SCALE–MAMBA v1.3 : Documentation

A. Aly M. Keller E. Orsini D. Rotaru P. Scholl N.P. Smart T. Wood

March 24, 2019

Contents

Changes

This section documents changes made since the "last" release of the software.

This is currently **version 1.3** of the SCALE and MAMBA software. There are two main components to the system, a run time system called SCALE,

Secure Computation Algorithms from LEuven

and a Python-like programming language called MAMBA

Multiparty AlgorithMs Basic Argot

Note, that MAMBA is a type of snake (like a Python), a snake has scales, and an argot is a "secret language".

The software is a major modification of the earlier SPDZ software. You *should not* assume it works the same so please read the documentation fully before proceeding.

Changes in version 1.3 from 1.2

1. New offline phase for full threshold called TopGear has been implemented [CS19]. This means we have changed a number of FHE related security parameters. In particular the new parameter sets are *more* secure. **But** they are likely to be different from the ones you have been using. Thus you will need to generate again FHE parameters (using `Setup.x`).
2. Bug fixed when using Shamir with a large number of parties and very very low threshold value.
3. Renaming of some bytecodes to make their meaning clearer.
4. Floating point operations are now (almost) fully supported. We have basic operations on floating point `sfloat` variables supported, as well as basic trigonometric functions. Still to be completed are the square root, logarithm and exponential functions. Note that, the implementation of floating point numbers is different from that in the original SPDZ compiler. The main alteration is an additional variable to signal error conditions within the number, and secure processing of this signal.

Changes in version 1.2 from 1.1

1. A lot of internal re-writing to make things easier to maintain.
2. There are more configuration options in `config.h` to enable; e.g. you can now give different values to the different places we use statistical security parameters.
3. Minor correction to how FHE parameters are created. This means that the FHE parameters are a bit bigger than they were before. Probably a good idea to re-run setup to generate new keys etc in the Full Threshold case.
4. Minor change to how `CRASH` works. If the IO handler returns, then the `RESTART` instruction is automatically called.
5. There is a new run time switch `maxI` for use when performing timings etc. This is only for use when combined with the `max` switch below.
6. Two new instructions `CLEAR_MEMORY` and `CLEAR_REGISTERS` have been added. These are called from MAMBA via `clear_memory()` and `clear_registers()`. The first can only be called with compilation is done with the flag `-M`, whilst the second *may* issue out of order (consider it experimental at present).
7. Bug fix for `sfix` version of `arcsin/arccos` functions.

8. When running Shamir with a large number of parties we now move to using an interactive method to produce the PRSS as opposed to the non-interactive method which could have exponential complexity. This means we can now cope with larger numbers of parties in the Shamir sharing case. An example ten party example is included to test these routines.

Changes in version 1.1 from 1.0

1. Major bug fix to IO processing of private input and output. This has resulted in a change to the bytecodes for these instructions.

2. We now support multiple FHE Factory threads, the precise number is controlled from a run-time switch.

3. The restart methodology now allows the use to programmatically create new schedules and program tapes if so desired. The "demo" functionality is however exactly as it was before. Please see the example functionality in the file `src/Input_Output/Input_Output_Simple.cpp`

4. We have added in some extra verbose output to enable timing of the offline phase. To time the offline phase, on say bit production, you can now use the program `Program/do_nothing.mpc`, and then execute the command for player zero.

   ```
   ./Player.x -verbose 1 -max 1,1,10000000  0 Programs/do_nothing/
   ```

   Note square production on its own is deliberately throttled so that when run in a real execution bit production is preferred over squares. By altering the constant in the program `Program/do_nothing.mpc` you can also alter the number of threads used for this timing operation. If you enter a negative number for verbose then verbose output is given for the online phase; i.e. it prints the bytecodes being executed.

5. The fixed point square root functions have now been extended to cope with full precision fixed point numbers.

6. The `PRINTxxx` bytecodes now pass their output via the `Input_Output` functionality. These bytecodes are meant for debugging purposes, and hence catching them via the IO functionality makes most sense. The relevant function in the IO class is `debug_output`.

7. We have added the ability to now also input and output `regint` values via the IO functionality, with associated additional bytecodes added to enable this.

8. The IO class now allows one to control the opening and closing of channels, this is aided by two new bytecodes for this purpose called `OPEN_CHAN` and `CLOSE_CHAN`.

9. Input of clear values via the IO functionality (i.e. for `cint` and `regint` values) is now internally checked to ensure that all players enter the same clear values. Note, this requires modification to any user defined derived classes from `Input_Output_Base`. See the chapter on IO for more details on this.

10. The way the chosen IO functionality is bound with the main program has now also been altered. See the chapter on IO for more details on this.

11. These changes have meant there are a number of changes to the specific byte codes, so you will need to recompile MAMBA programs. If you generate your own bytecodes then your backend will need to change as well.

Changes From SPDZ

Apart from the way the system is configured and run there are a number of functionality changes which we outline below.

Things no longer supported

1. We do not support any $GF(2^n)$ arithmetic in the run time environment. The compiler will no longer compile your programs.

2. There are much fewer switches in the main program, as we want to produce a system which is easier to support and more useful in building applications.

3. Socket connections, file, and other forms of IO to the main MPC engine is now unified into a single location. This allows *you* to extend the functionality without altering the compiler or run-time in any way (bar changing which IO class you load at compile time). See Section 6 for details.

Additions

1. The offline and online phases are now fully integrated. This means that run-times will be slower than you would have got with SPDZ, but the run-times obtained are closer to what you would expect in a "real" system. **Both** the online and offline phases are **actively** secure with abort.

2. Due to this change it can be slow to start a new instance and run a new program. So we provide a new (experimental) operation which "restarts" the run-time. This is described in Section 7. This operation is likely to be revamped and improved in the next release as we get more feedback on its usage.

3. We support various Q2 access structures now, which can be defined in various ways: Shamir threshold, via Replicated sharing, or via a general Monotone Span Programme (MSP). For replicated sharing you can define the structure via either the minimally qualified sets, or the maximally unqualified sets. For general Q2-MSPs you can input a non-multiplicative MSP and the system will find an equivalent multiplicative one for you using the method of [CDM00].

4. Offline generation for Q2 is done via Maurer's method [Mau06], but for Replicated you can choose between Maurer and the reduced communication method of Keller, Rotaru, Smart and Wood [KRSW18]. For general Q2-MSPs, and Shamir sharing, the online phase is the method described in Smart and Wood [SW18], with (*currently*) the offline phase utilizing Maurer's multiplication method [Mau06].

5. All player connections are now via SSL, this is not strictly needed for full threshold but is needed for the other access structures we now support.

6. We now have implemented more higher level mathematical functions for the `sfix` datatype, and corrected a number of bugs. A similar upgrade is expected in the next release for the `sfloat` type.

Changes

1. The **major** change is that the offline and online phases are now integrated. This means that to run quick test runs, using full threshold is going to take ages to set up the offline data. Thus for test runs of programs in the online phase it is best to test using one of the many forms of Q2 access structures. For example by using Shamir with three players and threshold one. Then once your online program is tested you can move to a production system with two players and full threshold if desired.

2. You now compile a program by executing

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

   where `Programs/tutorial` is a *directory* which contains a file called `tutorial.mpc`. Then the compiler puts all of the compiled tapes etc *into this directory*. This produces a much cleaner directory output etc. By typing `make pclean` you can clean up all pre-compiled directorys into their initial state.

3. The compiler picks up the prime being used for secret sharing after running the second part of `Setup.x`. So you need to recompile the `.mpc` files if you change the prime used in secret sharing, and you should not compile any SCALE `.mpc` programs before running `Setup.x`.

4. Internally (i.e. in the C++ code), a lot has been re-organized. The major simplification is removal of the `octetstream` class, and it's replacement by a combination of `stringstream` and `string` instead. This makes readibility much easier.

5. All opcodes in the range `0xB*` have been changed, so any byte codes you have generated from outside the python system will need to be changed.

6. We have tried to reduce dependencies between classes in the C++ code a lot more. Thus making the code easier to manage.

7. Security-wise we use the latest FHE security estimates for the FHE-based part, and this can be easily updated. See Chapter 9 on FHE security later.

Introduction

The SCALE system consists of three main sub-systems: An offline phase, an online phase and a compiler. Unlike the earlier SPDZ system, in SCALE the online and offline phases are fully integrated. Thus you can no longer time just the online phase, or just the offline phase. The combined online/offline phase we shall refer to as SCALE, the compiler takes a program written in our special language MAMBA, and then turns it into bytecode which can be executed by the SCALE system.

We provide switches (see below) to obtain different behaviours between how the online and offline phases are integrated together, which can allow for some form of timing approximation. The main reason for this change is to ensure that the system is "almost" secure out of the box, even if means it is less good for obtaining super-duper numbers for research papers.

An issue though is that the system takes a while to warm up the offline queues before the online phase can execute. This is especially true in the case of using a Full Threshold secret sharing scheme. Indeed in this case it is likely that the online phase will run so-fast that the offline phase is always trying to catch up. In addition, in this situation the offline phase needs to do a number of high-cost operations before it can even start. Thus using the system to run very small programs is going to be inefficient, although the execution time you get is indicative of the total run time you should be getting in a real system, it is just not going to be very impressive.

In order to enable efficient operation in the case where the offline phase is expensive (e.g. for Full Threshold secret sharing) we provide an *experimental* mechanism to enable the SCALE system to run as a seperate process (generating offline data), and then a MAMBA program can be compiled in a just-in-time manner and can then be dispated to the SCALE system for execution. Our methodology for this is not perfect, but has been driven by a real use case of the system. See Section 7 for more details.

But note SCALE/MAMBA is *an experimental research system* there has been no effort to ensure that the system meets rigourous production quality code control. In addition this also means it comes with limited support. If you make changes to any files/components you are on your own. If you have a question about the system we will endeavour to answer it.

Warnings:

- The Overdrive system [KPR18] for the offline phase for full-threshold access structures requires a distributed key generation phase for the underlying homomorphic encryption scheme. The SPDZ-2 system and paper does describe such a protocol, but it is only covertly secure. We do not provide the protocol to do this, instead the Setup program in this instance will generate a suitable key and distribute it to the different players. In any real system this entire setup phase will need investigating, with perhaps using HSMs to construct and deploy keys if no actively secure key generation protocol is available.

- There is a security hole in how we have implemented things. As the

  ```
  offline->sacrifice->online
  ```

  pipeline is run continously in separate threads, each with their own channels etc., we may **use** a data item in the online phase **before** the checking of a data item has **fully completed** (i.e. before in an associated sacrifice etc. has been MAC-checked, for full-threshold, or hash-checked, for other LSSS schemes). In a real system you will want to address this by having some other list of stuff (between sacrifice and online), which ensures that all checks are complete before online is allowed to use any data; or by ensuring MAC/hash-checking is done before the sacrifice queues are passed to the online phase. In our current system if something bad is introduced by an adversary then the system **will** halt. But, before doing so there is a *small* chance that some data will have leaked from the computation if the adversary can schedule the error at exactly the right point (depending on what is happening in other threads).

Architecture

The basic internal runtime architecture is as follows:

- Each MAMBA program (`.mpc` file) will initiate a number of threads to perform the online stage. The number of "online threads" needed is worked out by the compiler. You can programmatically start and stop threads using the python-like language (see later). Using multiple threads enables you to get high throughput. Almost all of our experimental results are produced using multiple threads.

- Each online is associated with another four "feeder" threads. One produces multiplication triples, one produces square pairs and one produces shared bits. The fourth thread performs the sacrificing step, as well as the preprocessing data for player IO. The chain of events is that the multiplication thread produces an unchecked triple. This triple is added to a triple-list (which is done in batches for efficiency). At some point the sacrifice thread decides to take some data off the triple-list and perform the triple checking via sacrificing. Once a triple passes sacrificing it is passed onto another list, called the sacrificed-list, for consumption by the online phase.

- By having the production threads aligned with an online thread we can avoid complex machinary to match produces with consumers. This however may (more like will) result in the over-production of offline data for small applications.

- In the case of Full Threshold we have another set of global threads (called the FHE Factory threads, or the FHE Industry) which produces level one random FHE ciphertexts which have passed the ZKPoK from the Top Gear protocol [CS19], which is itself a variant of the High Gear protocol in Overdrive [KPR18]. This is done in a small bunch of global threads as the ZKPoK would blow up memory if done for each thread that would consume data from the FHE thread. These threads are numbered from 10000 in the code. Any thread (offline production thread) can request a new ciphertext/plaintext pair from the FHE factory threads.

- We also implement a thread (number 20000 in the code) which implements pairwise OTs, which can be turned on with a magic undocumented switch. (TO DO:) *This is currently only for experimental purposes though. It will be fully integrated in a future release.*

Installation and Setup

Installation

You will require the following previously loaded programs and libraries:

- GCC/G++: Tested with version 7.2.1
- MPIR library, compiled with C++ support (use flag –enable-cxx when running configure) : Tested with version 3.0.0
- Python, ideally with ‘gmpy2‘ package (for testing): Tested with Python 2.7.5
- CPU supporting AES-NI and PCLMUL
- OpenSSL: Tested with version 1.1.0
- Crypto++: Tested with version 7.0

Developers will also require

- clang-format as to apply the standard C++ format to files. Tested with clang-format version 4.0.1.

Installing MPIR and OpenSSL

This bit, on explaining how to install MPIR and OpenSSL inside `$HOME/local`, is inspired from this blogpost. The target directory here can be changed to whatever you wish. If you follow this section we assume that you have **cloned** the main repository in your `$HOME` directory.

```
mylocal="$HOME/local"
mkdir -p ${mylocal}
cd ${mylocal}

# install MPIR 3.0.0
curl -O 'http://mpir.org/mpir-3.0.0.tar.bz2'
tar xf mpir-3.0.0.tar.bz2
cd mpir-3.0.0
./configure --enable-cxx --prefix="${mylocal}/mpir"
make && make check && make install

# install OpenSSL 1.1.0
cd $mylocal
curl -O https://www.openssl.org/source/openssl-1.1.0j.tar.gz
tar -xf openssl-1.1.0j.tar.gz
cd openssl-1.1.0j
./config --prefix="${mylocal}/openssl"
make && make install
```

Now export MPIR and OpenSSL paths by copying the following lines at the end of your `$HOME/.bashrc` configuration file.

```
# this goes at the end of your $HOME/.bashrc file
export mylocal="$HOME/local"

# export OpenSSL paths
export PATH="${mylocal}/openssl/bin/:${PATH}"
```

```
export C_INCLUDE_PATH="${mylocal}/openssl/include/:${C_INCLUDE_PATH}"
export CPLUS_INCLUDE_PATH="${mylocal}/openssl/include/:${CPLUS_INCLUDE_PATH}"
export LIBRARY_PATH="${mylocal}/openssl/lib/:${LIBRARY_PATH}"
export LD_LIBRARY_PATH="${mylocal}/openssl/lib/:${LD_LIBRARY_PATH}"

# export MPIR paths
export PATH="${mylocal}/mpir/bin/:${PATH}"
export C_INCLUDE_PATH="${mylocal}/mpir/include/:${C_INCLUDE_PATH}"
export CPLUS_INCLUDE_PATH="${mylocal}/mpir/include/:${CPLUS_INCLUDE_PATH}"
export LIBRARY_PATH="${mylocal}/mpir/lib/:${LIBRARY_PATH}"
export LD_LIBRARY_PATH="${mylocal}/mpir/lib/:${LD_LIBRARY_PATH}"
```

Change CONFIG.mine

We now need to copy the file `CONFIG` in the main directory to the file `CONFIG.mine`. Then we need to edit `CONFIG.mine`, so as to place the correct location of this ROOT directory correctly, as well as indicating where the OpenSSL library should be picked up from (this is likely to be different from the system installed one which GCC would automatically pick up). This is done by executing the following commands

```
cd $HOME/SCALE-MAMBA
cp CONFIG CONFIG.mine
echo "ROOT = $HOME/SCALE-MAMBA" >> CONFIG.mine
echo "OSSL = ${mylocal}/openssl" >> CONFIG.mine
```

You can also at this stage specify various compile time options such as various debug and optimisation options. We would recommend commenting out all DEBUG options from FLAGS and keeping `OPT = -O3`.

- The `DEBUG` flag is a flag which turns on checking for reading before writing on registers, thus it is mainly a flag for development testing of issues related to the compiler.
- The `DETERMINISTIC` flag turns off the use of true randomness. This is really for debugging to ensure we can replicate errors due. It should **not** be used in a real system for obvious reasons.

If you are going to use full threshold LSSSs then `MAX_MOD` needs to be set large enough to deal with the sizes of the FHE keys. Otherwise this can be set to just above the word size of your secret-sharing modulus to obtain better performance. As default we have set it for use with full threshold.

Change config.h

If wanted you can also now configure various bits of the system by editing the file

```
config.h
```

in the sub-directory `src`. The main things to watch out for here are the various security parameters; these are explained in more detail in Section 9. Note, to configure the statistical security parameter for the number representations in the compiler (integer comparison, fixed point etc) from the default of 40 you need to add the following commands to your MAMBA programs.

```
program.security = 100
sfix.kappa = 60
sfloat.kappa=30
```

However, in the case of the last two you *may* also need to change the precision or prime size you are using. See the documentation for `sfix` and `sfloat` for this.

Final Compilation

The only thing you now have to do is type

```
make progs
```

That's it! After make finishes then you should see a 'Player.x' executable inside the SCALE-MAMBA directory.

Creating and Installing Certificates

For a proper configuration you need to worry about the rest of this section. However, for a quick idiotic test installation jump down to the "Idiot Installation" of Section 3.4.

All channels will be TLS encrypted. For SPDZ this is not needed, but for other protocols we either need authenticated or secure channels. So might as well do everything over *mutually* authenticated TLS. We are going to setup a small PKI to do this. You thus first need to create keys and certificates for the main CA and the various players you will be using.

When running `openssl req ...` to create certificates, it is vitally important to ensure that each player has a different Common Name (CN), and that the CNs contain no spaces. The CN is used later to configure the main MPC system and be sure about each party's identity (in other words, they really are who they say they are).

First go into the certificate store

```
cd Cert-Store
```

Create CA authority private key

```
openssl genrsa -out RootCA.key 4096
```

Create the CA self-signed certificate:

```
openssl req -new -x509 -days 1826 -key RootCA.key -out RootCA.crt
```

Note, setting the DN for the CA is not important, you can leave them at the default values.

Now for *each* MPC player create a player certificate, e.g.

```
openssl genrsa -out Player0.key 2048
openssl req -new -key Player0.key -out Player0.csr
openssl x509 -req -days 1000 -in Player0.csr -CA RootCA.crt \
       -CAkey RootCA.key -set_serial 0101 -out Player0.crt -sha256
```

remembering to set a different Common Name for each player.

In the above we assumed a global shared file system. Obviously on a real system the private keys is kept only in the `Cert-Store` of that particular player, and the player public keys are placed in the `Cert-Store` on each player's computer. The global shared file system here is simply for test purposes. Thus a directory listing of `Cert-Store` for player one, in a four player installation, will look like

```
Player1.crt
Player1.key
Player2.crt
Player3.crt
Player4.crt
RootCA.crt
```

Running Setup

The program `Setup.x` is used to run a one-time setup for the networking and/or secret-sharing system being used. You must do networking before secret-sharing (unless you keep the number of players fixed), since the secret-sharing setup picks up the total number of players you configured when setting up networking.

- Just as above for OpenSSL key-generation, for demo purposes we assume a global file store with a single directory `Data`.

Running the program `Setup.x` and specifying the secret-sharing method will cause the program to generate files holding MAC and/or FHE keys and place them in the folder `Data`. When running the protocol on separate machines, you must then install the appropriate generated MAC key file `MKey-*.key` in the `Data` folder of each player's computer. If you have selected full-threshold, you also need to install the file `FHE-Key-*.key` in the same directory. You also need to make sure the public data files `NetworkData.txt` and `SharingData.txt` are in the directory `Data` on each player's computer. These last two files specify the configuration which you select with the `Setup.x` program.

We now provide more detail on each of the two aspects of the script `Setup.x`.

Data for networking

Input provided by the user generates the file `Data/NetworkData.txt` which defines the following

- The root certificate name.
- The number of players.
- For each player you then need to define
  - Which IP address is going to be used
  - The name of the certificate for that player
- Whether a fake offline phase is going to be used.
- Whether a fake sacrifice phase is going to be used.

Data for secret sharing:

You first define whether you are going to be using full threshold (as in traditional SPDZ), Shamir (with $t < n/2$), a Q2-Replicated scheme, or a Q2-MSP.

Full Threshold: In this case the prime modulus cannot be chosen directly, but needs to be selected to be FHE-friendly. Hence, in this case you can only specify the number of bits in the modulus[1]. For Full Threshold you can enter the size of the modulus you want (between 16 bits and 1024 bits). The system will then search for a modulus which is compatible with the FHE system we are using.

At this stage the MAC keys and FHE secret keys are setup and written into the files

`MKey-*.key` and `FHE-Key-*.key`

in the `Data` directory. This is clearly an insecure way of parties picking their MAC keys. But this is only a research system. At this stage we also generate a set of keys for distributed decryption of a level-one FHE scheme if needed. For the case of fake offline we assume these keys are on *each* computer, but using fake offline is only for test purposes in any case.

[1] In all other cases you select the prime modulus for the LSSS directly at this point.

Shamir Secret Sharing: Shamir secret sharing we assume is self-explanatory. For the Shamir setting we use an online phase using the reduced communication protocols of [KRSW18]; the offline phase (*currently*) only supports *Maurer*'s multiplication method [Mau06]. This will be changed in future releases to also support the new offline method from [SW18].

Replicated Secret Sharing: For Replicated sharing you should enter a complete monotone Q2 access structure. There are three options to do this,

1. As a set of maximally unqualified sets;
2. As a set of unqualified qualified sets;
3. As a simple threshold system.

If the first (resp. second) option is selected, then any set that is neither a superset nor a subset of a set provided as input will be assumed qualified (resp. unqualified). The last option is really for testing, as most threshold systems implemented using Replicated secret-sharing will be less efficient than using Shamir. Specifying either the first or second option and providing input results in the program computing all of the qualified and unqualified sets in the system.

For Replicated secret-sharing you can also decide what type of offline phase you want: one based on *Maurer*'s multiplication method [Mau06], or one based on our *Reduced* communication protocols [KRSW18].

Suppose we want to input the four-party Q2 access structure given by

$$\Gamma^- = \{\{P_1, P_2\}, \{P_1, P_3\}, \{P_1, P_4\}, \{P_2, P_3, P_4\}\}$$

and

$$\Delta^+ = \{\{P_1\}, \{P_2, P_3\}, \{P_2, P_4\}, \{P_3, P_4\}\}.$$

We can express this example in the following two ways:

```
4 parties, maximally unqualified
        1 0 0 0
        0 1 1 0
        0 1 0 1
        0 0 1 1
```

or as

```
4 parties, minimally qualified
        1 1 0 0
        1 0 1 0
        1 0 0 1
        0 1 1 1
```

As a second example with six parties with a more complex access structure for our Reduced Communication protocol consider:

$$\Gamma^- = \{\{P_1, P_6\}, \{P_2, P_5\}, \{P_3, P_4\}, \{P_1, P_2, P_3\}, \{P_1, P_4, P_5\}, \{P_2, P_4, P_6\}, \{P_3, P_5, P_6\}\}$$

and

$$\Delta^+ = \{\{P_1, P_2, P_4\}, \{P_1, P_3, P_5\}, \{P_2, P_3, P_6\}, \{P_4, P_5, P_6\}\}.$$

Each party is in a different pair of sets. We can represent it via:

```
6 parties, maximally unqualified sets
        1 1 0 1 0 0
        1 0 1 0 1 0
        0 1 1 0 0 1
        0 0 0 1 1 1
```

Q2-MSP Programs: A final way of entering a Q2 access structure is via a MSP, or equivalently, via the matrix which defines an underlying Q2 LSSS. For n parties we define the number of shares each party has to be n_i, and the dimension of the MSP to be k. The MSP is then given by a $(\sum n_i) \times k$ matrix G. The first n_1 rows of G are associated with player one, the next n_2 rows with player two and so on. An secret sharing of a value s is given by the vector of values

$$\mathbf{s} = G \cdot \mathbf{k}$$

where $\mathbf{k} = (k_i) \in \mathbb{F}_p^k$ and $\sum k_i = s$. A subset of parties A is qualified if the span of the rows they control contain the vector $(1, \ldots, 1) \in \mathbb{F}_p^k$.

This secret sharing scheme can be associated with a monotone access structure, and we call this scheme Q2 if the associated access structure is Q2. However, it is not the case (unlike for Shamir and Replicated sharing) that the Q2 MSP is itself multiplicative (which is crucial for our MPC protocols). Thus if you enter a Q2 MSP which is **not** multiplicative, we will automatically extend this for you into an equivalent multiplicative MSP using the method of [CDM00].

As in the Shamir setting we use an online phase using the reduced communication protocols of [KRSW18]; the offline phase (*currently*) only supports *Maurer*'s multiplication method [Mau06]. This will be changed in future releases to also support the new offline method from [SW18].

In the file `Auto-Test-Data/README.txt` we provide some examples of MSPs. If we take the MSP for Shamir (say) over three parties with threshold one we obtain the matrix

$$G = \begin{pmatrix} 1 & 2 \\ 1 & 3 \\ 1 & 4 \end{pmatrix}.$$

This is slightly different from the usual Shamir generating matrix as we assume the target vector is $(1, 1)$ and not $(1, 0)$ as is often the case in Shamir sharing.

Idiot's Installation

To install an installation of the three party Shamir based variant, with default certificates and all parties running on local host. Execute the following commands

```
cp Auto-Test-Data/Cert-Store/* Cert-Store/
cp Auto-Test-Data/1/* Data/
```

You can play with writing your own MAMBA programs and running them on the same local host (via IP address 127.0.0.1).

Simple Example

In this section we describe writing and compiling a simple function using the python-like language MAMBA. We also explain how to run the test scripts to test the system out, plus the run time options/switches you can pass to `Player.x` for different behaviours.

Compiling and running simple program

Look at the simple program in `Programs/tutorial/tutorial.mpc` given below:

```
# (C) 2017 University of Bristol. See License.txt
# (C) 2018 KU Leuven. See License.txt

def test(actual, expected):
    actual = actual.reveal()
    print_ln('expected %s, got %s', expected, actual)

# cint: clear integers modulo p
# sint: secret integers modulo p

a = sint(1)
b = cint(2)

test(a + b, 3)
test(a + a, 2)
test(a * b, 2)
test(a * a, 1)
test(a - b, -1)
test(a < b, 1)
test(a <= b, 1)
test(a >= b, 0)
test(a > b, 0)
test(a == b, 0)
test(a != b, 1)

clear_a = a.reveal()

# arrays and loops

a = Array(100, sint)

@for_range(100)
def f(i):
    a[i] = sint(i)**2

test(a[99], 99**2)

# conditional

if_then(cint(0))
a[0] = 123
else_then()
```

```
a[0] = 789
end_if()

test(a[0], 789)
```

This takes a secret integer `a` and a clear integer `b` and applies various operations to them. It then prints tests as to whether these operations give the desired results. Then an array of secret integers is created and assigned the values i^2. Finally a conditional expression is evaluated based on a clear value. Notice how the `tutorial.mpc` file is put into a directory called `tutorial`, this is crucial for the running of the compiler.

To compile this program we type

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

in the main directory. Notice how we run the compiler on the *directory* and not the program file itself. The compiler then places various "tape" files consisting of byte-code instructions into this directory, along with a schedule file `tutorial.sch`. It is this last file which tells the run time how to run the program, including how many threads to run, and which "tape" files to run first[2].

Having compiled our program we can now run it. To do so we simply need to execute the following commands, one on each of the computers in our MPC engine (assuming three players)

```
./Player.x 0 Programs/tutorial
./Player.x 1 Programs/tutorial
./Player.x 2 Programs/tutorial
```

Note players are numbers from zero, and again we run the *directory* and not the program file itself.

The Test Scripts

You will notice a bunch of test programs in directory `Programs`. These are for use with the test scripts in the directory `Scripts`. To use these test scripts you simply execute in the top level directory

```
Script/test.sh test_<name of test>
```

The test scripts place data in the clear memory dump at the end of a program, and test this cleared memory against a simulated run. the test-all facility of

```
Script/test.sh
```

If a test passes then the program will fail, with a (possibly cryptic) explanation of why.

We also provide a script which tests the **entire** system over various access structures. This can take a **very long time to run**, but if you want to run exhaustive tests then in the main directory execute

```
./run_tests.sh
```

Run Time Switches for Player.x

There are a number of switches which can be passed to the program `Player.x`; these are

- **-pnb** x: Sets the base portnumber to x. This by default is equal to 5000. With this setting we use all portnumbers in the range x to $x + n - 1$, where n is the number of players.

[2]Historical note, we call the byte code files "tapes" as they are roughly equivalent to simple programs, and the initial idea for scheduling came to Nigel when looking at the Harwell WITCH computer at TMNOC. They in some sense correspond to "largish" basic blocks in modern programming languages.

- **-pns** $x_1, \ldots, x_n$: This overides the `pnb` option, and sets the listening portnumber for player i to x_i. The same arguments must be supplied to each player, otherwise the players do not know where to connect to, and if this option is used there needs to be precisely n given distinct portnumbers.
- **-mem xxxx**: Where xxxx is either `old` or `empty`. The default is `empty`. See later for what we mean by memory.
- **-verbose n**: Sets the verbose level to n. The higher value of n the more diagnostic information is printed. This is mainly for our own testing purposes, but verbose level one gives you a method to time offline production (see the Changes section for version 1.1). If n is negative then the bytecodes being executed by the online phase are output (and no offline verbose output is produced).
- **-max m,s,b**: Stop running the offline phase for each online thread when we have generated m multiplication triples, s square pairs and b shared bits.
- **-min m,s,b**: Do not run the online phase in each thread until the associated offline threads have generated m multiplication triples, s square pairs and b shared bits. However, these minimums need to be less than the maximum sacrificed list sizes defined in `config.h`. Otherwise the maximums defined in that file will result in the program freezing.
- **-maxI i**: An issue when using the flag **-max** is that for programs with a large amount of input/output **-max** can cause the IO queue to stop being filled. Thus if you use **-max** and are in this situation then signal using this flag an upper bound on the number of amount IO data you will be consuming. We would recommend that you multiply the max amount per player by the number of players here.
- **-f 2**: The number of FHE factories to run in parallel. This only applies (obviously) to the Full Threshold situation. How this affects your installation depends on the number of cores and how much memory you have. We set the default to two.

For example by using high values of the variables set to **-min** you get the offline data queues full before you trigger the execution of the online program. For a small online program this will produce times close to that of running the offline phase on its own. Or alternatively you can stop these queues using **-max**. By combining the two together you can get something close to (but not exactly the same as) running the offline phase followed by the online phase.

Run Time Subsystem

This chapter describes the implementation of the MPC Virtual Machine. The virtual machine is driven by Bytecodes which are produced by the MAMBA compiler (see later). Of course you could compile Bytecodes from any compiler if you wanted to write one with the correct backend.

The virtual machine structure resembles that of a simple multi-core processor, and is a register-based machine. Each core corresponds to a seperate online thread of execution; so from now on we refer to these "cores" as threads. Each thread has a seperate set of registers, as well as a stack for *integer* values. To allow the saving of state, or the transfer of data between threads there is a global memory. This global memory (or at least the first 2^{20} values) are saved whenever the SCALE system gracefully shuts down. The loading of this saved memory into a future run of the system is controlled by the command line arguments passed to the `Player.x` program. The design is deliberately kept sparse to ensure a fast, low-level implementation, whilst more complex optimization decisions are intended to be handled at a higher level.

Overview

The core of the virtual machine is a set of threads, which execute sequences of instructions encoded in a bytecode format. Files of bytecode instructions are referred to as *tapes*, and each thread processes a single tape at a time. Each of these execution threads has a pairwise point-to-point communication channels with the associated threads in all other players' runtime environments. These communication channels, like all communication channels in SCALE, are secured via TLS. The threads actually have two channels to the correspond thread in the other parties; we call these different channels "connections". For the online thread "connection" zero is used for standard opening of shared data, whereas "connection" one is used for private input and output. This is to avoid conflicts beween the two (for example a `PRIVATE_OUTPUT` coming between a `STARTOPEN` and a `STOPOPEN`. Each online thread is supported by four other threads performing the offline phase, each again with pairwise TLS secured point-to-point channels. Currently the offline threads only communicate on "connection" zero.

In the case of Full Threshold secret sharing another set of threads act as a factory for FHE ciphertexts. Actively secure production of such ciphertexts is expensive, requiring complex zero-knowledge proofs (see Section 9). Thus the FHE-Factory threads locates this production into a single location. The number of FHE Factory threads can be controlled at run-time by the user.

In addition to bytecode files, each program to be run must have a *schedule*. This is a file detailing the execution order of the tapes, and which tapes are to be run in parallel. There is no limit to the number of concurrent tapes specified in a schedule, but in practice one will be restricted by the number of cores. The schedule file allows you to schedule concurrent threads of execution, it also defines the maximum number of threads a given run-time system will support. It also defines the specific bytecode sequences which are pre-loaded into the system. One can also programmatically control execution of new threads using the byte-code instructions `RUN_TAPE` and `JOIN_TAPE` (see below for details). The schedule is run by the *control thread*. This thread takes the tapes to be executed at a given step in the schedule, passes them to the execution threads, and waits for the threads to finish their execution before proceeding to the next stage of the schedule.

Communication between threads is handled by a global *main memory*, which all threads have access to. To avoid unnecessary stalls there is no locking mechanism provided to the memory. So if two simultaneously running threads execute a read and a write, or two writes, to the same memory location then the result is undefined since it is not specified as to which order the instructions will be performed in. Memory comes in both clear and secret forms.

Each execution thread also has its own local clear and secret registers, to hold temporary variables. To avoid confusion with the main memory The values of registers are not assumed to be maintained between an execution thread running one tape and the next tape, so all passing of values between two sequential tape executions must be done by reading and writing to the virtual machine's main memory. This holds even if the two consequtive bytecode sequences run on the same "core".

Bytecode instructions

The design of the Bytecode instructions within a tape are influenced by the RISC design strategy, coming in only a few basic types and mostly taking between one and three operands. The virtual machine also supports a limited form of SIMD instructions within a thread, whereby a single instruction is used to perform the same operation on a fixed size set of registers. These vectorized instructions are not executed in parallel as in traditional SIMD architectures, but exist to provide a compact way of executing multiple instructions within a thread, saving on memory and code size.

A complete set of byte codes and descriptions is given in the html file in

`$(HOME)/Documentation/Compiler_Documentation/index.html`

under the class `instructions`. Each encoded instruction begins with 32 bits reserved for the opcode. The rightmost nine bits specify the instruction to be executed[3]. The remaining 23 bits are used for vector instructions, specifying the size of the vector of registers being operated on. The remainder of an instruction encoding consists of 4-byte operands, which correspond to either indices of registers or immediate integer values.

- Note, vector instructions are not listed in the `html` document above. They have the same name as standard instructions, prefixed by 'V', with the opcode created as described above.

The basic syntax used in the above html file is as follows:

- 'c[w]': clear register, with the optional suffix 'w' if the register is written to.
- 's[w]': secret register, as above.
- 'r[w]': regint register, as above.
- 'i' : 32-bit integer signed immediate value.
- 'int' : 64-bit integer unsigned immediate value.
- 'p' : 32-bit number representing a player index.
- 'str' : A four byte string.

Memory comes in three varieties `sint`, `cint`, and `regint`; denoted by `S[i]`, `C[i]` and `R[i]`.

Load, Store and Memory Instructions

Being a RISC design the main operations are load/store operations, moving operations, and memory operations. Each type of instructions comes in either clear data, sharedata, or integer data formats. The integer data is pure integer arithemetic, say for controling loops, whereas clear data is integer arithmetic modulo p. For the clear values all values represented as integers in the range $(-\frac{p-1}{2}, \ldots, \frac{p-1}{2}]$. The integer stack is again mainly intended for loop control.

Basic Load/Store/Move Instructions:

`LDI`, `LDI`, `LDSI`, `LDINT`, `MOVC`, `MOVS`, `MOVINT`.

Loading to/from Memory:

`LDMC`, `LDMS`, `STMC`, `STMS`, `LDMCI`, `LDMSI`, `STMCI`, `STMSI`, `LDMINT`, `STMINT`, `LDMINTI`, `STMINTI`.

Accessing the integer stack:

`PUSHINT`, `POPINT`.

[3]The choice of nine is to enable extension of the system later, as eight is probably going to be too small.

Data Conversion

To convert from mod p to integer values and back we provide the conversion routines. `CONVINT`, `CONVMODP`. These are needed as the internal mod p representation of clear data is in Montgomery representation.

Preprocessing loading instructions

The instructions for loading data from the preprocessing phase are denoted `TRIPLE`, `SQUARE`, `BIT`, and they take as argument three, two, and one secret registers respectively. The associated data is loaded from the concurrently running offline threads and loaded into the registers given as arguments.

Open instructions

The process of opening secret values is covered by two instructions. The `STARTOPEN` instruction takes as input a set of m shared registers, and `STOPOPEN` an associated set of m clear registers, where m can be an arbitrary integer. This initiates the protocol to reveal the m secret shared register values, storing the result in the specified clear registers. The reason for splitting this into two instructions is so that local, independent operations may be placed between a `STARTOPEN` and `STOPOPEN`, to be executed whilst waiting for the communication to finish.

There is no limit on the number of operands to these instructions, allowing for communication to be batched into a single pair of instructions to save on network latency. However, note that when the `RunOpenCheck` function in the C++ class `Open_Protocol` is used to check MACs/Hashes then this can stall when the network buffer fills up, and hang indefinitely. On our test machines this happens when opening around 10000 elements at once, so care must be taken to avoid this when compiling or writing bytecode (the Python compiler could automatically detect and avoid this).

Threading tools

Various special instructions are provided to ease the workload when writing programs that use multiple tapes.

- The `LDTN` instruction loads the current thread number into a clear register.
- The `LDARG` instruction loads an argument that was passed when the current thread was called. Thread arguments are optional and consist of a single integer, which is specified in the schedule file that determines the execution order of tapes, or via the instruction `RUN_TAPE`.
- The `STARG` allows the current tape to change its existing argument.
- To run a specified pre-loaded tape in a given thread, with a given argument the `RUN_TAPE` command is executed.
- To wait until a specified thread has finished one executes the `JOIN_TAPE` function.

Basic Arithmetic

This is captured by the following instructions, with different instructions being able to be operated on clear, shared and integer types. `ADDC`, `ADDS`, `ADDM`, `ADDCI`, `ADDSI`, `ADDINT`, `SUBC`, `SUBS`, `SUBML`, `SUBMR`, `SUBCI`, `SUBSI`, `SUBCFI`, `SUBSFI`, `SUBINT`, `MULC`, `MULM`, `MULCI`, `MULSI`, and `MULINT`.

Advanced Arithmetic

More elaborate algorithms can clearly be executed directly on clear or integer values; without the need for complex protocols. These include logical, shift and number theoretic functions. `ANDC`, `XORC`, `ORC`, `ANDCI`, `XORCI`, `ORCI`, `NOTC`, `SHLC`, `SHRC`, `SHLCI`, `SHRCI`, `DIVC`, `DIVCI`, `DIVINT`, `MODC`, `MODCI`. `LEGENDREC`, and `DIGESTC`.

Debuging Output

To enable debugging we provide simple commands to send debugging information to the `Input_Output` class. These bytecodes are

```
PRINTINT,           PRINTMEM,          PRINTREG,          PRINTREGPLAIN,
PRINTCHR,           PRINTSTR,          PRINTCHRINT,       PRINTSTRINT,
PRINTFLOATPLAIN,    PRINTFIXPLAIN.
```

Data input and output

This is entirely dealt with in the later Chapter on IO. The associated bytecodes are

```
OUTPUT_CLEAR,          INPUT_CLEAR,
OUTPUT_SHARE,          INPUT_SHARE,
OUTPUT_INT,            INPUT_INT,
PRIVATE_INPUT,         PRIVATE_OUTPUT,
OPEN_CHAN,             CLOSE_CHAN
```

Branching

Branching is supported by the following instructions `JMP`, `JMPNZ`, `JMPEQZ`, `EQZINT`, `LTZINT`, `LTINT`, `GTINT`, `EQINT`, and `JMPI`.

Other Commands

The following byte codes are for fine tuning the machine

- `REQBL` this is output by the compiler to signal that the tape requires a minimal bit length. This forces the runtime to check the prime p satisfies this constraint.
- `CRASH` this enables the program to create a crash, if the programmer is feeling particularly destuctive.
- `RAND` this loads a pseudo-random value into a clear register. This is not a true random number, as all parties output the same random number at this point.
- `RESTART` which restarts the online runtime. See Section 7 for how this intended to be used.
- `CLEAR_MEMORY` which clears the current memory. See Section 7 for more details on how this is used.
- `CLEAR_REGISTERS` which clears the registers of this processor core (i.e. thread). See Section 7 for more details on how this is used.
- `START_TIMER` and `STOP_TIMER` are used to time different parts of the code. There are 100 times available in the system; each is initialized to zero at the start of the machine running. The operation `START_TIMER` re-initializes a specified timer, whereas `STOP_TIMER` prints the elapsed time since the last initialization (it does not actually reinitialise/stop the timer itself). These are accessed from MAMBA via the functions `start_timer(n)` and `stop_timer(n)`. The timers use an internal class to measure time command in the C runtime.

New IO Class

A major change in SCALE over SPDZ is the way input and output is handled in the system to an outside data source/sync. In SPDZ various hooks were placed within the compiler/bytecode to enable external applications to connect. This resulted in code-bloat as the run-time had to support all possible ways someone would want to connect.

In SCALE this is simplified to a simple IO procedure for getting data in and out of the MPC engine. It is then up to the application developer to write code (see below) which catches and supplies this data in the way that is wanted by their application. This should be done without needing to modify then bytecode, runtime system, or any compiler being used.

Adding your own IO Processing

We identify a number of different input/output scenarios which are captured in the C++ abstract class

```
src/Input_Output/Input_Output_Base.h
```

To use this interface, an application programmer will have to write their own derived class, just like in the example class

```
src/Input_Output/Input_Output_Simple.h
```

given in the distribution. Then to compile the system they will need to place the name of their derived class the correct place in the main program,

```
src/Player.cpp
```

including any configuration needed. To load your own IO functionality you alter the line

```
unique_ptr<Input_Output_Simple> io(new Input_Output_Simple);
```

To configure your IO functionality you alter the line

```
io->init(cin,cout,true);
```

Internally the IO class can maintain any number of "channels" for each of the various operations below. The runtime bytecodes can then pass the required channel to the IO class; if no channel is specified in the MAMBA language then channel zero is selected by default (although for the `input_shares` and `output_shares` commands you *always* need to specify a channel. Channels are assumed to be bidirectional, i.e. they can communicate for both reading and writing. Note, these are logical channels purely for the IO class; they are nothing to do with the main communication channels between the players.

Types of IO Processing

In this section we outline the forms of input and output supported by the new IO system. All IO instructions must be executed from thread zero, this is to ensure that the order of IO functions is consistent across all executions.

Private Output

To obtain an $\mathbb{F}_p$ element privately to one party one executes the bytecode

```
PRIVATE_OUTPUT.
```

These correspond to a combination of the old SPDZ bytecodes `STARTPRIVATEOUTPUT` and `STOPPRIVATEOUTPUT`. The instruction enables a designated party to obtain output of one of the secretly shared variables, this is then passed to the sytem by a call to the function `IO.private_output_gfp(const gfp& output,unsigned int channel)` in the C++ IO class.

Private Input

This is the oppposite operation to the one above and it is accomplised by the bytecode

```
PRIVATE_INPUT.
```

This correspond to the old SPDZ bytecodes `STARTINPUT` and `STOPINPUT`. The instruction enables a designated party to enter a value into the computation. The value that the player enters is obtained via a call to the member function `IO.private_input_gfp(unsigned int channel)` in the C++ IO class.

Public Output

To obtain public output, i.e. the output of an opened variable, then the bytecode is `OUTPUT_CLEAR`, corresponding to old SPDZ bytecode `RAWOUTPUT`. This output needs to be caught by the C++ IO class in the member function `IO.public_output_gfp(const gfp& output,unsigned int channel)`.

For the same functionality but for `regint` type we have that the bytecode is `OUTPUT_INT`. This output needs to be caught by the C++ IO class in the associated member function `IO.public_output_int(...)`.

Public Input

A clear public input value is something which is input to *all* players; and must be the same input for all players. This hooks into the C++ IO class function `IO.public_input_gfp(unsigned int channel)`, and corresponds to the bytecode `INPUT_CLEAR`. This is a bit like the old SPDZ bytecode `PUBINPUT` but with slightly different semantics. For the `regint` type this becomes the bytecode `INPUT_INT` and the C++ IO class function `IO.public_input_int(...)`.

Any derived class from `Input_Output_Base` needs to call the function `IO.Update_Checker(y,channel)` from within the function `IO.public_input_gfp(...)`, or the function `IO.public_input_int(...)`. See the example given in the demonstration `Input_Output_Simple` class provided.

Share Output

In some situations the system might want to store some internal state, in particular the shares themselves. To enable this we provide the `OUTPUT_SHARE` bytecode (corresponding roughly to the old `READFILESHARE` bytecode). The IO class can do whatever it would like with this share obtained. However, one needs to be careful that any usage does not break the MPC security model. The member function hook to deal with this type of output is the function `IO.output_share(const Share& S,unsigned int channel)`.

Share Input

Finally, shares can be input from an external source (note they need to be correct/suitably MAC'd). This is done via the bytecode `INPUT_SHARE` and the member function `IO.input_share(unsigned int channel)`. Again the same issues re careful usage of this function apply, as they did for the `OUTPUT_SHARE` bytecode.

Other IO Processing

Opening and Closing Channels

As some IO functionalities will require the explicit opening and closing of specific channels we provide two functions for this purpose; `IO.open_channel(unsigned int n)` and `IO.close_channel(unsigned int n)`. These correspond to the bytecodes `OPEN_CHAN` and `CLOSE_CHAN`. For our default IO functionality one does not need to explicitly call these functions before, or after, using a channel. However, it is good programming practice to do so, just in case the default IO functionality is replaced by another user of your code. The `open_channel` command returns a `regint` value which could be used to signal a problem opening a channel.

Trigger

There is a special function `IO.trigger(Schedule& schedule)` which is used for the parties to signal that they are content with performing a `RESTART` command. See Section 7 for further details.

Debug Output

There is a function `IO.debug_output(const stringstream &ss)` which passes the responsed from the `PRINTxxx` bytecodes. In the default instantiation this just sends the contents of the `stringstream` to standard output.

Crashing

The bytecode `CRASH` calls the function `IO.crash(unsigned int PC, unsigned int thread_num)`. This enables the IO class to be able to process any crash in an application specific way if needs be. If the `IO.crash` command returns, as opposed to causing a system shut-down, then the `RESTART` instruction will be called immediately. This means an online program can crash, and then a new one can be restarted without loosing all the offline data that has been produced.

MAMBA Hooks

These functions can be called from the MAMBA language via, for `a` of type `sint`, `b` of type `cint` and `c` of type `regint`.

```
# Private Output to player 2 on channel 0 and on channel 11
a.reveal_to(2)
a.reveal_to(2,11)

# Private Input from player 0 on channel 0 and on channel 15
a=sint.get_private_input_from(0)
a=sint.get_private_input_from(0,15)

# Public Output on channel 0 and on channel 5
b=cint.public_output()
b.public_output(5)

# Public Input on default channel 0 and on channel 10
b=cint.public_input()
b=public_input(10)

# Share Output on channel 1000
output_shares(1000,[sint(1)])

# Share Input on channel 2000
inp=[sint()]
input_shares(2000,*inp)

# Regint input and output on channel 4
open_channel(4)
e=regint.public_input(4)
e.public_output(4)
close_channel(4)
```

The `IO.trigger(Schedule& schedule)` is called only when a `restart()` command is executed from MAMBA.

Programmatic Restarting

Because, unlike in SPDZ, the offline and online phases are totally integrated, it can takes a long time for the queues of pre-processed triples to be full. This means that quickly starting an instance to run a short program can be very time-consuming. For reactive systems in which the precise program which is going to be run is not known until just before execution this can be a problem. We therefore provide a mechanism, which we call `RESTART`, to enable a program to *programatically* restart the online runtime, whilst still maintaining the current offline data. In the bytecode this is accessed by the op-code `RESTART` and in MAMBA it is accessed by the function `restart(Schedule &schedule)`. Both must be called in online thread zero.

The basic model is as follows. You start an instance of the SCALE engine with some (possibly dummy) program. This program instance will define the total number of online threads you will ever need; this is because we do not want a restart operation to have to spawn new offline threads. You then ensure that the last operation performed by this program is a `RESTART`/`restart(Schedule &schedule)`. This *must* be executed in thread zero, and to avoid undefined behaviour *should* only happen when all other thread programs are in a wait state.

After calling `RESTART`/`restart(Schedule &schedule)` the runtime waits for a trigger to be obtained from *each* player on the IO functionality (see Section 6). This trigger can be anything, but in our default IO class it is simply each player needing to enter a value on the standard input. The reason for this triggering is that the system which is calling SCALE *may* want to replace the underlying schedule file and tapes; thus enabling the execution of a new program entirely.

The specific `restart(Schedule &schedule)` function in the IO functionality will now be executed. In the provided variant of IO functionality, given in `Input_Output_Simple`, the schedule file is reloaded from the *same directory* as the original program instance. This also reloads the tapes etc.

You could implement your own `restart(Schedule &schedule)` function in your own IO functionality which programmatically alters the underlying schedule to be executed, via the argument to `restart`. This enables a more programmatic control of the SCALE engine.

To see this in operation we provide the following simple examples, in `Programs/restart_1/restart.mpc` and `Programs/restart_2/restart.mpc`. To see these in operation execute the following commands from the main directory (assuming you are using the default IO functionality,

```
\cp Programs/restart_1/restart.mpc Programs/restart/
./compile.py Programs/restart
```

Now run the resulting program as normal, i.e. using `Programs/restart` as the program. When this program has finished it asks (assuming you are using the default IO class) for all players to enter a number. *Do not do this yet!* First get the next program ready to run by executing

```
\cp Programs/restart_2/restart.mpc Programs/restart/
./compile.py Programs/restart
```

Now enter a number on each players console, and the second program should now run. Since it also ends with a `RESTART` bytecode, you can continue in this way forever.

Memory Management While Restarting

As part of the restart functionality we also add some extra commands to aid safety of the overall system.

The first of these is `clear_memory()`, which corresponds to the bytecode `CLEAR_MEMORY`. What this does is initialize all system memory to zero. Thus before calling a `restart()` one can zero out all the memory, meaning the memory cannot be used by the next program to be executed. This might be useful when you want to avoid the next program to be run accessing memory from the current state. This could be a concern if you are unsure if the byte-code to be executed does not contain invalid memory accesses by mistake. Thus `CLEAR_MEMORY` can be considered as an instruction which avoids security errors due to programmer mistakes.

Note, that the `clear_memory()` MAMBA instruction can only be used when your program is compiled with `-M` as it relies on the memory operations being compiled in the correct order. Also note that if it is executed in one thread,

then this means the memory in any other thread will now be not what is expected. So the instruction should only really be executed in thread zero, once all other threads have finished. A demonstration of the `clear_memory()` command called from MAMBA is available in the program directory `/Programs/mem_clear_demo`.

The second command is `clear_registers()`, which corresponds to the bytecode `CLEAR_REGISTERS`. This resets the registers in the calling processor to zero; it does nothing to the register files in the other threads. The purpose is again to avoid mistakes from programmers. One should consider emitting this instruction at the end of the code of every thread. However, the instruction ordering when compiled from MAMBA *may* be erratic (we have not fully tested it); thus its usage should be considered experimental. The exact order of it being emitted can be checked by running the player with a negative verbose number (which outputs the instructions being executed in the order they are found). Again the usage is demonstrated in the program directory `/Programs/mem_clear_demo`.

The MAMBA Programming Language

Getting Started

Setup

Dependencies To run the compiler, the following Python packages are required:

- `gmpy2` — for big integer arithmetic
- `networkx` — for graph algorithms

If these are not available on your system, you should be able to install them via `easy_install`:

```
easy_install --install-dir <path> <package-name>
```

Directory structure The Compiler package should be located in the same root directory as the `compile.py` script, alongside a folder Programs with sub-folders for actual programs. The final directory structure should look like:

```
root_dir/
|-- compile.py
|-- Compiler/
|   |-- ...
|   |-- ...
|-- Programs/
|   |-- some_program/
|   |   |-- some_program.mpc
```

MAMBA programs have the extension `.mpc` and each MAMBA program is placed in its own sub-directory within the directory `Programs`.

Compilation Compiling a program located in Programs/some_program/some_program.mpc is performed by the command:

```
compile.py [options] Programs/some_program
```

The compiled bytecode and schedule files are then stored in the same directory as the main program.

The options available are as follows (there are a number of other options, related to re-ordering of instructions and compiler testing, but we do not document these as they are not as relevant for the casual user).

`-n --nomerge`

Don't optimize the program by merging independent rounds of communication.

`-o --output <name>`

Specify a prefix `name-` for output bytecode files (defaults to the input file name).

`-d --debug`

Keep track of the call stack and display when an error occurs.

-c --comparison <type>

Specify the algorithm used for comparison of secret integers. Can be one of:

- `log`: logarithmic rounds.
- `plain`: constant rounds.

Default is `log`.

-a --asm-output

Produces ASM output file for potential debugging use.

-D --dead-code-elimination

This eliminates instructions which produce an unused result.

There are a number of other options which are mainly for testing or developers purposes. These are given by

```
-r --noreorder

-M --preserve-mem-order

-u --noreallocate

-m  -max-parallel-open <MAX_PARALLEL_OPEN>

-P --profile

-C --continuous

-s --stop
```

We refer the reader to the compiler help for usage of these compiler options.

Understanding the compilation output

The compilation output includes important information related to your code. It is an intuitive collection of parameters that can be easily interpreted. We include a short analysis of the compilation output and a basic description of common output parameters based on the Simple Example from Section 4. The output is meant to be an informative revision on the different tasks performed by the compiler. In case you have correctly followed the instructions for compilation, the output should resemble the following:

```
Compiling program in Programs/tutorial
tutorial
p = 34359738368
Prime size: 32
Default bit length: 24
Default statistical security parameter: 6
Compiling file Programs/tutorial/tutorial.mpc
Compiling basic block tutorial-0--0
Compiling basic block tutorial-0-begin-loop-1
Compiling basic block tutorial-0-end-loop-2
Compiling basic block tutorial-0-if-block-3
Compiling basic block tutorial-0-else-block-4
Compiling basic block tutorial-0-end-if-5
```

```
Processing tape tutorial-0 with 6 blocks
Processing basic block tutorial-0--0, 0/6, 4386 instructions
Program requires 7 rounds of communication
Program requires 440 invocations
Processing basic block tutorial-0-begin-loop-1, 1/6, 21 instructions
WARNING: Order of memory instructions not preserved, errors possible
Program requires 1 rounds of communication
Program requires 1 invocations
Processing basic block tutorial-0-end-loop-2, 2/6, 18 instructions
Program requires 1 rounds of communication
Program requires 1 invocations
Processing basic block tutorial-0-if-block-3, 3/6, 2 instructions
Processing basic block tutorial-0-else-block-4, 4/6, 2 instructions
Processing basic block tutorial-0-end-if-5, 5/6, 14 instructions
Program requires 1 rounds of communication
Program requires 1 invocations
Tape register usage: defaultdict
(<function <lambda> at 0xf27c80>, {'c': 1278, 'ci': 10, 's': 3793})
modp: 1278 clear, 3793 secret
Re-allocating...
Register(s) [ci0] never used, assigned by 'popint ci0' in <omitted>
Compile offline data requirements...
Tape requires 211 triples in modp, 100 squares in modp, 184 bits in modp
Tape requires prime bit length 32
Program requires:
{('modp', 'triple'): 211, ('modp', 'square'): 100, ('modp', 'bit'): 184}
Cost: 0.102230483271
Memory size: defaultdict(<function <lambda> at 0xf27b18>,
{'c': 8192, 'ci': 8192, 's': 8292})
Compiling basic block tutorial-0-memory-usage-6
Writing to Programs/tutorial/tutorial.sch
Writing to Programs/tutorial/tutorial-0.bc
Writing to Programs/tutorial/tutorial-0.bc
```

The compilation output in this case corresponds to a 3-party scenario using Shamir Secret Sharing for both, online and offline phases. The output introduces first some program level parameters, followed by the rolling of the tape (converting all operations to bytecode instructions), and offline data requirements. We now proceed to analyze the output more into detail.

Program Level Parameters

`p`

This is the modulus of the finite field which secret sharing is defined over. It is defined through `Setup.x`. The program `Setup.x` then stores it in the `Data` folder, more specifically in the file `SharingData.txt`

`Prime Size`

Is the bit size of the prime p above.

`Default Bit Length`

Is the default maximum bit length of emulated integer values inputs, for operations like integer comparison below. In

particular see the value k below in Section 8.2.1. Because of mechanisms such as comparisons, implemented under statistical security parameters, the input size has to be adjusted so some power of 2 smaller than the modulus.

Default Statistical Security

By default, some bits are reserved for the statistical security of operations such as comparisons. This reduces the actual input size. The field size has to be greater than the inputs bit-length k plus the statistical security bit-length κ such that no overflow occurs. When the prime p is 128-bits in length the default values of $\kappa = 40$ and $k = 64$ are chosen. These can be altered by using the commands

```
program.security = 100
program.bit_length = 20
```

Remember, the requirement is that $k + \kappa$ must be less than the bit length of the prime p. These are the parameters for the base `sint` data type when we intepret the value it holds as an integer, as opposed to an element modulo p. There are also κ statistical security parameters associated with the `sfix` and `sfloat` data types; whose default values are also 40. These can be set by

```
sfix.kappa=50
sfloat.kappa=50
```

Compilation comments regarding the tape enrollment:

The compiler output showcases a typical expample of instructions from the compiler. They reflect the tasks being performed while writting on the operations tape.

Compiling basic block

Signals the start of compilation of a new basic block on the tape. It also adds operations to such block.

Processing tape

Signals the start of the optimization of the contents of the tape. This includes merging blocks to reduce rounds. It also elminates dead code.

Processing basic block

While Processing tape, blocks get optimized, code reviewed to eliminated dead code and also merged.

Program requires X rounds of communication

Total amount of communication rounds (latency) of the program after compilation and its optimization.

Program requires X invocations

Total amount of multiplications needed for the compiled program (amount of work) to process secret data.

Offline data Requirements:

Tape requires X triples in modp, Y squares in modp, Z bits in modp

Signal a recount of the amount of different triples needed for the protocol execution.

Memory Size

This is an estimation of the memory allocation needed to execute the protocol.

Writing Programs

Programs for the compiler are written in a restricted form of Python code, with an additional library of functions and classes specifically for MPC functionalities. The compiler then executes the Python code, and the MPC library functions cause this to output bytecode suitable for running on the virtual machine. Whilst the compiler will, in theory, accept any valid Python code as input, some language features may not work as expected. The key point is that any native Python constructs are evaluated at compile time, the effect being that all Python loops will be unrolled and all functions inlined. So to get the most out of the compiler and write efficient programs, it is important to understand a little about the internals of the underlying implementation.

This section documents the library functions available to an MPC source code file, and how to perform basic operations on the associated data types. Basic knowledge of the workings of the Virtual Machine (chapter 5) and the Python language are assumed. The definitions of all library functions are given in `library.py`.

Data Types

A complete set of all types and their methods can be found in the html file in

```
$(HOME)/Documentation/Compiler_Documentation/index.html
```

under the class `types`. Details of advanced algorithms can be found under the heading `Files`.

The compiler uses separate classes for clear and secret integer, fixed point and floating point data types. The integer classes `cint` and `sint` correspond directly to a single clear or secret register in the virtual machine. Integer types lie in the range $[-2^{k-1}, \ldots, 2^{k-1}]$, for some parameter t with $p > 2^k$. Internally, these are represented in $\mathbb{F}_p$ by the mapping $x \mapsto x \bmod p$, so will not wrap around modulo 2^k by default. However, all clear values are reduced into the proper range before comparison operations, to ensure correct results. The parameters k and κ are chosen depending on the field size x as follows:

- $x < 48 : k = 24$
- $48 \leq x < 85 : k = 32$
- 128-bit field: $k = 64$

These choices allow some room for secure comparison with a statistical security parameter of at least 30 bits, apart from the 32-bit field, which should only be used for test purposes. More precisely, the default security parameter is chosen as follows:

- `prime_bit_size < 48`:
  Security parameter κ is fixed to 6 by Default.
- `48 < prime_bit_size  < 85`:
  Security parameter κ is fixed to 32.
- `prime_bit_size > 85`:
  Default security parameter κ is fixed to 40. Thus this will be the security parameter bit size for 128 bit finite fields (which is the recommended prime size for MAMBA programs).

The default values of k and κ can be modified using the commands described above if desired.

There is also `regint` class, which works like normal 32-bit integer, and it can be used for example as array index, since values used in range-loops and some of while-loops are `regints`. Also value in MemValue(described later) is stored as `regint`. Operations on regint are faster then on cint. `regint` values map directly to the integer values in the Bytecodes, where as `cint` map to the clear type in the Bytecode.

Fixed and floating point types can be instantiated using the `sfix` and `sfloat` classes, and are described in later part of documentation about Advanced Algorithms.

Creating data

sint(value)

Loads the integer or clear integer `value` into a secret register and returns the register.
Example:

```
i = sint(5)
```

Note: value type can be: regint, int, sint and cint.

cint(value)

Loads the integer `value` into a clear register and returns the register.
Example:

```
i = cint(5)
```

Note: value type can be: regint, cint and int.

regint(value)

Loads the integer `value` into a `regint` register and returns the register.
Example:

```
i = regint(5)
```

Note: value type can be: regint and int.

sfix(value)

It instantiates an `sfix` register based on `value`. In case `value` is a publicly available value, it loads it as a secret shared fix point number. In case is a secret shared integer, it places `value` to the mantissa `v` of the `sfix` instance.
Example:

```
i = sfix(5.5)
```

Note: value type can be of any clear or secret numeric type.

cfix(value)

It instantiates an `sfix` register based on `value`. Note that `value` is a publicly available value, it loads it as a secret fix point.
Example:

```
i = cfix(5.5)
```

Note: value type can be of any clear numeric type.

sfloat(value)

It instantiates an `sfloat` register based on `value` and returns an `sfloat` object. Its instantiation logic mimics its fixed point counterpart. Current implementation supports basic logical and arithmetic operations with all data-types. We describe the formatting later in this section.
Example:

```
i = sfloat(5.5)
```

Note: value type can be: int, float, sint, sfloat and sfix.

cfloat(value)

It is the clear register counterpart of sfloat. It instantiates a cfloat register based on value and returns a cfloat object. It mimics sfloat number representation, and its main function is to serve as an interface when operating between sfloat instances and clear types and registers.
Example:

```
i = cfloat(5.5)
```

Note: value type can be: int, float, sint, sfloat and sfix.

load_int_to_secret_vector(vector)

Loads a list of integers vector into a vectorized sintand operates on the vector as in a single instance (vectorized instructions).
Example:

```
A= [1, 2, 3]
SA= load_int_to_secret(A)
print_ln("Values from A: \%s", SA) #the output is 123.
```

load_secret_mem(address)
load_clear_mem(address)

Returns the value stored in memory address of the according type. The value had to be previously stored on SCALE. Memory in this context refers to a data-storage from SCALE and not physical memory. Users select a memory address when storing data, and the same address needs to be used to extract it. The calls can be implemented as follows:
Example:

```
i =cint(5)
i.store_in_mem(0)
ci= load_secret_mem(i)
%print_ln("Values from A: \%s", ci) #the output is 5 and type cint.
```

Note: address type can be: regint, int and cint. It does the same as functions below and can store any data-type.

x.store_in_mem(address)

Stores register x into a given address of the appropriate memory type. This basically implies it can be later retrieved by a load_mem instruction. Memory addresses are decided by the user and are stored by the compiler on SCALE.
Example:

```
i =sint(5)
i.store_in_mem(0)
si= load_secret_mem(i)
print_ln("Value stored on memory address 0: \%s", si.reveal()) #the output is 5.
```

Note: address type can be: regint, int and cint. Note: operation supported for sint, cint, sfix and sfloat.

x.load_mem(address)

Loads the value stored in memory address to the register. The address is selected during the invocation of a store_in_memcall.
Example:

```
i =sint(5)
i.store_in_mem(0)
si= sint.load_mem(0)
print_ln("Value stored on memory address 0: \%s", si.reveal()) #the output is 5.
```

Note: address type can be: regint, int and cint. Note: operation supported for sint, cint, sfix and sfloat.

`x.reveal()`

Opens the value in a secret register and returns a clear data-type, also referred as register for the now publicly available register.
Example:

```
si= sint(5)
print_ln("Value in the clear from si: \%s", si.reveal()) #the output is 5.
```

Note: x type can be: sint, sfix, and sfloat.

Operations on Data Types

Most of the usual arithmetic operators ($+, -, *, /, \%, <<, >>$) can be used with clear and secret integer types, implemented with operator overloading. This extends also to the case where one operand is a Python integer and one is a clear or secret register. Exponentation ($**$) is implemented for immediate exponents and immediate base 2. For secret registers, there are some limitations:

- The modulo operation can only be computed for immediate powers of two.
- Division by secret registers is not possible.
- $>>$ is not implemented for clear registers as the right operand. All other shift operations are implemented

Boolean operations $\&, \wedge, |, \sim$ are supported on clear registers, which function as expected for integers of bit length t.

Note that upon compilation all of the above operators return a new register – it is not possible to directly modify the value of a specific register from MAMBA. For example, in the following code, the $\star$ operator creates a fresh clear register, and assigns the result of the multiplication into this. The name `c` is then bound to the new register, and the previous register is no longer accessible. The reason for this is to simplify the register allocation procedure, details of which are in section 8.4.3.

When operating between different types, the result will be secret if one of the operands was a secret register. Additionally, as in any other conventional programming language, the returned type will correspond to the type of the strongest precision.

The goal of providing clear registers, is to provide the means to the user to interact and operate with secret values. Our examples make use of secret registers, but as mentioned, We now provide some examples for some basic operations. All of these operations are supported also in between secret and clear registers. As a cautionary note, although supported, multiplication between fixed and secret float registers might cause some loss of precision, hence discouraged.

Multiplication: As before, multiplication is supported for secret and non-secret, integer and fractional data-types. They can be invoked as follows:

```
c = sint(12)
c = c * c
f = sfix(12)
f = f * f
d = c * f
g = sfloat(12)
g = g * g
h = c * g
i = f * g
```

In this small example, we can see how to multiply among different and the same data types. As a result $c = 12^2$, $f = 12.0^2$, $d = 12.0^4$, $g = 12.0^2$, $h = 12.0^4$, and $i = 12.0^4$. Note that `d` is of type `sfix`, whereas `g`, `h` and `i` are of type `sfloat`.

Additions and Substractions: We follow the same principle as before:

```
c = sint(12)
c = c + c
f = sfix(12)
f = f - f
d = c + f
g = sfloat (12)
g = g - g
h = c + g
i = f + g
```

As before in this case the type fo `d` is `sfix`, whereas the type of `g`, `h` and `i` is `sfloat`

Division and Modulus: The Compiler can handle also division and modulus operations. However, these are not generic operations. Let us start by showing some basic constructions for division:

```
c= sint(12)
c= c / 3
f = sfix(3)
f = f/2
g = sfloat(3)
g = g/2
```

The results for `c`, `f` and `g` are 4, 1.5 and 1.5. Indeed, this is a quite natural way to call division on integers and decimal types. But it has to be noted that the division on integers is constructed as a multiplication between the numerator and the multiplicative inverse of the denominator. This is because of the modulo arithmetic the protocols are built upon. Moreover, modulus operations are indeed somewhat different as we may see:

```
c = sint(2)
d = sint(3)
c = c \% 2
d = d \% 2
```

The operations will return, in the case of `c` 0 and for `d` 1. As with divisions, there are some observations: modulo operations can only be performed to powers of 2. Furthermore, note that modulo operations cannot be performed on non-integer types: `sfix` and `sfloat`.

Shift operations : We have included bit shifts operations for our basic integer data-types. Such shifts do not work on fractional types.

```
c = sint(2)
d = c << 1
e = e >> 1
```

In this case, the output of `d` is 4 as expected and from `e` 1. Note that bit shifts only work on integer data-types.

Exponentiation: We also provide a built-in exponentiation operator for exponentiation over integer and fractional data-types ($**$) when the base is secret shared. For the fractional case, This overload provides a simple implementation for successive multiplications. We provide more comprehensive protocols for exponentiation of fractional inputs in the following sections. Note that, for integer registers, the exponent can be also secret shared, however the base and the exponent cannot be both secrets on the same operation execution. We show how these cases can be easily implemented:

```
a = sint(2)
b = cint(3)
f = sfix(1.5)

c = a**b  #returns $2**3$
c = b**a  #returns $3**2$
f = f**2  #returns $1.5**2$
```

NOTE: For the reasons exposed in this section, and with respect to fractional types, the exponent has to be of *native Python integer* type. This is not the case for integer types. We invite the reader to revise the Advance Protocols section for a revision of alternatives to compute the exponent on fractional data-types.

Comparisons (Inequality Tests) : We have in-built operators for comparisons as well. Indeed, comparison of secret values is supported, and returns a secret output containing 0 (false) or 1 (true). They work on both integer and fractional data-types. In this sense, they can be used as follows:
Example:

```
a = sint(1)
b = sint(2)
c = a < b
      d = sfloat(3)
      f = a < d
```

Clear Data-types. Before comparison of clear integers (`cint`) is done, all operands are reduced into the range $[-2^{t-1}, \ldots, 2^{t-1}]$. Note that the virtual machine has no boolean type, so any comparison operation returns a clear register value of 0 (false) or 1 (true).
Secret Data-types. The bit length of the comparison operators defaults to the parameter t, which is set-up by the compiler based on the input modulus, but if a different length is required it can be specified with the following functions:

```
x.less_than(y, bit_length, sec_param)
x.greater_than(y, bit_length, sec_param)
x.less_equal(y, bit_length, sec_param)
x.greater_equal(y, bit_length, sec_param)
x.equal(y, bit_length, sec_param)
x.not_equal(y, bit_length, sec_param)
```

The following simple example is applicable to all these methods:

```
a= sint(2)
b= sint(1)
c= a.less_than(b,128,40)
```

The output in this case is 1 as expected. The 2 last parameters are not obligatory.

Loading preprocessing data and sources of randomness

For some programs, data from the preprocessing phase of SPDZ may be required (as source of randomness). Note that some kinds of randomness can be generated during the program's offline phase. The function is implemented over the `sint` class and cannot be accessed through any other data-type. Randomness can be accessed in the following ways:

`sint.get_random_triple()`

Returns three secret registers a, b, c such that $a \cdot b = c$.

```
a,b,c =sint.get_random_triple()
print_ln("these 2 results are equal %s, %s", (a*b).reveal() c.reveal())
```

The code above will show in this case c and the result of the multiplication of a and b, that should be equal.

sint.get_random_bit()

Returns a secret value b, with value in $\{0, 1\}$. The function can be used as follows:

```
b =sint.get_random_bit()
print_ln("the result is either 0 or 1 %s", b.reveal())
```

The code will get a secret shared random bit.

sint.get_random_square()

Returns two secret values a, b such that $a^2 = b$. Let us see the following example:

```
a,b =sint.get_random_square()
print_ln("these 2 results are equal %s, %s", (a*a).reveal() b.reveal())
```

The code will output the value of $a \cdot a$ versus b, which are equal values.

sint.get_random_int(nbits)

Parameters:
`nbits`: bitsize of the secret shared randomness to be produced. Must be a native Python integer variable (not any MAMBA data-type).
The function returns a random integer of size `nbits`. This does not come directly from preprocessed data, but instead loads `nbits` random bits and uses these to create the random secret integer. The function can be used as follows:

```
a =sint.get_random_bit(5)
print_ln("the result is smaller that 2^a %s", a.reveal())
```

The output is a bounded integer by 2^5. Note that `nbits` can be a public input by the parties.

Printing

We provide the following functions to printout outputs:

print_str(string, *args)
print_ln(string, *args)

Both of the functions do the same thing, only difference is that print_ln adds $newline$ after execution. Arguments are inserted into string in places of $\%$s respectively.
Example:

```
x = 13
y = cint(5)
z = sint(x)
print_ln("x = %s, y = %s, z = %s", x, y, z.reveal())
```

Will print x = 13, y = 5, z = 13.

How to print Vectorized data

Suppose we have two vectorized data types such as sint, sfix, or sfloat. After we have done some operations then we want to print them. We will demonstrate here with sints:

```
n = 10
x = sint(13, size=n)
y = sint(25, size=n)
z = x * y # this is now 325 on each of the 10 slots
```

```
z_array = sint.Array(n) # allocate memory to copy z
z.store_in_mem(z_array.address) # now z_array is full of z's data

for i in range(n):
  print_str("%s ", z_array[i].reveal())
```

This might seem useless - why do we want to do the same multiplication but 10 fold? Well we can put different data in each slot by first dumping same length arrays to x and y and then multiplication is going to be faster due to SIMD.

Advanced Data Type Explanation

class sfix

The `sfix` class is based on Catrina and Saxena's work on processing fixed point precision arithmetic within MPC [CS10]. Basically, we use an integer mapping to encode a rational element represented up to certain precision. Given integer values v, $\beta = 2$ and f, in SCALE, we can represent a rational value as follows:

$$x \approx v \cdot \beta^f.$$

The results might be slightly different given the truncation of the information contained by the number. You can think of f as the bitwise precision for the given fixed point representation.

Data Components: The following are the most important data stored by the class:

v

Accessed by: `Default`.
Type: `sint`.
S Stores a register of type `sint` on the $\{-2^k - 1, 2^k - 1\}$ interval, encoding of the rational original value.

f

Accessed by: `Default`.
Type: `int`.
Stores the bitwise bit precision of the value to be stored by the instance.

k

Accessed by: `Default`.
Type: `int`.
Defines the mapping space. Basically, we can map numbers from -2^{k-1} to 2^{k-1}, such that $k - f \geq 0$ so that no overflow occurs.

Special Operations:

sfix.set_precision(f, k = None)

Accessed by: `Default`.
Parameters:

- `f`: New bitwise precision value.
- `k`: New bitwise k, interval. `default: NONE`.

Returns: No return value.
Description: Let you change the precision for the `sfix` type. The requirement is that $f < k$ and $2 \cdot k + \kappa < \log_2 p$. By default the values $(f, k) = (20, 41)$ are chosen internally in the compiler. It is used to fix the default precision on `types.py`.
Example: To change the precision of `sfix`:

```
fixed_f=20
fixed_k=41
sfix.set_precision(fixed_f, fixed_k)
```

sfix.load_int(v)

Accessed by: `Default`.
Parameters:

`v`: Integer value to load into a `sfix` instance.

Returns: No return value.
Description: It is used to do explicit initialization of an `sfix` value from any integer instantiation value.
Example: To initialize a `sfix` value with an integer:

```
a = sfix()
a.load_int(5)
b = a*3.0
print_ln(b.reveal(b))  #the output is 15
```

sfix.conv()

Accessed by: `Default`.
Parameters: N/A
Returns: `sint` value corresponding to the mantissa (`v` value) mapping.
Description: Function obtains the mantissa (`v` value) mapping of the sorted value by the instance.
Example: To obtain the value of the mantissa:

```
a =sfix()
a.load_int(4.5)
v = a.conv()
print_ln(v.reveal())  # the output is 4718592
```

sfix.sizeof()

Accessed by: `Default`.
Parameters: N/A
Returns: Python native `int` value corresponding to the size of memory slots occupied by the instance.
Description: It returns the global_vector_size times 1.
Example: To obtain reciprocal you can execute:

```
a =sfix()
a.load_int(4.5)
r = a.sizeof()
print_ln(r)  # the output is 1. By Default the global_vector
_size is set to 1.
```

sfix.compute_reciprocal()

Accessed by: `Default`.
Parameters: N/A
Returns: `sfix` value corresponding to the reciprocal of the instance.
Description: It calculates and returns the reciprocal of an instance in secret shared form, in whatever precision is supported by `sfix`.
Example: To obtain reciprocal you can execute:

```
a =sfix()
a.load_int(4.5)
r = a.compute_reciprocal()
print_ln(r.reveal())  # the output is 0.222222
```

Observations:

- The class should not be initialized from a `sint` object. Application level invocations should use the function `load_int`.
- The default precision and mantissa size, for `sfix`, are fixed assuming at least a 128 bit modulus. The values $(f, k) = (20, 41)$ are fixed directly on `types.py` and where fixed taken size restrictions into account. Note that by default the internal parameter $kappa$ is fixed to 40 bits.

class cfix

We also provide users with an equivalent data type for clear inputs. It represents rational numbers in the same way, using Catrina and Saxena's [CS10]. Only this time, v, is **not** secret shared. This basically means that an instance of the same number, on `sfix` and `cfix`, would encode the number in the same fashion, as long as they are using the same precision parameters. Note that since `cfix` values are held in the clear there is no notion of a statistical security parameter κ for `cfix` values.

Data Components: The following are the most important data stored by the class:

v

Accessed by: `Default`.
Type: `int`.
Stores a register on the $\{-2^k - 1, 2^k - 1\}$ interval, encoding of the rational original value.

f

Accessed by: `Default`.
Type: `int`.
Stores the bitwise bit precision of the value to be stored by the instance.

k

Accessed by: `Default`.
Type: `int`.
Defines the mapping space. We can map numbers from -2^{k-1} to 2^{k-1}, such that $k - f \geq 0$ so that no overflow occurs.

Special Operations:

cfix.set_precision(f, k = None)

Accessed by: Default.
Parameters:

- f: New bitwise precision value.

- k: New bitwise k, interval. default: NONE.

Returns: No return value.
Description: Let you change the precision for cfix. The precision parameters should be in line with those of cfix, given that this class is used to interface operations between public available values and sfix instances.

By default the values $(f, k) = (20, 41)$ are chosen internally in the compiler. It is used to fix the default precision on types.py.
Example: To change the precision of a cfix:

```
fixed_f=20
fixed_k=41
cfix.set_precision(fixed_f, fixed_k)
```

cfix.load_int(v)

Accessed by: Default.
Parameters:

v: Public integer value to load into this cfix instance.

Returns: No return value.
Description: It is used to do explicit initialization of an cfix value from any integer instantiation value.
Example: To initialize a cfix value with an integer:

```
a = cfix()
a.load_int(5)
b = a*3.0
print_ln(b)  #the output is 15
```

cfix.conv()

Accessed by: Default.
Parameters: N/A
Returns: cint value corresponding to the mantissa (v value) mapping.
Description: Function obtains the mantissa (v value) mapping of the sorted value by the instance.
Example: To obtain the value of the mantissa:

```
a =cfix()
a.load_int(4.5)
v = a.conv()
print_ln(v)  # the output is 4718592
```

cfix.sizeof()

Accessed by: `Default`.
Parameters: N/A
Returns: Python native `int` value corresponding to the size of memory slots occupied by the instance.
Description: It returns the global_vector_size times 1.
Example: To obtain reciprocal you can execute:

```
a =cfix()
a.load_int(4.5)
r = a.sizeof()
print_ln(r)  # the output is 4. By Default the global_vector
_size is set to 1.
```

cfix.compute_reciprocal()

Accessed by: `Default`.
Parameters: N/A
Returns: `cfix` value corresponding to the reciprocal of the instance.
Description: It calculates and returns the reciprocal of the instance, on whatever precision is supported by `cfix`.
Example: To obtain reciprocal you can execute:

```
a =sfix()
a.load_int(4.5)
r = a.compute_reciprocal()
print_ln(r)  # the output is 0.222222
```

Observations:

- A class instance cannot be initialized directly via a secret shared input. In case you are loading an `cint` value, we recommend you to use `load_int`. You should work with a `cfix`, as you would do with a `sfix` type.

- `cfix` by default, uses the same precision and mantissa size as `sfix`. That means it requires atleast a 128 bit modulus. The values $(f, k) = (20, 41)$ are fixed directly on `types.py` and where fixed taken size restrictions into account.

class sfloat

A floating point number x is represented as

$$x \approx (1 - 2 \cdot s) \cdot (1 - z) \cdot v \cdot 2^p.$$

where s is the sign bit, z is a bit to signal a zero or not, v is the *significand* or *mantissa*, and p is the *exponent*. We also maintain a flag err which determines whether some form of error state in the floating point variable has occured (e.g. underflow, overflow, division by zero, taking square roots of negative numbers etc). Note, like all floating point representations such an encoding is an approximation. The reader should note that our encoding does not directly emulate IEEE floating point standards, but tries to mimic them (thus precision etc of results will be different from IEEE standard). Our encoding is in line with the one described in detail by Aliasgari et. al [ABZS13] and used across various complex protocols for mathematical operations on MPC. The values s, z, v, p and err are kept in secret shared format.

We also maintain a statistical security parameter κ associated with an `sfloat` which needs to satisfy

$$2 \cdot \texttt{vlen} + \kappa < \log_2 p.$$

You can change the default sizes for `sfloat` values using the commands.

```
    sfloat.plen=5
    sfloat.vlen=10
    sfloat.kappa=20
\endd{mylisting}
The default values being $8$, $24$ and $40$ respectively.

The $\err$ flag needs to be treated with some care.
For efficiency reasons this can be {\em any} integer, but only
a value of zero represents a valid number.
The flag is initialized to $0$ and this value would only change in case an error is
    produce.
We propagate the error by adding the $\err$ flags of the operating instances.
Thus any positive value indicates an error, and also could, if revealed, reveal how
many errors have occured in a calculation.
Thus before revealing an \verb|sfloat| instance, we first obtain the bit $ b = (err ==
     0)$,
and multiply it by the data components of the instance. So an \verb|sfloat| value
will equal zero if {\em any} error has occured.
That way we guarantee that the output would not leak any information
related to the inputs or any intermediary value during the calculations.

The \verb|sfloat| type supports basic arithmetic and logic operations with
any clear or secret register, as well as standard data-types.
Public values, as well as standard types are implicitly cast to \verb|cfloat|
before operating on an \verb|sfloat| instance. The result will always be of type \verb
    |sfloat|.
It is not recommended to utilize fixed and floating point operations unless
precision loss is taken into account.

\paragraph{Data Components}
The following is a detailed description of the most important member state variables
    of \verb|sfloat|, their behaviour, and properties:
    \func{v}
    \textbf{Accessed by:} \verb|Default|.\\
    \textbf{Type:} \verb|sint|.\\
    Stores the mantissa/significand of the encoding in secret form.
    \func{p}
    \textbf{Accessed by:} \verb|Default|.\\
    \textbf{Type:} \verb|sint|.\\
    Stores the exponent of the encoding in secret form.
    \func{s}
    \textbf{Accessed by:} \verb|Default|.\\
    \textbf{Type:} \verb|sint|.\\
    Stores a $\{0,1\}$ value storing the sign of the instance, where $\sshare{0}$
        means is positive and $\sshare{1}$ otherwise.
    \func{z}
    \textbf{Accessed by:} \verb|Default|.\\
    \textbf{Type:} \verb|sint|.\\
    Stores a $\{0,1\}$ value to flag when the \verb|sfloat| instance's value is zero.
    When this variable takes a value of $\sshare{0}$, it means that, the \verb|sfloat|
         instance is a non-zero value and $\sshare{1}$ otherwise.
    \func{err}
    \textbf{Accessed by:} \verb|Default|.\\
    \textbf{Type:} \verb|sint|.\\
```

```
    Stores a register of type \verb|sint|,
    that serves to flag whether the instance is reporting a numeric error.
    When this variable takes a value of $\sshare{0}$,
    it means that the \verb|sfloat| instance is valid and non-zero otherwise.
    Although not recommended,
    in case of an error, if this value is later opened,
    it would disclose the number of chained operations executed on t
    op its operation tree since the error was produced.

\paragraph{Special Operations:}

%\func{sfloat.load_mem(address, mem_type)}

\func{sfloat.set_error(error)}
      \textbf{Accessed by:} \verb|Default|.    \\
      \textbf{Parameters:}
      \begin{description}
        \item[]\verb|error|:  sets the default precision error.
      \end{description}
      \textbf{Returns:} N/A. \\
      \textbf{Description:} Not to be confused with the
$\err$ flag. The error, in this case, pertains to the precision of the representation
    which is initialized in 0. It can be increased as follows:
      \begin{mylisting}
          cls.error += error - cls.error * error
```

The formulation follows the literature on this topic.
Example: To alter the error you can do the following:

```
a =sfix(1.5)
a.set_error(0.1)
```

sfloat.convert_float(v,vlen,plen)

Accessed by: `Default`.
Parameters:

`v`: data publicly available value (numeric Python data-type) to be transformed into float representation.

`vlen`: bit-lenght of the mantissa that will encode `v`.

`plen`: bit-lenght of the exponential encoding of `v`.

Returns: A tuple composed by the 5 data components of a `sfloat`. It can be used to instantiate a new `sfloat` obtect.
Description: Transforms and secret shares an input in plain text to our floating point representation. Note that both `vlen` and `plen` parameters should be sufficiently large to handle `v`.
Example: An examplle, using would look as follows:

```
float_value= sfloat(9999, 14 ,1 ,0 , 0)
print_ln("The float representation of 9999 is \%s", float_value.reveal())
```

sfloat.sizeof()

Accessed by: `Default`.
Parameters: N/A
Returns: Python native `int` value corresponding to the size of memory slots occupied by the instance.
Description: Note that each `sfloat` instance is composed by 5 different values (v, p, z, s, and err), it returns the `global_vector_size` times 5. In case the instance is vectorized, it would return the vector size time 5.
Example: To obtain reciprocal you can execute:

```
a =sfloat(4.5)
r = a.sizeof()
print_ln(r)  # the output is 5. By Default the
    global_vector
_size is set to 1.
```

class cfloat

The main purpose of this class is to offer interoperability between public registers and Python numeric types, and instances of our secret `sfloat` register. We use the same representation as it secret shared counterpart, this way we are able to perform operations in between `sfloat` and publicly available values, once they are cast into `cfloat` values. This basically means that an instance of the same number, on `sfloat` and `cfloat`, would encode the number in the same fashion, as long as they are using the same precision parameters. However, `cfloat` values do not include an err parameter, given that such flag is not necessary.

Data Components The following is a detailed description of the most important member state variables of `cfloat`, and how it stores its value representation:

v

Accessed by: `Default`.
Type: `cint`.
Stores the significand in clear form.

p

Accessed by: `Default`.
Type: `cint`.
Stores the exponent in clear form.

s

Accessed by: `Default`.
Type: `cint`.
Stores a $\{0, 1\}$ value storing the sign of the instance, where 0 means is positive and 1 otherwise. Our aim is to mimic the behaviour of its secret shared counterpart.

z

Accessed by: `Default`.
Type: `cint`.
Stores a $\{0, 1\}$ value to flag when the `cfloat` instance's value is zero. When this variable takes a value of 0, it means that the `cfloat` instance is a non-zero value and 1 otherwise. Our aim is to mimic the behaviour of its secret shared counterpart.

Special Operations:

cfloat.sizeof()

Accessed by: `Default`.
Parameters: N/A
Returns: Python native `int` value corresponding to the size of memory slots occupied by the instance.
Description: Note that each `sfloat` instance is composed by 4 different values (v, p, z and s), it returns the `global_vector_size` times 4. In case the instance is vectorized, it would return the vector size time 4.
Example: To obtain reciprocal you can execute:

```
a =cfloat(4.5)
r = a.sizeof()
print_ln(r)  # the output is 4. By Default the
    global_vector
_size is set to 1.
```

Branching and Looping

As discussed earlier, all native Python loops and branches are evaluated at compile time. So to allow for control flow breaks depending on register values that are not known at compile time, and also to prevent unrolling of loops (reducing code size), the special library functions for looping and branching should be used. Because of the constraints of the compiling process, there are several possibilities for both branching and looping. Some of them require that the body of a loop or branch is written in a separate function. The loop or branch is then initiated by a special function call, taking the loop or branch function as a parameter.

Branching There are two ways of doing the if statement:

if_statement(condition, if_true, if_false=None)	if_then(condition)

def if_true():
  print_ln("True")
 	
def if_false():
  print_ln("False")
 	
c = cint(1)
if_statement(c, if_true, if_false)
	c = cint(1)
	
	if_then(c)
	print_ln("True")
	else_then()
	print_ln("False")
	end_if()

Note that the functions in if_statement do not need to be named as above, but a clear naming convention like this is recommended to avoid ambiguity.

In that case value in register `c` is non-zero (1), so in both cases `True` will be printed in the terminal. More complex if/else statements (e.g. with multiple else conditions), can be created by simply nesting further `if_statement` calls within `if_false` in the left example, or adding another statement after `else_then` in the right example.

Looping All types of while-loops and range-loops with simple examples, like in other languages we can implement same function using different types of loops.

do_loop(condition, loop_fn)

Executes the code in `loop_body` once, and continues to do so while the clear register returned by `loop_body`

is non-zero. Unlike with if and else statements, here the loop function must always return the loop condition register. Otherwise there is no way of knowing which register holds the condition after each iteration, since the original condition register cannot be modified:

```
def loop_body(cond):
    print_ln("%s", cond)
    return cond-1

t = cint(5)
do_loop(t, loop_body)
```

Prints numbers from 5 to 1.

`while_loop(loop_body, condition, arg1, arg2, ...)`

Here `condition` has to be a function returning a clear integer, and `arg1, arg2, ...` are the initial arguments given to that function. `loop_body` has to return the arguments given `condition` after every loop iteration. In addition, these arguments are given to `loop_body`, which can be used to store the state between loop iterations. This is how the example looks like and also in a more conventional order:

while_loop	**@while_do**

def loop_body(i):
  print_ln("%s",i+1)
  return i+1
 	
while_loop(loop_body, lambda x: x < 5, 0)
	@while_do(lambda x: x < 5, 0)
	def loop_body(i):
	  print_ln("%s",i+1)
	  return i+1

Both prints numbers from 1 to 5.

`range_loop(loop_body, stop)`
`range_loop(loop_body, start, stop[, step])`

For-range loops can also be implemented in two ways, second way is not very different from clasic for loop, if start value is not declared it is automaticaly set to be 0, as on the example:

range_loop	**@for_range**

def loop_body(i):
  print_ln("%s",i+1)
 	
range_loop(loop_body, 5)
	@for_range(5)
	def loop_body(i):
	  print_ln("%s",i+1)

Both prints numbers from 1 to 5.

`do_while(loop_body)`

Finally, it is the most traditional variant of a do-while loop. The loop is stopped when the return value is zero or false. However, variables declared outside the function cannot be modified inside the function, but they can be read. So in order to create finite loop, memory have to be used, MemValue is the easiest way to deal with memory:

do_while	**@do_while**

def loop_body():
  m.write(m+1)
  print_ln("%s", m)
  return m < 5
 	
m = MemValue(cint(0))
do_while(loop_body)
	m = MemValue(cint(0))
	
	@do_while
	def loop_body():
	  m.write(m+1)
	  print_ln("%s", m)
	  return m < 5

Both prints numbers from 1 to 5.

Memory in a classical way can also be used, address in function is read only, same function once again:

do_while	**@do_while**

def loop_body():
  m = cint.load_mem(address)
  print_ln("%s", m)
  store_in_mem(m+1, address)
  return m < 5
 	
address = program.malloc(1, 'c')
store_in_mem(cint(1),address)
do_while(loop_body)
	address = program.malloc(1, 'c')
	store_in_mem(cint(1), address)
	
	@do_while
	def loop_body():
	  m = cint.load_mem(address)
	  print_ln("%s", m)
	  store_in_mem(m+1, address)
	  return m < 5

Both prints numbers from 1 to 5.

Arrays

Arrays are made very similar to the other programming languages, using the array[index] indexing method. To declare array:

type.Array(size)

Where size is just a compile-time known integer, type is the data type of single array element, for exaple sint or cint. Note that in previous versions, the array declaration was Array(size, type). This is still suported but consider it as deprecated.

```
new_array = sint.Array(10)

@for_range(len(new_array))
def range_body(i):
    new_array[i] = sint(i+1)

@while_do(lambda x: x < 5, 0)
def while_body(i):
    print_ln("%s",new_array[i].reveal())
    return i+1
```

Declares new array of size 10, then fills it with values from 1 to 10, at the end prints first five values, so it prints numbers from 1 to 5. Note that the values of array can be modified inside the function, they are exacly like MemValue.

type.Matrix(rows, columns)

2D arrays are also implemented, but they are called matrices, matrix can be used in the same way as array:

```
new_matrix = sint.Matrix(3,2)

@for_range(len(new_matrix))
def range_body(i):
    new_matrix[i][0] = sint(2*i)
    new_matrix[i][1] = sint(2*i+1)

@while_do(lambda x: x < 3, 0)
def while_body(i):
```

```
    @for_range(2)
    def range_body(j):
        print_ln("%s",new_matrix[i][j].reveal())
    return i+1
```

Matrix is just an array of arrays so length of matrix is just number of arrays inside, as it is shown on the example the first value of declaration is number of arrays inside, the second is length of an arrays(example prints numbers from 0 to 5).

type.MultiArray([n_1,...,n_k])

kD arrays are also implemented. They are meant to be just containers (retrieve and set data) - no fancy functions added on top of them such as assign_all, etc:

```
n = 3
m = 4
p = 5
a = sint.MultiArray([n,m,p])
b = sint.MultiArray([n,m,p])
c = sint.MultiArray([n,m,p])

for i in range(n):
  for j in range(m):
    for k in range(p):
      a[i][j][k] = i + j + k
      b[i][j][k] = 2 * (i + j + k)
      c[i][j][k] = (a[i][j][k] + b[i][j][k])

# now c[i][j][k] = 3 * (i + j + k)
```

Short-circuit evalution The following functions provide short-circuit evaluation, which means that only necessary terms are executed, just as `&&` and `||` in C or `or` and `and` in Python:

and_(term1, term2, ...)
or_(term1, term2, ...)
not_(term)

Since the direct specification of a term would trigger immediate executiation and thus lose the short-circuit property, only functions are permitted inputs:

```
    def compare():
        return a < b
    and_(lambda: x < y, compare)
```

However, the functions should be combined directly because they already output lambda functions:

```
    not_(and_(lambda: x < y, lambda: a < b), lambda: c < d)
```

It is possible to use short-circuit evaluation for branching and `do_while` loops but not `do_loop` loops because the condition also represents the state in the latter case.

```
if_then(and_(lambda: x < y, lambda: a < b))
...
```

```
end_if()

@do_while
def f():
    ...
    return and_(lambda: x < y, lambda: a < b)
```

Multi-threading

Creating and running multiple, concurrent tapes on the virtual machine is supported via a simple threading interface. A thread is created by creating an `MPCThread` object referring to a function, which contains the code to be executed by the thread.

```
def func1():
    store_in_mem(sint(1),0)
    store_in_mem(sint(1),0)
    for i in range(8):
        a = load_secret_mem(i)
        b = load_secret_mem(i+1)
        store_in_mem(a+b, i+2)

t = MPCThread(func1, 't1')
t.start()
t.join()
```

Stores secret Fibonacci numbers in first 10 places of memory.

Multiple threads can be run in parallel and sequentially, but threads cannot themselves spawn threads as this is not yet supported by the virtual machine.

Testing

Testing of programs is done using the `test.sh` script, and the library functions `test` and `test_mem`. To test output, execution of the library functions is emulated in Python by the `test_result.py` script, and compared with the actual output from running the virtual machine. To trigger tests when running the script, calls to the following functions must be inserted into the source code file.

`test(value, lower=None, upper=None)`

Tests the value of register `value` by storing it to memory and then reading the memory file output by the virtual machine. If `lower` and `upper` are not specified, checks that `value` corresponds exactly to the emulated result. Otherwise, checks that `value` lies in the range `[lower,upper]`.

`test_mem(value, address, lower=None, upper=None)`

Tests whether the value in a given memory address is equal to the integer `value`. If `lower` and `upper` are specified, ignore `value` and instead check the result lies in the given range.

Emulation is not currently supported for all library functions. Loops and branches must be tested manually using `test_mem`, and any programs that rely on reading or writing to or from main memory may not emulate correctly either.

The Compilation Process

Compilation is performed by a single call to the Python function `execfile` on the main source code file. The creation and optimization of bytecode tapes is done on-the-fly, as the program is being parsed. As soon as a tape is finished parsing it is optimized and written to a bytecode file, and all its resources freed to save on memory usage.

During parsing, instructions are created such that every output of an instruction is a new register. This ensures that registers are only written to once, which simplifies the register allocation procedure later. The main goal of the optimization process is to minimize the number of rounds of communication within basic blocks of each tape, by merging independent `startopen` and `stopopen` instructions. This is done through instruction reordering and analysis of the program dependency graph in each basic block. After optimization, a simple register allocation procedure is carried out to minimize register usage in the virtual machine.

Program Representation

The data structures used to represent a program are implemented in `program.py`, and here the relevant classes are described.

`Program`

The main class for a program being compiled.

tapes: List of tapes in the program.

schedule: The schedule of the tapes is a list of pairs of the form `('start|stop', list_of_tapes)`. Each pair instructs the virtual machine to either start or stop running the given list of tapes.

curr_tape: A reference to the tape that is currently being processed.

curr_block: A reference to the basic block that is currently being processed (within the current tape). Any new instructions created during parsing are added to this block.

restart_main_thread(): Force the current tape to end and restart a fresh tape. Can be useful for breaking tapes up to speed up the optimization process.

`Tape`

Contains basic blocks and data for a tape. Each tape has its own set of registers

basicblocks: List of basic blocks in this tape.

optimize(): Optimize the tape. First optimizes communication, then inserts any branching instructions (which would otherwise interfere with the previous optimization) and finally allocates registers.

req_num: Dictionary storing the required numbers of preprocessed triples, bits etc. for the tape.

purge(): Clear all data stored by the tape to reduce memory usage.

new_reg(reg_type, size=None): Creates a register of type `reg_type` (either 's' or 'c' for secret or clear) for this Tape. If `size` is specified and > 1 then a vector of registers is returned.

`BasicBlock`

A basic block contains a list of instructions with no branches, but may end with a branch to another block depending on a certain condition. Basic blocks within the same tape share the same set of registers.

instructions: The instructions in this basic block.

set_exit(condition, exit_block): Sets the exit block to which control flow will continue, if the instruction `condition` evaluates to true. If `condition` returns false or `set_exit` is not called, control flow implicitly proceeds to the next basic block in the parent Tape's list.

add_jump(): Appends the exit condition instruction onto the list of instructions. This must be done *after* optimization, otherwise instruction reordering during merging of opens will cause the branch to behave incorrectly.

adjust_jump(): Calculates and sets the value of the relative jump offset for the exit condition instruction. This must be done after `add_jump` has been called on *every basic block in the tape*, in order that the correct jump offset is calculated.

`Tape.Register`

The class for clear and secret registers. This is enclosed within a Tape, as registers should only be created by calls to a Tape's `new_reg` method (or, preferably, the high-level library functions).

`Instruction`

This is the base class for all instructions. The field `__slots__` should be specified on every derived class to speed up and reduce the memory usage from processing large quantities of instructions. Creating new instructions should be fairly simple by looking at the existing code. In most cases, `__init__` will not need to be overridden; if this is necessary, ensure that the original method is still called using `super`.

code: The integer opcode. This should be stored in the `opcodes` dictionary.

arg_format: The format for the instruction's arguments is given as a list of strings taking one of the following:

- 'c[w]': clear register, with the optional suffix 'w' if the register is written to.
- 's[w]': secret register, as above.
- 'r[w]': regint register, as above.
- 'i' : 32-bit integer signed immediate value.
- 'int' : 64-bit integer unsigned immediate value.
- 'p' : 32-bit number representing a player index.
- 'str' : A four byte string.

For example, to implement a basic addition instruction for adding a secret and a clear register, the following code suffices.

```
class addm(Instruction):
    """ Add a secret and clear register into a secret register. """
    __slots__ = [] # contents inherited from parent
    code = opcodes['ADDM']
    arg_format = ['sw', 's', 'c']
```

`Memory`

Memory comes in three varieties `sint`, `cint`, and `regint`; denoted by `S[i]`, `C[i]` and `R[i]`.

Optimizing Communication

The first observation is that communication of small data items costs, so we want to pack as much data together in a start/stop open command. The technique for automating this is as follows:

- Calculate the *program dependence graph* G, whose vertices correspond to instructions in the byte-code; treating start/stop open instructions as a single instruction. Two vertices (v_i, v_j) are connected by a directed edge if an output from instruction v_i is an input to instruction v_j.
- Now consider the graph H, whose vertices also correspond to instructions; insert a directed edge into H from every `startopen` vertex to any other vertex where there is a path in G not going through another `startopen` vertex.
- Label vertices in H with their *maximal* distance from any source in H that is a `startopen`.

- Merge all start/stop opens with the same label in H.
- Create another graph H' with vertices corresponding to instructions and edges from every non-open vertex to any `startopen` where there is a path not going though another `startopen` vertex.
- Label non-open vertices in H' with the minimum of the labels (in H) of their successors in H'.
- Compute a topological ordering the merged graph such that sources with H'-label i appear after the open with label $i-1$ and such that sinks with H-label j before the open with label j. Furthermore, let non-open vertices with H-label i and H'-label j such that $j - i \geq 2$ appear between the `startopen` and `stopopen` with label $i+1$.
- Output the resulting instruction sequence.

Register allocation

After reordering instructions to reduce communication costs, we are left with a sequence of instructions where every register is written to no more than once. To reduce the number of registers and memory requirements we now perform register allocation.

Traditionally, this is done by graph colouring, which requires constructing the *interference graph*, where nodes correspond to registers, with an undirected edge (a, b) if the 'lifetimes' of registers a and b overlap. Algorithms for creating this graph, however, typically run in $O(n^2)$ time for n instructions. Since we unroll loops and restrict the amount of branching that can be done, basic blocks are a lot larger then usual, so constructing the interference graph becomes impractical. We instead use a simple method that takes advantage of the fact that the program is in SSA form. We iterate backwards over the instructions, and whenever a variable is assigned to, we know that the variable will not be used again and the corresponding register can be re-used.

Note that register allocation should be done *after* the merging of open instructions: if done first, this will severely limit the possibilities for instruction reordering due to reuse of registers.

Notes

The following are mainly notes for the development team, so we do not forget anything. Please add stuff here when you notice something which might be useful for people down the line.

1. Instructions/ByteCodes can be re-ordered. Sometimes this is a bad idea, e.g. for IO instructions. All instructions/bytecodes which inherit from `IOInstruction` are never reordered.
2. Instructions executed in the test scripts need to be emulated in python. So do not use an instruction here which does not have an emulation. The emulation stuff is in `core.py`. Essentially, there's two options for testing:A
   - If you use the 'test(x,y)' syntax you have to make sure that all functions (including classes) called for x are defined in `core.py`, but they don't have to do anything. For example:

     ```
     \verb+ print_ln = lambda *args: None+
     ```

     This allows to call print_ln in tested programs, but nothing really happens when testing the results.
   - If you use the 'test(x)' syntax you have to make sure that the functionality is replicated in `core.py` and understood by `Scripts/test_results.py`. This is more involved, but it allows to write tests with less effort.

FHE Security

In this chapter we detail how FHE parameters are selected. The first part deals with how the basic "sizes" are derived, and the second part gives the mathematical justification (i.e. noise analysis) for the equations used in the code for setting up parameters. Before proceeding it is perhaps worth detailing how different security parameters are used within the code.

- sacrifice_sec (default 80): This defined how many sacrifices we do per triple, to get the precise number you compute $\lceil \mathsf{sacrifice\_sec}/\log_2 p \rceil$. Thus this statistical security parameter is effectively $\log_2 p$ unless p is very small.
- macs_sec (default 80): For the full threshold case this defined how many MACs are held per data item, again to get the precise number you compute $\lceil \mathsf{macs\_sec}/\log_2 p \rceil$. Thus, again, this statistical security parameter is effectively $\log_2 p$ unless p is very small.
- Sound_sec (default 128): This defines the soundness error of the ZKPoKs below, the value of $2^{-\mathsf{Sound\_sec}}$ defines the probability that an adversary can cheat in a single ZKPoK.
- ZK_sec (default 80): This defines the statistical distance between the coefficients of the ring elements in an honest ZKPoK transcript and one produced by a simulation.
- DD_sec (default 80): This defines the statistical distance between the coefficients of the ring elements in the distribution produced in the distributed decryption protocol below to the uniform distribution.
- `comp_sec` (default 128): This defines the computational security parameter for the FHE scheme.
- ϵ (default 55): This defines the noise bounds for the FHE scheme below in the following sense. A FHE ciphertext in the protocol is gauranteed to decrypt correctly, even under adversarial inputs assuming the ZKPoKs are applied correctly, with probability $1 - 2^{-\epsilon}$. In fact this is an under-estimate of the probability. From ϵ we define e_i such that $\mathsf{erfc}(e_i)^i \approx 2^{-\epsilon}$ and then we set $\mathfrak{c}_i = e_i^i$.

All of these parameters can be tweaked in the file `config.h`. Another two parameters related to the FHE scheme are `hwt`, which is defined in `Setup.cpp` to be equal to $h = 64$. This defines the number of non-zero coefficients in the FHE secret key. Another parameter is σ which is the standard deviation for our approximate discrete Gaussians, which we hardwire to be $3.16 = \sqrt{10}$ (as we use the NewHope method of producing approximate Gaussians).

Main Security Parameters

Our Ring-LWE samples are (essentially) always from an approximate Gaussian distribution with standard deviation 3.16 and from a ring with a two-power degree of N. This is not strictly true as some noise samples come from small Hamming Weight distributions, and some come from distributions over $\{-1, 0, 1\}$. But the above are the main parameters, and have been used in prior works to estimate Ring-LWE security in SHE settings.

Given these settings we need to determine the maximum (ciphertext) ring modulus q that still gives us security. For this we use Martin Albrecht's estimator which can be found at

`https://bitbucket.org/malb/lwe-estimator`

For various values of n and (symmetric equivalent) security levels `comp_sec` we then find the minimum secure value of q. This is done by running the following `sage` code

```
load("estimator.py")
n = 1024
comp_sec = 128
for i in range(10, 500, 5):
   q= 2^i
```

```
    costs = estimate_lwe(n, 3.16*sqrt(2*pi)/q, q, reduction_cost_model=BKZ.sieve,  \
                         skip=["arora-gb", "bkw", "dec", "mitm"])
    if any([cost.values()[0]<2^comp_sec for cost in costs.values()]):
        break
print i-5
```

This will print a bunch of rubbish and then the number 29. This means any $q < 2^{29}$ will be "secure" by the above definition of secure.

We did this in Feb 2018 and obtained the following table of values, giving maximum values of q in the form of 2^x for the value x from the following table.

N	comp_sec=80	comp_sec=128	comp_sec=256
1024	44	29	16
2048	86	56	31
4096	171	111	60
8192	344	220	120
16384	690	440	239
32768	998	883	478

Any updates to this table needs to be duplicated in the file `FHE_Params.cpp` in the code base.

Distributions and Norms

Given an element $a \in R = \mathbb{Z}[X]/(X^N + 1)$ (represented as a polynomial) where $X^N + 1$ is the $m = 2 \cdot N$-th cyclotomic polynomial (recall N is always a power of two). We define $\|a\|_p$ to be the standard p-norm of the coefficient vector (usually for $p = 1, 2$ or ∞). We also define $\|a\|_p^{\mathsf{can}}$ to be the p-norm of the same element when mapped into the canonical embedding i.e.

$$\|a\|_p^{\mathsf{can}} = \|\kappa(a)\|_p$$

where $\kappa(a) : R \longrightarrow \mathbb{C}^{\phi(m)}$ is the canonical embedding. The key three relationships are that

$$\|a\|_\infty \leq c_m \cdot \|a\|_\infty^{\mathsf{can}} \quad \text{and} \quad \|a\|_\infty^{\mathsf{can}} \leq \|a\|_1 \quad \text{and} \quad \|a\|_1 \leq \phi(m) \cdot \|a\|_\infty.$$

for some constant c_m depending on m. Since in our protocol we select m to be a power of two, we have $c_m = 1$. In particular (which will be important later in measuring the noise of potentially dishonestly generated ciphertexts) we have

$$\|a\|_\infty^{\mathsf{can}} \leq \phi(m) \cdot \|a\|_\infty.$$

We also define the *canonical embedding norm reduced modulo q* of an element $a \in R$ as the smallest canonical embedding norm of any a' which is congruent to a modulo q. We denote it as

$$|a|_q^{can} = \min\{\ \|a'\|_\infty^{\mathsf{can}} :\ a' \in R,\ a' \equiv a \pmod q\ \}.$$

We sometimes also denote the polynomial where the minimum is obtained by $[a]_q^{\mathsf{can}}$, and call it the *canonical reduction* of a modulo q.

Following [GHS12][Appendix A.5] we examine the variances of the different distributions utilized in our protocol.

- $\mathcal{HWT}(h, N)$: This generates a vector of length N with elements chosen at random from $\{-1, 0, 1\}$ subject to the condition that the number of non-zero elements is equal to h.
- $\mathcal{ZO}(0.5, N)$: This generates a vector of length N with elements chosen from $\{-1, 0, 1\}$ such that the probability of coefficient is $p_{-1} = 1/4$, $p_0 = 1/2$ and $p_1 = 1/4$.
- $\mathcal{DG}(\sigma^2, N)$: This generates a vector of length N with elements chosen according to an approximation to the discrete Gaussian distribution with variance σ^2.

- $\mathcal{RC}(0.5, \sigma^2, N)$: This generates a triple of elements (v, e_0, e_1) where v is sampled from $\mathcal{ZO}_s(0.5, N)$ and e_0 and e_1 are sampled from $\mathcal{DG}_s(\sigma^2, N)$.
- $\mathcal{U}(q, N)$: This generates a vector of length N with elements generated uniformly modulo q.

Let ζ_m denote any complex primitive m-th root of unity. Sampling $a \in R$ from $\mathcal{HWT}(h, \phi(m))$ and looking at $a(\zeta_m)$ produces a random variable with variance h, when sampled from $\mathcal{ZO}(0.5, \phi(m))$ we obtain variance $\phi(m)/2$, when sampled from $\mathcal{DG}(\sigma^2, \phi(m))$ we obtain variance $\sigma^2 \cdot \phi(m)$ and when sampled from $\mathcal{U}(q, \phi(m))$ we obtain variance $q^2 \cdot \phi(m)/12$. By the law of large numbers we can use $\mathfrak{c}_1 \cdot \sqrt{V}$, where V is the above variance, as a high probability bound on the size of $a(\zeta_m)$ (the probability is $1 - 2^{-\epsilon}$, and this provides a bound on the canonical embedding norm of a.

If we take a product of t such elements with variances $V_1, V_2, \ldots, V_t$ then we use $\mathfrak{c}_t \cdot \sqrt{V_1 \cdot V_2 \cdots V_t}$ as the resulting bounds. In our implementation we approximate $\mathcal{DG}(\sigma^2, n)$ using the binomial method from the NewHope paper, with a standard deviation of $3.16 = \sqrt{10}$. In particular this means any vector sampled from $\mathcal{DG}(\sigma^2, n)$ will have infinity norm bounded by 20.

The FHE Scheme and Noise Analysis

Note that whilst the scheme is "standard" some of the analysis is slightly different from that presented in the [DKL+13] paper and that presented in the [GHS12] paper. We use the notation of the [DKL+13] paper in this section, and assume the reader is familiar with prior work. The key differences are:

- Unlike in SPDZ-2 we assume a perfect distributed key generation method amongst the n players.
- Unlike in SPDZ-2 we are going to use an actively secure ZKPoK (see later), which will make the actual noise analysis much more complex.

Key Generation:

The main distinction between the assumptions here and those used in the [DKL+13] paper, is that in the latter a specific distributed key generation protocol for the underlying threshold FHE keys was assumed. In SCALE we assume a 'magic black box' which distributes these keys, which we leave to the application developer to create. The secret key $\mathfrak{s}$ is selected from a distribution with Hamming weight h, i.e. $\mathcal{HWT}(h, \phi(m))$, and then it is distributed amongst the n parties by simply producing a random linear combination, and assigning each party one of the sums. The switching key data is produced in the standard way, i.e. in a non-distributed trusted manner. We assume a two-leveled scheme with moduli p_0 and p_1 with $q_1 = p_0 \cdot p_1$ and $q_0 = p_0$. We require

$$\begin{aligned} p_1 &\equiv 1 \pmod{p}, \\ p_0 - 1 \equiv p_1 - 1 \equiv p - 1 &\equiv 0 \pmod{\phi(m)}. \end{aligned}$$

In particular the public key is of the form (a, b) where

$$a \leftarrow \mathcal{U}(q, \phi(m)) \quad \text{and} \quad b = a \cdot \mathfrak{s} + p \cdot \epsilon$$

where $\epsilon \leftarrow \mathcal{DG}(\sigma^2, \phi(m))$ and the switching key data $(a_{\mathfrak{s},\mathfrak{s}^2}, b_{\mathfrak{s},\mathfrak{s}^2})$ is of the form

$$a_{\mathfrak{s},\mathfrak{s}^2} \leftarrow \mathcal{U}(q, \phi(m)) \quad \text{and} \quad b_{\mathfrak{s},\mathfrak{s}^2} = a_{\mathfrak{s},\mathfrak{s}^2} \cdot \mathfrak{s} + p \cdot e_{\mathfrak{s},\mathfrak{s}^2} - p_1 \cdot \mathfrak{s}^2$$

where $e_{\mathfrak{s},\mathfrak{s}^2} \leftarrow \mathcal{DG}(\sigma^2, \phi(m))$. We take $\sigma = 3.16$ as described above in what follows.

Encryption:

To encrypt an element $m \in R$, we choose $v, e_0, e_1 \leftarrow \mathcal{RC}(0.5, \sigma^2, n)$, i.e.

$$v \leftarrow \mathcal{ZO}(0.5, \phi(m)) \text{ and } e_0, e_1 \leftarrow \mathcal{DG}(\sigma^2, \phi(m))$$

Then we set $c_0 = b \cdot v + p \cdot e_0 + m,\ c_1 = a \cdot v + p \cdot e_1$, and set the initial ciphertext as $\mathfrak{c}' = (c_0, c_1)$. We calculate a bound (with high probability) on the output noise of an honestly generated ciphertext to be

$$\begin{aligned}
\|c_0 - \mathfrak{s} \cdot c_1\|_\infty^{\mathsf{can}} &= \|((a \cdot \mathfrak{s} + p \cdot \epsilon) \cdot v + p \cdot e_0 + m - (a \cdot v + p \cdot e_1) \cdot \mathfrak{s}\|_\infty^{\mathsf{can}} \\
&= \|m + p \cdot (\epsilon \cdot v + e_0 - e_1 \cdot \mathfrak{s})\|_\infty^{\mathsf{can}} \\
&\leq \|m\|_\infty^{\mathsf{can}} + p \cdot \left(\|\epsilon \cdot v\|_\infty^{\mathsf{can}} + \|e_0\|_\infty^{\mathsf{can}} + \|e_1 \cdot \mathfrak{s}\|_\infty^{\mathsf{can}}\right) \\
&\leq \phi(m) \cdot p/2 + p \cdot \sigma \cdot \left(\mathfrak{c}_2 \cdot \phi(m)/\sqrt{2} + \mathfrak{c}_1 \cdot \sqrt{\phi(m)} + \mathfrak{c}_2 \cdot \sqrt{h \cdot \phi(m)}\right) = B_{\mathsf{clean}}.
\end{aligned}$$

Note this is a probablistic bound and not an absolute bound.

However, below we will only be able to guarantee the m, v, e_0 and e_1 values of a **sum** of n fresh ciphertexts (one from each party) are selected subject to

$$\|v\|_\infty \leq 2^{\mathsf{ZK\_sec}+U/2+2} \cdot n \quad \text{and} \quad \|e_0\|_\infty, \|e_1\|_\infty \leq 20 \cdot 2^{\mathsf{ZK\_sec}+U/2+2} \cdot n \quad \text{and} \quad \|m\|_\infty \leq 2^{\mathsf{ZK\_sec}/2+U/2+1} \cdot n \cdot p,$$

where $\mathsf{ZK\_sec}$ is our soundness parameter for the zero-knowledge proofs and U is selected so that $(2 \cdot \phi(m))^U > 2^{\mathsf{Sound\_sec}}$. In this situation we obtain the bound, using the inequality above between the infinity norm in the polynomial embedding and the infinity norm in the canonical embedding,

$$\begin{aligned}
\|c_0 - \mathfrak{s} \cdot c_1\|_\infty^{\mathsf{can}} &\leq \sum_{i=1}^{n} \|2 \cdot m_i\|_\infty^{\mathsf{can}} + p \cdot \Big(\|\epsilon\|_\infty^{\mathsf{can}} \cdot \|2 \cdot e_{2,i}\|_\infty^{\mathsf{can}} + \|2 \cdot e_{0,i}\|_\infty^{\mathsf{can}} \\
&\qquad + \|\mathfrak{s}\|_\infty^{\mathsf{can}} \cdot \|2 \cdot e_{1,i}\|_\infty^{\mathsf{can}}\Big) \\
&\leq 2 \cdot \phi(m) \cdot 2^{\mathsf{ZK_sec}+U/2+1} \cdot n \cdot p/2 \\
&\qquad + p \cdot \Big(\mathfrak{c}_1 \cdot \sigma \cdot \phi(m)^{3/2} \cdot 2 \cdot 2^{\mathsf{ZK_sec}+U/2+1} \cdot n \\
&\qquad\quad + \phi(m) \cdot 2 \cdot 2^{\mathsf{ZK_sec}+U/2+1} \cdot n \cdot 20 \\
&\qquad\quad + \mathfrak{c}_1 \cdot \sqrt{h} \cdot \phi(m) \cdot 2 \cdot 2^{\mathsf{ZK_sec}+U/2+1} \cdot n \cdot 20\Big) \\
&= \phi(m) \cdot 2^{\mathsf{ZK_sec}+U/2+2} \cdot n \cdot p \cdot \Big(\frac{41}{2} + \mathfrak{c}_1 \cdot \sigma \cdot \phi(m)^{1/2} + 20 \cdot \mathfrak{c}_1 \cdot \sqrt{h}\Big) \\
&= B_{\mathsf{clean}}^{\mathsf{dishonest}}.
\end{aligned}$$

Again this is a probabilistic bound (assuming validly distributed key generation), but assumes the worst case ciphertext bounds.

$\mathsf{SwitchModulus}((c_0, c_1))$**:**

This takes as input a ciphertext modulo q_1 and outputs a ciphertext mod q_0. The initial ciphertext is at level $q_1 = p_0 \cdot p_1$, with $q_0 = p_0$. If the input ciphertext has noise bounded by ν in the canonical embedding then the output ciphertext will have noise bounded by ν' in the canonical embedding, where

$$\nu' = \frac{\nu}{p_1} + B_{\mathsf{scale}}.$$

The value B_{scale} is an upper bound on the quantity $\|\tau_0 + \tau_1 \cdot \mathfrak{s}\|_\infty^{\mathsf{can}}$, where $\kappa(\tau_i)$ is drawn from a distribution which is close to a complex Gaussian with variance $\phi(m) \cdot p^2/12$. We therefore, we can (with high probability) take the upper bound to be

$$B_{\mathsf{scale}} = c_1 \cdot p \cdot \sqrt{\phi(m)/12} + c_2 \cdot p \cdot \sqrt{\phi(m) \cdot h/12},$$

This is again a probabilistic analysis, assuming validly generated public keys.

$\mathsf{Dec}_{\mathfrak{s}}(\mathfrak{c})$**:**

As explained in [DKL+13, GHS12] this procedure works when the noise ν (in the canonical embedding) associated with a ciphertext satisfies $c_m \cdot \nu < q_0/2$. However, as we always take power of two cyclotomics we have $c_m = 1$

$\mathsf{DistDec}_{\{\mathfrak{s}_i\}}(\mathfrak{c})$**:**

There are two possible distributed decryption protocols. The first one is from [DPSZ12] is for when we want to obtain a resharing of an encrypted value, along with a fresh ciphertext. The second version is from [KPR18], where we do not need to obtain a fresh encryption of the value being reshared.

Reshare $-$ 1

Input is $e_{\mathbf{m}}$, where $e_{\mathbf{m}} = \mathsf{Enc}_{\mathfrak{pk}}(\mathbf{m})$ is a public ciphertext.
Output is a share $\mathbf{m}_i$ of $\mathbf{m}$ to each player P_i; and a ciphertext $e'_{\mathbf{m}}$.
The idea is that $e_{\mathbf{m}}$ could be a product of two ciphertexts, which Reshare converts to a "fresh" ciphertext $e'_{\mathbf{m}}$. Since Reshare uses distributed decryption (that may return an incorrect result), it is not guaranteed that $e_{\mathbf{m}}$ and $e'_{\mathbf{m}}$ contain the same value, but it *is* guaranteed that $\sum_i \mathbf{m}_i$ is the value contained in $e'_{\mathbf{m}}$.

1. Each player P_i samples a uniform $\mathbf{f}_i \in (\mathbb{F}_p)^N$. Define $\mathbf{f} := \sum_{i=1}^n \mathbf{f}_i$.
2. Each player P_i computes and broadcasts $e_{\mathbf{f}_i} \leftarrow \mathsf{Enc}_{\mathfrak{pk}}(\mathbf{f}_i)$.
3. Each player P_i runs the proof below as a prover on $e_{\mathbf{f}_i}$. The protocol aborts if any proof fails and if successful each player obtains $e_{\mathbf{f}} \leftarrow e_{\mathbf{f}_1} \boxplus \cdots \boxplus e_{\mathbf{f}_n}$.
4. The players compute $e_{\mathbf{m}+\mathbf{f}} \leftarrow e_{\mathbf{m}} \boxplus e_{\mathbf{f}}$.
5. The players decrypt $e_{\mathbf{m}+\mathbf{f}}$ as follows:
   (a) Player one computes $\mathbf{v}_1 = e^{(0)}_{\mathbf{m}+\mathbf{f}} - \mathfrak{s}_1 \cdot e^{(1)}_{\mathbf{m}+\mathbf{f}}$ and player $i \neq 1$ computes. $\mathbf{v}_i = -\mathfrak{s}_i \cdot e^{(1)}_{\mathbf{m}+\mathbf{f}}$.
   (b) All players compute $\mathbf{t}_i = \mathbf{v}_i + p \cdot \mathbf{r}_i$ for some random element with infinity norm given by $2^{\mathsf{DD\_sec}} \cdot B/p$, where DD_sec is the parameter defining statistical distance for the distributed decryption.
   (c) The parties broadcast $\mathbf{t}_i$.
   (d) The parties compute $\mathbf{m} + \mathbf{f} = \sum \mathbf{t}_i \pmod p$.
6. P_1 sets $\mathbf{m}_1 \leftarrow \mathbf{m} + \mathbf{f} - \mathbf{f}_1$, and each player P_i ($i \neq 1$) sets $\mathbf{m}_i \leftarrow -\mathbf{f}_i$.
7. All players set $e'_{\mathbf{m}} \leftarrow \mathsf{Enc}_{pk}(\mathbf{m}+\mathbf{f}) \boxminus e_{\mathbf{f}}$, where a default value for the randomness is used when computing $\mathsf{Enc}_{pk}(\mathbf{m}+\mathbf{f})$.

Figure 1: The sub-protocol for additively secret sharing a plaintext $\mathbf{m} \in (\mathbb{F}_{p^k})^s$ on input a ciphertext $e_{\mathbf{m}} = \mathsf{Enc}_{\mathfrak{pk}}(\mathbf{m})$.

Reshare Version 1: This is described in Figure 1. The value B in the protocol is an upper bound on the noise in the canonical embedding ν associated with a ciphertext we will decrypt in our protocols. To ensure valid distributed decryption we require

$$2 \cdot (1 + n \cdot 2^{\mathsf{DD\_sec}}) \cdot B < q_0.$$

Given a value of B, we therefore will obtain a lower bound on p_0 by the above inequality. The addition of a random term with infinity norm bounded by $2^{\mathsf{DD\_sec}} \cdot B/p$ in the distributed decryption procedure ensures that the individual *coefficients* of the sum $\mathbf{t}_1 + \cdots + \mathbf{t}_n$ are statistically indistinguishable from random, with probability $2^{-\mathsf{DD\_sec}}$. This does not imply that the adversary has this probability of distinguishing the simulated execution in [DPSZ12] from the real execution; since each run consists of the exchange of $\phi(m)$ coefficients, and the protocol is executed many times over the execution of the whole protocol. We however feel that setting concentrating solely on the statistical indistinguishability of the coefficients is valid in a practical context.

Reshare Version 2: This is given in Figure 2. This protocol is simpler as it does not require the resharing to a new ciphertext. So in particular players do not need to provide encryptions of their $\mathbf{f}_i$ values, and hence there is no need to execute the ZKPoK needed in the previous protocol.

Reshare $-$ 2

Furthermore, let B denote a bound on the noise, that is $\|c_0 - s \cdot c_1\|_\infty$.

1. Party i samples $\mathbf{f}_i \leftarrow [0, B \cdot 2^{\mathsf{DD\_sec}}]^N$.
2. Party 1 computes $\mathbf{v}_i := (c_0 - s_1 \cdot c_1) - \mathbf{f}_1 \bmod q$, and every other party i computes $\mathbf{v}_i := -s_i \cdot c_1 - \mathbf{f}_i$.
3. Party i broadcasts $\mathbf{v}_i$.
4. Party 1 outputs $\mathbf{m}_1 := (\sum_{i=1}^n \mathbf{v}_i \bmod q + \mathbf{f}_1) \bmod p$, and every other party i outputs $\mathbf{m}_i := \mathbf{f}_i \bmod p$.

Figure 2: Distributed decryption to secret sharing

$\mathsf{SwitchKey}(d_0, d_1, d_2)$**:**

In order to estimate the size of the output noise term in the canonical embedding we need first to estimate the size of the term (again probabilistically assuming validly generated public keys)

$$\|p \cdot d_2 \cdot \epsilon_{\mathfrak{s},\mathfrak{s}^2}\|_\infty^{\mathsf{can}}.$$

Following [GHS12] we assume heuristically that d_2 behaves like a uniform polynomial with coefficients drawn from $[-q_0/2, \ldots, q_0/2]$ (remember we apply SwitchKey at level zero). So we expect

$$\begin{aligned}\|p \cdot d_2 \cdot e_{\mathfrak{s},\mathfrak{s}^2}\|_\infty^{\mathsf{can}} &\leq p \cdot c_2 \cdot \sqrt{q_0^2/12 \cdot \sigma^2 \cdot \phi(m)^2} \\ &= p \cdot c_2 \cdot q_0 \cdot \sigma \cdot \phi(m)/\sqrt{12} \\ &= B_{\mathsf{KS}} \cdot q_0.\end{aligned}$$

Thus

$$B_{\mathsf{KS}} = p \cdot c_2 \cdot \sigma \cdot \phi(m)/\sqrt{12}.$$

Then if the input to SwitchKey has noise bounded by ν then the output noise value in the canonical embedding will be bounded by

$$\nu + \frac{B_{\mathsf{KS}} \cdot p_0}{p_1} + B_{\mathsf{scale}}.$$

$\mathsf{Mult}(\mathfrak{c}, \mathfrak{c}')$**:**

Combining the all the above, if we take two ciphertexts of level one with input noise in the canonical embedding bounded by ν and ν', the output noise level from multiplication will be bounded by

$$\nu'' = \left(\frac{\nu}{p_1} + B_{\mathsf{scale}}\right) \cdot \left(\frac{\nu'}{p_1} + B_{\mathsf{scale}}\right) + \frac{B_{\mathsf{KS}} \cdot p_0}{p_1} + B_{\mathsf{scale}}.$$

Application to the Offline Phase:

In all of our protocols the most complext circuit we will be evaluating is the following one: We first add n ciphertexts together and perform a multiplication, giving a ciphertext with respect to modulus p_0 with noise in the canonical embedding bounded by

$$U_1 = \left(\frac{B_{\mathsf{clean}}^{\mathsf{dishonest}}}{p_1} + B_{\mathsf{scale}}\right)^2 + \frac{B_{\mathsf{KS}} \cdot p_0}{p_1} + B_{\mathsf{scale}}.$$

Recall that $B_{\mathsf{clean}}^{\mathsf{dishonest}}$ is the bound for a sum of n ciphertexts, one from each party. We then add on another n ciphertexts, which are added at level one and then reduced to level zero. We therefore obtain a final upper bound on the noise in the canonical embedding for our adversarially generated ciphertexts of

$$U_2 = U_1 + \frac{B_{\mathsf{clean}}^{\mathsf{dishonest}}}{p_1} + B_{\mathsf{scale}}.$$

To ensure valid (distributed) decryption, we require

$$2 \cdot U_2 \cdot (1 + n \cdot 2^{\mathsf{DD\_sec}}) < p_0,$$

i.e. we take $B = U_2$ in our distributed decryption protocol. Note, that here we take the worst case bound for the ciphertext noise, but probabilistic analysis everywhere else. Since the key generation is assumed to be honestly performed.

This lower bound on p_0 ensure valid decryption in our offline phase. To obtain a valid and secure set of parameters, we search over all tuples (N, p_0, p_1) satisfying our various constraints for a given "size" of plaintext modulus p; including the upper bound on $q_1 = p_0 \cdot p_1$ obtained from the Albrecht's tool via the table above.

The above circuit is needed for obtaining the sharings of the c value, where we need to obtain a fresh encryption of c in order to be able to obtain the MAC values. For the obtaining of the MAC values of the a and b values we need a simpler circuit, and we use the Distributed Decryption protocol from [KPR18][Appendix A].

Zero Knowledge Proof

The precise ZKPoK we use is from TopGear [CS19], which is an extension of the HighGear prtoocol from [KPR18]. The precise protocol is given in Figure 3 and Figure 4. This is an amortized version of the proof from [DPSZ12], in the sense that the input ciphertexts from all players are simultaneously proved to be correct. The proof takes as input a variable flag, which if set to Diag imposes the restriction that the input plaintexts are "diagonal" in the sense that they encrypt they same value in each slot. This is needed to encrypt the MAC values at the start of the protcol. Being diagonal equates to the input plaintext polynomial being the constant polynomial.

Note the soundness security in the case of $\mathsf{flag} = \mathsf{Diag}$ is only equal to $2^{-}U$ as opposed to $(2 \cdot \phi(m))^U$. However, by repeating the ZKPoK a number of times we obtain the desired soundness security. As this is only done for the ciphertexts encrypting the MAC keys this is not a problem in practice.

Protocol Π_{ZKPoK}: Commitment, Challenge and Response Phases

The protocol is parametrized by an integer parameter U and a $\mathsf{flag} \in \{\mathsf{Diag}, \perp\}$.

INPUT: Each (honest) party P_i enters U plaintexts $\mathbf{m}_i \in \mathbb{R}_p^U$ (considered as elements of $\mathbb{R}_{q_1}^U$) and U randomness triples $R_i \in \mathbb{R}_{q_1}^{U\times 3}$, where each row of R_i $(r_i^{(j,1)}, r_i^{(j,2)}, r_i^{(j,3)})$ is generated from $\mathcal{RC}\left(\sigma^2, 0.5, N\right)$. For honest P_i we therefore have, for all $j \in [U]$, $\|\mathbf{m}_i^{(j)}\|_\infty \leq \frac{p}{2}$ and $\|r_i^{(j,k)}\|_\infty \leq \rho_k$, where $\rho_1 = \rho_2 = 20$ and $\rho_3 = 1$. We write $V = 2 \cdot U - 1$.

Commitment Phase: Commit

1. P_i computes the BGV encryptions by applying the BGV encryption algorithm to the j plaintext/randomness vectors in turn to obtain $C_i \leftarrow \mathsf{Enc}(\mathbf{m}_i, R_i; \mathfrak{pk}) \in \mathbb{R}_{q_1}^{U\times 2}$.
2. The players broadcast C_i.
3. Each P_i samples V pseudo-plaintexts $\mathbf{y}_i \in \mathbb{R}_{q_1}^V$ and pseudo-randomness vectors $S_i = (s_i^{(j,k)}) \in \mathbb{R}_{q_1}^{V\times 3}$ such that, for all $j \in [V]$, $\|\mathbf{y}_i^{(j)}\|_\infty \leq 2^{\mathsf{ZK\_sec}-1} \cdot p$ and $\|s_i^{(j,k)}\|_\infty \leq 2^{\mathsf{ZK\_sec}} \cdot \rho_k$.
4. Party P_i computes $A_i \leftarrow \mathsf{Enc}(\mathbf{y}_i, S_i; \mathfrak{pk}) \in \mathbb{R}_{q_1}^{V\times 2}$.
5. The players broadcast A_i.

Challenge Phase: Chal

1. Parties call agree on a random vector $\mathbf{e} = (e_i)$ such that $\mathbf{e} \in \left\{\{X^i\}_{i=0\ldots,2\cdot N-1}\right\}^U$ if $\mathsf{flag} = \perp$ and $\mathbf{e} \in \{0,1\}^U$ if $\mathsf{flag} = \mathsf{Diag}$.

Response Phase: $\mathsf{Response}$

1. Parties define the $V \times U$ matrix $M_{\mathbf{e}}$ where

$$M_{\mathbf{e}}^{(k,l)} = \begin{cases} e_{k-l+1} & \text{if } 1 \leq k-l+1 \leq U \\ 0 & \text{otherwise} \end{cases}$$

2. Each P_i computes $\mathbf{z}_i \leftarrow \mathbf{y}_i + M_{\mathbf{e}} \cdot \mathbf{m}_i$ and $T_i \leftarrow S_i + M_{\mathbf{e}} \cdot R_i$.
3. Party P_i sets $\mathsf{resp}_i \leftarrow (\mathbf{z}_i, T_i)$, and broadcasts resp_i.

Figure 3: Protocol for global proof of knowledge of a set of ciphertexts: Part I

Protocol Π_{ZKPoK}: Verification Phase

Verification Phase: Verify

1. Each party P_i computes $D_i \leftarrow \mathsf{Enc}(\mathbf{z}_i, T_i; \mathfrak{pk})$.
2. The parties compute $A \leftarrow \sum_{i=1}^n A_i$, $C \leftarrow \sum_{i=1}^n C_i$, $D \leftarrow \sum_{i=1}^n D_i$, $T \leftarrow \sum_{i=1}^n T_i$ and $\mathbf{z} \leftarrow \sum_{i=1}^n \mathbf{z}_i$.
3. The parties check whether $D = A + M_{\mathbf{e}} \cdot C$, and then whether the following inequalities hold, for $j \in [V]$,

$$\|\mathbf{z}^{(j)}\|_\infty \leq n \cdot 2^{\mathsf{ZK\_sec}} \cdot p, \quad \|T^{(j,k)}\|_\infty \leq 2 \cdot n \cdot 2^{\mathsf{ZK\_sec}} \cdot \rho_k \text{ for } k = 1, 2, 3.$$

4. If $\mathsf{flag} = \mathsf{Diag}$ then the proof is rejected if $\mathbf{z}^{(j)}$ is not a constant polynomial (i.e. a "diagonal" plaintext element).
5. If all checks pass, the parties output C.

Figure 4: Protocol for global proof of knowledge of a set of ciphertexts: Part II

Advanced Protocols

This section details the protocols needed to perform complex functionalities on LSSS style secret shared values in MAMBA. As well as the documentation here further, details can be found at

```
$(HOME)/Documentation/Compiler_Documentation/index.html
```

under the heading `Files`.

The protocols break the "arithmetic circuit" model of computation, as they make heavy use of pre-processed data and the ability to Open shared values. In particular we assume three lists of pre-processed data:

$$\mathsf{MultList}, \quad \mathsf{SquaresList}, \quad \mathsf{BitsList}.$$

An entry on the MultList is of the form $(\langle a\rangle, \langle b\rangle, \langle c\rangle)$ where $c = a \cdot b \pmod p$, an entry on the SquaresList is of the form $(\langle a\rangle, \langle b\rangle)$ where $b = a^2 \pmod p$, whilst an entry on the BitsList is of the form $\langle b\rangle$ where $b \in \{0, 1\}$. We also assume a function Random() which can generate a random secret sharing (this can be implemented by just taking the first element from a multiplication triple). We add and multiply secret shared elements in what follows using the notation

$$\langle a + b\rangle \leftarrow \langle a\rangle + \langle b\rangle, \quad \langle a \cdot b\rangle \leftarrow \langle a\rangle \cdot \langle b\rangle.$$

The protocols in this chapter allow us to do more advanced operations, without resorting to using full blown arithmetic circuits. We describe here what we have implemented as a form of documentation, both for us and for others. Many of the protocol ideas can be found in the five documents

- Unconditionally Secure Constant-Rounds Multi-party Computation for Equality, Comparison, Bits and Exponentiation. *TCC 2006*, [DFK+06].
- Improved Primitives for Secure Multiparty Integer Computation. *SCN 2010*, [CdH10].
- D9.2 of the EU project *secureSCM*, [sec].
- Secure Computation with Fixed-Point Numbers *FC 2010* [CS10].
- Secure Computation on Floating Point Numbers *NDSS 2013* [ABZS13].

Many of the protocols operating on integers make use of a statistical security parameter κ. If the integer being operated on is k bits long, then we often require $(k + \kappa) < \log_2 p$. For ease of implementation of the protocols we recommend k is always a power of two, and we assume this in the write up below. If this is not the case, obvious tweaks can be made to the protocols.

Due to experience with the SPDZ system we prefer logarithmic round protocols over constant round protocols, it appears that in practice the logarithmic round protocols outperform the constant round ones. The MAMBA compiler can execute non-constant round protocols, by flicking a compile time switch. But here we only document logarithmic round protocols.

Basic Protocols

Inv($\langle x \rangle$):

This produces a share $\langle z \rangle$ of $1/x \pmod p$, with an abort if $x = 0$.

1. $\langle a \rangle \leftarrow$ Random().
2. $\langle y \rangle \leftarrow \langle a \rangle \cdot \langle x \rangle$.
3. $y \leftarrow$ Open($\langle y \rangle$).
4. If $y = 0$ then abort.
5. $t \leftarrow 1/y \pmod p$.
6. $\langle z \rangle \leftarrow t \cdot \langle a \rangle$.
7. Return $\langle z \rangle$.

MAMBA Example: To obtain the inverse of a `sint` or a `cint` can be done as follows:

```
from Compiler import floatingpoint
d = sint(5)
d_inv =  floatingpoint.Inv(d)
print_ln("inverse is correct if 1: %s", (d*d_inv).reveal())
```

Ran$_p^*$():

This produces a random sharing of a value x and its inverse $1/x$. This is faster than generating x, and then performing the above operaton.

1. Take a triple $(\langle a \rangle, \langle b \rangle, \langle c \rangle)$ from MultList.
2. $c \leftarrow$ Open($\langle c \rangle$).
3. If $c = 0$ then return to the first step.
4. $\langle a^{-1} \rangle \leftarrow c^{-1} \cdot \langle b \rangle$.
5. Output $(\langle a \rangle, \langle a^{-1} \rangle)$.

This function does not exist "as is" in the MAMBA language, as it is used in place within the python compiler. Thus it is only here for documentation reasons.

PreMult($\langle a_1 \rangle, \ldots, \langle a_t \rangle, T$):

This computes the prefix multiplication, i.e. the values

$$\langle a_{i_0,i_1} \rangle = \left\langle \prod_{i=i_0}^{i_1} a_i \right\rangle$$

where $(i_0, i_1) \in T$ and $1 \leq i_0 \leq i_1 \leq t$.

1. For $i \in [0, \ldots, t]$ do.
   (a) $(\langle b_i \rangle, \langle b_i^{-1} \rangle) \leftarrow$ Ran$_p^*$().
2. For $i \in [0, \ldots, t]$ do.

(a) $\langle t \rangle \leftarrow \langle b_{i-1} \rangle \cdot \langle a_i \rangle$.

(b) $\langle d_i \rangle \leftarrow \langle t \rangle \cdot \langle b_i^{-1} \rangle$.

(c) $d_i \leftarrow \mathsf{Open}(\langle d_i \rangle)$.

3. For $(i_0, i_1) \in T$

(a) $d_{i_0,i_1} \leftarrow \prod_{i=i_0}^{i_1} d_i$.

(b) $\langle a_{i_0,i_1} \rangle \leftarrow d_{i_0,i_1} \cdot \langle b_{i_0-1}^{-1} \rangle \cdot \langle b_{i_1} \rangle$.

Again, this function does not exist "as is" in the MAMBA language, as it is used in place within the python compiler. Thus it is only here for documentation reasons.

Bit Oriented Operations

OR($\langle a \rangle, \langle b \rangle$):

This computes the logical OR of two input shared bits:

1. Return $\langle a \rangle + \langle b \rangle - \langle a \rangle \cdot \langle b \rangle$.

MAMBA Example: To obtain the `or` of two `sint` or `cint` numbers you can:

```
a= sint(1)
b= sint(0)
print_ln("or is correct if 1: %s", (a or b).reveal())
```

XOR($\langle a \rangle, \langle b \rangle$):

This computes the logical XOR of two input shared bits:

1. Return $\langle a \rangle + \langle b \rangle - 2 \cdot \langle a \rangle \cdot \langle b \rangle$.

MAMBA Example: To obtain the `xor` of two `sint` or `cint` numbers you can:

```
from Compiler import floatingpoint
a= sint(1)
b= sint(0)
print_ln("or is correct if 1: %s", floatingpoint.xor_op(a, b).reveal())
```

KOp($\odot, \langle a_1 \rangle, \ldots, \langle a_k \rangle, k$):

This computes the operation $\langle p \rangle = \bigodot_{i=1}^{k} \langle a_i \rangle$ given a binary operator $\odot$.

1. If $k > 1$ then
   (a) For $i \in [1, \ldots, k/2]$ do
      i. $\langle u_i \rangle \leftarrow \langle a_{2 \cdot i} \rangle \odot \langle a_{2 \cdot i - 1} \rangle$.
   (b) $\langle p \rangle \leftarrow$ KOp$(\odot, \langle u_{k/2} \rangle, \ldots, \langle u_1 \rangle, k/2)$.
2. Else
   (a) $\langle p \rangle \leftarrow \langle a_1 \rangle$.
3. Return $\langle p \rangle$.

MAMBA Example: Note that we basically want to achieve a construction capable to call any function in an iterative fashion reducing computation time. In this sense a call to the function could be perform in the following way:

```
from Compiler import floatingpoint

# addition
def addition(a, b):
    return a + b

ar=[1]*16
# (k is exctracted from ar directly on the implementation)
print_ln("KOpL is correct if 32: %s", (floatingpoint.KOpL(func,ar)).reveal())
```

PreOp$(\odot, \langle a_1\rangle, \ldots, \langle a_k\rangle, k)$**:**

This computes the prefix operator $\langle p_j\rangle = \odot_{i=1}^{j}\langle a_i\rangle$, for $1 \leq j \leq k$.

1. For $i \in [1, \ldots, \log_2 k]$ do
   (a) For $j \in [1, \ldots, k/2^i]$ do
      i. $y \leftarrow 2^{i-1} + j \cdot 2^i$.
      ii. For $z \in [1, \ldots, 2^{i-1}]$ do
         A. $\langle a_{y+z}\rangle \leftarrow \langle a_y\rangle \odot \langle a_{y+z}\rangle$.
2. Return $(\langle a_1\rangle, \ldots, \langle a_k\rangle)$.

MAMBA Example: Similarly, we basically want to achieve a construction capable to call any function in an iterative fashion reducing computation time. In this case, however, we return all the list of intermediate values. The function call could be performed in the following way:

```
from Compiler import floatingpoint

def addition(a, b):
    return a + b

def addition_triple(a, b, c):
  # c is a boolean parameter
    return a + b

e = sint(2)

ar = [e]*16
# k is stracted from ar.
print_ln("PreOpL is correct if 32: %s", (floatingpoint.PreOpL(addition_triple,ar))
    [15].reveal())
print_ln("PreOpN is correct if 32: %s", (floatingpoint.PreOpN(addition,ar))[15].reveal
    ())
```

Note that both methods are implementations of the functionality with slightly different communication and round complexity. With the `PreOpL` function corresponding to the pseudo-code above.

Solved-Bits$(BitsList, k)$**:**

This outputs a shared integer x in the range $[0, \ldots, 2^k)$ with $2^k < p$ and the shared bits making up its binary representation.

1. Take k shares $\{\langle x_i\rangle\}_{i=0}^{k-1}$ from BitsList.
2. $\langle x\rangle \leftarrow \sum_{i=0}^{k-1} 2^i \cdot \langle x_i\rangle$.
3. Output $(\langle x\rangle, \{\langle x_i\rangle\}_{i=0}^{k-1})$.

To ease notation in what follows we write $\langle x\rangle_B = \{\langle x_i\rangle\}_{i=0}^{k-1}$.

MAMBA Example: We can reconstruct a number from its bits as follows:

```
from Compiler import floatingpoint
a = [sint(0)]*program.bit_length
# k is taken from a
print_ln("solved_bits is correct if 0: %s", floatingpoint.SolvedBits(a, program.
    bit_length).reveal())
```

PRandM(k, m, κ)**:**

This generates two random shares $r' \in [0, \ldots, 2^{k+\kappa-m} - 1]$ and $r \in [0, \ldots, 2^m - 1]$, along with the shares the bits of r.

1. $\langle r \rangle, \langle r \rangle_B \leftarrow$ Solved-Bits(m).
2. $\langle r' \rangle, \langle r' \rangle_B \leftarrow$ Solved-Bits$(k + \kappa - m)$.
3. Return $\langle r' \rangle, \langle r \rangle, \langle r \rangle_B$.

MAMBA Example: We obtain the randomness and its bits as follows:

```
from Compiler import comparison

# x, y,z are returned values
x = sint()
y = sint()
z = [sint() for i in range(3)]

# k, m, kappa are parameters
k = 5
m = 3
kappa = 7
comparison.PRandM(x, y, z, k, m, kappa)
```

CarryOut$(\langle a \rangle_B, \langle b \rangle_B, k)$**:**

This protocol computes the carry-out of a binary addition of two k bit shared values, when presented via shared bits. The protocol can easily be adapted to the case when either the bits of a, or the bits of b, are given in the clear. We give a logarithmic round version, which requires $\log k$ rounds of interaction. It requires a sub-routine CarryOutAux which we give below.

1. For $i \in [0, \ldots, k-1]$ do
   (a) $\langle d_i \rangle_B \leftarrow (\mathsf{XOR}(\langle a_i \rangle, \langle b_i \rangle), \langle a_i \rangle \cdot \langle b_i \rangle)$. [Note, $\langle d_i \rangle_B$ is a set of two shared bits, one being the XOR of a_i and b_i, whilst the other the AND].
2. $\langle d \rangle_B \leftarrow$ CarryOutAux$(\langle d_{k-1} \rangle_B, \ldots, \langle d_0 \rangle_B, k)$.
3. $(\langle p \rangle, \langle g \rangle) \leftarrow \langle d \rangle_B$.
4. Return $\langle g \rangle$.

MAMBA Example: The carry out operation is executed as follows:

```
from Compiler import comparison

res = sint() # last carry bit in addition of a and b
a = [cint(i) for i in [1,0]] # array of clear bits
b = [sint(i) for i in [0,1]] # array of secret bits (same length as a)
c = 0 # initial carry-in bit
kappa = 16
comparison.CarryOut(res, a, b, c, 16)
```

$\mathsf{CarryOutAux}(\langle d_k\rangle_B, \ldots, \langle d_1\rangle_B, k, \kappa)$**:**

This function uses the $\circ$ operator for carry propagation on two bit inputs which is defined as

$$\circ : \begin{cases} \{0,1\}^2 \times \{0,1\}^2 & \longrightarrow & \{0,1\} \\ (p_2, g_2) \circ (p_1, g_1) & \longmapsto & (p_1 \wedge p_2, g_2 \vee (p_2 \wedge g_1)) \end{cases}$$

This is computed using arithmetic operations (i.e. where the values are held as bits modulo p) as $(p, g) = (p_2, g_2) \circ (p_1, g_1)$ via

$$\begin{aligned} p &= p_1 \cdot p_2, \\ g &= g_2 + p_2 \cdot g_1. \end{aligned}$$

Given this operation the function $\mathsf{CarryOutAux}$ is defined by, which is just a specialisation of the protocol KOp above,

1. If $k > 1$ then
   (a) For $i \in [1, \ldots, k/2]$ do
      i. $\langle u_i\rangle_B \leftarrow \langle d_{2\cdot i}\rangle_B \circ \langle d_{2\cdot i-1}\rangle_B$.
   (b) $\langle d\rangle_B \leftarrow \mathsf{CarryOutAux}(\langle u_{k/2}\rangle_B, \ldots, \langle u_1\rangle_B, k/2)$.
2. Else
   (a) $\langle d\rangle_B \leftarrow \langle d_1\rangle_B$.
3. Return $\langle d\rangle_B$.

MAMBA Example: This method is thought as a subroutine for `CarryOut` and should not be used outside that context. the code is invoked in the following way:

```
from Compiler import comparison
# this is how it is invoked, where res, is the return, and 16 is the kappa
# k can be extracted from the array size
comparison.CarryOutAux(res, [::d -1], 16)
# it could be interpreted as follows when called:
kappa = 16
x = [cint(i) for i in [1,0]]
res = sint()
comparison.CarryOut(z, x, 16)
```

$\mathsf{BitAdd}((\langle a_{k-1}\rangle, \ldots, \langle a_0\rangle), (\langle b_{k-1}\rangle, \ldots, \langle b_0\rangle), k)$**:**

This function also makes use of the operator $\circ$. The inputs are shared bits. The case where one set of inputs is in the clear is obviously more simple, and we do not detail this here.

1. For $i \in [0, \ldots, k-1]$ do
   (a) $\langle d_i\rangle_B \leftarrow (\mathsf{XOR}(\langle a_i\rangle, \langle b_i\rangle), \langle a_i\rangle \cdot \langle b_i\rangle)$.
2. $\langle c_{k-1}, t_{k-1}\rangle, \ldots, \langle c_0, t_0\rangle \leftarrow \mathsf{PreOp}(\circ, \langle d_{k-1}\rangle_B, \ldots, \langle d_0\rangle_B, k)$.
3. $\langle s_0\rangle \leftarrow \mathsf{XOR}(\langle a_0\rangle, \langle a_1\rangle)$.
4. For $i \in [1, \ldots, k-1]$ do
   (a) $\langle s_i\rangle \leftarrow \langle a_i\rangle + \langle b_i\rangle + \langle c_{i-1}\rangle - 2 \cdot \langle c_i\rangle$.
5. $\langle s_k\rangle \leftarrow \langle c_{k-1}\rangle$.
6. Return $(\langle s_k\rangle, \ldots, \langle s_0\rangle)$.

MAMBA Example: The addition of two numbers expressed in bits, can be performed as follows:

```
from Compiler import floatingpoint
a_bits = [sint(i) for i in [0,1,0,1,1]]
b_bits = [sint(i) for i in [0,1,0,1,1]]
# k can be extracted from the array size
b = floatingpoint.BitAdd(a_bits, b_bits)
```

BitLT$(a, \langle b \rangle_B, k)$**:**

This computes the sharing of the bit $a < b$, where a is a public value. Both a and b are assumed to be k bit values, with $a = \sum_{i=0}^{k-1} a_i \cdot 2^i$ and $b = \sum_{i=0}^{k-1} b_i \cdot 2^i$.

1. For $i \in [0, \ldots, k-1]$
   (a) $\langle b'_i \rangle \leftarrow 1 - \langle b_i \rangle$.
2. $\langle s \rangle \leftarrow 1 - \mathsf{CarryOut}((a_{k-1}, \ldots, a_0), \langle b \rangle_B)$.
3. Return $\langle s \rangle$.

MAMBA Example: Comparing an open register with a secret shared number decomposed in bits can be achieved as follows:

```
from Compiler import comparison
x = cint(5)
y = [sint(i) for i in [1,0,1]]
z = sint()
kappa = 16
# k can be extracted from the array size
# in this case the bit that is the answer is contained in z
comparison.BitLTL(z, x, y, kappa)
```

BitDec$(\langle a \rangle, k, m)$**:**

This outputs the m least significant bits in the 2's complement representation of $a \in \mathbb{Z}_{\langle k \rangle}$.

1. $\langle r' \rangle, \langle r \rangle, \langle r \rangle_B \leftarrow \mathsf{PRandM}(k, m, \kappa)$.
2. $c \leftarrow \mathsf{Open}(\langle a \rangle + 2^k + 2^{k+\kappa} - \langle r \rangle - 2^m \cdot \langle r' \rangle)$.
3. $(\langle a_{m-1} \rangle, \ldots, \langle a_0 \rangle) \leftarrow \mathsf{BitAdd}(c, (\langle r_{m-1} \rangle, \ldots, \langle r_0 \rangle))$.
4. Return $(\langle a_{m-1} \rangle, \ldots, \langle a_0 \rangle)$.

MAMBA Example: A secret shared value can be decomposed into bits as shown in the following snippet:

```
from Compiler import comparison
a = sint(23)
k = 5
m = 5
kappa = 20
# where b is bit array of type sint
b = floatingpoint.BitDec(a, k, m, kappa)
```

Arithmetic with Signed Integers

In this section we define basic arithmetic on signed integers. We define $\mathbb{Z}_{\langle k \rangle}$ as the set of integers $\{x \in \mathbb{Z} : -2^{k-1} \leq x \leq 2^{k-1} - 1\}$, which we embed into $\mathbb{F}_p$ via the map $x \mapsto x \pmod{p}$.

TruncPR$(\langle a \rangle, k, m, \kappa)$**:**

An approximate truncation algorithm which is faster than a fully accurate Trunc. Given $a \in \mathbb{Z}_{\langle k \rangle}$, $m \in [1, \ldots, k-1]$ this outputs $\lfloor a/2^m \rfloor + u$ where u is a random (and unknown) bit. It gives the actual correct nearest integer with probability $1 - \alpha$, where α is the distance between $a/2^m$ and that integer.

1. $\langle r' \rangle, \langle r \rangle, \langle r \rangle_B \leftarrow \mathsf{PRandM}(k, m, \kappa)$.
2. $c \leftarrow \mathsf{Open}(\langle a \rangle + 2^{k-1} + \langle r \rangle + 2^m \cdot \langle r' \rangle)$.
3. $c' \leftarrow c \pmod{2^m}$.
4. $t \leftarrow 1/2^m \pmod{p}$.
5. $\langle d \rangle \leftarrow t \cdot (\langle a \rangle - c' + \langle r \rangle)$.
6. Return $\langle d \rangle$.

MAMBA Example: A secret shared fractional register can be approximately truncated as follows:

```
from Compiler import floatingpoint
a = sint(23)
k = 5
m = 3
kappa = 20
# where b is a register of type sint
b=floatingpoint.TruncPr(a, k, m, kappa)
```

Mod2m$(\langle a_{prime} \rangle, \langle a \rangle, k, m, \kappa, signed)$**:**

Given $a \in \mathbb{Z}_{\langle k \rangle}$, $m \in [1, \ldots, k-1]$ this outputs $a \pmod{2^m}$. Use this protocol when $m > 1$, for $m = 1$ use Mod2 below.

1. $\langle r' \rangle, \langle r \rangle, \langle r \rangle_B \leftarrow \mathsf{PRandM}(k, m, \kappa)$.
2. $c \leftarrow \mathsf{Open}(\langle a \rangle + 2^{k-1} + \langle r \rangle + 2^m \cdot \langle r' \rangle)$.
3. $c' \leftarrow c \pmod{2^m}$.
4. $\langle u \rangle \leftarrow \mathsf{BitLT}(c', (\langle r_{m-1} \rangle, \ldots, \langle r_0 \rangle), m)$.
5. $\langle a' \rangle \leftarrow c' - \langle r \rangle + 2^m \cdot \langle u \rangle$.
6. Return $\langle a' \rangle$.

MAMBA Example: The `mod` to a power of 2 of a secret shared integer register can be obtain as follows:

```
from Compiler import comparison

a_prime = sint(0) # a % 2 ^ m
a = sint(100)
k = 16 # bit length of a
m = 2 # modulo of 2^m
kappa = 8
signed = True # True/False

# where a is a register of type sint
r_dprime, r_prime, c, c_prime, u, t, c2k1 = \
    comparison.Mod2m(a_prime, a, k, m, kappa, signed)
```

Trunc$(\langle a\rangle, k, m, kappa)$**:**

An exact version of Trunc above

1. $\langle a'\rangle \leftarrow \mathsf{Mod2m}(\langle a\rangle, k, m, \kappa)$.
2. $t \leftarrow 1/2^m \pmod p$.
3. $\langle d\rangle \leftarrow t \cdot (\langle a\rangle - \langle a'\rangle)$.
4. Return $\langle d\rangle$.

Below we will give a version of Trunc in which m is kept secret shared.

MAMBA Example: You truncate a number as follows:

```
from Compiler import floatingpoint
# a = sint(23)
# k = 5
# m = 3
# kappa = 20
# where a is a register of type sint
a= floatingpoint.Trunc(a, k, m, kappa)
```

Mod2$(\langle a\rangle, k, \kappa, signed)$**:**

1. $\langle r'\rangle, \langle r\rangle, \langle r_0\rangle \leftarrow \mathsf{PRandM}(k, 1, \kappa)$.
2. $c \leftarrow \mathsf{Open}(\langle a\rangle + 2^{k-1} + \langle r\rangle + 2 \cdot \langle r'\rangle)$.
3. $\langle a_0\rangle \leftarrow c_0 + \langle r_0\rangle - 2 \cdot c_0 \cdot \langle r_0\rangle$.
4. Return $\langle a_0\rangle$.

MAMBA Example: You obtain the modulo two of a number as follows:

```
from Compiler import comparison

a = sint(1)
a_0 = sint()
k = 1
kappa = 8
```

```
signed = False
# y stores the result of X % 2
comparison.Mod2(a_0, A, k, kappa, signed)
```

$\mathsf{LTZ}(\langle a\rangle, k, \kappa)$:

Given $a \in \mathbb{Z}_{\langle k\rangle}$ this tests whether $a < 0$ or not, resulting in a shared bit.

1. $\langle s\rangle \leftarrow -\mathsf{Trunc}(\langle a\rangle, k, k-1)$.

MAMBA Example: You determine whether a number is less than zero as follows:

```
from Compiler import comparison
a = sint(1)
b = sint()
k=80
kappa=40
# b stores the result of x < 0
comparison.LTZ(b, a, k, kappa)
```

Like many commands in this section this can be abbreviated to

```
b=a<0
```

In which case the default value of $\kappa = 40$ is chosen (when using a 128-bit prime modulus), this default value can be altered by using the command

```
program.security = 100
```

The default value of $k = 64$ is used in this setting, and this can be altered by executing

```
program.bit_length = 40
```

The requirement is that $k + \kappa$ must be less than the bit length of the prime p.

$\mathsf{EQZ}(\langle a\rangle, k, \kappa)$:

Given $a \in \mathbb{Z}_{\langle k\rangle}$ this tests whether $a = 0$ or not, resulting in a shared bit.

1. $\langle r'\rangle, \langle r\rangle, \langle r\rangle_B \leftarrow \mathsf{PRandM}(k, k, \kappa)$.
2. $c \leftarrow \mathsf{Open}(\langle a\rangle + 2^{k-1} + 2^k \cdot \langle r'\rangle + \langle r\rangle)$.
3. Let $c_{k-1}, \ldots, c_0$ be the bits of c.
4. For $i \in [0, \ldots, k-1]$ do
   (a) $\langle d_i\rangle \leftarrow c_i + \langle r_i\rangle - 2 \cdot c_i \cdot \langle r_i\rangle$.
5. $\langle z\rangle \leftarrow 1 - \mathsf{KOp}(\mathsf{OR}, \langle d_{k-1}\rangle, \ldots, \langle d_0\rangle, k)$.
6. Return $\langle z\rangle$.

MAMBA Example: You determine whether a number is equal to zero as follows:

```
from Compiler import floatingpoint
a = sint(1)
b = sint()
k = 80
kappa = 40
# b stores the result of x == 0
floatingpoint.EQZ(b, a, k, kappa)
```

Comparison Operators:

We can now define the basic comparison operators on shared representations from $\mathbb{Z}_{\langle k \rangle}$.

Operator	Protocol Name	Construction
$a > 0$	$\mathsf{GTZ}(\langle \mathsf{a} \rangle)$	$\mathsf{LTZ}(-\langle a \rangle)$
$a \leq 0$	$\mathsf{LEZ}(\langle \mathsf{a} \rangle)$	$1 - \mathsf{LTZ}(-\langle a \rangle)$
$a \geq 0$	$\mathsf{GEZ}(\langle \mathsf{a} \rangle)$	$1 - \mathsf{LTZ}(\langle a \rangle)$
$a = b$	$\mathsf{EQ}(\langle \mathsf{a} \rangle, \langle \mathsf{b} \rangle)$	$\mathsf{EQZ}(\langle a \rangle - \langle b \rangle)$
$a < b$	$\mathsf{LT}(\langle \mathsf{a}, \langle \mathsf{b} \rangle \rangle)$	$\mathsf{LTZ}(\langle a \rangle - \langle b \rangle)$
$a > b$	$\mathsf{GT}(\langle \mathsf{a}, \langle \mathsf{b} \rangle \rangle)$	$\mathsf{LTZ}(\langle b \rangle - \langle a \rangle)$
$a \leq b$	$\mathsf{LE}(\langle \mathsf{a}, \langle \mathsf{b} \rangle \rangle)$	$1 - \mathsf{LTZ}(\langle b \rangle - \langle a \rangle)$
$a \geq b$	$\mathsf{GE}(\langle \mathsf{a}, \langle \mathsf{b} \rangle \rangle)$	$1 - \mathsf{LTZ}(\langle a \rangle - \langle b \rangle)$

Addition, Multiplication in $\mathbb{Z}_{\langle k \rangle}$

Addition and multiplication $\odot$ of two elements $\langle a \rangle, \langle b \rangle$ to obtain $\langle c \rangle$, where $a, b, c \in \mathbb{Z}_{\langle k \rangle}$ is then easy to define by performing

1. $\langle d \rangle \leftarrow \langle a \rangle \odot \langle b \rangle$.
2. $\langle c \rangle \leftarrow \mathsf{Mod2m}(\langle d \rangle, k', k)$, where $k' = k + 1$ is $\odot = +$ and $k' = 2 \cdot k$ if $\odot = \cdot$.
3. Return $\langle c \rangle$.

These functions are not directly callable from MAMBA, they are included here purely for documentation reasons.

$\mathsf{Pow2}(\langle a \rangle, k, \kappa)$**:**

This computes $\langle 2^a \rangle$ where $a \in [0, \ldots, k)$

1. $m \leftarrow \lceil \log_2 k \rceil$.
2. $\langle a_{m-1} \rangle, \ldots, \langle a_0 \rangle \leftarrow \mathsf{BitDec}(\langle a \rangle, m, m)$.
3. For $i \in [0, \ldots, m-1]$ do
   (a) $\langle v_i \rangle \leftarrow 2^{2^i} \cdot \langle a_i \rangle + 1 - \langle a_i \rangle$.
4. $\langle x_0 \rangle, \ldots, \langle x_{m-1} \rangle \leftarrow \mathsf{PreMult}(\langle v_0 \rangle, \ldots, \langle v_{m-1} \rangle, \{(1,1), \ldots, (1,m)\})$
5. Return $\langle x_{m-1} \rangle$.

MAMBA Example: You can obtain the value of two raised to a secret shared number as follows:

```
from Compiler import floatingpoint
a = sint(23)
l = 32
kappa = 20
# y stores the result of 2^23
y =floatingpoint.Pow2(a, l, kappa)
```

$\mathsf{B2U}(\langle a\rangle, k, \kappa)$**:**

This converts the integer $a \in [0, \ldots, k)$ into unary form. It outputs k bits, of which the last a bits are zero to one, with the others set to zero.

1. $\langle 2^a\rangle \leftarrow \mathsf{Pow2}(\langle a\rangle, k)$.
2. $\langle r'\rangle, \langle r\rangle, \langle r\rangle_B \leftarrow \mathsf{PRandM}(k, k, \kappa)$.
3. $c \leftarrow \mathsf{Open}(\langle 2^a\rangle + \langle r\rangle + 2^k \cdot \langle r'\rangle)$.
4. Let $c_{k-1}, \ldots, c_0$ be the bits of c.
5. For $i \in [0, \ldots, k = 1]$ do
   (a) $\langle x_i\rangle \leftarrow c_i + \langle r_i\rangle - 2 \cdot c_i \cdot \langle r_i\rangle$.
6. $\langle y_{k-1}\rangle, \ldots, \langle y_0\rangle \leftarrow \mathsf{PreOp}(\mathsf{OR}, \langle x_{k-1}\rangle, \ldots, \langle x_0\rangle, k)$.
7. For $i \in [0, \ldots, k = 1]$ do
   (a) $\langle a_i\rangle \leftarrow 1 - \langle y_i\rangle$.
8. Return $\langle a_0\rangle, \ldots, \langle a_{k-1}\rangle$.

Note, in the function Trunc below we also require the value $\langle 2^a\rangle$ to be returned so as to avoid recomputing it.

MAMBA Example: You can transform a number into its unary form as follows:

```
from Compiler import floatingpoint
a = sint(3)
l=5
kappa=20
b,c = floatingpoint.B2U(a, l, kappa)
```

$\mathsf{Trunc}(\langle a\rangle, k, \langle m\rangle, \kappa)$**:**

This does the same operation as Trunc above, but m is now secret shared.

1. $\langle x_0\rangle, \ldots, \langle x_{k-1}\rangle, \langle 2^m\rangle \leftarrow \mathsf{B2U}(\langle m\rangle, k)$.
2. $\langle 2^{-m}\rangle \leftarrow \mathsf{Inv}(\langle 2^m\rangle)$.
3. $\langle r''\rangle, \langle r\rangle, \langle r\rangle_B \leftarrow \mathsf{PRandM}(k, k, \kappa)$.
4. $\langle r'\rangle \leftarrow \sum_{i=0}^{k-1} 2^i \cdot \langle x_i\rangle \cdot \langle r_i\rangle$
5. $c \leftarrow \mathsf{Open}(\langle a\rangle + \langle r''\rangle + \langle r\rangle)$.
6. For $i \in [1, \ldots, k = 1]$ do $c'_i \leftarrow c \pmod{2^i}$.
7. $\langle c''\rangle \leftarrow \sum_{i=1}^{k-1} c'_i \cdot (\langle x_{i-1}\rangle - \langle x_i\rangle)$.
8. $\langle d\rangle \leftarrow \mathsf{LT}(\langle c''\rangle, \langle r'\rangle, k)$.
9. $\langle b\rangle \leftarrow (\langle a\rangle - \langle c''\rangle + \langle r'\rangle) \cdot \langle 2^{-m}\rangle - \langle d\rangle$.
10. Return $\langle b\rangle$.

MAMBA Example: You truncate a number as follows:

```
from Compiler import floatingpoint
a = sint(23)
k = 5
m = sint(3)
kappa = 20
# where a is a register of type sint
b= floatingpoint.Trunc(a, k, m, kappa)
```

Mod2m$(a_prime, \langle a \rangle, k, \langle m \rangle, \kappa)$**:**

This does the same operation as Mod2m above, but m is now secret shared.

1. $\langle x_0 \rangle, \ldots, \langle x_{k-1} \rangle, \langle 2^m \rangle \leftarrow$ B2U$(\langle m \rangle, k)$.
2. $\langle 2^{-m} \rangle \leftarrow$ Inv$(\langle 2^m \rangle)$.
3. $\langle r'' \rangle, \langle r \rangle, \langle r \rangle_B \leftarrow$ PRandM(k, k, κ).
4. $\langle r' \rangle \leftarrow \sum_{i=0}^{k-1} 2^i \cdot \langle x_i \rangle \cdot \langle r_i \rangle$
5. $c \leftarrow$ Open$(\langle a \rangle + \langle r'' \rangle + \langle r \rangle)$.
6. For $i \in [1, \ldots, k = 1]$ do $c'_i \leftarrow c \pmod{2^i}$.
7. $\langle c'' \rangle \leftarrow \sum_{i=1}^{k-1} c'_i \cdot (\langle x_{i-1} \rangle - \langle x_i \rangle)$.
8. $\langle d \rangle \leftarrow$ LT$(\langle c'' \rangle, \langle r' \rangle, k)$.
9. $\langle b \rangle \leftarrow \langle c'' \rangle - \langle r' \rangle + \langle 2^m \rangle \cdot \langle d \rangle$.
10. Return $\langle b \rangle$.

MAMBA Example: The `mod` to a secret shared power of 2 of a secret shared integer register can be obtain as follows:

```
from Compiler import comparison

a_prime = param_a_prime # a % 2^m
a sint(2137)
k = 16
m = sint(2)
kappa = 8
signed = True # True/False, describes a

# where sb is a register of type sint
r_dprime, r_prime, c, c_prime, u, t, c2k1 = \
    comparison.Mod2m(a_prime, a, k, m, kappa, signed)
```

Arithmetic with Fixed Point Numbers

In this section we define basic arithmetic on fixed point numbers. We mainly follow the algorithms given in

- Secure Computation with Fixed-Point Numbers *FC 2010* [CS10].

We define $\mathbb{Q}_{\langle k,f\rangle}$ as the set of rational numbers $\{x \in \mathbb{Q} : x = \overline{x} \cdot 2^{-f}, \overline{x} \in \mathbb{Z}_{\langle k\rangle}\}$. We represent $x \in \mathbb{Q}$ as the integer $x \cdot 2^f = \overline{x} \in \mathbb{Z}_{\langle k\rangle}$, which is then represented in $\mathbb{F}_p$ via the mapping used above. Thus $x \in \mathbb{Q}$ is in the range $[-2^e, 2^e - 2^{-f}]$ where $e = k - f$. As we are working with fixed point numbers we assume that the parameters f and k are public. For our following algorithms to work (in particular fixed point multiplication and division) we require that $q > 2^{2 \cdot k}$. By abuse of notation we write $\langle a\rangle$ to mean $\langle \overline{a}\rangle$, i.e. the secret sharing of the fixed point number a, is actually the secret sharing of the integer representative $\overline{a}$.

Scale$(\langle a\rangle, k, f_1, f_2)$:

Sometimes we want to scale the input fixed point number a from $\mathbb{Q}_{\langle k,f_1\rangle}$ to $\mathbb{Q}_{\langle k,f_2\rangle}$.

1. $m \leftarrow f_2 - f_1$.
2. If $m \geq 0$ then $\langle a'\rangle \leftarrow 2^m \cdot \langle a\rangle$.
3. Else $\langle a'\rangle \leftarrow$ TruncPR$(\langle a\rangle, k, -m)$.
4. Return $\langle a'\rangle$.

This is not directly callable from MAMBA it is here purely for documentation reasons.

FxEQZ, FxLTZ, FxEQ, FxLT, etc:

All of the comparison operators for integers given above carry over to fixed point numbers (if the inputs have the same f-values).

FxAbs$(\langle a\rangle, k, f)$

1. $\langle s\rangle \leftarrow$ LTZ$(\langle a\rangle)$.
2. $\langle a\rangle \leftarrow (1 - 2 \cdot \langle s\rangle) \cdot \langle a\rangle$.
3. Return $\langle a\rangle$.

MAMBA Example: To obtain the absolute value of a number as follows:

```
from Compiler import mpc_math
b = -1.5
sb = sfix(b)

# k and f are extracted from b
# returns unsigned b and receives signed b
ub = mpc_math.abs_fx(sb)
```

FxNeg$(\langle a\rangle, k, f)$

1. Return $-\langle a\rangle$.

MAMBA Example: To obtain the original value times -1 as follows:

```
b = -1.5
sb = sfix(b)
# k and f are extracted from b
# returns the value of signed b times -1.
nb = -sb
```

FxAdd$(\langle a\rangle, \langle b\rangle, k, f)$**:**

Given $a, b \in \mathbb{Q}_{\langle k,f\rangle}$ this is just the integer addition algorithm for elements in $\mathbb{Z}_{\langle k\rangle}$ given above

1. $\langle d\rangle \leftarrow \langle a\rangle + \langle b\rangle$.
2. $\langle c\rangle \leftarrow$ Mod2m$(\langle d\rangle, k+1, k)$.
3. Return $\langle c\rangle$.

Obviously from FxNeg and FxAdd we can define FxSub.

MAMBA Example: To obtain the addition of two fixed point secret shared values you can do as follows:

```
a = sfix(3.5)
b = sfix(1.5)
# k and f are extracted from b or a
#returns secret shared 5
a_plus_b = a+b
```

FxMult$(\langle a\rangle, \langle b\rangle, k, f)$**:**

Given $a, b \in \mathbb{Q}_{\langle k,f\rangle}$ this requires integer multiplication followed by a suitable truncation.

1. $\langle d\rangle \leftarrow \langle a\rangle \cdot \langle b\rangle$.
2. $\langle c\rangle \leftarrow$ TrunkPR$(\langle d\rangle, 2 \cdot k, f)$.
3. Return $\langle c\rangle$.

MAMBA Example: To obtain the multiplication of two fixed point secret shared values you can do as follows:

```
a = sfix(3.5)
b = sfix(1.5)
# k and f are extracted from b or a
#returns secret shared 5.25
a_mult_b = a*b
```

FxDiv$(\langle a\rangle, b, k, f)$**:**

We first give division for when $a, b \in \mathbb{Q}_{\langle k,f\rangle}$ and b is in the clear.

1. Compute $x \in \mathbb{Q}_{\langle k,f\rangle}$ such that $x \approx 1/b$.
2. $\langle y\rangle \leftarrow$ TruncPR$(\overline{x} \cdot \langle a\rangle, k, f)$.
3. Return $\langle y\rangle$.

MAMBA Example: To divide two fixed point values (where only one is secret shared) you can do as follows:

```
a = sfix(3.5)
b = 1.5
# k and f are extracted from b
#returns secret shared 2.333333
a_div_b = a/b
```

FxDiv$(\langle a \rangle, \langle b \rangle, k, f)$**:**

This operation is more complex and we use method of Goldschmidt, which is recommended by Catrina et. al. The following routine makes use of the two subroutines which follow

1. $\theta \leftarrow \lceil \log_2(k/3.5) \rceil$.
2. $\overline{\alpha} \leftarrow 2^{2 \cdot f}$. Note that $\overline{\alpha}$ is the integer representative of 1.0 in $\mathbb{Q}_{\langle k, 2 \cdot f \rangle}$.
3. $\langle w \rangle \leftarrow$ AppRcr$(\langle b \rangle, k, f)$.
4. $\langle x \rangle \leftarrow \overline{\alpha} - \langle b \rangle \cdot \langle w \rangle$.
5. $\langle y \rangle \leftarrow \langle a \rangle \cdot \langle w \rangle$.
6. $\langle y \rangle \leftarrow$ TruncPr$(\langle y \rangle, 2 \cdot k, f)$.
7. For $i \in [1, \ldots, \theta - 1]$ do
   (a) $\langle y \rangle \leftarrow \langle y \rangle \cdot (\overline{\alpha} + \langle x \rangle)$.
   (b) $\langle x \rangle \leftarrow \langle x \rangle^2$.
   (c) $\langle y \rangle \leftarrow$ TruncPr$(\langle y \rangle, 2 \cdot k, 2 \cdot f)$.
   (d) $\langle x \rangle \leftarrow$ TruncPr$(\langle x \rangle, 2 \cdot k, 2 \cdot f)$.
8. $\langle y \rangle \leftarrow \langle y \rangle \cdot (\overline{\alpha} + \langle x \rangle)$.
9. $\langle y \rangle \leftarrow$ TruncPr$(\langle y \rangle, 2 \cdot k, 2 \cdot f)$.
10. Return $\langle y \rangle$.

MAMBA Example: To obtain the division of two fixed point values that are secret shared, you can do as follows:

```
a = sfix(3.5)
b = 1.5
# k and f are extracted from b
#returns secret shared 2.333333
a_div_b = a/b
```

AppRcr$(\langle b \rangle, k, f)$**:**

1. $\overline{\alpha} \leftarrow 2.9142 \cdot 2^k$. Note that $\overline{\alpha}$ is the integer representative of 2.9142 in $\mathbb{Q}_{\langle k, f \rangle}$.
2. $(\langle c \rangle, \langle v \rangle) \leftarrow$ Norm$(\langle b \rangle, k, f)$.
3. $\langle d \rangle \leftarrow \overline{\alpha} - 2 \cdot \langle c \rangle$.
4. $\langle w \rangle \leftarrow \langle d \rangle \cdot \langle v \rangle$.
5. $\langle w \rangle \leftarrow$ TruncPR$(\langle w \rangle, 2 \cdot k, 2 \cdot (k - f))$.
6. Return $\langle w \rangle$.

This is not callable from MAMBA, it is here purely for documentation reasons.

$\mathsf{MSB}(\langle b\rangle, k)$:

Returns index array $\langle z\rangle$ of size k, such that it holds a 1 in the position of the most significative bit of $\langle b\rangle$ and $\langle 0\rangle$ otherwise. This function is used internally on `NormSQ` and `SimplifiedNormSQ`.

1. $\langle s\rangle \leftarrow 1 - 2 \cdot \mathsf{LTZ}(\langle b\rangle, k)$.
2. $\langle x\rangle \leftarrow \langle s\rangle \cdot \langle b\rangle$.
3. $\langle x_{k-1}\rangle, \ldots, \langle x_0\rangle \leftarrow \mathsf{BitDec}(\langle x\rangle, k, k)$.
4. $\langle y_{k-1}\rangle, \ldots, \langle y_0\rangle \leftarrow \mathsf{PreOp}(\mathsf{OR}, \langle x_{k-1}\rangle, \ldots, \langle x_0\rangle, k)$.
5. For $i \in [0, \ldots, k-1]$ do
6. $z \leftarrow (\langle\rangle 0_1), ..., \langle\rangle 0_{k+1-k} \quad \% \quad _2))$
   (a) $\langle z_i\rangle \leftarrow \langle y_i\rangle - \langle y_{i+1}\rangle$.
7. $\langle z_{k-1}\rangle \leftarrow \langle y_{k-1}\rangle$.
8. Return $\langle z\rangle$.

$\mathsf{Norm}(\langle b\rangle, k, f)$:

This returns the value c such that $2^{k-1} \leq c < 2^k$ and v' such that $b \cdot v' = c$, and if $2^{m-1} \leq |b| < 2^m$ then $v' = \pm 2^{k-m}$.

1. $\langle s\rangle \leftarrow 1 - 2 \cdot \mathsf{LTZ}(\langle b\rangle, k)$.
2. $\langle x\rangle \leftarrow \langle s\rangle \cdot \langle b\rangle$.
3. $\langle x_{k-1}\rangle, \ldots, \langle x_0\rangle \leftarrow \mathsf{BitDec}(\langle x\rangle, k, k)$.
4. $\langle y_{k-1}\rangle, \ldots, \langle y_0\rangle \leftarrow \mathsf{PreOp}(\mathsf{OR}, \langle x_{k-1}\rangle, \ldots, \langle x_0\rangle, k)$.
5. For $i \in [0, \ldots, k-2]$ do
   (a) $\langle z_i\rangle \leftarrow \langle y_i\rangle - \langle y_{i+1}\rangle$.
6. $\langle z_{k-1}\rangle \leftarrow \langle y_{k-1}\rangle$.
7. $\langle v\rangle \leftarrow \sum_{i=0}^{k-1} 2^{k-i-1} \cdot \langle z_i\rangle$.
8. $\langle c\rangle \leftarrow \langle x\rangle \cdot \langle v\rangle$.
9. $\langle v'\rangle \leftarrow \langle s\rangle \cdot \langle v\rangle$.
10. Return $(\langle c\rangle, \langle v'\rangle)$.

MAMBA Example: To obtain the norm of a secret shared fix point register you can do as follows:

```
from Compiler import library
kappa = 40
b = sfix(1.5)
#returns the norm
 c, v = library.Norm(b, b.k, b.f, kappa, True)
```

$\mathsf{NormSQ}(\langle b \rangle, k)$:

As above, but now we assume $b \geq 0$, and we also output shares of m and w such that $w = 2^{m/2}$ if m is even and $2^{(m-1)/2}$ if m is odd. Furthermore $v = 2^{k-m}$. Note that we have introduced some adaptations from the original paper:

1. $z \leftarrow \mathsf{MSB}(\langle b \rangle, k, f)$.
2. $\langle v \rangle \leftarrow \sum_{i=0}^{k-1} 2^{k-i-1} \cdot \langle z_i \rangle$.
3. $\langle c \rangle \leftarrow \langle b \rangle \cdot \langle v \rangle$.
4. $\langle m \rangle \leftarrow \sum_{i=0}^{k-1} (i+1) \cdot \langle z_i \rangle$.
5. For $i \in [1, \ldots, k/2]$ do
   (a) $\langle w_i \rangle = \langle z_{2 \cdot i - 1} \rangle + \langle z_{2 \cdot i} \rangle$.
6. $\langle w_0 \rangle \leftarrow 0$.
7. $\langle w \rangle \leftarrow \sum_{i=0}^{k/2} 2^i \cdot \langle w_i \rangle$.
8. Return $\langle c \rangle, \langle v \rangle, \langle m \rangle, \langle w \rangle$.

$\mathsf{SimplifiedNormSQ}(\langle b \rangle, k)$:

Same as above, but in this case we only return m, and w, together with a $\{0, 1\}$ bit signaling whether m is odd.

1. $z \leftarrow \mathsf{MSB}(\langle b \rangle, k, f)$.
2. $\langle m \rangle \leftarrow \sum_{i=0}^{k-1} (i+1) \cdot \langle z_i \rangle$.
3. For $i \in [0, \ldots, k-1]$ do
   (a) $\langle m \rangle = \langle m \rangle + (i+1) + \langle z_i \rangle$.
   (b) If $(i \quad \% \quad 2 == 0)$:
      i. $\langle m_{odd} \rangle \leftarrow m_{odd} + \langle z_i \rangle$
4. For $i \in [1, \ldots, k/2]$ do
   (a) $\langle w_i \rangle = \langle z_{2 \cdot i - 1} \rangle + \langle z_{2 \cdot i} \rangle$.
5. $\langle w_0 \rangle \leftarrow 0$.
6. $\langle w \rangle \leftarrow \sum_{i=0}^{k/2} 2^i \cdot \langle w_i \rangle$.
7. Return $\langle m_{odd} \rangle, \langle m \rangle, \langle w \rangle$.

Arithmetic with Floating Point Numbers

For floating point numbers we utilize the methods described in

- Secure Computation on Floating Point Numbers *NDSS 2013* [ABZS13].

However we make explicit use of an error flag which we carry throughout a computation, as detailed in previous sections. The processing of overflow and underflow detection is expensive, and thus we enable the user to turn this off via means of a compile time flag `fdflag` by passing the option `-f` or `--fdflag` when using `compile.py`. This can also be turned on/off within a program by assigning to the variable `program.fdflag`. The error flag is still needed however to catch other forms of errors in computations (such as division by zero, taking square roots of negative numbers etc). Thus setting `fdflag` equal to false does not necessarily result in err always equaling zero.

Floating point numbers are defined by two global, public integer parameters (ℓ, k) which define the size of the mantissa and the exponent respectively. Each floating point number is represented as a five tuple $(v, p, z, s, \mathsf{err})$, where

- $v \in [2^{\ell-1}, 2^{\ell})$ is an ℓ-bit significand with it's most most bit always set to one.
- $p \in \mathbb{Z}_{\langle k \rangle}$ is the signed exponent.
- z is a bit to define whether the number is zero or not.
- s is a sign bit (equal to zero if non-negative).
- err is the error flag (equal to zero if no error has occurred, it holds a non-zero value otherwise).

Thus assuming $\mathsf{err} = 0$ this tuple represents the value

$$u = (1 - 2 \cdot s) \cdot (1 - z) \cdot v \cdot 2^p.$$

We adopt the conventions that when $u = 0$ we also have $z = 1, v = 0$ and $p = 0$, and when $\mathsf{err} = 1$ then the values of v, p, z and s are meaningless.

The standard arithmetic operations of addition, multiplication and comparison are then implemented in MAMBA for this datetype using operator overloading. The precise algorithms which are executed are detailed below.

FlowDetect($\langle p \rangle$):

1. If fdflag then
   (a) $\langle s \rangle \leftarrow -2 \cdot (\langle p \rangle < 0) + 1$.
   (b) $\langle \mathsf{err} \rangle \leftarrow \mathsf{GT}(\langle p \rangle \cdot \langle s \rangle, 2^{k-1} - 1, k + 1)$.
2. Return $\langle \mathsf{err} \rangle$.

FLNeg($(\langle v \rangle, \langle p \rangle, \langle z \rangle, \langle s \rangle, \langle \mathsf{err} \rangle)$):

1. $\langle s \rangle \leftarrow 1 - \langle s \rangle$.
2. Return $(\langle v \rangle, \langle p \rangle, \langle z \rangle, \langle s \rangle, \langle \mathsf{err} \rangle)$

FLAbs($(\langle v \rangle, \langle p \rangle, \langle z \rangle, \langle s \rangle, \langle \mathsf{err} \rangle)$):

1. $\langle s \rangle \leftarrow 0$.
2. Return $(\langle v \rangle, \langle p \rangle, \langle z \rangle, \langle s \rangle, \langle \mathsf{err} \rangle)$

Obviously from FLNeg and FLAdd we can define FLSub.

$\mathsf{FLMult}((\langle v_1\rangle, \langle p_1\rangle, \langle z_1\rangle, \langle s_1\rangle, \langle \mathsf{err}_1\rangle), (\langle v_2\rangle, \langle p_2\rangle, \langle z_2\rangle, \langle s_2\rangle, \langle \mathsf{err}_2\rangle))$**:**

For floating point operations multiplication is much easier than addition, so we deal with this first.

1. $\langle v\rangle \leftarrow \langle v_1\rangle \cdot \langle v_2\rangle$.
2. $\langle v\rangle \leftarrow \mathsf{Trunc}(\langle v\rangle, 2 \cdot \ell, \ell - 1)$.
3. $\langle b\rangle \leftarrow \mathsf{LT}(\langle v\rangle, 2^\ell, \ell + 1)$.
4. $\langle v'\rangle \leftarrow \langle v\rangle + \langle b\rangle \cdot \langle v\rangle$.
5. $\langle v\rangle \leftarrow \mathsf{Trunc}(\langle v'\rangle, \ell + 1, 1)$.
6. $\langle z\rangle \leftarrow \mathsf{OR}(\langle z_1\rangle, \langle z_2\rangle)$.
7. $\langle s\rangle \leftarrow \mathsf{XOR}(\langle s_1\rangle, \langle s_2\rangle)$.
8. $\langle p\rangle \leftarrow (\langle p_1\rangle + \langle p_2\rangle + \ell - \langle b\rangle) \cdot (1 - \langle z\rangle)$.
9. $\langle \mathsf{err}\rangle \leftarrow \langle \mathsf{err}_1\rangle + \langle \mathsf{err}_2\rangle + \mathsf{FlowDetect}(\langle p\rangle)$.
10. Return $(\langle v\rangle, \langle p\rangle, \langle z\rangle, \langle s\rangle, \langle \mathsf{err}\rangle)$.

$\mathsf{FLAdd}((\langle v_1\rangle, \langle p_1\rangle, \langle z_1\rangle, \langle s_1\rangle, \langle \mathsf{err}_1\rangle), (\langle v_2\rangle, \langle p_2\rangle, \langle z_2\rangle, \langle s_2\rangle, \langle \mathsf{err}_2\rangle))$**:**

1. $\langle a\rangle \leftarrow \mathsf{LT}(\langle p_1\rangle, \langle p_2\rangle, k$.
2. $\langle b\rangle \leftarrow \mathsf{EQ}(\langle p_1\rangle, \langle p_2\rangle, k$.
3. $\langle c\rangle \leftarrow \mathsf{LT}(\langle v_1\rangle, \langle v_2\rangle, k$.
4. $\langle p_{\max}\rangle \leftarrow \langle a\rangle \cdot \langle p_2\rangle + (1 - \langle a\rangle) \cdot \langle p_1\rangle$.
5. $\langle p_{\min}\rangle \leftarrow (1 - \langle a\rangle) \cdot \langle p_2\rangle + \langle a\rangle \cdot \langle p_1\rangle$.
6. $\langle a \cdot b\rangle \leftarrow \langle a\rangle \cdot \langle b\rangle$.
7. $\langle b \cdot c\rangle \leftarrow \langle b\rangle \cdot \langle c\rangle$.
8. $\langle v_{\max}\rangle \leftarrow (\langle a \cdot b\rangle - \langle a\rangle - \langle b \cdot c\rangle) \cdot (\langle v_1\rangle - \langle v_2\rangle) + \langle v_1\rangle$.
9. $\langle v_{\min}\rangle \leftarrow (\langle a \cdot b\rangle - \langle a\rangle - \langle b \cdot c\rangle) \cdot (\langle v_2\rangle - \langle v_1\rangle) + \langle v_2\rangle$.
10. $\langle s_3\rangle \leftarrow \mathsf{XOR}(\langle s_1\rangle, \langle s_2\rangle)$.
11. $\langle d\rangle \leftarrow \mathsf{LT}(\ell, \langle p_{\max}\rangle - \langle p_{\min}\rangle, k)$.
12. $\langle 2^\Delta\rangle \leftarrow \mathsf{Pow2}((1 - \langle d\rangle) \cdot (\langle p_{\max}\rangle - \langle p_{\min}\rangle), \ell + 1)$.
13. $\langle v_3\rangle \leftarrow 2 \cdot (\langle v_{\max}\rangle - \langle s_3\rangle) + 1$.
14. $\langle v_4\rangle \leftarrow \langle v_{\max}\rangle \cdot \langle 2^\Delta\rangle + (1 - 2 \cdot \langle s_3\rangle) \cdot \langle v_{\min}\rangle$.
15. $\langle v\rangle \leftarrow (\langle d\rangle \cdot \langle v_3\rangle + (1 - \langle d\rangle) \cdot \langle v_4\rangle) \cdot 2^\ell \cdot \mathsf{Inv}(\langle 2^\Delta\rangle)$.
16. $\langle v\rangle \leftarrow \mathsf{Trunc}(\langle v\rangle, 2 \cdot \ell + 1, \ell - 1)$.
17. $\langle u_{\ell+1}\rangle, \ldots, \langle u_0\rangle \leftarrow \mathsf{BitDec}(\langle v\rangle, \ell + 2, \ell + 2)$.
18. $\langle h_0\rangle, \ldots, \langle h_{\ell+1}\rangle \leftarrow \mathsf{PreOp}(\mathsf{OR}, \langle u_{\ell+1}\rangle, \ldots, \langle u_0\rangle, k)$.

19. $\langle p_0 \rangle \leftarrow \ell + 2 - \sum_{i=0}^{\ell+1} \langle h_i \rangle$.
20. $\langle 2^{p_0} \rangle \leftarrow 1 + \sum_{i=0}^{\ell+1} 2^i \cdot (1 - \langle h_i \rangle)$.
21. $\langle v \rangle \leftarrow \mathsf{Trunc}(\langle 2^{p_0} \rangle \cdot \langle v \rangle, \ell + 2, 2)$.
22. $\langle p \rangle \leftarrow \langle p_{\max} \rangle - \langle p_0 \rangle + 1 - \langle d \rangle$.
23. $\langle z_1 \cdot z_2 \rangle \leftarrow \langle z_1 \rangle \cdot \langle z_2 \rangle$.
24. $\langle v \rangle \leftarrow (1 - \langle z_1 \rangle - \langle z_2 \rangle + \langle z_1 \cdot z_2 \rangle) \cdot \langle v \rangle + \langle z_1 \rangle \cdot \langle v_2 \rangle + \langle z_2 \rangle \cdot \langle v_1 \rangle$.
25. $\langle z \rangle \leftarrow \mathsf{EQZ}(\langle v \rangle, \ell)$.
26. $\langle p \rangle \leftarrow (1 - \langle z_1 \rangle - \langle z_2 \rangle + \langle z_1 \cdot z_2 \rangle) \cdot \langle p \rangle + \langle z_1 \rangle \cdot \langle p_2 \rangle + \langle z_2 \rangle \cdot \langle p_1 \rangle) \cdot (1 - \langle z \rangle)$.
27. $\langle s \rangle \leftarrow (\langle a \rangle - \langle a \cdot b \rangle) \cdot \langle s_2 \rangle + (1 - \langle a \rangle - \langle b \rangle + \langle a \cdot b \rangle) \cdot \langle s_1 \rangle) + \langle b \cdot c \rangle \cdot \langle s_2 \rangle + (\langle b \rangle - \langle b \cdot c \rangle) \cdot \langle s_1 \rangle)$.
28. $\langle s \rangle \leftarrow (1 - \langle z_1 \rangle - \langle z_2 \rangle + \langle z_1 \cdot z_2 \rangle) \cdot \langle s \rangle + (\langle z_2 \rangle - \langle z_1 \cdot z_2 \rangle) \cdot \langle s_1 \rangle + (\langle z_1 \rangle - \langle z_1 \cdot z_2 \rangle) \cdot \langle s_2 \rangle$.
29. $\langle \mathsf{err} \rangle \leftarrow \langle \mathsf{err}_1 \rangle + \langle \mathsf{err}_2 \rangle + \mathsf{FlowDetect}(\langle p \rangle)$.
30. $\langle \mathsf{err} \rangle \leftarrow \mathsf{FlowDetect}(\langle p \rangle, \langle \mathsf{err} \rangle)$.
31. Return $(\langle v \rangle, \langle p \rangle, \langle z \rangle, \langle s \rangle, \langle \mathsf{err} \rangle)$.

$\mathsf{SDiv}(\langle a \rangle, \langle b \rangle, \ell)$**:**

1. $\theta \leftarrow \lceil \log_2 \ell \rceil$.
2. $\langle x \rangle \leftarrow \langle b \rangle$.
3. $\langle y \rangle \leftarrow \langle a \rangle$.
4. For $i \in [1, \ldots, \theta - 1]$ do
   (a) $\langle y \rangle \leftarrow \langle y \rangle \cdot (2^{\ell+1} - \langle x \rangle)$.
   (b) $\langle y \rangle \leftarrow \mathsf{TruncPr}(\langle y \rangle, 2 \cdot \ell + 1, \ell)$.
   (c) $\langle x \rangle \leftarrow \langle x \rangle \cdot (2^{\ell+1} - \langle x \rangle)$.
   (d) $\langle x \rangle \leftarrow \mathsf{TruncPr}(\langle x \rangle, 2 \cdot \ell + 1, \ell)$.
5. $\langle y \rangle \leftarrow \langle y \rangle \cdot (2^{\ell+1} - \langle x \rangle)$.
6. $\langle y \rangle \leftarrow \mathsf{TruncPr}(\langle y \rangle, 2 \cdot \ell + 1, \ell)$.
7. Return $\langle y \rangle$.

$\mathsf{FLDiv}((\langle v_1 \rangle, \langle p_1 \rangle, \langle z_1 \rangle, \langle s_1 \rangle, \langle \mathsf{err}_1 \rangle), (\langle v_2 \rangle, \langle p_2 \rangle, \langle z_2 \rangle, \langle s_2 \rangle, \langle \mathsf{err}_2 \rangle))$**:**

1. $\langle v \rangle \leftarrow \mathsf{SDiv}(\langle v_1 \rangle, \langle v_2 \rangle + \langle z_2 \rangle, \ell)$.
2. $\langle b \rangle \leftarrow \mathsf{LT}(\langle v \rangle, 2^\ell, \ell + 1)$.
3. $\langle v' \rangle \leftarrow \langle v \rangle + \langle b \rangle \cdot \langle v \rangle$.
4. $\langle v \rangle \leftarrow \mathsf{Trunc}(\langle v' \rangle, \ell + 1, 1)$.
5. $\langle z \rangle \leftarrow \langle z_1 \rangle$.
6. $\langle s \rangle \mathsf{XOR}(\langle s_1 \rangle, \langle s_2 \rangle)$.

7. $\langle p \rangle \leftarrow (\langle p_1 \rangle - \langle p_2 \rangle - \ell + 1 - \langle b \rangle) \cdot (1 - \langle z \rangle)$.
8. $\langle \mathsf{err} \rangle \leftarrow \langle \mathsf{err}_1 \rangle + \langle \mathsf{err}_2 \rangle$.
9. $\langle \mathsf{err} \rangle \leftarrow \langle \mathsf{err} \rangle + \langle z_2 \rangle$.
10. $\langle \mathsf{err} \rangle \leftarrow \langle \mathsf{err} \rangle + \mathsf{FlowDetect}(\langle p \rangle)$.
11. Return $(\langle v \rangle, \langle p \rangle, \langle z \rangle, \langle s \rangle, \langle \mathsf{err} \rangle)$.

$\mathsf{FLLTZ}((\langle v \rangle, \langle p \rangle, \langle z \rangle, \langle s \rangle, \langle \mathsf{err} \rangle))$**:**

1. Return $\langle s \rangle \cdot (1 - \langle z \rangle) \cdot \mathsf{EQZ}(\langle \mathsf{err} \rangle, k)$.

$\mathsf{FLEQZ}((\langle v \rangle, \langle p \rangle, \langle z \rangle, \langle s \rangle, \langle \mathsf{err} \rangle))$**:**

1. Return $\langle z \rangle \cdot \mathsf{EQZ}(\langle \mathsf{err} \rangle, k)$.

$\mathsf{FLGTZ}((\langle v \rangle, \langle p \rangle, \langle z \rangle, \langle s \rangle, \langle \mathsf{err} \rangle))$**:**

1. Return $(1 - \langle s \rangle) \cdot (1 - \langle z \rangle)\mathsf{EQZ}(\langle \mathsf{err} \rangle, k)$.

$\mathsf{FLLEZ}((\langle v \rangle, \langle p \rangle, \langle z \rangle, \langle s \rangle, \langle \mathsf{err} \rangle))$**:**

1. Return $\langle s \rangle \cdot (1 - \mathsf{EQZ}(\langle \mathsf{err} \rangle, k))$.

$\mathsf{FLGEZ}((\langle v \rangle, \langle p \rangle, \langle z \rangle, \langle s \rangle, \langle \mathsf{err} \rangle))$**:**

1. Return $(1 - \langle s \rangle) \cdot \mathsf{EQZ}(\langle \mathsf{err} \rangle, k)$.

$\mathsf{FLEQ}((\langle v_1 \rangle, \langle p_1 \rangle, \langle z_1 \rangle, \langle s_1 \rangle, \langle \mathsf{err}_1 \rangle), (\langle v_2 \rangle, \langle p_2 \rangle, \langle z_2 \rangle, \langle s_2 \rangle, \langle \mathsf{err}_2 \rangle))$**:**

1. $\langle b_1 \rangle \leftarrow \mathsf{EQ}(\langle v_1 \rangle, \langle v_2 \rangle, \ell)$.
2. $\langle b_2 \rangle \leftarrow \mathsf{EQ}(\langle p_1 \rangle, \langle p_2 \rangle, k)$.
3. $\langle b_3 \rangle \leftarrow \langle z_1 \rangle \cdot \langle z_2 \rangle$.
4. $\langle b_4 \rangle \leftarrow \langle s_1 \rangle \cdot \langle s_2 \rangle$.
5. $\langle t \rangle \leftarrow \langle err_1 \rangle + \langle err_2 \rangle$.
6. $\langle t \rangle \leftarrow (\mathsf{EQZ}(\langle t \rangle, k)$.
7. Return $(\langle b_1 \rangle \cdot \langle b_2 \rangle \cdot \langle b_3 \rangle \cdot (1 - \langle b_4 \rangle) + \langle b_4 \rangle) \cdot \langle t \rangle$.

$\mathsf{FLLT}((\langle v_1 \rangle, \langle p_1 \rangle, \langle z_1 \rangle, \langle s_1 \rangle, \langle \mathsf{err}_1 \rangle), (\langle v_2 \rangle, \langle p_2 \rangle, \langle z_2 \rangle, \langle s_2 \rangle, \langle \mathsf{err}_2 \rangle))$**:**

1. $\langle a \rangle \leftarrow \mathsf{LT}(\langle p_1 \rangle, \langle p_2 \rangle, k)$.
2. $\langle c \rangle \leftarrow \mathsf{EQ}(\langle p_1 \rangle, \langle p_2 \rangle, k)$.
3. $\langle d \rangle \leftarrow \mathsf{LT}((1 - 2 \cdot \langle s_1 \rangle) \cdot \langle v_1 \rangle, (1 - 2 \cdot \langle s_2 \rangle) \cdot \langle v_2 \rangle, \ell + 1)$.
4. $\langle a \cdot c \rangle \leftarrow \langle a \rangle \cdot \langle c \rangle$.
5. $\langle c \cdot d \rangle \leftarrow \langle c \rangle \cdot \langle d \rangle$.
6. $\langle b^+ \rangle \leftarrow \langle c \cdot d \rangle + (\langle a \rangle - \langle a \cdot c \rangle)$.

7. $\langle b^- \rangle \leftarrow \langle c \cdot d \rangle + (1 - \langle c \rangle - \langle a \rangle + \langle a \cdot c \rangle)$.
8. $\langle z_1 \cdot z_2 \rangle \leftarrow \langle z_1 \rangle \cdot \langle z_2 \rangle$.
9. $\langle s_1 \cdot s_2 \rangle \leftarrow \langle s_1 \rangle \cdot \langle s_2 \rangle$.
10. $\langle b \rangle \leftarrow \langle z_1 \cdot z_2 \rangle \cdot (\langle s_2 \rangle - 1 - \langle s_1 \cdot s_2 \rangle) + \langle s_1 \cdot s_2 \rangle \cdot (\langle z_1 \rangle + \langle z_2 \rangle - 1) + \langle z_1 \rangle \cdot (1 - \langle s1 \rangle - \langle s_2 \rangle) + \langle s_1 \rangle$.
11. $\langle t \rangle \leftarrow \langle err_1 \rangle + \langle err_2 \rangle$.
12. $\langle t \rangle \leftarrow \mathsf{EQZ}(\langle t \rangle, k)$.
13. $\langle b \rangle \leftarrow \langle b \rangle + (1 - \langle z_1 \rangle - \langle z_2 \rangle + \langle z_1 \cdot z_2 \rangle \cdot ((1 - \langle s_1 \rangle - \langle s_2 \rangle + \langle s_1 \cdot s_2 \rangle) \cdot \langle b^+ \rangle + \langle s_1 \cdot s_2 \rangle \cdot \langle b^- \rangle) \cdot \langle t \rangle$.
14. Return $\langle b \rangle$.

$\mathsf{FLGT}((\langle v_1 \rangle, \langle p_1 \rangle, \langle z_1 \rangle, \langle s_1 \rangle, \langle \mathsf{err}_1 \rangle), (\langle v_2 \rangle, \langle p_2 \rangle, \langle z_2 \rangle, \langle s_2 \rangle, \langle \mathsf{err}_2 \rangle))$**:**

1. $\langle v_r \rangle, \langle p_r \rangle, \langle z_r \rangle, \langle s_r \rangle, \langle err_r \rangle \leftarrow \mathsf{FLAdd}((\langle v_2 \rangle, \langle p_2 \rangle, \langle z_2 \rangle, 1 - \langle s_2 \rangle, \langle \mathsf{err}_2 \rangle), (\langle v_1 \rangle, \langle p_1 \rangle, \langle z_1 \rangle, \langle s_1 \rangle, \langle \mathsf{err}_1 \rangle))$.
2. Return $\mathsf{FLLTZ}(\langle v_r \rangle, \langle p_r \rangle, \langle z_r \rangle, \langle s_r \rangle, \langle err_r \rangle)$.

$\mathsf{FLLET}((\langle v_1 \rangle, \langle p_1 \rangle, \langle z_1 \rangle, \langle s_1 \rangle, \langle \mathsf{err}_1 \rangle), (\langle v_2 \rangle, \langle p_2 \rangle, \langle z_2 \rangle, \langle s_2 \rangle, \langle \mathsf{err}_2 \rangle))$**:**

1. $\langle b \rangle \leftarrow \mathsf{FLGT}((\langle v_1 \rangle, \langle p_1 \rangle, \langle z_1 \rangle, \langle s_1 \rangle, \langle \mathsf{err}_1 \rangle), (\langle v_2 \rangle, \langle p_2 \rangle, \langle z_2 \rangle, \langle s_2 \rangle, \langle \mathsf{err}_2 \rangle))$.
2. Return $1 - \langle b \rangle$

$\mathsf{FLGET}((\langle v_1 \rangle, \langle p_1 \rangle, \langle z_1 \rangle, \langle s_1 \rangle, \langle \mathsf{err}_1 \rangle), (\langle v_2 \rangle, \langle p_2 \rangle, \langle z_2 \rangle, \langle s_2 \rangle, \langle \mathsf{err}_2 \rangle))$**:**

1. $\langle b \rangle \leftarrow \mathsf{FLLT}((\langle v_1 \rangle, \langle p_1 \rangle, \langle z_1 \rangle, \langle s_1 \rangle, \langle \mathsf{err}_1 \rangle), (\langle v_2 \rangle, \langle p_2 \rangle, \langle z_2 \rangle, \langle s_2 \rangle, \langle \mathsf{err}_2 \rangle))$.
2. Return $1 - \langle b \rangle$

Conversion Routines

$\mathsf{FLRound}((\langle v_1\rangle, \langle p_1\rangle, \langle z_1\rangle, \langle s_1\rangle, \langle \mathsf{err}_1\rangle, \mathsf{mode})$**:**

This, depending on mode, computes either the floating point representation of the floor (if $\mathsf{mode} = 0$) or the ceiling (if $\mathsf{mode} = 1$) of the input floating point number.

1. $\langle a\rangle \leftarrow \mathsf{LTZ}(\langle p_1\rangle, k)$.
2. $\langle b\rangle \leftarrow \mathsf{LT}(\langle p_1\rangle, -\ell + 1, k)$.
3. $\langle a \cdot b\rangle \leftarrow \langle a\rangle \cdot \langle b\rangle$.
4. $\langle v_2\rangle, \langle 2^{-p_1}\rangle \leftarrow \mathsf{Mod2m}(\langle v_1\rangle, \ell, (\langle a \cdot b\rangle - \langle a\rangle) \cdot \langle p_1\rangle)$. [Note, we save the computation of $\langle 2^{-p_1}\rangle$ which this routine computes when $0 \leq p_1 < -\ell$, otherwise the returned share is of 2^0.]
5. $\langle c\rangle \leftarrow \mathsf{EQZ}(\langle v_2\rangle, \ell)$.
6. $\langle v\rangle \leftarrow \langle v_1\rangle - \langle v_2\rangle + (1 - \langle c\rangle) \cdot \langle 2^{-p_1}\rangle \cdot \mathsf{XOR}(\mathsf{mode}, \langle s_1\rangle)$.
7. $\langle d\rangle \leftarrow \mathsf{EQ}(\langle v\rangle, 2^\ell, \ell + 1)$.
8. $\langle v\rangle \leftarrow 2^{\ell-1} \cdot \langle d\rangle + (1 - \langle d\rangle) \cdot \langle v\rangle$.
9. $\langle v\rangle \leftarrow (\langle a\rangle - \langle a \cdot b\rangle) \cdot \langle v\rangle + \langle a \cdot b\rangle \cdot (\mathsf{mode} - \langle s_1\rangle) + (1 - \langle a\rangle) \cdot \langle v_1\rangle$.
10. $\langle s\rangle \leftarrow (1 - \langle b\rangle \cdot \mathsf{mode}) \cdot \langle s_1\rangle$.
11. $\langle z\rangle \leftarrow \mathsf{OR}(\mathsf{EQZ}(\langle v\rangle, \ell), \langle z_1\rangle)$.
12. $\langle v\rangle \leftarrow \langle v\rangle \cdot (1 - \langle z\rangle)$.
13. $\langle p\rangle \leftarrow (\langle p_1\rangle + \langle d\rangle \cdot (\langle a\rangle - \langle a \cdot b\rangle)) \cdot (1 - \langle z\rangle)$.
14. $\langle \mathsf{err}\rangle \leftarrow \langle \mathsf{err}_1\rangle$.
15. Return $(\langle v\rangle, \langle p\rangle, \langle z\rangle, \langle s\rangle, \langle \mathsf{err}\rangle)$.

MAMBA Example: To round a `sfloat` value we could invoke the function as follows:

```
from Compiler import floatingpoint
x = sfloat(5.5)
mode = 0
# v, p, z, s, err are extracted from x
# retunrs the floor of x
y = floatingpoint.FLRound(x,mode)
```

When `mode=2` then we get the ceil operation.

$\mathsf{Int2Fx}(\langle a\rangle, k, f)$**:**

Given an integer $a \in \mathbb{Z}_{\langle k\rangle}$ this gives the equivalent integer b in $\mathbb{Q}_{\langle k,f\rangle}$, namely $\overline{b} = a \cdot 2^f$. Note this means, to ensure correctness, that $|a| \leq 2^{k-f}$.

1. Return $2^f \cdot \langle a\rangle$.

MAMBA Example: To cast an `int` or `sint` register into a `sfix` one, you could execute the following:

```
x = sfix(5.5)
# k, f are extracted from x
y = sfix.load_sint(x)
```

Int2FL$(\langle a \rangle, \gamma, \ell)$:

We assume $a \in \mathbb{Z}_{\langle \gamma \rangle}$, this could loose precision if $\gamma - 1 > \ell$.

1. $\lambda \leftarrow \gamma - 1$.
2. $\langle s \rangle \leftarrow \mathsf{LTZ}(\langle a \rangle, \gamma)$.
3. $\langle z \rangle \leftarrow \mathsf{EQZ}(\langle a \rangle, \gamma)$.
4. $\langle a \rangle \leftarrow (1 - 2 \cdot \langle s \rangle) \cdot \langle a \rangle$.
5. $\langle a_{\lambda-1} \rangle, \ldots, \langle a_0 \rangle \leftarrow \mathsf{BitDec}(\langle a \rangle, \lambda, \lambda)$.
6. $\langle b_0 \rangle, \ldots, \langle b_{\lambda-1} \rangle \leftarrow \mathsf{PreOp}(\mathsf{OR}, \langle a_{\lambda-1} \rangle, \ldots, \langle a_0 \rangle, \gamma)$.
7. $\langle v \rangle \leftarrow \langle a \rangle \cdot (1 + \sum_{i=0}^{\lambda-1} 2^i \cdot (1 - \langle b_i \rangle))$.
8. $\langle p \rangle \leftarrow -(\lambda - \sum_{i=0}^{\lambda-1} \langle b_i \rangle)$.
9. If $(\gamma - 1) > \ell$ then
   (a) $\langle v \rangle \leftarrow \mathsf{Trunc}(\langle v \rangle, \gamma - 1, \gamma - \ell - 1)$.
10. Else
    (a) $\langle v \rangle \leftarrow 2^{\ell - \gamma + 1} \cdot \langle v \rangle$.
11. $\langle p \rangle \leftarrow (\langle p \rangle + \gamma - 1 - \ell) \cdot (1 - \langle z \rangle)$.
12. $\langle \mathsf{err} \rangle \leftarrow 0$.
13. Return $(\langle v \rangle, \langle p \rangle, \langle z \rangle, \langle s \rangle, \langle \mathsf{err} \rangle)$.

MAMBA Example: To cast an `int` or `sint` register into a `sfloat` one, you could execute the following:

```
x = sfloat(5.5)
# gamma and l are extracted from the system
y = sfloat(x)
```

Fx2Int$(\langle a \rangle, k, f)$:

Given a value $a \in \mathbb{Q}_{\langle k,f \rangle}$ this gives the integer $\lfloor \overline{a}/2^f \rfloor$.

1. Return $\mathsf{Trunc}(\langle a \rangle, k, f)$.

MAMBA Example: To extract the integral component of a `sfix` register, which is then encapsulated on a `sint` register, and taking into account what is currently implemented, you could execute the following:

```
from Compiler import floatingpoint
x = sifx(5.5)
# y stores 5 in a sint register
y = floatingpoint.Trunc(x.v, x.k - x.f, x.f, x.kappa)
```

FxFloor$(\langle a \rangle, k, f)$:

Given a value $a \in \mathbb{Q}_{\langle k,f \rangle}$ this does the same, but gives the result as a fixed point value.

1. Return $2^f \cdot \mathsf{Trunc}(\langle a \rangle, k, f)$.

MAMBA Example: To floor an `sfix` register, you could execute the following:

```
from Compiler import mpc_math
x = sifx(5.5)
# k and f are extracted from x
# y stores 5 in a sfix register
y = mpc_math.floor_fx(x)
```

Fx2FL$(\langle g \rangle, \gamma, f, \ell, k)$**:**

Converts $g \in \mathbb{Q}_{\langle k,f \rangle}$ into a floating point number

1. $(\langle v \rangle, \langle p \rangle, \langle z \rangle, \langle s \rangle, \langle \mathsf{err} \rangle) \leftarrow$ Int2FL$(\langle g \rangle, \gamma, \ell)$.
2. $\langle p \rangle \leftarrow (\langle p \rangle - f) \cdot (1 - \langle z \rangle)$.
3. Return $(\langle v \rangle, \langle p \rangle, \langle z \rangle, \langle s \rangle, \langle \mathsf{err} \rangle)$.

MAMBA Example: To cast from `sfix` to `sfloat`, you could execute the following:

```
# stores 5.5 on a sfloat register
x = sfloat(sfix(5.5))
```

FL2Fx$((\langle v \rangle, \langle p \rangle, \langle z \rangle, \langle s \rangle, \langle \mathsf{err} \rangle), \ell, k, \gamma, f)$**:**

1. $\langle b \rangle \leftarrow$ LT$(\langle p \rangle, 2^{k-1} - f, k)$.
2. $\langle g \rangle \leftarrow$ FL2Int$((\langle v \rangle, \langle p \rangle + f, \langle z \rangle, \langle s \rangle, \langle \mathsf{err} \rangle), \ell, k, \gamma)$.
3. Return $\langle g \rangle \cdot \langle b \rangle$.

MAMBA Example: To cast an `sfloat` register into a `sfix` one, you could execute the following:

```
# stores 5.5 on a sfix register
# v, p, z, s, err are extracted from x and l, k, gamma are system parameters
x = sfix(sfloat(5.5))
```

SQRT Functions

(TO DO:) *These functions are only currently supported in their fixed point versions.* The original description of the protocols for the fixed point square root algorithm are included in:

- Secure Distributed Computation of the Square Root and Applications, *ISPEC 2012* [Lie12].

The floating point variant is in

- Secure Computation on Floating Point Numbers *NDSS 2013* [ABZS13].(TO DO:) *This is not currently implemented*

Additionally, we provide a simplified implementation for the Square Root on fixed point variables, that is appropriate for inputs of any size. We make use Liedel's protocol, when the input's size makes it possible, as we explain later in this section. Both algorithms makes use of two constants

$$\alpha = -0.8099868542, \quad \beta = 1.787727479.$$

These are the solutions of the system of equations

$$E(x) = \frac{\alpha \cdot x + \beta - \frac{1}{\sqrt{x}}}{\frac{1}{\sqrt{x}}},$$

$$M = \frac{\sqrt{3}}{3} \cdot \sqrt{\frac{-\beta}{\alpha}} \cdot \left(\frac{2}{3} \cdot \beta - \frac{\sqrt{3}}{\sqrt{\frac{-\beta}{\alpha}}} \right),$$

$$E\left(\frac{1}{2}\right) = E(1) = -M.$$

ParamFxSqrt$(\langle x \rangle, k, f)$**:**

This algorithm uses the sub-algorithm LinAppSQ defined below, note that `LinAppSQ` returns an scaled $1/\sqrt{x} \cdot 2^f$. The algorithm only works when $3 \cdot k - 2 \cdot f$ is less than the system precision, which is by default equal to 20. In a future release we will extend the sqrt function to cope with other input ranges.

1. $\theta \leftarrow \lceil \log_2(k/5.4) \rceil$.
2. $\langle y_0 \rangle \leftarrow$ LinAppSQ$(\langle x \rangle, k, f)$.
3. $\langle y_0 \rangle \leftarrow \langle y_0 \rangle \cdot 1/2^f$.
4. $\langle g_0 \rangle \leftarrow \langle y_0 \rangle \cdot \langle x \rangle$.
5. $\langle g_0 \rangle \leftarrow \langle y_0 \rangle \cdot 1/2^f$.
6. $\langle g_0 \rangle \leftarrow \langle y_0 \rangle \cdot 1/2$.
7. $\langle gh_0 \rangle \leftarrow \langle g_0 \rangle \cdot \langle h_0 \rangle$.
8. $\langle g \rangle \leftarrow \langle g_0 \rangle$.
9. $\langle h \rangle \leftarrow \langle h_0 \rangle$.
10. $\langle gh \rangle \leftarrow \langle gh_0 \rangle$.
11. For $i \in [1, \ldots, \theta - 2]$ do
    (a) $\langle r \rangle \leftarrow 3/2 - \langle gh \rangle$.

(b) $\langle g \rangle \leftarrow \langle g \rangle \cdot \langle r \rangle$.

(c) $\langle h \rangle \leftarrow \langle h \rangle \cdot \langle r \rangle$.

(d) $\langle gh \rangle \leftarrow \langle g \rangle \cdot \langle h \rangle$.

12. $\langle r \rangle \leftarrow 3/2 - \langle gh \rangle$.
13. $\langle h \rangle \leftarrow \langle h \rangle \cdot \langle r \rangle$.
14. $\langle H \rangle \leftarrow 4 \cdot (\langle h \rangle^2)$.
15. $\langle H \rangle \leftarrow \langle H \rangle \cdot \langle x \rangle$.
16. $\langle H \rangle \leftarrow (3) - \langle H \rangle$.
17. $\langle H \rangle \leftarrow \langle h \rangle \cdot \langle H \rangle$.
18. $\langle g \rangle \leftarrow \langle H \rangle \cdot \langle x \rangle$.
19. Return $\langle g \rangle$.

SimplifiedFxSqrt$(\langle x \rangle, k, f)$**:**

This algorithm uses the sub-algorithm NormSQ defined above. Among the values it returns, we base our approximation by directly using $w = 2^{m/2}$. From that point it approximates the value of $\sqrt{x}$ by calculating $\frac{x}{2^{m/2}}$. To avoid any loss of precision, we reuse `sfix` instantiation process. The algorithm work on the precision of the system and can solve values on any range. Its behaviour is still experimental. The function is designed in such a way that there is no restriction on the size f.

1. $\theta \leftarrow \max(\lceil \log_2(k) \rceil, 6)$.
2. $\langle m_{odd} \rangle, \langle w \rangle \leftarrow$ SimplifiedNormSQ$(\langle x \rangle, k)$.
3. $\langle m_{odd} \rangle \leftarrow (1 - 2 \cdot \langle m_{odd} \rangle) \cdot f$.
4. $\langle w \rangle \leftarrow (2 \cdot \langle w \rangle - \langle w \rangle) \cdot (1 - \langle m_{odd} \rangle) \cdot (f \mod 2) + \langle w \rangle$.
5. $\langle w \rangle \leftarrow sfix(\langle w \rangle \cdot 2^{\frac{f - f \mod 2}{2}})$.
6. $\langle w \rangle \leftarrow (\sqrt{2} \cdot \langle w \rangle - \langle w \rangle) \cdot \langle m_{odd} \rangle + \langle w \rangle$.
7. $\langle y_0 \rangle \leftarrow \frac{1}{\langle w \rangle}$.
8. $\langle g_0 \rangle \leftarrow \langle y_0 \rangle \cdot \langle x \rangle$.
9. $\langle g_0 \rangle \leftarrow \langle y_0 \rangle \cdot 1/2$.
10. $\langle gh_0 \rangle \leftarrow \langle g_0 \rangle \cdot \langle h_0 \rangle$.
11. $\langle g \rangle \leftarrow \langle g_0 \rangle$.
12. $\langle h \rangle \leftarrow \langle h_0 \rangle$.
13. $\langle gh \rangle \leftarrow \langle gh_0 \rangle$.
14. For $i \in [1, \ldots, \theta - 2]$ do

    (a) $\langle r \rangle \leftarrow 3/2 - \langle gh \rangle$.

    (b) $\langle g \rangle \leftarrow \langle g \rangle \cdot \langle r \rangle$.

(c) $\langle h \rangle \leftarrow \langle h \rangle \cdot \langle r \rangle$.
(d) $\langle gh \rangle \leftarrow \langle g \rangle \cdot \langle h \rangle$.

15. $\langle r \rangle \leftarrow 3/2 - \langle gh \rangle$.
16. $\langle h \rangle \leftarrow \langle h \rangle \cdot \langle r \rangle$.
17. $\langle H \rangle \leftarrow 4 \cdot (\langle h \rangle^2)$.
18. $\langle H \rangle \leftarrow \langle H \rangle \cdot \langle x \rangle$.
19. $\langle H \rangle \leftarrow (3) - \langle H \rangle$.
20. $\langle H \rangle \leftarrow \langle h \rangle \cdot \langle H \rangle$.
21. $\langle g \rangle \leftarrow \langle H \rangle \cdot \langle x \rangle$.
22. Return $\langle g \rangle$.

FxSqrt($\langle x \rangle, k \leftarrow$ `sfix.k`, $f \leftarrow$ `sfix.f`)**:**

Our `FxSqrt` functionality returns the square root of any fixed point input. It receives an input value $\langle x \rangle$, from which it calculates the square root, and optional parameters regarding its bit-length and bit-wise precision. The functionality is going to make use of our `SimplifiedFxSqrt` process by default, and the somewhat more efficient Liedel's method instead when the $3 \cdot k - 2 \cdot f <$ `sfix.f` bound, provided by his paper, is met.

1. if $(3 \cdot k - 2 \cdot f >=$ `sfix.f`$)$:
   (a) Return SimplifiedFxSqrt($\langle x \rangle, k, f$).
2. else:
   (a) $\langle x \rangle \leftarrow Trunc(\langle x \rangle \cdot 2^f, sfix.k, sfix.k - sfix.f)$
   (b) Return ParamFxSqrt($\langle x \rangle, k, f$).

MAMBA Example: To obtain the `sqrt` of any value, you could execute the following:

```
from Compiler import mpc_math
k = 5
f = 2

x = sfix(6.25)
y = sfix(144)
z = sfix (257.5)

# returns the sqrt of the number, i.e. 2.5
# inputs have to be expressed such that:
# x * 2^f \in Z_q
# and 3*k -2*f < sfix.f (system precision)
# by default system precision is 20 bits.
a = mpc_math.sqrt(x, k, f)

# when you don't specify k and f, the system uses
# the default sfix values, and hence the simplified
# version for any value range, at the cost of an
# additional division call.

b = mpc_math.sqrt(y)
c = mpc_math.sqrt(z)
```

$\mathsf{LinAppSQ}(\langle b\rangle, k, f)$**:**

We based this section on the contents of the original paper, [Lie12]. However we corrected the typos from the original work, the result is as follows:

1. $\alpha \leftarrow (-0.8099868542) \cdot 2^k$.
2. $\beta \leftarrow (1.787727479) \cdot 2^{2\cdot k}$.
3. $(\langle c\rangle, \langle v\rangle, \langle m\rangle, \langle W\rangle) \leftarrow \mathsf{NormSQ}(\langle b\rangle, k, f)$.
4. $\langle w\rangle \leftarrow \alpha \cdot \langle c\rangle + \beta$.
5. $\langle m\rangle \leftarrow \mathsf{Mod2}(\langle m\rangle, \lceil \log_2 k\rceil)$.
6. $\langle w\rangle \leftarrow \langle w\rangle \cdot \langle W\rangle \cdot \langle v\rangle$.
7. $\langle w\rangle \leftarrow \mathsf{FxDiv}(\langle w\rangle, 2^{f/2}, w.k, w.f)$.
8. $\langle w\rangle \leftarrow \mathsf{FxDiv}(\langle w\rangle, 3\cdot k - 2\cdot f, w.k, w.f)$.
9. $\langle w\rangle \leftarrow (1 - \langle m\rangle) \cdot \langle w\rangle \cdot 2^f + (\sqrt{2}\cdot 2^f) \cdot \langle m\rangle \cdot \langle w\rangle$.
10. Return $\langle w\rangle$

$\mathsf{FLSqrt}((\langle v_1\rangle, \langle p_1\rangle, \langle z_1\rangle, \langle s_1\rangle, \langle \mathsf{err}_1\rangle))$**:**

Below we let ℓ_0 denote the lsb of ℓ, $(v_\alpha, p_\alpha, z_\alpha, s_\alpha)$ (resp. $(v_\beta, p_\beta, z_\beta, s_\beta)$) denote the floating point representation of the constant α (resp. β) given above, and $v_{\sqrt{2}}$ and $p_{\sqrt{2}}$ represent the ℓ-bit significand and exponent of $\sqrt{2}$ in floating point representation.

1. $\langle b\rangle \leftarrow \mathsf{BitDec}(\langle p_1\rangle, \ell, 1)$.
2. $\langle c\rangle \leftarrow \mathsf{XOR}(\langle b\rangle, \ell_0)$.
3. $\langle p\rangle \leftarrow 2^{-1}\cdot(\langle p_1\rangle - \langle b\rangle) + \lfloor \ell/2\rfloor + \mathsf{OR}(\langle b\rangle, \ell_0)$.
4. $(\langle v_2\rangle, \langle p_2\rangle, \langle z_2\rangle, \langle s_2\rangle, \langle \mathsf{err}_2\rangle) \leftarrow \mathsf{FLMult}((\langle v_1\rangle, -\ell, 0, 0, 0), (v_\alpha, p_\alpha, z_\alpha, s_\alpha, 0))$.
5. $(\langle v_0\rangle, \langle p_0\rangle, \langle z_0\rangle, \langle s_0\rangle, \langle \mathsf{err}_0\rangle) \leftarrow \mathsf{FLAdd}((\langle v_2\rangle, \langle p_2\rangle, \langle z_2\rangle, \langle s_2\rangle, \langle \mathsf{err}_2\rangle), (v_\beta, p_\beta, z_\beta, s_\beta, 0))$.
6. $(\langle v_g\rangle, \langle p_g\rangle, \langle z_g\rangle, \langle s_g\rangle, \langle \mathsf{err}_g\rangle) \leftarrow \mathsf{FLMult}((\langle v_1\rangle, -\ell, 0, 0, 0), (\langle v_0\rangle, \langle p_0\rangle, \langle z_0\rangle, \langle s_0\rangle, \langle \mathsf{err}_0\rangle))$.
7. $(\langle v_h\rangle, \langle p_h\rangle, \langle z_h\rangle, \langle s_h\rangle, \langle \mathsf{err}_h\rangle) \leftarrow (\langle v_0\rangle, \langle p_0\rangle - 1, \langle z_0\rangle, \langle s_0\rangle, \langle \mathsf{err}_0\rangle)$.
8. For $i \in [1, \ldots, \lceil \ell/5.4\rceil - 1]$ do
   (a) $(\langle v_2\rangle, \langle p_2\rangle, \langle z_2\rangle, \langle s_2\rangle, \langle \mathsf{err}_2\rangle) \leftarrow \mathsf{FLMult}((\langle v_g\rangle, \langle p_g\rangle, \langle z_g\rangle, \langle s_g\rangle, \langle \mathsf{err}_g\rangle), (\langle v_h\rangle, \langle p_h\rangle, \langle z_h\rangle, \langle s_h\rangle, \langle \mathsf{err}_h\rangle))$.
   (b) $(\langle v_2\rangle, \langle p_2\rangle, \langle z_2\rangle, \langle s_2\rangle, \langle \mathsf{err}_2\rangle) \leftarrow \mathsf{FLSub}((3\cdot 2^{\ell-2}, -(\ell-1), 0, 0, 0), (\langle v_2\rangle, \langle p_2\rangle, \langle z_2\rangle, \langle s_2\rangle, \langle \mathsf{err}_2\rangle))$.
   (c) $(\langle v_g\rangle, \langle p_g\rangle, \langle z_g\rangle, \langle s_g\rangle, \langle \mathsf{err}_g\rangle) \leftarrow \mathsf{FLMult}((\langle v_g\rangle, \langle p_g\rangle, \langle z_g\rangle, \langle s_g\rangle, \langle \mathsf{err}_g\rangle), (\langle v_2\rangle, \langle p_2\rangle, \langle z_2\rangle, \langle s_2\rangle, \langle \mathsf{err}_2\rangle))$.
   (d) $(\langle v_h\rangle, \langle p_h\rangle, \langle z_h\rangle, \langle s_h\rangle, \langle \mathsf{err}_h\rangle) \leftarrow \mathsf{FLMult}((\langle v_h\rangle, \langle p_h\rangle, \langle z_h\rangle, \langle s_h\rangle, \langle \mathsf{err}_h\rangle), (\langle v_2\rangle, \langle p_2\rangle, \langle z_2\rangle, \langle s_2\rangle, \langle \mathsf{err}_2\rangle))$.
9. $(\langle v_{h^2}\rangle, \langle p_{h^2}\rangle, \langle z_{h^2}\rangle, \langle s_{h^2}\rangle, \langle \mathsf{err}_{h^2}\rangle) \leftarrow \mathsf{FLMult}((\langle v_h\rangle, \langle p_h\rangle, \langle z_h\rangle, \langle s_h\rangle, \langle \mathsf{err}_h\rangle), (\langle v_h\rangle, \langle p_h\rangle, \langle z_h\rangle, \langle s_h\rangle, \langle \mathsf{err}_h\rangle))$.
10. $(\langle v_2\rangle, \langle p_2\rangle, \langle z_2\rangle, \langle s_2\rangle, \langle \mathsf{err}_2\rangle) \leftarrow \mathsf{FLMult}((\langle v_1\rangle, -\ell, 0, 0, 0), (\langle v_{h^2}\rangle, \langle p_{h^2}\rangle, \langle z_{h^2}\rangle, \langle s_{h^2}\rangle, \langle \mathsf{err}_{h^2}\rangle))$.
11. $(\langle v_2\rangle, \langle p_2\rangle, \langle z_2\rangle, \langle s_2\rangle, \langle \mathsf{err}_2\rangle) \leftarrow \mathsf{FLSub}((3\cdot 2^{\ell-2}, -(\ell-1), 0, 0, 0), (\langle v_2\rangle, \langle p_2\rangle + 1, \langle z_2\rangle, \langle s_2\rangle, \langle \mathsf{err}_2\rangle))$.
12. $(\langle v_h\rangle, \langle p_h\rangle, \langle z_h\rangle, \langle s_h\rangle, \langle \mathsf{err}_h\rangle) \leftarrow \mathsf{FLMult}((\langle v_h\rangle, \langle p_h\rangle, \langle z_h\rangle, \langle s_h\rangle, \langle \mathsf{err}_h\rangle), (\langle v_2\rangle, \langle p_2\rangle, \langle z_2\rangle, \langle s_2\rangle, \langle \mathsf{err}_2\rangle))$.

13. $(\langle v_2\rangle, \langle p_2\rangle, \langle z_2\rangle, \langle s_2\rangle, \langle \mathsf{err}_2\rangle) \leftarrow \mathsf{FLMult}((\langle v_1\rangle, -\ell, 0, 0, 0), (\langle v_h\rangle, \langle p_h\rangle + 1, \langle z_h\rangle, \langle s_h\rangle, \langle \mathsf{err}_h\rangle))$.

14. $(\langle v_2\rangle, \langle p_2\rangle, \langle z_2\rangle, \langle s_2\rangle, \langle \mathsf{err}_2\rangle) \leftarrow \mathsf{FLMult}((\langle v_2\rangle, \langle p_2\rangle, \langle z_2\rangle, \langle s_2\rangle, \langle \mathsf{err}_2\rangle)(2^{\ell-1} \cdot (1 - \langle c\rangle) + v_{\sqrt{2}} \cdot \langle c\rangle, -(1 - \langle c\rangle) \cdot (\ell - 1) + p_{\sqrt{2}} \cdot \langle c\rangle, 0, 0, 0)$.

15. $\langle p\rangle \leftarrow (\langle p_2\rangle + \langle p\rangle) \cdot (1 - \langle z_1\rangle)$.

16. $\langle v\rangle \leftarrow \langle v_2\rangle \cdot (1 - \langle z_1\rangle)$.

17. $\langle \mathsf{err}\rangle \leftarrow \mathsf{OR}(\langle \mathsf{err}_2\rangle, \langle s_1\rangle)$.

18. Return $(\langle v\rangle, \langle p\rangle, \langle z_1\rangle, \langle s_1\rangle, \langle \mathsf{err}\rangle)$.

(TO DO:) *Have we picked up* err *correctly here?*

EXP and LOG Functions

(TO DO:) *These functions are only currently supported in their fixed point versions.*

A secure fixed point exponentiation and logarithm algorithm is not found anywhere, so this is our own one derived from the identities in the book.

- *Computer Approximations* by Hart from 1968 [Har78].

The floating point variants are in

- Secure Computation on Floating Point Numbers *NDSS 2013* [ABZS13].

Once we have defined FxExp2 and FxLog2 (resp. FLExp2 and FLLog2) we can define the following functions from the usual identities for non-secret values of the base b:

$$\begin{aligned} \log_b x &= (\log_b 2) \cdot \mathsf{Log2}(x), \\ x^y &= \mathsf{Exp2}(y \cdot \mathsf{Log2}(x)), \\ \exp x &= \mathsf{Exp2}(x \cdot \log_2 e), \end{aligned}$$

Note, we can define these operations

FxExp2($\langle a \rangle, k, f$)**:**

This algorithm computes 2^a as a fixed point calculation. First takes the integer and fractional part of the input $|a/2^f|$, which we denote by b and c. We then compute $d = 2^b$, which will clearly overflow if $b > k - f$, but we ignore this error (if the user is stupid enough to put in garbage, they get garbage out). We then compute $e = 2^c$, as $0 \leq c \leq 1$ via the polynomial $P_{1045}(X)$ from Hart [Har78] with coefficients

0	1	+.10000 00077 44302 1686
1	0	+.69314 71804 26163 82779 5756
2	0	+.24022 65107 10170 64605 384
3	-1	+.55504 06862 04663 79157 744
4	-2	+.96183 41225 88046 23749 77
5	-2	+.13327 30359 28143 78193 29
6	-3	+.15510 74605 90052 57397 8
7	-4	+.14197 84739 97656 06711
8	-5	+.18633 47724 13796 7076

which gives a relative error of at most $10^{-12.11}$ if computed exactly. The table should be read as line (i, a, b) giving the ith coefficient of the polynomial being $b \cdot 10^a$. Given d and e one can now compute $2^{|a|} = 2^{b+c} = 2^b \cdot 2^c = d \cdot e$, and the final dealing with the sign of a can be done by an inversion. We denote by $\mathsf{FxPol}(P_{1045}, \langle x \rangle, k, f)$ the evaluation of the polynomial P_{1045} on the fixed point input $\langle x \rangle$ where $x \in \mathbb{Q}_{\langle k,f \rangle}$. This is done by Horner's rule.

1. $\langle s \rangle = \mathsf{FxLTZ}(\langle a \rangle)$.
2. $\langle a \rangle \leftarrow (1 - 2 \cdot \langle s \rangle) \cdot \langle a \rangle$.
3. $\langle b \rangle \leftarrow \mathsf{Fx2Int}(\langle a \rangle, k, f)$.
4. $\langle c \rangle \leftarrow \langle a \rangle - \mathsf{Int2Fx}(\langle b \rangle, k, f)$.
5. $\langle d \rangle \leftarrow \mathsf{Int2Fx}(\mathsf{Pow2}(\langle b \rangle, k), k, f)$. [This will produce an invalid result if b is too big, in which case the result cannot be held in an Fx in any case]
6. $\langle e \rangle \leftarrow \mathsf{FxPol}(P_{1045}, \langle c \rangle, k, f)$.

7. $\langle g \rangle \leftarrow \mathsf{FxMult}(\langle d \rangle, \langle e \rangle, k, f)$.
8. $\langle g^{-1} \rangle \leftarrow \mathsf{FxDiv}(2^f, \langle g \rangle, k, f)$.
9. $\langle a \rangle \leftarrow (1 - \langle s \rangle) \cdot g + \langle s \rangle \cdot \langle g^{-1} \rangle$.
10. Return $\langle a \rangle$.

The above works, but we have found a little numerical instability due to the division operation.

MAMBA Example: To obtain 2^y where y is secret shared you could run the following:

```
from Compiler import mpc_math
# import comparison
sfloat.vlen = 15   # Length of mantissa in bits
sfloat.plen = 10   # Length of exponent in bits
sfloat.kappa = 4  # Statistical security parameter for floats

y =sfix(4)

# returns 2^4
# extracts k and f from y
exp2_y=mpc_math.exp2_fx(sfix(y))
```

$\mathsf{FLExp2}((\langle v_1 \rangle, \langle p_1 \rangle, \langle z_1 \rangle, \langle s_1 \rangle, \langle \mathsf{err}_1 \rangle))$**:**

This method assumes that $k \leq \ell$. We do not support a method if $k > \ell$, and so if this happens we will signal an error.

1. If $k > \ell$ then $\mathsf{err}_1 \leftarrow 1$.
2. $\max \leftarrow \lceil \log_2(2^{k-1} - 1 + \ell) - \ell + 1 \rceil$.
3. $\langle a \rangle \leftarrow \mathsf{LT}(\langle p_1 \rangle, \max, k)$.
4. $\langle b \rangle \leftarrow \mathsf{LT}(\langle p_1 \rangle, -\ell + 1, k)$.
5. $\langle c \rangle \leftarrow \mathsf{LT}(\langle p_1 \rangle, -2 \cdot \ell + 1, k)$.
6. $\langle (1 - c) \cdot a \rangle \leftarrow (1 - \langle c \rangle) \cdot \langle a \rangle$.
7. $\langle p_2 \rangle \leftarrow -\langle (1 - c) \cdot a \rangle \cdot (\langle b \rangle \cdot \ell + \langle p_1 \rangle)$.
8. $\langle x \rangle, \langle 2^{p_2} \rangle \leftarrow \mathsf{Trunc}(\langle v_1 \rangle, \ell, \langle p_2 \rangle)$.
9. $\langle y \rangle \leftarrow \langle v_1 \rangle - \langle x \rangle \cdot \langle 2^{p_2} \rangle$.
10. $\langle d \rangle \leftarrow \mathsf{EQZ}(\langle y \rangle, \ell)$.
11. $\langle b \cdot s_1 \rangle \leftarrow \langle b \rangle \cdot \langle s_1 \rangle$.
12. $\langle (1 - d) \cdot s_1 \rangle \leftarrow (1 - \langle d \rangle) \cdot \langle s_1 \rangle$.
13. $\langle x \rangle \leftarrow (1 - \langle b \cdot s_1 \rangle) \cdot (\langle x \rangle - \langle (1 - d) \cdot s_1 \rangle) + \langle b \cdot s_1 \rangle \cdot (2^\ell - 1 + \langle d \rangle - \langle x \rangle)$.
14. $\langle y \rangle \leftarrow \langle (1 - d) \cdot s_1 \rangle \cdot (\langle 2^{p_2} \rangle - \langle y \rangle) + (1 - \langle s_1 \rangle) \cdot \langle y \rangle$.
15. $\langle w \rangle \leftarrow \langle (1 - c) \cdot a \rangle \cdot ((1 - \langle b \rangle) \cdot \langle x \rangle + \langle b \cdot s_1 \rangle) \cdot (1 - 2 \cdot \langle s_1 \rangle) - \langle c \rangle \cdot \langle s_1 \rangle$.
16. $\langle u \rangle \leftarrow \langle (1 - c) \cdot a \rangle \cdot (\langle b \rangle \cdot \langle x \rangle + (1 - \langle b \rangle) \cdot 2^\ell \cdot \mathsf{Inv}(\langle 2^{p_2} \rangle) \cdot \langle y \rangle) + (2^\ell - 1) \cdot \langle c \rangle \langle s_1 \rangle$.

17. $\langle u_\ell \rangle, \ldots, \langle u_1 \rangle \leftarrow \mathsf{BitDec}(\langle u \rangle, \ell, \ell)$.

18. For $i \in [1, \ldots, \ell]$ do

    (a) [In this loop $(cv_i, cp_i, 0, 0)$ represents the floating point number $2^{2^{-i}}$].

    (b) $\langle a_i \rangle \leftarrow 2^{\ell-1} \cdot (1 - \langle u_i \rangle) + cv_i \cdot \langle u_i \rangle$.

    (c) $\langle b_i \rangle \leftarrow -(\ell - 1) \cdot (1 - \langle u_i \rangle) + cp_i \cdot \langle u_i \rangle$.

19. $(\langle v_u \rangle, \langle p_i \rangle, 0, 0) \leftarrow \mathsf{FLProd}((\langle a_1 \rangle, \langle b_1 \rangle, 0, 0), \ldots, (\langle a_\ell \rangle, \langle b_\ell \rangle, 0, 0))$. [This implements a product of ℓ floating point values, which is performed via a binary tree style method.]

20. $\langle p \rangle \leftarrow \langle a \rangle \cdot (\langle w \rangle + \langle p_u \rangle) + 2^{k-1} \cdot (1 - \langle a \rangle) \cdot (1 - 2\langle s_1 \rangle)$.

21. $\langle v \rangle \leftarrow 2^{\ell-1} \cdot \langle z_1 \rangle + (1 - \langle z_1 \rangle) \cdot \langle v_u \rangle$.

22. $\langle p \rangle \leftarrow -\langle z_1 \rangle \cdot (\ell - 1) + (1 - \langle z_1 \rangle) \cdot \langle p \rangle$.

23. $\langle \mathsf{err} \rangle \leftarrow \mathsf{FlowDetect}(\langle p \rangle, \langle \mathsf{err}_1 \rangle)$

24. Return $(\langle v \rangle, \langle p \rangle, 0, 0, \langle \mathsf{err} \rangle)$.

$\mathsf{FxLog2}(\langle a \rangle, k, f)$**:**

We first map a to a value v in the interval $[1/2, 1]$ by essentially converting to a floating point number. So we a have $a = (v/2^k) \cdot 2^p$ where $v, p \in \mathbb{Z}_{\langle k \rangle}$, and $v/2^k \in [1/2, 1]$. Thus we have $\log_2 a = p + \log_2(v/2^k)$, and we then treat v as a fixed point number and apply the Pade approximation P_{2524}/Q_{2524} from Hart's book [Har78], which produces an *absolute* error of $10^{-8.32}$. We denote by $\mathsf{FxPade}(P_{2524}, Q_{2524}, \langle x \rangle, k, f)$ the evaluation of the rational function P_{2524}/Q_{2524} on the fixed point input $\langle x \rangle$ where $x \in \mathbb{Q}_{\langle k,f \rangle}$. The Pade approximation is given be the rational function defined by the following table

P	0	1	-.20546 66719 51
P	1	1	-.88626 59939 1
P	2	1	+.61058 51990 15
P	3	1	+.48114 74609 89
Q	0	0	+.35355 34252 77
Q	1	1	+.45451 70876 29
Q	2	1	+.64278 42090 29
Q	3	1	+.1

1. $(\langle v \rangle, \langle p \rangle, \langle z \rangle, \langle s \rangle, \langle \mathsf{err} \rangle) \leftarrow \mathsf{Fx2FL}(\mathsf{a}, k, f, k, k)$.

2. $\langle a \rangle \leftarrow \mathsf{FxPade}(P_{2524}, Q_{2524}, \langle v \rangle, k, k)$.

3. $\langle a \rangle \leftarrow \langle a \rangle + \langle p \rangle$.

4. $\langle a \rangle \leftarrow \langle a \rangle \cdot (1 - \langle z \rangle) \cdot (1 - \langle s \rangle) \cdot (1 - \cdot \langle \mathsf{err} \rangle)$.

5. Return $\langle a \rangle$.

MAMBA Example: To obtain `log2`(x) where y is secret shared you could run the following:

```
from Compiler import mpc_math
# import comparison
sfloat.vlen = 15   # Length of mantissa in bits
sfloat.plen = 10   # Length of exponent in bits
sfloat.kappa = 4  # Statistical security parameter for floats

x =sfix(4)
# extracts k and f from y
# returns log_2(4)
log2_x=mpc_math.log2_fx(sfix(x))
```

FLLog2$((\langle v_1\rangle, \langle p_1\rangle, \langle z_1\rangle, \langle s_1\rangle, \langle \mathsf{err}_1\rangle))$**:**

In the following algorithm $(cv_i, cp_i, 0, 0)$ represents the floating point constant $(2 \cdot \log_2 e)/(2 \cdot i + 1)$.

1. $M \leftarrow \lceil \ell/(2 \cdot \log_2 3) - 1/2 \rceil$.
2. $(\langle v_2\rangle, \langle p_2\rangle, 0, 0, 0) \leftarrow \mathsf{FLSub}((2^{\ell-1}, -(\ell-1), 0, 0, 0), (\langle v_1\rangle, -\ell, 0, 0, 0))$.
3. $(\langle v_3\rangle, \langle p_3\rangle, 0, 0, 0) \leftarrow \mathsf{FLAdd}((2^{\ell-1}, -(\ell-1), 0, 0, 0), (\langle v_1\rangle, -\ell, 0, 0, 0))$.
4. $(\langle v_y\rangle, \langle p_y\rangle, 0, 0, 0) \leftarrow \mathsf{FLDiv}((\langle v_2\rangle, \langle p_2\rangle, 0, 0, 0), (\langle v_3\rangle, \langle p_3\rangle, 0, 0, 0))$.
5. $(\langle v_{y^2}\rangle, \langle p_{y^2}\rangle, 0, 0, 0) \leftarrow \mathsf{FLMult}((\langle v_y\rangle, \langle p_y\rangle, 0, 0, 0), (\langle v_y\rangle, \langle p_y\rangle, 0, 0, 0))$.
6. $(\langle v\rangle, \langle p\rangle, 0, 0, 0) \leftarrow \mathsf{FLMult}((\langle v_y\rangle, \langle p_y\rangle, 0, 0), (cv_0, cp_0, 0, 0))$.
7. For $i \in [1, \ldots, M]$ do
   (a) $(\langle v_y\rangle, \langle p_y\rangle, 0, 0, 0) \leftarrow \mathsf{FLMult}((\langle v_y\rangle, \langle p_y\rangle, 0, 0, 0), (\langle v_{y^2}\rangle, \langle p_{y^2}\rangle, 0, 0, 0))$.
   (b) $(\langle v_2\rangle, \langle p_2\rangle, 0, 0, 0) \leftarrow \mathsf{FLMult}((\langle v_y\rangle, \langle p_y\rangle, 0, 0, 0), (cv_i, cp_i, 0, 0, 0))$.
   (c) $(\langle v\rangle, \langle p\rangle, 0, 0, 0) \leftarrow \mathsf{FLAdd}((\langle v\rangle, \langle p\rangle, 0, 0, 0), (\langle v_2\rangle, \langle p_2\rangle, 0, 0, 0))$.
8. $(\langle v_2\rangle, \langle p_2\rangle, \langle z_2\rangle, \langle s_2\rangle, \langle \mathsf{err}_2\rangle) \leftarrow \mathsf{Int2FL}(\ell, -\langle p\rangle, \ell, \ell)$.
9. $(\langle v\rangle, \langle p\rangle, \langle z\rangle, \langle s\rangle, \langle \mathsf{err}\rangle) \leftarrow \mathsf{FLSub}((\langle v_2\rangle, \langle p_2\rangle, \langle z_2\rangle, \langle s_2\rangle, \langle \mathsf{err}_2\rangle), (\langle v\rangle, \langle p\rangle, 0, 0, 0))$.
10. $\langle a\rangle \leftarrow \mathsf{EQ}(\langle p_1\rangle, -(\ell-1), k)$.
11. $\langle b\rangle \leftarrow \mathsf{EQ}(\langle v_1\rangle, 2^{\ell-1}, \ell)$.
12. $\langle z\rangle \leftarrow \langle a\rangle \cdot \langle b\rangle$.
13. $\langle v\rangle \leftarrow \langle v\rangle \cdot (1 - \langle z\rangle)$.
14. $\langle \mathsf{err}\rangle \leftarrow \mathsf{OR}(\langle \mathsf{err}\rangle, \langle \mathsf{err}_1\rangle)$.
15. $\langle \mathsf{err}\rangle \leftarrow \mathsf{OR}(\mathsf{OR}(\langle z_1\rangle, \langle s_1\rangle), \mathsf{err})$.
16. $\langle p\rangle \leftarrow \langle p\rangle \cdot (1 - \langle z\rangle)$.
17. Return $(\langle v\rangle, \langle p\rangle, \langle z_1\rangle, \langle s_1\rangle, \langle \mathsf{err}\rangle)$.

Trigonometic Functions

All three basic trigonometric functions support inputs of either fixed point or floating point precision. With the output type being equal to the input type. The computation of $\sin(x)$ and $\cos(x)$ are performed using polnoymial approximations, with the computation of $\tan(x)$ done via $\sin(x)/\cos(x)$. The basic idea for $\sin(x)$ and $\cos(x)$ is to first reduce the argument x into the range $[0, \ldots, 2\pi)$, so as to obtain a new argument (which we call y) We then compute a bit b_1 to test as to whether $y \in [0, \pi)$ or $[\pi, 2\pi)$ (with 0 being the former). We then reduce y to z by reducing it into the range $[0, \pi)$, and compute a bit b_2 which says whether z is in the range $[0, \pi/2)$ or $[\pi/2, \pi)$. We finally reduce z into the range $[0, \pi/2)$ resulting in w. Then a polynomial is used to compute $\sin(w)$ or $\cos(w)$, which means we need to now *scale* w into the range $[0, 1)$ to obtain v. For the polynomial approximations to the basic functions in the range $[0, \pi/2)$, where the argument is given as $w = v \cdot \pi/2$ we use the following approximations from Hart's book [Har78]

$$\sin(w) = v \cdot P_{3307}(v^2),$$
$$\cos(w) = P_{3508}(v^2).$$

Where we have

	P_{3307}		P_{3508}	
0	1	+.15707 96326 79489 66192 31314 989	0	+.99999 99999 99999 99999 99914 771
1	0	-.64596 40975 06246 25365 51665 255	0	-.49999 99999 99999 99999 91637 437
2	-1	+.79692 62624 61670 45105 15876 375	-1	+.41666 66666 66666 66653 10411 988
3	-2	-.46817 54135 31868 79164 48035 89	-2	-.13888 88888 88888 88031 01864 15
4	-3	+.16044 11847 87358 59304 30385 5	-4	+.24801 58730 15870 23300 45157
5	-5	-.35988 43235 20707 78156 5727	-6	-.27557 31922 39332 25642 1489
6	-7	+.56921 72920 65732 73962 4	-8	+.20876 75698 16541 25915 59
7	-9	-.66880 34884 92042 33722	-10	-.11470 74512 67755 43239 4
8	-11	+.60669 10560 85201 792	-13	+.47794 54394 06649 917
9	-13	-.43752 95071 18174 8	-15	-.15612 26342 88277 81
10	-15	+.25002 85418 9303	-18	+.39912 65450 7924

NOTE: Polynomial tables are described by the monomial number, the degree of the approximation p and its significand s. The coefficient of the ith monomial can be obtained by multiplying the significand by 10^{p_i} as follows: $s_i \cdot 10^{p_i}$.

$\mathsf{F} \star \mathsf{TrigSub}(\langle x \rangle)$**:**

1. $\langle f \rangle \leftarrow \mathsf{F} \star \mathsf{Mult}(\langle x \rangle, (1/(2 \cdot \pi))$
2. $\langle f \rangle \leftarrow \mathsf{F} \star \mathsf{Floor}(\langle f \rangle)$.
3. $\langle y \rangle \leftarrow \mathsf{F} \star \mathsf{Mult}(\langle f \rangle, (2 \cdot \pi))$.
4. $\langle y \rangle \leftarrow \mathsf{F} \star \mathsf{Add}(\langle x \rangle, -\langle y \rangle)$.
5. $\langle b_1 \rangle \leftarrow \mathsf{F} \star \mathsf{GE}(\langle y \rangle, (\pi))$.
6. $\langle f \rangle \leftarrow \mathsf{F} \star \mathsf{Add}(2 \cdot \pi, -\langle y \rangle)$
7. $w \leftarrow \mathsf{F} \star \mathsf{Choose}(\langle f \rangle, \langle y \rangle, \langle b_1 \rangle)$.
8. $\langle b_2 \rangle \leftarrow \mathsf{F} \star \mathsf{GE}(\langle 2 \rangle, (\pi/2))$.
9. $\langle f \rangle \leftarrow \mathsf{F} \star \mathsf{Add}(\pi, -\langle w \rangle)$
10. $w \leftarrow \mathsf{F} \star \mathsf{Choose}(\langle f \rangle, \langle w \rangle, \langle b_2 \rangle)$.
11. Return $(\langle w \rangle, \langle b_1 \rangle, \langle b_2 \rangle)$.

MAMBA Example: To reduce the angle you could execute the following (note that this function call is meant to be used internally):

```
from Compiler import mpc_math
x = sfix(4) # sfloat(4)
# returns an angle in the [0,pi/2) interval in w and flags b1 and b2.
w, b1, b2 = mpc_math.sTrigSub_fx(x)
```

$\mathsf{F} \star \mathsf{Sin}(\langle x \rangle)$

We present these routines as generic routines given the specific helper subroutine above; we assume an obvious overloading/translation of arguments. We let $\mathsf{F} \star \mathsf{Choose}(\langle x \rangle, \langle y \rangle, \langle b \rangle)$, for a shared bit b, denote an operation which produces $\langle x \rangle$ if $\langle b \rangle = 1$ and $\langle y \rangle$ otherwise This is easily obtained by securely multiplying each component share of the data representing $\langle x \rangle$ etc by $\langle b \rangle$. So for fixed point representations this becomes, irrespective of the values k and f,

1. $\langle a \rangle \leftarrow \langle b \rangle \cdot \langle x \rangle + (1 - \langle b \rangle) \cdot \langle y \rangle$

For floating point representations this becomes

1. $\langle w \rangle, \langle b_1 \rangle, \langle b_2 \rangle \leftarrow \mathsf{F} \star \mathsf{TrigSub}(\langle x \rangle)$
2. $\langle v \rangle \leftarrow \langle w \rangle \cdot (1/(\pi/2))$.
3. $\langle b \rangle \leftarrow \mathsf{F} \star \mathsf{Choose}(\langle -1 \rangle, \langle 1 \rangle, \langle b_1 \rangle)$.
4. $\langle \sin(v) \rangle \leftarrow \langle v \rangle \cdot \mathsf{F} \star \mathsf{Pol}(P_{3307}, \langle v^2 \rangle)$.
5. Return $(\langle b \rangle \cdot \langle \sin(v) \rangle)$.

MAMBA Example: To obtain the `sin` of any value, you could execute the following:

```
from Compiler import mpc_math
x = sfix(4) # sfloat(4)
# returns the sin of a number of any interval
y = mpc_math.sin(x)
```

$\mathsf{F} \star \mathsf{Cos}(\langle x \rangle)$

Likewise this becomes

1. $\langle w \rangle, \langle b_1 \rangle, \langle b_2 \rangle \leftarrow \mathsf{F} \star \mathsf{TrigSub}(\langle x \rangle)$
2. $\langle v \rangle \leftarrow \langle w \rangle$.
3. $\langle b \rangle \leftarrow \mathsf{F} \star \mathsf{Choose}(\langle -1 \rangle, \langle 1 \rangle, \langle b_2 \rangle)$.
4. $\langle \cos(v) \rangle \leftarrow \mathsf{F} \star \mathsf{Pol}(P_{3308}, \langle v^2 \rangle)$.
5. Return $(\langle b \rangle \cdot \langle \cos(v) \rangle)$.

MAMBA Example: To obtain the `sin` of any value, you could execute the following:

```
from Compiler import mpc_math
x = sfix(4) # sfloat(4)
# returns the cos of an angle on any interval
y = mpc_math.cos(x)
```

$\mathsf{F} \star \mathsf{Tan}(\langle x \rangle)$

Likewise this becomes

1. $(\langle w \rangle, \langle b_1 \rangle, \langle b_2 \rangle) \leftarrow \mathsf{F} \star \mathsf{TrigSub}(\langle x \rangle)$.
2. $\langle v \rangle \leftarrow \langle w \rangle \cdot (1/(\pi/2))$.
3. $\langle b \rangle \leftarrow \mathsf{F} \star \mathsf{Choose}(\langle -1 \rangle, \langle 1 \rangle, \langle b_1 \rangle)$.
4. $\langle \sin(v) \rangle \leftarrow \langle v \rangle \cdot \mathsf{F} \star \mathsf{Pol}(P_{3307}, \langle v^2 \rangle)$.
5. $\langle \sin(x) \rangle \leftarrow (\langle b \rangle \cdot \langle \sin(v) \rangle)$.
6. $\langle v \rangle \leftarrow \langle w \rangle$.
7. $\langle b \rangle \leftarrow \mathsf{F} \star \mathsf{Choose}(\langle -1 \rangle, \langle 1 \rangle, \langle b_2 \rangle)$.
8. $\langle \cos(v) \rangle \leftarrow \mathsf{F} \star \mathsf{Pol}(P_{3308}, \langle v^2 \rangle)$.
9. $\langle \sin(x) \rangle \leftarrow (\langle b \rangle \cdot \langle \cos(v) \rangle)$.
10. $\langle \tan(x) \rangle \leftarrow \mathsf{F} \star \mathsf{Div}(\langle \sin(x) \rangle, \langle \cos(x) \rangle)$.
11. Return $\langle \tan(x) \rangle$.

MAMBA Example: To obtain the `tan` of any value, you could execute the following:

```
from Compiler import mpc_math
x = sfix(4) # sfloat(4)
# returns the tan of an angle on any interval
y = mpc_math.tan(x)
```

Inverse Trigonometric Functions

(TO DO:) *Given that SCALE-MAMBA currently only supports square root operations for sfix inputs, inverse trigonometric functions are restricted to sfix inputs.* To obtain $\arcsin$ and $\arccos$ one makes use of the formula:

$$\arcsin(x) = \arctan\left(\frac{x}{\sqrt{1-x^2}}\right),$$
$$\arccos(x) = \frac{\pi}{2} - \arcsin(x).$$

Note, that $\arcsin$ and $\arccos$ are only defined when $|x| \leq 1$. The value of $\arctan(x)$ is however defined for all real x. For $\arctan(x)$ we first reduce to positive values of x by using the formula

$$\arctan(-x) = -\arctan(x).$$

We then reduce to the interval $[0, 1)$ using the formula

$$\arctan(x) = \frac{\pi}{2} - \arctan\left(\frac{1}{x}\right).$$

The final approximation to $\arctan(x)$ for $x \in [0, 1)$ is obtained using the Pade approximation P_{5102}/Q_{5102} from Hart's book [Har78]. Where the polynomials are represented as in our earlier descriptions.

	P_{5102}		Q_{5102}	
0	5	+.21514 05962 60244 19331 93254 468	5	+.21514 05962 60244 19331 93298 234
1	5	+.73597 43380 28844 42408 14980 706	5	+.80768 78701 15592 48851 76713 209
2	6	+.10027 25618 30630 27849 70511 863	6	+.12289 26789 09278 47762 98743 322
3	5	+.69439 29750 03225 23370 59765 503	5	+.97323 20349 05355 56802 60434 387
4	5	+.25858 09739 71909 90257 16567 793	5	+.42868 57652 04640 80931 84006 664
5	4	+.50386 39185 50126 65579 37791 19	5	+.10401 13491 56689 00570 05103 878
6	3	+.46015 88804 63535 14711 61727 227	4	+.12897 50569 11611 09714 11459 55
7	2	+.15087 67735 87003 09877 17455 528	2	+.68519 37831 01896 80131 14024 294
8	-1	+.75230 52818 75762 84445 10729 539	1	+.1

The following protocol for `arcsin` implements the formulas from before. The protocol and its implementation, make use of our secure implementation of Square Root. This method is implemented, and it is used to derive `arccos`.

$\mathsf{F} \star \mathsf{ArcSin}(\langle x \rangle)$

1. $\langle x^2 \rangle \leftarrow \mathsf{F} \star \mathsf{Mult}(\langle x \rangle, \langle x \rangle)$.
2. $\langle -x^2 \rangle \leftarrow \mathsf{F} \star \mathsf{Neg}(\langle x^2 \rangle)$.
3. $\langle 1 - x^2 \rangle \leftarrow \mathsf{F} \star \mathsf{Add}(1, \langle -x^2 \rangle)$.
4. $\langle \sqrt{1-x^2} \rangle \leftarrow \mathsf{F} \star \mathsf{Sqrt}(1, \langle 1 - x^2 \rangle)$.
5. $\langle v \rangle \leftarrow \mathsf{F} \star \mathsf{Div}(\langle x \rangle, \langle \sqrt{1-x^2} \rangle)$.
6. $\langle y \rangle \leftarrow \mathsf{F} \star \mathsf{ArcTan}(\langle v \rangle)$.
7. If $\star = \mathsf{L}$
   (a) $\langle |x| \rangle \leftarrow \mathsf{FLAbs}(\langle x \rangle)$.
   (b) $\langle y_{\mathsf{err}} \rangle \leftarrow \mathsf{FLGT}(\langle |x| \rangle, 1.0)$.
8. Return $\langle y \rangle$.

MAMBA Example: To obtain the `arcSin` on the (-1,1) interval, you could execute the following:

```
from Compiler import mpc_math
x =sfix(0.5)
# returns the tan of an angle on any interval
y = mpc_math.asin(x)
```

F $\star$ ArcCos($\langle x \rangle$)

1. $\langle y \rangle \leftarrow$ F $\star$ ArcSin($\langle x \rangle$).
2. $\langle -y \rangle \leftarrow$ F $\star$ Neg($\langle y \rangle$).
3. $\langle \pi/2 - y \rangle \leftarrow$ F $\star$ Add($\pi/2, \langle -y \rangle$).
4. Return $\langle y \rangle$.

MAMBA Example: To obtain the `arcCos` of any value on the (-1,1) interval, you could execute the following:

```
from Compiler import mpc_math
x =sfix(0.5)
# returns the tan of an angle on any interval
y = mpc_math.acos(x)
```

F $\star$ ArcTan($\langle x \rangle$)

1. $\langle s \rangle \leftarrow$ F $\star$ LTZ($\langle z \rangle$).
2. $\langle |x| \rangle \leftarrow$ F $\star$ Abs($\langle x \rangle$).
3. $\langle b \rangle \leftarrow$ F $\star$ GT($\langle |x| \rangle, 1.0$).
4. $\langle v \rangle \leftarrow$ F $\star$ Div($1, \langle |x| \rangle$).
5. $\langle v \rangle \leftarrow$ F $\star$ Choose($\langle |x| \rangle, \langle v \rangle, 1 - \langle b \rangle$).
6. $\langle v^2 \rangle \leftarrow$ F $\star$ Mul($\langle v \rangle, \langle v \rangle$).
7. $\langle y \rangle \leftarrow$ FxPade($P_{5102}, Q_{5102}, \langle v^2 \rangle$).
8. $\langle y \rangle \leftarrow$ F $\star$ Mul($\langle v \rangle, \langle y \rangle$).
9. $\langle \pi/2 - y \rangle \leftarrow$ F $\star$ Sub($\pi/2, \langle y \rangle$).
10. $\langle y \rangle \leftarrow$ F $\star$ Choose($\langle y \rangle, \langle \pi/2 - y \rangle, 1 - \langle b \rangle$).
11. $\langle -y \rangle \leftarrow$ F $\star$ Neg($\langle y \rangle$).
12. $\langle y \rangle \leftarrow$ F $\star$ Choose($\langle y \rangle, \langle -y \rangle, 1 - \langle s \rangle$).
13. Return $\langle y \rangle$.

MAMBA Example: To obtain the `arcTan` of any value, you could execute the following:

```
from Compiler import mpc_math
x =sfix(0.5)
# returns the tan of an angle on any interval
y = mpc_math.atan(x)
```

References

[ABZS13] Mehrdad Aliasgari, Marina Blanton, Yihua Zhang, and Aaron Steele. Secure computation on floating point numbers. In *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*. The Internet Society, 2013.

[CdH10] Octavian Catrina and Sebastiaan de Hoogh. Improved primitives for secure multiparty integer computation. In Juan A. Garay and Roberto De Prisco, editors, *Security and Cryptography for Networks, 7th International Conference, SCN 2010, Amalfi, Italy, September 13-15, 2010. Proceedings*, volume 6280 of *Lecture Notes in Computer Science*, pages 182–199. Springer, 2010.

[CDM00] Ronald Cramer, Ivan Damgård, and Ueli M. Maurer. General secure multi-party computation from any linear secret-sharing scheme. In Bart Preneel, editor, *Advances in Cryptology - EUROCRYPT 2000, International Conference on the Theory and Application of Cryptographic Techniques, Bruges, Belgium, May 14-18, 2000, Proceeding*, volume 1807 of *Lecture Notes in Computer Science*, pages 316–334. Springer, 2000.

[CS10] Octavian Catrina and Amitabh Saxena. Secure computation with fixed-point numbers. In Radu Sion, editor, *Financial Cryptography and Data Security, 14th International Conference, FC 2010, Tenerife, Canary Islands, January 25-28, 2010, Revised Selected Papers*, volume 6052 of *Lecture Notes in Computer Science*, pages 35–50. Springer, 2010.

[CS19] Daniele Cozzo and Nigel P. Smart. Using TopGear in Overdrive: A more efficient ZKPoK for SPDZ, 2019.

[DFK+06] Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In Shai Halevi and Tal Rabin, editors, *Theory of Cryptography, Third Theory of Cryptography Conference, TCC 2006, New York, NY, USA, March 4-7, 2006, Proceedings*, volume 3876 of *Lecture Notes in Computer Science*, pages 285–304. Springer, 2006.

[DKL+13] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *Computer Security - ESORICS 2013 - 18th European Symposium on Research in Computer Security, Egham, UK, September 9-13, 2013. Proceedings*, volume 8134 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2013.

[DPSZ12] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Safavi-Naini and Canetti [SC12], pages 643–662.

[GHS12] Craig Gentry, Shai Halevi, and Nigel P. Smart. Homomorphic evaluation of the AES circuit. In Safavi-Naini and Canetti [SC12], pages 850–867.

[Har78] John F. Hart. *Computer Approximations*. Krieger Publishing Co., Inc., Melbourne, FL, USA, 1978.

[KPR18] Marcel Keller, Valerio Pastro, and Dragos Rotaru. Overdrive: Making SPDZ great again. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part III*, volume 10822 of *Lecture Notes in Computer Science*, pages 158–189. Springer, 2018.

[KRSW18] Marcel Keller, Dragos Rotaru, Nigel P. Smart, and Tim Wood. Reducing communication channels in MPC. In Dario Catalano and Roberto De Prisco, editors, *Security and Cryptography for Networks - 11th International Conference, SCN 2018, Amalfi, Italy, September 5-7, 2018, Proceedings*, volume 11035 of *Lecture Notes in Computer Science*, pages 181–199. Springer, 2018.

[Lie12] Manuel Liedel. Secure distributed computation of the square root and applications. In Mark Dermot Ryan, Ben Smyth, and Guilin Wang, editors, *Information Security Practice and Experience - 8th International Conference, ISPEC 2012, Hangzhou, China, April 9-12, 2012. Proceedings*, volume 7232 of *Lecture Notes in Computer Science*, pages 277–288. Springer, 2012.

[Mau06] Ueli M. Maurer. Secure multi-party computation made simple. *Discrete Applied Mathematics*, 154(2):370–381, 2006.

[SC12] Reihaneh Safavi-Naini and Ran Canetti, editors. *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, volume 7417 of *Lecture Notes in Computer Science*. Springer, 2012.

[sec] secureSCM. Deliverable D9.2. `https://www1.cs.fau.de/filepool/publications/octavian_securescm/SecureSCM-D.9.2.pdf`.

[SW18] Nigel P. Smart and Tim Wood. Error-detecting in monotone span programs with application to communication efficient multi-party computation, 2018.