compiler.ml*), 118
concat() (*Compiler.types.Array method*), 37
concat() (*Compiler.types.Matrix method*), 42
concat() (*Compiler.types.MultiArray method*), 50
concat_columns() (*Compiler.types.Matrix method*), 42
concats (*class in Compiler.instructions*), 166
concatsb (*class in Compiler.GC.instructions*), 200
cond_print_plain (*class in Compiler.instructions*), 167
cond_print_str (*class in Compiler.instructions*), 167
cond_print_strb (*class in Compiler.GC.instructions*), 200
cond_swap() (*Compiler.types.cgf2n method*), 59
cond_swap() (*Compiler.types.cint method*), 62
cond_swap() (*Compiler.types.regint method*), 68
cond_swap() (*Compiler.types.sgf2n method*), 79
cond_swap() (*Compiler.types.sint method*), 85
conv2ds (*class in Compiler.instructions*), 167
convcbit (*class in Compiler.GC.instructions*), 200
convcbit2s (*class in Compiler.GC.instructions*), 200
convcbitvec (*class in Compiler.GC.instructions*), 200
convcint (*class in Compiler.GC.instructions*), 200
convcintvec (*class in Compiler.GC.instructions*), 201
convert_graph() (*in module Compiler.dijkstra*), 138
convint (*class in Compiler.instructions*), 168
convmodp (*class in Compiler.instructions*), 168
convsint (*class in Compiler.GC.instructions*), 201
cos() (*in module Compiler.mpc_math*), 113
crash (*class in Compiler.instructions*), 168
crash() (*in module Compiler.library*), 105
create_from() (*Compiler.types.Array class method*), 37
create_from() (*Compiler.types.Matrix static method*), 42
CryptoPlayer (*C++ class*), 247
CryptoPlayer::CryptoPlayer (*C++ function*), 248

D

dabit (*class in Compiler.instructions*), 168
decompose_demux() (*in module Compiler.oram*), 134
delshuffle (*class in Compiler.instructions*), 169
delshuffleb (*class in Compiler.GC.instructions*), 201
demux_matrix() (*in module Compiler.oram*), 134
Dense (*class in Compiler.ml*), 118
diag() (*Compiler.types.Matrix method*), 42
diag() (*Compiler.types.MultiArray method*), 50
digest() (*Compiler.types.cint method*), 62
digestc (*class in Compiler.instructions*), 169
dijkstra() (*in module Compiler.dijkstra*), 138
direct (*Compiler.GC.instructions.ldmcbi attribute*), 203
direct (*Compiler.GC.instructions.ldmsbi attribute*), 203

direct (*Compiler.GC.instructions.stmcbi attribute*), 206
direct (*Compiler.GC.instructions.stmsbi attribute*), 206
direct (*Compiler.instructions.ldmci attribute*), 176
direct (*Compiler.instructions.ldminti attribute*), 177
direct (*Compiler.instructions.ldmsi attribute*), 177
direct (*Compiler.instructions.stmci attribute*), 190
direct (*Compiler.instructions.stminti attribute*), 191
direct (*Compiler.instructions.stmsi attribute*), 191
direct_mul() (*Compiler.types.Matrix method*), 42
direct_mul() (*Compiler.types.MultiArray method*), 50
direct_mul_trans() (*Compiler.types.Matrix method*), 42
direct_mul_trans() (*Compiler.types.MultiArray method*), 50
direct_trans_mul() (*Compiler.types.Matrix method*), 42
direct_trans_mul() (*Compiler.types.MultiArray method*), 50
divc (*class in Compiler.instructions*), 169
divci (*class in Compiler.instructions*), 169
divint (*class in Compiler.instructions*), 169
do_while() (*in module Compiler.library*), 105
dot() (*Compiler.types.Array method*), 37
dot() (*Compiler.types.Matrix method*), 43
dot() (*Compiler.types.MultiArray method*), 51
dot() (*Compiler.types.sfix method*), 72
dot_product() (*Compiler.types.sfix class method*), 72
dot_product() (*Compiler.types.sgf2n class method*), 79
dot_product() (*Compiler.types.sint class method*), 85
dotprods (*class in Compiler.instructions*), 169
Dropout (*class in Compiler.ml*), 118

E

easyConv2d() (*in module Compiler.ml*), 126
easyMaxPool() (*in module Compiler.ml*), 126
edabit (*class in Compiler.instructions*), 170
eqc (*class in Compiler.instructions*), 170
equal() (*Compiler.GC.types.sbitfixvec method*), 96
equal() (*Compiler.GC.types.sbitintvec method*), 100
equal() (*Compiler.GC.types.sbitvec method*), 104
equal() (*Compiler.types.sfix method*), 72
equal() (*Compiler.types.sgf2n method*), 79
equal() (*Compiler.types.sint method*), 85
eqzc (*class in Compiler.instructions*), 171
eval() (*Compiler.ml.Adam method*), 116
eval() (*Compiler.ml.Optimizer method*), 120
eval() (*Compiler.ml.SGD method*), 122
exp2_fx() (*in module Compiler.mpc_math*), 113
expand_to_vector() (*Compiler.types.Array method*), 37
extract_min() (*Compiler.path_oblivious_heap.PathObliviousHeap method*), 137
extract_min() (*Compiler.path_oblivious_heap.POHToHeapQAdapter method*), 136
extract_min() (*Compiler.path_oblivious_heap.UniquePathObliviousHeap method*), 138
extract_min() (*Compiler.path_oblivious_heap.UniquePOHToHeapQAdapter method*), 137

F

FHE_KeyPair (*C++ class*), 279
FHE_KeyPair::FHE_KeyPair (*C++ function*), 279
FHE_KeyPair::generate (*C++ function*), 279
FHE_KeyPair::pk (*C++ member*), 280
FHE_KeyPair::sk (*C++ member*), 280
FHE_Params (*C++ class*), 279
FHE_Params::basic_generation_mod_prime (*C++ function*), 279
FHE_Params::FHE_Params (*C++ function*), 279
FHE_Params::pack (*C++ function*), 279
FHE_Params::unpack (*C++ function*), 279
FHE_PK (*C++ class*), 280
FHE_PK::encrypt (*C++ function*), 280
FHE_PK::pack (*C++ function*), 280
FHE_PK::unpack (*C++ function*), 280
FHE_SK (*C++ class*), 280
FHE_SK::decrypt (*C++ function*), 280
FHE_SK::pack (*C++ function*), 280
FHE_SK::unpack (*C++ function*), 280
field_div() (*Compiler.types.cgf2n method*), 59
field_div() (*Compiler.types.cint method*), 62
field_div() (*Compiler.types.sgf2n method*), 80
field_div() (*Compiler.types.sint method*), 85
file_base (*class in Compiler.instructions*), 171
find_cycles() (*Compiler.sparse.SparseMatrix method*), 141
find_min() (*Compiler.path_oblivious_heap.PathObliviousHeap method*), 137
find_min() (*Compiler.path_oblivious_heap.POHToHeapQAdapter method*), 136
find_min() (*Compiler.path_oblivious_heap.UniquePathObliviousHeap method*), 138
find_min() (*Compiler.path_oblivious_heap.UniquePOHToHeapQAdapter method*), 137
fit() (*Compiler.decision_tree.TreeClassifier method*), 128
fit() (*Compiler.ml.Adam method*), 116
fit() (*Compiler.ml.Optimizer method*), 120
fit() (*Compiler.ml.SGD method*), 122
fit() (*Compiler.ml.SGDLinear method*), 124
fit() (*Compiler.ml.SGDLogistic method*), 125
fit_with_testing() (*Compiler.decision_tree.TreeClassifier method*),

128
fit_with_testing() (*Compiler.ml.SGDLinear method*), 124
fit_with_testing() (*Compiler.ml.SGDLogistic method*), 125
FixAveragePool2d (*class in Compiler.ml*), 118
FixConv2d (*class in Compiler.ml*), 119
fixinput (*class in Compiler.instructions*), 171
floatoutput (*class in Compiler.instructions*), 171
floordivc (*class in Compiler.instructions*), 171
for_range() (*in module Compiler.library*), 105
for_range_multithread() (*in module Compiler.library*), 106
for_range_opt() (*in module Compiler.library*), 106
for_range_opt_multithread() (*in module Compiler.library*), 106
for_range_parallel() (*in module Compiler.library*), 107
foreach_enumerate() (*in module Compiler.library*), 107
forward() (*Compiler.ml.Adam method*), 117
forward() (*Compiler.ml.Optimizer method*), 121
forward() (*Compiler.ml.SGD method*), 123
from_binary() (*Compiler.types.Array method*), 37
from_binary() (*Compiler.types.Matrix method*), 43
from_binary() (*Compiler.types.MemValue method*), 48
from_binary() (*Compiler.types.MultiArray method*), 51
from_graph() (*Compiler.sparse.SparseMatrix class method*), 141
from_hex() (*Compiler.GC.types.sbitintvec class method*), 100
from_hex() (*Compiler.GC.types.sbitvec class method*), 104
function() (*in module Compiler.library*), 108
FunctionArgument (*C++ class*), 225
FunctionArgument::FunctionArgument (*C++ function*), 225
FusedBatchNorm (*class in Compiler.ml*), 119

G

Gelu (*class in Compiler.ml*), 119
gen_multi_bit_perm() (*in module Compiler.sorting*), 139
gensecshuffle (*class in Compiler.instructions*), 171
gensecshuffleb (*class in Compiler.GC.instructions*), 201
get() (*Compiler.types.Array method*), 37
get_arg() (*in module Compiler.library*), 108
get_cmdline_arg() (*in module Compiler.library*), 108
get_column() (*Compiler.types.Matrix method*), 43
get_column() (*Compiler.types.MultiArray method*), 51
get_dabit() (*Compiler.types.sint class method*), 86
get_def() (*Compiler.instructions.call_tape method*), 164
get_def() (*Compiler.instructions.dotprods method*), 170
get_def() (*Compiler.instructions.mulrs method*), 181
get_edabit() (*Compiler.types.sint class method*), 86
get_input_from() (*Compiler.GC.types.sbit class method*), 95
get_input_from() (*Compiler.GC.types.sbitfixvec class method*), 97
get_input_from() (*Compiler.GC.types.sbits class method*), 102
get_input_from() (*Compiler.types.sfix class method*), 72
get_input_from() (*Compiler.types.sfloat class method*), 77
get_input_from() (*Compiler.types.sgf2n class method*), 80
get_input_from() (*Compiler.types.sint class method*), 86
get_number_of_players() (*in module Compiler.library*), 108
get_part() (*Compiler.types.Array method*), 37
get_part() (*Compiler.types.Matrix method*), 43
get_part() (*Compiler.types.MultiArray method*), 51
get_part_vector() (*Compiler.types.Array method*), 37
get_part_vector() (*Compiler.types.Matrix method*), 43
get_part_vector() (*Compiler.types.MultiArray method*), 51
get_player_id() (*in module Compiler.library*), 108
get_random() (*Compiler.GC.types.sbitfixvec class method*), 97
get_random() (*Compiler.types.regint class method*), 68
get_random() (*Compiler.types.sfix class method*), 72
get_random() (*Compiler.types.sint class method*), 86
get_random_bit() (*Compiler.types.sgf2n class method*), 80
get_random_bit() (*Compiler.types.sint class method*), 86
get_random_input_mask_for() (*Compiler.types.sgf2n class method*), 80
get_random_input_mask_for() (*Compiler.types.sint class method*), 86
get_random_int() (*Compiler.types.sint class method*), 86
get_random_inverse() (*Compiler.types.sgf2n class method*), 80
get_random_inverse() (*Compiler.types.sint class method*), 86
get_random_square() (*Compiler.types.sgf2n class method*), 80
get_random_square() (*Compiler.types.sint class method*), 86
get_random_triple() (*Compiler.types.sgf2n class method*), 80

get_random_triple() (*Compiler.types.sint class method*), 87
get_reverse_vector() (*Compiler.types.Array method*), 38
get_slice_vector() (*Compiler.types.Matrix method*), 43
get_slice_vector() (*Compiler.types.MultiArray method*), 51
get_thread_number() (*in module Compiler.library*), 108
get_threshold() (*in module Compiler.library*), 108
get_tight_row_matrix() (*Compiler.sparse.CellSparseMatrix method*), 140
get_type() (*Compiler.GC.types.cbits class method*), 93
get_type() (*Compiler.GC.types.sbit class method*), 95
get_type() (*Compiler.GC.types.sbitintvec class method*), 100
get_type() (*Compiler.GC.types.sbits class method*), 102
get_type() (*Compiler.GC.types.sbitvec class method*), 104
get_used() (*Compiler.instructions.call_tape method*), 164
get_used() (*Compiler.instructions.dotprods method*), 170
get_used() (*Compiler.instructions.mulrs method*), 181
get_vector() (*Compiler.types.Array method*), 38
get_vector() (*Compiler.types.Matrix method*), 44
get_vector() (*Compiler.types.MultiArray method*), 51
get_vector_by_indices() (*Compiler.types.Matrix method*), 44
get_vector_by_indices() (*Compiler.types.MultiArray method*), 52
gfp_ (*C++ class*), 218
gfp_::gfp_ (*C++ function*), 218
gfp_::init_default (*C++ function*), 219
gfp_::init_field (*C++ function*), 219
gfp_::operator>> (*C++ function*), 219
gfp_::operator<< (*C++ function*), 219
gfp_::output (*C++ function*), 218
gfp_::pack (*C++ function*), 218
gfp_::pr (*C++ function*), 219
gfp_::randomize (*C++ function*), 218
gfp_::sqrRoot (*C++ function*), 218
gfp_::unpack (*C++ function*), 218
gfpvar_ (*C++ class*), 219
GlobalPRNG (*C++ class*), 274
greater_equal() (*Compiler.GC.types.sbitfixvec method*), 97
greater_equal() (*Compiler.types.sfix method*), 72
greater_equal() (*Compiler.types.sint method*), 87
greater_than() (*Compiler.GC.types.sbitfixvec method*), 97
greater_than() (*Compiler.types.sfix method*), 72
greater_than() (*Compiler.types.sint method*), 87
GroupOr() (*in module Compiler.sparse*), 140
groupsumint (*class in Compiler.instructions*), 172
gtc (*class in Compiler.instructions*), 172

H

half_adder() (*Compiler.GC.types.cbits method*), 93
half_adder() (*Compiler.GC.types.sbit method*), 95
half_adder() (*Compiler.GC.types.sbitintvec method*), 100
half_adder() (*Compiler.GC.types.sbits method*), 102
half_adder() (*Compiler.GC.types.sbitvec method*), 104
half_adder() (*Compiler.types.cgf2n method*), 59
half_adder() (*Compiler.types.cint method*), 62
half_adder() (*Compiler.types.regint method*), 68
half_adder() (*Compiler.types.sgf2n method*), 81
half_adder() (*Compiler.types.sint method*), 87

I

iadd() (*Compiler.GC.types.sbitfixvec method*), 97
iadd() (*Compiler.GC.types.sbitintvec method*), 100
iadd() (*Compiler.types.cfix method*), 56
iadd() (*Compiler.types.cgf2n method*), 59
iadd() (*Compiler.types.cint method*), 62
iadd() (*Compiler.types.Matrix method*), 44
iadd() (*Compiler.types.MemValue method*), 48
iadd() (*Compiler.types.MultiArray method*), 52
iadd() (*Compiler.types.regint method*), 68
iadd() (*Compiler.types.sfix method*), 73
iadd() (*Compiler.types.sfloat method*), 77
iadd() (*Compiler.types.sgf2n method*), 81
iadd() (*Compiler.types.sint method*), 87
ieee_float (*class in Compiler.circuit*), 130
if_() (*in module Compiler.library*), 108
if_e() (*in module Compiler.library*), 109
if_else() (*Compiler.GC.types.cbits method*), 93
if_else() (*Compiler.GC.types.sbit method*), 95
if_else() (*Compiler.GC.types.sbits method*), 102
if_else() (*Compiler.types.cgf2n method*), 59
if_else() (*Compiler.types.cint method*), 63
if_else() (*Compiler.types.regint method*), 68
if_else() (*Compiler.types.sgf2n method*), 81
if_else() (*Compiler.types.sint method*), 87
inc() (*Compiler.types.regint class method*), 69
incint (*class in Compiler.instructions*), 172
index_structure (*Compiler.oram.RecursiveORAM attribute*), 134
init_client_connection() (*in module Compiler.library*), 109
initclientconnection (*class in Compiler.instructions*), 172
input_from() (*Compiler.types.Array method*), 38
input_from() (*Compiler.types.Matrix method*), 44
input_from() (*Compiler.types.MultiArray method*), 52

input_tensor_from() (*Compiler.types.sfix class method*), 73
input_tensor_from() (*Compiler.types.sfloat class method*), 77
input_tensor_from() (*Compiler.types.sgf2n class method*), 81
input_tensor_from() (*Compiler.types.sint class method*), 88
input_tensor_from_client() (*Compiler.types.sfix class method*), 73
input_tensor_from_client() (*Compiler.types.sfloat class method*), 77
input_tensor_from_client() (*Compiler.types.sgf2n class method*), 81
input_tensor_from_client() (*Compiler.types.sint class method*), 88
input_tensor_via() (*Compiler.types.sfix class method*), 73
input_tensor_via() (*Compiler.types.sfloat class method*), 77
input_tensor_via() (*Compiler.types.sgf2n class method*), 81
input_tensor_via() (*Compiler.types.sint class method*), 88
inputb (*class in Compiler.GC.instructions*), 201
InputBase (*C++ class*), 215
InputBase::add_from_all (*C++ function*), 216
InputBase::add_mine (*C++ function*), 215
InputBase::add_other (*C++ function*), 216
InputBase::exchange (*C++ function*), 216
InputBase::finalize (*C++ function*), 216
InputBase::reset (*C++ function*), 215
InputBase::reset_all (*C++ function*), 215
inputbvec (*class in Compiler.GC.instructions*), 202
inputfix (*class in Compiler.instructions*), 173
inputfloat (*class in Compiler.instructions*), 173
inputmask (*class in Compiler.instructions*), 173
inputmaskreg (*class in Compiler.instructions*), 173
inputmixed (*class in Compiler.instructions*), 173
inputmixedreg (*class in Compiler.instructions*), 174
inputpersonal (*class in Compiler.instructions*), 174
insert() (*Compiler.path_oblivious_heap.PathObliviousHeap method*), 137
insert() (*Compiler.path_oblivious_heap.POHToHeapQAdapter method*), 136
insert() (*Compiler.path_oblivious_heap.UniquePathObliviousHeap method*), 138
insert() (*Compiler.path_oblivious_heap.UniquePOHToHeapQAdapter method*), 138
int_div() (*Compiler.types.sint method*), 88
int_mod() (*Compiler.types.sint method*), 88
intoutput (*class in Compiler.instructions*), 174
inv2m (*class in Compiler.instructions*), 175
inverse (*class in Compiler.instructions*), 175
inverse_permutation (*class in Compiler.instructions*), 175
InvertSqrt() (*in module Compiler.mpc_math*), 114

J

jmp (*class in Compiler.instructions*), 175
jmpeqz (*class in Compiler.instructions*), 175
jmpi (*class in Compiler.instructions*), 175
jmpnz (*class in Compiler.instructions*), 175
join_tape (*class in Compiler.instructions*), 176
join_tapes() (*Compiler.program.Program method*), 132

L

LayerNorm (*class in Compiler.ml*), 119
layers (*Compiler.ml.Adam property*), 117
layers (*Compiler.ml.Optimizer property*), 121
layers (*Compiler.ml.SGD property*), 123
layers_from_torch() (*in module Compiler.ml*), 126
ldarg (*class in Compiler.instructions*), 176
ldbits (*class in Compiler.GC.instructions*), 202
ldi (*class in Compiler.instructions*), 176
ldint (*class in Compiler.instructions*), 176
ldmc (*class in Compiler.instructions*), 176
ldmcb (*class in Compiler.GC.instructions*), 202
ldmcbi (*class in Compiler.GC.instructions*), 202
ldmci (*class in Compiler.instructions*), 176
ldmint (*class in Compiler.instructions*), 176
ldminti (*class in Compiler.instructions*), 177
ldms (*class in Compiler.instructions*), 177
ldmsb (*class in Compiler.GC.instructions*), 203
ldmsbi (*class in Compiler.GC.instructions*), 203
ldmsi (*class in Compiler.instructions*), 177
ldsi (*class in Compiler.instructions*), 177
ldtn (*class in Compiler.instructions*), 177
left_shift() (*Compiler.types.sint method*), 89
legendre() (*Compiler.types.cint method*), 63
legendrec (*class in Compiler.instructions*), 177
less_equal() (*Compiler.GC.types.sbitfixvec method*), 97
less_equal() (*Compiler.types.sfix method*), 73
less_equal() (*Compiler.types.sint method*), 89
less_than() (*Compiler.GC.types.sbitfixvec method*), 97
less_than() (*Compiler.types.cfix method*), 56
less_than() (*Compiler.types.cint method*), 63
less_than() (*Compiler.types.sfix method*), 74
less_than() (*Compiler.types.sint method*), 89
LinearORAM (*class in Compiler.oram*), 134
listen (*class in Compiler.instructions*), 178
listen_for_clients() (*in module Compiler.library*), 109
load_mem() (*Compiler.types.cfix class method*), 56
load_mem() (*Compiler.types.cgf2n class method*), 59
load_mem() (*Compiler.types.cint class method*), 63

load_mem() (*Compiler.types.regint class method*), 69
load_mem() (*Compiler.types.sfix class method*), 74
load_mem() (*Compiler.types.sfloat class method*), 77
load_mem() (*Compiler.types.sgf2n class method*), 82
load_mem() (*Compiler.types.sint class method*), 89
localint (*class in Compiler.types*), 65
log2_fx() (*in module Compiler.mpc_math*), 114
log_fx() (*in module Compiler.mpc_math*), 114
ltc (*class in Compiler.instructions*), 178
ltzc (*class in Compiler.instructions*), 178

M

MAC_Check_Base (*C++ class*), 216
MAC_Check_Base::Check (*C++ function*), 216
MAC_Check_Base::CheckFor (*C++ function*), 216
MAC_Check_Base::exchange (*C++ function*), 216
MAC_Check_Base::finalize_open (*C++ function*), 216
MAC_Check_Base::get_alphai (*C++ function*), 216
MAC_Check_Base::init_open (*C++ function*), 216
MAC_Check_Base::open (*C++ function*), 216
MAC_Check_Base::POpen (*C++ function*), 216
MAC_Check_Base::prepare_open (*C++ function*), 216
malloc() (*Compiler.types.cgf2n class method*), 59
malloc() (*Compiler.types.cint class method*), 63
malloc() (*Compiler.types.regint class method*), 69
malloc() (*Compiler.types.sgf2n class method*), 82
malloc() (*Compiler.types.sint class method*), 89
map_sum_opt() (*in module Compiler.library*), 109
map_sum_simple() (*in module Compiler.library*), 110
matmuls (*class in Compiler.instructions*), 178
matmulsm (*class in Compiler.instructions*), 178
Matrix (*class in Compiler.types*), 41
Matrix() (*Compiler.types.cfix class method*), 56
Matrix() (*Compiler.types.cgf2n class method*), 58
Matrix() (*Compiler.types.cint class method*), 61
Matrix() (*Compiler.types.regint class method*), 66
Matrix() (*Compiler.types.sfix class method*), 71
Matrix() (*Compiler.types.sfloat class method*), 76
Matrix() (*Compiler.types.sgf2n class method*), 78
Matrix() (*Compiler.types.sint class method*), 84
max() (*Compiler.GC.types.sbitfixvec method*), 98
max() (*Compiler.GC.types.sbitintvec method*), 100
max() (*Compiler.types.cfix method*), 56
max() (*Compiler.types.cgf2n method*), 59
max() (*Compiler.types.cint method*), 63
max() (*Compiler.types.MemValue method*), 48
max() (*Compiler.types.regint method*), 69
max() (*Compiler.types.sfix method*), 74
max() (*Compiler.types.sfloat method*), 77
max() (*Compiler.types.sgf2n method*), 82
max() (*Compiler.types.sint method*), 89
MaxPool (*class in Compiler.ml*), 120
maybe_get() (*Compiler.types.Array method*), 38
maybe_set() (*Compiler.types.Array method*), 38
MemValue (*class in Compiler.types*), 47
MemValue() (*Compiler.GC.types.cbits class method*), 93
MemValue() (*Compiler.GC.types.sbit class method*), 94
MemValue() (*Compiler.GC.types.sbitfixvec class method*), 96
MemValue() (*Compiler.GC.types.sbits class method*), 101
MemValue() (*Compiler.types.cfix class method*), 56
MemValue() (*Compiler.types.cgf2n class method*), 58
MemValue() (*Compiler.types.cint class method*), 61
MemValue() (*Compiler.types.regint class method*), 66
MemValue() (*Compiler.types.sfix class method*), 71
MemValue() (*Compiler.types.sfloat class method*), 76
MemValue() (*Compiler.types.sgf2n class method*), 78
MemValue() (*Compiler.types.sint class method*), 84
min() (*Compiler.GC.types.sbitfixvec method*), 98
min() (*Compiler.GC.types.sbitintvec method*), 100
min() (*Compiler.types.cfix method*), 56
min() (*Compiler.types.cgf2n method*), 59
min() (*Compiler.types.cint method*), 63
min() (*Compiler.types.MemValue method*), 48
min() (*Compiler.types.regint method*), 69
min() (*Compiler.types.sfix method*), 74
min() (*Compiler.types.sfloat method*), 77
min() (*Compiler.types.sgf2n method*), 82
min() (*Compiler.types.sint method*), 89
MixedProtocolSet (*C++ class*), 214
MixedProtocolSet::check (*C++ function*), 214
MixedProtocolSet::MixedProtocolSet (*C++ function*), 214
MixedProtocolSetup (*C++ class*), 214
MixedProtocolSetup::MixedProtocolSetup (*C++ function*), 214
mod2m() (*Compiler.types.cint method*), 63
mod2m() (*Compiler.types.regint method*), 69
mod2m() (*Compiler.types.sint method*), 89
modc (*class in Compiler.instructions*), 179
modci (*class in Compiler.instructions*), 179
module
    Compiler.circuit, 129
    Compiler.decision_tree, 128
    Compiler.dijkstra, 138
    Compiler.GC.instructions, 197
    Compiler.GC.types, 93
    Compiler.instructions, 160
    Compiler.library, 105
    Compiler.ml, 115
    Compiler.mpc_math, 113
    Compiler.oram, 133
    Compiler.path_oblivious_heap, 135
    Compiler.program, 131
    Compiler.sorting, 139
    Compiler.sparse, 140

Compiler.sqrt_oram, 135
Compiler.types, 35
movc (*class in Compiler.instructions*), 179
movint (*class in Compiler.instructions*), 180
movs (*class in Compiler.instructions*), 180
movsb (*class in Compiler.GC.instructions*), 203
mr() (*in module Compiler.ml*), 126
mul_base (*class in Compiler.instructions*), 180
mul_trans() (*Compiler.types.Matrix method*), 44
mul_trans() (*Compiler.types.MultiArray method*), 52
mul_trans_to() (*Compiler.types.Matrix method*), 44
mul_trans_to() (*Compiler.types.MultiArray method*), 52
mul_without_removal() (*Compiler.sparse.SparseMatrix method*), 141
mulbits (*class in Compiler.instructions*), 180
mulc (*class in Compiler.instructions*), 180
mulcbi (*class in Compiler.GC.instructions*), 203
mulci (*class in Compiler.instructions*), 180
mulint (*class in Compiler.instructions*), 181
mulm (*class in Compiler.instructions*), 181
mulrs (*class in Compiler.instructions*), 181
muls (*class in Compiler.instructions*), 181
muls_base (*class in Compiler.instructions*), 182
mulsi (*class in Compiler.instructions*), 182
MultiArray (*class in Compiler.types*), 49
MultiOutput (*class in Compiler.ml*), 120
MultiPlayer (*C++ class*), 247
multithread() (*in module Compiler.library*), 110

N

Names (*C++ class*), 244
Names::~Names (*C++ function*), 245
Names::DEFAULT_PORT (*C++ member*), 245
Names::get_name (*C++ function*), 245
Names::get_portnum_base (*C++ function*), 245
Names::init (*C++ function*), 244
Names::my_num (*C++ function*), 245
Names::Names (*C++ function*), 244, 245
Names::num_players (*C++ function*), 245
new_tape() (*Compiler.program.Program method*), 132
not_equal() (*Compiler.GC.types.sbitfixvec method*), 98
not_equal() (*Compiler.types.sfix method*), 74
not_equal() (*Compiler.types.sgf2n method*), 82
not_equal() (*Compiler.types.sint method*), 90
notc (*class in Compiler.instructions*), 182
notcb (*class in Compiler.GC.instructions*), 203
nots (*class in Compiler.GC.instructions*), 203
nplayers (*class in Compiler.instructions*), 182

O

octetStream (*C++ class*), 248
octetStream::append (*C++ function*), 249
octetStream::append_random (*C++ function*), 249
octetStream::clear (*C++ function*), 248
octetStream::concat (*C++ function*), 249
octetStream::consume (*C++ function*), 249, 250
octetStream::done (*C++ function*), 248
octetStream::empty (*C++ function*), 248
octetStream::exchange (*C++ function*), 251
octetStream::get (*C++ function*), 249, 250
octetStream::get_data (*C++ function*), 248
octetStream::get_data_ptr (*C++ function*), 248
octetStream::get_int (*C++ function*), 249
octetStream::get_length (*C++ function*), 248
octetStream::get_max_length (*C++ function*), 248
octetStream::get_no_resize (*C++ function*), 250
octetStream::get_ptr (*C++ function*), 248
octetStream::get_total_length (*C++ function*), 248
octetStream::hash (*C++ function*), 248
octetStream::input (*C++ function*), 250
octetStream::left (*C++ function*), 248
octetStream::octetStream (*C++ function*), 248
octetStream::operator== (*C++ function*), 249
octetStream::output (*C++ function*), 250
octetStream::Receive (*C++ function*), 250
octetStream::reset_read_head (*C++ function*), 249
octetStream::reset_write_head (*C++ function*), 249
octetStream::resize (*C++ function*), 248
octetStream::Send (*C++ function*), 250
octetStream::store (*C++ function*), 249, 250
octetStream::store_int (*C++ function*), 249
octetStream::str (*C++ function*), 248
op() (*Compiler.instructions.addint method*), 161
op() (*Compiler.instructions.divint method*), 169
op() (*Compiler.instructions.eqc method*), 170
op() (*Compiler.instructions.gtc method*), 172
op() (*Compiler.instructions.ltc method*), 178
op() (*Compiler.instructions.mulint method*), 181
op() (*Compiler.instructions.subint method*), 192
OptimalORAM() (*in module Compiler.oram*), 134
Optimizer (*class in Compiler.ml*), 120
options_from_args() (*Compiler.program.Program method*), 132
orc (*class in Compiler.instructions*), 182
orci (*class in Compiler.instructions*), 183
Output (*class in Compiler.ml*), 121
output() (*Compiler.decision_tree.TreeClassifier method*), 128
output() (*Compiler.types.localint method*), 65
output_decision_tree() (*in module Compiler.decision_tree*), 129

P

path_oblivious_sort() (*in module Compiler.path_oblivious_heap*), 138

PathObliviousHeap (*class in Compiler.path_oblivious_heap*), 137
permute() (*Compiler.types.Array method*), 38
permute() (*Compiler.types.Matrix method*), 44
permute() (*Compiler.types.MultiArray method*), 52
personal (*class in Compiler.types*), 65
personal_base (*class in Compiler.instructions*), 183
picks (*class in Compiler.instructions*), 183
picksb (*class in Compiler.GC.instructions*), 204
PlainPlayer (*C++ class*), 247
PlainPlayer::PlainPlayer (*C++ function*), 247
Plaintext (*C++ class*), 280
Plaintext::element (*C++ function*), 280
Plaintext::num_slots (*C++ function*), 280
Plaintext::pack (*C++ function*), 281
Plaintext::Plaintext (*C++ function*), 280
Plaintext::set_element (*C++ function*), 281
Plaintext::unpack (*C++ function*), 281
Player (*C++ class*), 245
Player::Broadcast_Receive (*C++ function*), 246
Player::Check_Broadcast (*C++ function*), 246
Player::exchange (*C++ function*), 246
Player::exchange_relative (*C++ function*), 246
Player::my_num (*C++ function*), 245
Player::num_players (*C++ function*), 245
Player::partial_broadcast (*C++ function*), 247
Player::pass_around (*C++ function*), 246
Player::receive_all (*C++ function*), 246
Player::receive_player (*C++ function*), 246
Player::receive_relative (*C++ function*), 246
Player::send_all (*C++ function*), 245, 246
Player::send_receive_all (*C++ function*), 246, 247
Player::send_relative (*C++ function*), 246
Player::send_to (*C++ function*), 246
Player::unchecked_broadcast (*C++ function*), 246
playerid (*class in Compiler.instructions*), 183
POHToHeapQAdapter (*class in Compiler.path_oblivious_heap*), 136
pop() (*Compiler.path_oblivious_heap.POHToHeapQAdapter method*), 136
pop() (*Compiler.path_oblivious_heap.UniquePOHToHeapQAdapter method*), 138
pop() (*Compiler.types.regint class method*), 69
popcnt() (*Compiler.GC.types.sbit method*), 95
popcnt() (*Compiler.GC.types.sbitintvec method*), 100
popcnt() (*Compiler.GC.types.sbits method*), 102
popcnt() (*Compiler.GC.types.sbitvec method*), 104
popint (*class in Compiler.instructions*), 183
pow2() (*Compiler.GC.types.sbitintvec method*), 101
pow2() (*Compiler.types.sint method*), 90
pow_fx() (*in module Compiler.mpc_math*), 114
predict() (*Compiler.decision_tree.TreeClassifier method*), 128
predict() (*Compiler.ml.SGDLinear method*), 124
predict() (*Compiler.ml.SGDLogistic method*), 125
predict_proba() (*Compiler.ml.SGDLogistic method*), 125
prefix_sum() (*Compiler.types.sint method*), 90
prefixsums (*class in Compiler.instructions*), 183
prep (*class in Compiler.instructions*), 183
preprocess_pandas() (*in module Compiler.decision_tree*), 129
Preprocessing (*C++ class*), 216
Preprocessing::basic_setup (*C++ function*), 217
Preprocessing::get_bit (*C++ function*), 217
Preprocessing::get_dabit (*C++ function*), 217
Preprocessing::get_edabitvec (*C++ function*), 217
Preprocessing::get_random (*C++ function*), 217
Preprocessing::get_triple (*C++ function*), 217
Preprocessing::setup (*C++ function*), 217
Preprocessing::teardown (*C++ function*), 217
prime_type (*Compiler.ml.Gelu attribute*), 119
prime_type (*Compiler.ml.Relu attribute*), 122
prime_type (*Compiler.ml.Square attribute*), 125
print_both() (*in module Compiler.library*), 111
print_char (*class in Compiler.instructions*), 184
print_char4 (*class in Compiler.instructions*), 184
print_float_plain (*class in Compiler.instructions*), 184
print_float_plain() (*Compiler.types.cfloat method*), 57
print_float_plainb (*class in Compiler.GC.instructions*), 204
print_float_prec (*class in Compiler.instructions*), 184
print_float_precision() (*in module Compiler.library*), 111
print_if() (*Compiler.types.cint method*), 64
print_if() (*Compiler.types.regint method*), 69
print_int (*class in Compiler.instructions*), 184
print_ln() (*in module Compiler.library*), 111
print_ln_if() (*in module Compiler.library*), 111
print_ln_to() (*in module Compiler.library*), 111
print_plain() (*Compiler.types.cfix method*), 56
print_reg (*class in Compiler.instructions*), 184
print_reg_plain (*class in Compiler.instructions*), 184
print_reg_plain() (*Compiler.types.cgf2n method*), 59
print_reg_plain() (*Compiler.types.cint method*), 64
print_reg_plain() (*Compiler.types.regint method*), 69
print_reg_plainb (*class in Compiler.GC.instructions*), 204
print_reg_plains (*class in Compiler.instructions*), 184
print_reg_plainsb (*class in Compiler.GC.instructions*), 204
print_reg_signed (*class in Compiler.GC.instructions*), 204
print_regb (*class in Compiler.GC.instructions*), 205
print_reveal_nested() (*Compiler.types.Array method*), 38

print_reveal_nested() (*Compiler.types.Matrix method*), 45
print_reveal_nested() (*Compiler.types.MultiArray method*), 52
print_str() (*in module Compiler.library*), 111
print_str_if() (*in module Compiler.library*), 112
private_division() (*Compiler.types.sint method*), 90
privateoutput (*class in Compiler.instructions*), 185
PRNG (*C++ class*), 272
PRNG::get (*C++ function*), 274
PRNG::get_bit (*C++ function*), 273
PRNG::get_doubleword (*C++ function*), 273
PRNG::get_octets (*C++ function*), 273
PRNG::get_uchar (*C++ function*), 273
PRNG::get_uint (*C++ function*), 273
PRNG::get_word (*C++ function*), 273
PRNG::PRNG (*C++ function*), 272
PRNG::randomBnd (*C++ function*), 273
PRNG::ReSeed (*C++ function*), 272
PRNG::SeedGlobally (*C++ function*), 272
PRNG::SetSeed (*C++ function*), 273
PRNG::SetSeedFrom (*C++ function*), 273
Program (*class in Compiler.program*), 131
protect_memory() (*Compiler.program.Program method*), 132
ProtocolBase (*C++ class*), 215
ProtocolBase::exchange (*C++ function*), 215
ProtocolBase::finalize_dotprod (*C++ function*), 215
ProtocolBase::finalize_mul (*C++ function*), 215
ProtocolBase::finalize_mult (*C++ function*), 215
ProtocolBase::init (*C++ function*), 215
ProtocolBase::init_dotprod (*C++ function*), 215
ProtocolBase::init_mul (*C++ function*), 215
ProtocolBase::mul (*C++ function*), 215
ProtocolBase::next_dotprod (*C++ function*), 215
ProtocolBase::prepare_dotprod (*C++ function*), 215
ProtocolBase::prepare_mul (*C++ function*), 215
ProtocolSet (*C++ class*), 213
ProtocolSet::check (*C++ function*), 213
ProtocolSet::ProtocolSet (*C++ function*), 213
ProtocolSetup (*C++ class*), 212
ProtocolSetup::ProtocolSetup (*C++ function*), 212
ProtocolSetup::set_batch_size (*C++ function*), 213
pubinput (*class in Compiler.instructions*), 185
public_input() (*Compiler.program.Program method*), 132
public_input() (*in module Compiler.library*), 112
push() (*Compiler.types.regint class method*), 70
pushint (*class in Compiler.instructions*), 185

Q

qs_perm (*class in Compiler.instructions*), 185
quicksort() (*in module Compiler.sorting*), 139

R

radix_sort() (*in module Compiler.sorting*), 139
rand (*class in Compiler.instructions*), 185
randomfulls (*class in Compiler.instructions*), 185
randomize() (*Compiler.types.Array method*), 38
randomize() (*Compiler.types.Matrix method*), 45
randomize() (*Compiler.types.MultiArray method*), 52
randoms (*class in Compiler.instructions*), 186
raw_right_shift() (*Compiler.types.sgf2n method*), 82
raw_right_shift() (*Compiler.types.sint method*), 90
rawinput (*class in Compiler.instructions*), 186
read() (*Compiler.types.MemValue method*), 48
read_fix() (*Compiler.types.personal class method*), 66
read_from_file() (*Compiler.types.Array method*), 39
read_from_file() (*Compiler.types.cgf2n class method*), 59
read_from_file() (*Compiler.types.cint class method*), 64
read_from_file() (*Compiler.types.Matrix method*), 45
read_from_file() (*Compiler.types.MultiArray method*), 53
read_from_file() (*Compiler.types.regint class method*), 70
read_from_file() (*Compiler.types.sfix class method*), 74
read_from_file() (*Compiler.types.sgf2n class method*), 82
read_from_file() (*Compiler.types.sint class method*), 90
read_from_socket() (*Compiler.types.Array method*), 39
read_from_socket() (*Compiler.types.cfix class method*), 56
read_from_socket() (*Compiler.types.cint class method*), 64
read_from_socket() (*Compiler.types.Matrix method*), 45
read_from_socket() (*Compiler.types.MultiArray method*), 53
read_from_socket() (*Compiler.types.regint class method*), 70
read_from_socket() (*Compiler.types.sint class method*), 91
read_int() (*Compiler.types.personal class method*), 66
readfileclear (*class in Compiler.instructions*), 186
readsharesfromfile (*class in Compiler.instructions*), 186
readsocketc (*class in Compiler.instructions*), 187
readsocketint (*class in Compiler.instructions*), 187
readsockets (*class in Compiler.instructions*), 187

receive_from_client() (*Compiler.types.sfix class method*), 74
receive_from_client() (*Compiler.types.sint class method*), 91
receive_outputs() (*ExternalIO.client.Client method*), 255
receive_plain_values() (*ExternalIO.client.Client method*), 255
RecursiveORAM (*class in Compiler.oram*), 134
regint (*class in Compiler.types*), 66
Relu (*class in Compiler.ml*), 122
relu() (*in module Compiler.ml*), 127
relu_prime() (*in module Compiler.ml*), 127
ReluMultiOutput (*class in Compiler.ml*), 122
rep_type (*Compiler.types.sfix attribute*), 75
reqbl (*class in Compiler.instructions*), 187
reset() (*Compiler.ml.Adam method*), 117
reset() (*Compiler.ml.Optimizer method*), 121
reset() (*Compiler.ml.SGD method*), 123
reveal (*class in Compiler.GC.instructions*), 205
reveal() (*Compiler.GC.types.sbitfixvec method*), 98
reveal() (*Compiler.types.Array method*), 39
reveal() (*Compiler.types.cgf2n method*), 60
reveal() (*Compiler.types.cint method*), 64
reveal() (*Compiler.types.Matrix method*), 45
reveal() (*Compiler.types.MemValue method*), 48
reveal() (*Compiler.types.MultiArray method*), 53
reveal() (*Compiler.types.regint method*), 70
reveal() (*Compiler.types.sfix method*), 75
reveal() (*Compiler.types.sfloat method*), 78
reveal() (*Compiler.types.sgf2n method*), 82
reveal() (*Compiler.types.sint method*), 91
reveal_correctness() (*Compiler.ml.Adam method*), 117
reveal_correctness() (*Compiler.ml.Optimizer method*), 121
reveal_correctness() (*Compiler.ml.SGD method*), 123
reveal_list() (*Compiler.types.Array method*), 39
reveal_list() (*Compiler.types.Matrix method*), 45
reveal_list() (*Compiler.types.MultiArray method*), 53
reveal_model_to_binary() (*Compiler.ml.Adam method*), 117
reveal_model_to_binary() (*Compiler.ml.Optimizer method*), 121
reveal_model_to_binary() (*Compiler.ml.SGD method*), 123
reveal_nested() (*Compiler.types.Array method*), 39
reveal_nested() (*Compiler.types.Matrix method*), 45
reveal_nested() (*Compiler.types.MultiArray method*), 53
reveal_print_hex() (*Compiler.GC.types.sbitintvec method*), 101
reveal_print_hex() (*Compiler.GC.types.sbitvec method*), 104
reveal_sort() (*in module Compiler.sorting*), 140
reveal_to() (*Compiler.types.Array method*), 39
reveal_to() (*Compiler.types.personal method*), 66
reveal_to() (*Compiler.types.sfix method*), 75
reveal_to() (*Compiler.types.sgf2n method*), 83
reveal_to() (*Compiler.types.sint method*), 91
reveal_to_binary_output() (*Compiler.types.Array method*), 39
reveal_to_binary_output() (*Compiler.types.Matrix method*), 45
reveal_to_binary_output() (*Compiler.types.MultiArray method*), 53
reveal_to_clients() (*Compiler.types.Array method*), 39
reveal_to_clients() (*Compiler.types.Matrix method*), 45
reveal_to_clients() (*Compiler.types.MemValue method*), 48
reveal_to_clients() (*Compiler.types.MultiArray method*), 53
reveal_to_clients() (*Compiler.types.sfix class method*), 75
reveal_to_clients() (*Compiler.types.sint class method*), 91
reveal_to_socket_by_party() (*Compiler.types.Array method*), 39
reveal_to_socket_by_party() (*Compiler.types.Matrix method*), 45
reveal_to_socket_by_party() (*Compiler.types.MemValue method*), 48
reveal_to_socket_by_party() (*Compiler.types.MultiArray method*), 53
right_shift() (*Compiler.types.cint method*), 64
right_shift() (*Compiler.types.sgf2n method*), 83
right_shift() (*Compiler.types.sint method*), 91
round() (*Compiler.types.sint method*), 91
round_nearest (*Compiler.GC.types.sbitfixvec attribute*), 98
round_nearest (*Compiler.types.sfix attribute*), 75
round_to_int() (*Compiler.types.sfloat method*), 78
run() (*Compiler.ml.Adam method*), 117
run() (*Compiler.ml.Optimizer method*), 121
run() (*Compiler.ml.SGD method*), 123
run_decision_tree() (*in module Compiler.decision_tree*), 129
run_tape (*class in Compiler.instructions*), 188
run_tapes() (*Compiler.program.Program method*), 132
runtime_error() (*in module Compiler.library*), 112
runtime_error_if() (*in module Compiler.library*), 112

S

same_shape() (*Compiler.types.Array method*), 40
same_shape() (*Compiler.types.Matrix method*), 45
same_shape() (*Compiler.types.MultiArray method*), 53
sbit (*class in Compiler.GC.types*), 94
sbitfixvec (*class in Compiler.GC.types*), 95
sbitintvec (*class in Compiler.GC.types*), 99
sbits (*class in Compiler.GC.types*), 101
sbitvec (*class in Compiler.GC.types*), 103
schur() (*Compiler.types.Matrix method*), 46
schur() (*Compiler.types.MultiArray method*), 53
secshuffle (*class in Compiler.instructions*), 188
secure_permute() (*Compiler.types.Array method*), 40
secure_permute() (*Compiler.types.Matrix method*), 46
secure_permute() (*Compiler.types.MultiArray method*), 54
secure_shuffle() (*Compiler.types.Array method*), 40
secure_shuffle() (*Compiler.types.Matrix method*), 46
secure_shuffle() (*Compiler.types.MultiArray method*), 54
security (*Compiler.program.Program property*), 132
sedabit (*class in Compiler.instructions*), 188
SeededPRNG (*C++ class*), 274
send_private_inputs() (*ExternalIO.client.Client method*), 256
send_public_inputs() (*ExternalIO.client.Client method*), 256
sendpersonal (*class in Compiler.instructions*), 188
set_bit_length() (*Compiler.program.Program method*), 133
set_column() (*Compiler.types.Matrix method*), 46
set_column() (*Compiler.types.MultiArray method*), 54
set_layers_with_inputs() (*Compiler.ml.Adam method*), 117
set_layers_with_inputs() (*Compiler.ml.Optimizer method*), 121
set_layers_with_inputs() (*Compiler.ml.SGD method*), 124
set_precision() (*Compiler.GC.types.sbitfixvec class method*), 98
set_precision() (*Compiler.types.cfix class method*), 57
set_precision() (*Compiler.types.sfix class method*), 75
sfix (*class in Compiler.types*), 71
sfloat (*class in Compiler.types*), 76
SGD (*class in Compiler.ml*), 122
SGDLinear (*class in Compiler.ml*), 124
SGDLogistic (*class in Compiler.ml*), 124
sgf2n (*class in Compiler.types*), 78
sha256() (*in module Compiler.circuit*), 130
sha3_256() (*in module Compiler.circuit*), 131
shlc (*class in Compiler.instructions*), 188
shlcbi (*class in Compiler.GC.instructions*), 205
shlci (*class in Compiler.instructions*), 189
shrc (*class in Compiler.instructions*), 189
shrcbi (*class in Compiler.GC.instructions*), 205
shrci (*class in Compiler.instructions*), 189
shrsi (*class in Compiler.instructions*), 189
shuffle (*class in Compiler.instructions*), 189
shuffle() (*Compiler.types.Array method*), 40
shuffle() (*Compiler.types.regint method*), 70
shuffle_base (*class in Compiler.instructions*), 190
sigmoid() (*in module Compiler.ml*), 127
sigmoid_prime() (*in module Compiler.ml*), 127
SignedZ2 (*C++ class*), 219
SignedZ2::SignedZ2 (*C++ function*), 220
sin() (*in module Compiler.mpc_math*), 114
sint (*class in Compiler.types*), 83
sintbit (*class in Compiler.types*), 92
softmax() (*in module Compiler.ml*), 127
solve_cholesky() (*in module Compiler.ml*), 127
solve_linear() (*in module Compiler.ml*), 127
solve_lower() (*in module Compiler.ml*), 127
solve_upper() (*in module Compiler.ml*), 127
sort() (*Compiler.types.Array method*), 40
sort() (*Compiler.types.Matrix method*), 46
sort() (*Compiler.types.MultiArray method*), 54
SparseMatrix (*class in Compiler.sparse*), 140
split (*class in Compiler.GC.instructions*), 205
sqrt() (*in module Compiler.mpc_math*), 114
SqrtOram (*class in Compiler.sqrt_oram*), 135
square (*class in Compiler.instructions*), 190
Square (*class in Compiler.ml*), 125
square() (*Compiler.GC.types.sbitfixvec method*), 98
square() (*Compiler.GC.types.sbitintvec method*), 101
square() (*Compiler.types.cfix method*), 57
square() (*Compiler.types.cgf2n method*), 60
square() (*Compiler.types.cint method*), 64
square() (*Compiler.types.MemValue method*), 48
square() (*Compiler.types.regint method*), 70
square() (*Compiler.types.sfix method*), 75
square() (*Compiler.types.sfloat method*), 78
square() (*Compiler.types.sgf2n method*), 83
square() (*Compiler.types.sint method*), 92
starg (*class in Compiler.instructions*), 190
start (*class in Compiler.instructions*), 190
start_timer() (*in module Compiler.library*), 112
stmc (*class in Compiler.instructions*), 190
stmcb (*class in Compiler.GC.instructions*), 206
stmcbi (*class in Compiler.GC.instructions*), 206
stmci (*class in Compiler.instructions*), 190
stmint (*class in Compiler.instructions*), 190
stminti (*class in Compiler.instructions*), 191
stms (*class in Compiler.instructions*), 191
stmsb (*class in Compiler.GC.instructions*), 206
stmsbi (*class in Compiler.GC.instructions*), 206
stmsi (*class in Compiler.instructions*), 191

stop (*class in Compiler.instructions*), 191
stop_timer() (*in module Compiler.library*), 112
store_in_mem() (*Compiler.GC.types.sbitfixvec method*), 98
store_in_mem() (*Compiler.types.cfix method*), 57
store_in_mem() (*Compiler.types.cgf2n method*), 60
store_in_mem() (*Compiler.types.cint method*), 64
store_in_mem() (*Compiler.types.regint method*), 70
store_in_mem() (*Compiler.types.sfix method*), 75
store_in_mem() (*Compiler.types.sfloat method*), 78
store_in_mem() (*Compiler.types.sgf2n method*), 83
store_in_mem() (*Compiler.types.sint method*), 92
subc (*class in Compiler.instructions*), 191
subcfi (*class in Compiler.instructions*), 191
subci (*class in Compiler.instructions*), 192
subint (*class in Compiler.instructions*), 192
subml (*class in Compiler.instructions*), 192
submr (*class in Compiler.instructions*), 192
subs (*class in Compiler.instructions*), 192
subsfi (*class in Compiler.instructions*), 193
subsi (*class in Compiler.instructions*), 193
sum() (*Compiler.types.Array method*), 40

T

tan() (*in module Compiler.mpc_math*), 115
tanh() (*in module Compiler.mpc_math*), 115
Tensor() (*Compiler.types.cfix class method*), 56
Tensor() (*Compiler.types.cgf2n class method*), 58
Tensor() (*Compiler.types.cint class method*), 61
Tensor() (*Compiler.types.regint class method*), 67
Tensor() (*Compiler.types.sfix class method*), 71
Tensor() (*Compiler.types.sfloat class method*), 76
Tensor() (*Compiler.types.sgf2n class method*), 78
Tensor() (*Compiler.types.sint class method*), 84
test_dijkstra() (*in module Compiler.dijkstra*), 139
threshold (*class in Compiler.instructions*), 193
time (*class in Compiler.instructions*), 193
to_binary_with_right_length() (*Compiler.types.Array method*), 40
to_binary_with_right_length() (*Compiler.types.Matrix method*), 47
to_binary_with_right_length() (*Compiler.types.MemValue method*), 48
to_binary_with_right_length() (*Compiler.types.MultiArray method*), 54
to_column_matrix() (*Compiler.types.Array method*), 40
to_regint() (*Compiler.types.cint method*), 64
to_regint_by_bit() (*Compiler.types.cint method*), 65
to_row_matrix() (*Compiler.types.Array method*), 41
trace() (*Compiler.types.Matrix method*), 47
trace() (*Compiler.types.MultiArray method*), 55
train() (*Compiler.decision_tree.TreeTrainer method*), 129
train_with_testing() (*Compiler.decision_tree.TreeTrainer method*), 129
trainable_variables (*Compiler.ml.Adam property*), 117
trainable_variables (*Compiler.ml.Optimizer property*), 121
trainable_variables (*Compiler.ml.SGD property*), 124
trans (*class in Compiler.GC.instructions*), 206
trans_mul() (*Compiler.types.Matrix method*), 47
trans_mul() (*Compiler.types.MultiArray method*), 55
trans_mul_to() (*Compiler.types.Matrix method*), 47
trans_mul_to() (*Compiler.types.MultiArray method*), 55
transpose() (*Compiler.types.Matrix method*), 47
transpose() (*Compiler.types.MultiArray method*), 55
transvec (*class in Compiler.GC.instructions*), 207
tree_reduce() (*in module Compiler.library*), 112
tree_reduce_multithread() (*in module Compiler.library*), 112
TreeClassifier (*class in Compiler.decision_tree*), 128
TreeTrainer (*class in Compiler.decision_tree*), 128
triple (*class in Compiler.instructions*), 193
trunc_pr (*class in Compiler.instructions*), 193
TwoQuadOutput (*class in Compiler.ml*), 125

U

UniquePathObliviousHeap (*class in Compiler.path_oblivious_heap*), 138
UniquePOHToHeapQAdapter (*class in Compiler.path_oblivious_heap*), 137
unsplit (*class in Compiler.instructions*), 194
update() (*Compiler.GC.types.cbits method*), 94
update() (*Compiler.GC.types.sbit method*), 95
update() (*Compiler.GC.types.sbitfixvec method*), 99
update() (*Compiler.GC.types.sbits method*), 103
update() (*Compiler.path_oblivious_heap.POHToHeapQAdapter method*), 136
update() (*Compiler.path_oblivious_heap.UniquePathObliviousHeap method*), 138
update() (*Compiler.path_oblivious_heap.UniquePOHToHeapQAdapter method*), 138
update() (*Compiler.types.cgf2n method*), 60
update() (*Compiler.types.cint method*), 65
update() (*Compiler.types.regint method*), 70
update() (*Compiler.types.sfix method*), 75
update() (*Compiler.types.sfloat method*), 78
update() (*Compiler.types.sgf2n method*), 83
update() (*Compiler.types.sint method*), 92
use (*class in Compiler.instructions*), 194
use_dabit (*Compiler.program.Program attribute*), 133
use_edabit (*class in Compiler.instructions*), 194

use_edabit() (*Compiler.program.Program method*), 133
use_inp (*class in Compiler.instructions*), 194
use_invperm() (*Compiler.program.Program method*), 133
use_matmul (*class in Compiler.instructions*), 194
use_prep (*class in Compiler.instructions*), 195
use_split() (*Compiler.program.Program method*), 133
use_square() (*Compiler.program.Program method*), 133
use_trunc_pr (*Compiler.program.Program property*), 133

V

var() (*in module Compiler.ml*), 127

W

while_do() (*in module Compiler.library*), 113
write() (*Compiler.types.MemValue method*), 49
write_fully_to_socket() (*Compiler.types.sint method*), 92
write_shares_to_socket() (*Compiler.types.sfix class method*), 76
write_shares_to_socket() (*Compiler.types.sint class method*), 92
write_to_file() (*Compiler.types.Array method*), 41
write_to_file() (*Compiler.types.cgf2n class method*), 60
write_to_file() (*Compiler.types.cint class method*), 65
write_to_file() (*Compiler.types.Matrix method*), 47
write_to_file() (*Compiler.types.MultiArray method*), 55
write_to_file() (*Compiler.types.regint class method*), 70
write_to_file() (*Compiler.types.sfix class method*), 76
write_to_file() (*Compiler.types.sgf2n class method*), 83
write_to_file() (*Compiler.types.sint class method*), 92
write_to_socket() (*Compiler.types.Array method*), 41
write_to_socket() (*Compiler.types.cfix class method*), 57
write_to_socket() (*Compiler.types.cgf2n class method*), 60
write_to_socket() (*Compiler.types.cint class method*), 65
write_to_socket() (*Compiler.types.Matrix method*), 47
write_to_socket() (*Compiler.types.MultiArray method*), 55
write_to_socket() (*Compiler.types.regint class method*), 71
write_to_socket() (*Compiler.types.sint class method*), 92
writefileclear (*class in Compiler.instructions*), 195
writesharestofile (*class in Compiler.instructions*), 195
writesockets (*class in Compiler.instructions*), 195
writesocketshare (*class in Compiler.instructions*), 196

X

xorc (*class in Compiler.instructions*), 196
xorcb (*class in Compiler.GC.instructions*), 207
xorcbi (*class in Compiler.GC.instructions*), 207
xorci (*class in Compiler.instructions*), 196
xorm (*class in Compiler.GC.instructions*), 207
xors (*class in Compiler.GC.instructions*), 208

Z

Z2 (*C++ class*), 219
zips (*class in Compiler.instructions*), 196
zipsb (*class in Compiler.GC.instructions*), 208