# Manual on Usage of PICCO – A General-Purpose Compiler for Private Distributed Computation

The source code of PICCO mainly consists of two directories: *compiler* and *compute*. The compiler directory contains the source code of the PICCO compiler whose functionality is to translate a user's program into its secure implementation. The compute directory contains the source code of the computational framework that will be used to securely execute the user's translated program in a distributed setting.

## Source Code Dependencies

To compile or run user programs using PICCO code, a machine should have the following libraries installed:

- GMP library
- Openssl library

## Compilation of PICCO

To use PICCO, the following three programs need to be compiled:

1) a PICCO compiler,
2) a utility program that will be used to generate secret shares of private inputs and to assemble the output of computational results upon the completion of secure computation,
3) a seed program that will be used to generate and transmit secret random seeds to each of the computational parties at the time of program initiation (so that shares of random values can be locally produced by each party without interaction).

To compile all three programs, one needs to go to the directory compiler/ and run the shell program "compile.sh" (by typing "./compile.sh"), which will produce three executable files `picco`, `picco-utility`, and `picco-seed` and put them in the directory compiler/bin/. The three executable files correspond to the PICCO compiler, the utility, and the seed program, respectively, and can be placed in any directory of user's choice at a later time.

These executable programs are used to compile a user's program written in an extension of C into its secure implementation, produce shares of private inputs to be used in secure computation, and reconstruct the output from the shares after secure execution of the user program.

## Translation of user programs

Before describing the procedure for compiling a user program, we explain the composition of a **SMC config** file that the program expects.

- **SMC config.** The SMC config is a text file that consists of five lines, with each of them being in the format of "$P$:$V$", where $P$ indicates an SMC parameter and $V$ indicates its value. The value of each parameter will be retrieved from the line at which it is positioned in the file, not based on the string $P$ with which it is labeled. For instance, the value of $V$ on the first line will always store the modulus size regardless of $P$. The reason for including $P$ in the config file is to provide intuitive information to end-users when they edit the file, so that each parameter is assigned a correct value.
  Out of the five lines, the first three specify parameters of a secret sharing scheme, which are: the modulus size, the number of computational parties, and the threshold value (in that order). Specifying the modulus size is optional, and if a programmer is uncertain what modulus size should be used, its value $V$ should be left blank in the config file. In this case, the PICCO compiler will determine the optimal modulus size by analyzing the program at the translation time, and later make it available to the utility program.
  The last two lines specify the number of input and output parties in the computation. By specifying these two values, a user is able to run a program with inputs distributed multiple parties and produce multiple outputs with each of them being sent to a distinct output party.
  Later, it will be assumed that input/output/computational parties are numbered sequentially from 1 up until the specified number of parties. For example, if the number of inputs parties is $N$, they are expected to be numbered 1 through $N$. The same entity can take on different roles (e.g., input party 1 can also be output party 2).
- **Program compilation.** To compile a user's program into its secure implementation, one needs to execute the following command: `picco <user program> <SMC config> <translated program> <utility config>`.
  Here, the arguments that the executable `picco` takes are:
  1) the name of the file containing user program to be translated;
  2) the of the file containing SMC config as described above;

3) the name of a file that will store the result of user program translation as a C++ program;
4) the name of a file that will store information that needs to be communicated to the utility program (such as the setup information found in the SMC config file and the list of variables used in I/O), i.e., a config file for the utility program.

The executable takes two files as its input and produces two files as its output.

## GENERATION OF INPUTS FOR USER PROGRAMS

Before secure computation can take place, the input parties need to prepare input data (that could be private, public, or both) and send them to the computational parties. Assuming that at least one of the inputs is private, an input party needs to call the program `picco-utility` to produce shares of private inputs. The same program is also used to assemble the output of upon completion of secure computation, as described later. In what follows, we first describe the usage of built-in I/O functions within user programs, then the format of files that contains plaintext input of an input party, followed by usage of the utility program for input generation.

- **Built-in I/O functions.** Input and output in user programs is handled through built-in I/O functions `smcinput` and `smcoutput`, respectively. Both of them have the same interface and take as the first argument the name of a variable and as the second argument the id of an input (output) party from whom the variable will be read (to whom the variable will be written). If the variable is an array, the sizes of each array dimension need to be specified as additional arguments (i.e., the number of arguments depends on the variable type). For instance, if `x` was declared as a two-dimensional array with both dimensions being 10, the program can specify `smcinput(x,1,10,10)` to read shares of 100 elements from input party 1. Note that if an array variable was declared to have a larger number of elements than the number of elements being read from the input, the read values will always be placed in the beginning of the array (or in the beginning of each row for the specified number of rows for two-dimensional arrays).
  In the current implementation, both `smcinput` and `smcoutput` are not allowed to be placed within loops or iterations. That is, a user should not write code such as:
  ```
  for(i = 0; i < 10; i++)
      for(j = 0; j < 10; j++)
          smcinput(x[i][j], 1);
  ```
  The reason is that the parser extracts each call to `smcinput`/`smcoutput` by a scan without performing a more complex analysis or trial execution of the program. Thus, the code above will result in a single integer variable (i.e., `x[i][j]`) being read from the input instead of the entire 100-element array.
- **Input file format.** Each input party needs to prepare her input data in a text file as described below. This text file will be used by the utility program. Note that the user program may read input from multiple parties, and each input party prepares her data independently of other inputs parties.
  - *Order of input data*: A user's program may read more than one variable from an input party, and the order of variables in an input file for that party needs to be the same as their relative order in the user program. For instance, three I/O statements `smcinput(x,1,2)`, `smcinput(y,2,10)`, and `smcinput(z,1,10,10)` appear in a user program in that order, the value of `x` should precede that of `z` in the input file of party 1.
  - *Data format*: If a variable is a single integer/real or a one-dimensional array, its value should be listed on a separate single line as `var = value1,value2,...` for both public and private variables. For instance, in the above example, we will have `x = 1,2` when `x` is a private array containing two elements `1` and `2`. If a variable is a two-dimensional array of integers, each row in a matrix (if we think of a two-dimensional array as a matrix) should be listed on a separate line with the name of the variable repeated on each line.
- **Generating input data**. After an input party generates input data in the specified format and saves it in a file of the user's choice, the the utility program can be invoked as follows:
  `picco-utility -I <input party ID> <input filename> <utility config> <shares filename>`.
  The utility program `picco-utility` takes a flag and four other arguments, which are:
  1) a flag that tells the utility program to either generate inputs (`-I`) or assemble outputs (`-O`);
  2) the ID of the input party;
  3) the name of the input file prepared by the input party in the format described above (storing values for both public and private variables input into the computation);
  4) the name of the file produced during program translation;
  5) a prefix of output files in which generated input shares will be stored.

  The utility program will read the input data and utility config and produce the same number of output files with shares as the number of computational parties $N$. The program's output will be stored in files named "`<shares filename>ID`", where ID is the identification number for each computational party between 1 and $N$. The values of public variables are copied unmodified from the input file into the shares files, while the values of private variables are secret shared, with each computational party obtaining a single share stored in the corresponding shares file.

## EXECUTION OF USER PROGRAMS

In order to run a user's translated program in a distributed setting, one needs to compile it using a native C++ compiler to produce a binary executable file, create a runtime config file, and send the send the executable to each computational party together with the runtime config and a file that stores input shares for that party. These steps are discussed in more detail below.

- To **compile** the translated program, the program should be placed in the compute/ directory at the compilation time, as it needs library functions stored in the directory compute/smc-compute/. Moreover, the makefile in the compute/ directory needs to be updated to have rules for the name of the program. That is, if the translated program is stored in a file named `X.cpp`, `X.cpp` and `X.o` need to be added to the lists of source and object files, respectively, and the makefile also needs to be updated to include build rule for `X`. For ease of use, every appearance of `test-code` can simply be substituted with `X` in the makefile.
  Then the binary executable `X` of the translated program can be produced by typing `make X` in the compute/ directory. The resulting executable file will be stored in the same directory and can later be placed in any other directory. Notice that, when one runs `make X` for the first time, the makefile automatically compiles source files of the SMC library stored in directory compute/smc-compute. This will be performed only once (i.e., from the second running, the library source won't be compiled any more).
- The **runtime config** will be used during program execution by computational parties and needs to be formed as follows. It is a text file that consists of $N$ text lines, where $N$ is the number of computational parties running the secure computation. Each line specifies information about the runtime setup and, in particular, contains the following four values separated by commas:
  1) an ID of a computational party between 1 and $N$;
  2) an IP address or a domain name of the computational party;
  3) an open port number to connecting to that party;
  4) a file name of the public key of that party for establishing a secure communication channel (this can be specified using a path or just the file name, but in either case it must be locatable by the running program; i.e., in the latter case the file must reside in the same directory as the program being executed).

  The four values should be listed in the specified order on each line. Note that the same runtime config file should be distributed to all computational parties.
  All programs compiled by PICCO use pair-wise secure channels protected using symmetric key cryptogrpaphy, and the parties' public keys are used to communicate the key material. Each computational party must have a public-private key pair, and the name of a file containing a computational node's public key is stored in the runtime configuration file. In the current implementation, only RSA keys are supported and a key stored in a file needs to be in a format compatible with what OpenSSL uses.
- The **execution** uses $N + 1$ machines that can communicate with each other, where $N$ is the number of computational parties participating in the computation. Out of these machines, $N$ machines correspond to computational nodes and the remaining machine is a additional node that supplies shared randomness to the computational parties using the seed program `picco-seed` (produced at the time of PICCO compilation) and can be controlled by the data owners. Strictly speaking, the seeds need to be communicated to the given set of computational parties only once, after which the computational parties can execute secure implementations of various programs any number of times. However, for simplicity, our implementation expects communication from `picco-seed` for each program execution, and the time for generating and transmitting the seeds is not counted in the program execution time.
  To initiate secure computation, each computational party needs to execute the following command:
  `X <ID> <runtime config> <privkey file> M K <share file 1> ... <share file M> <output 1> ... <output K>`.
  The first two arguments to the program are the ID of the computational party and the name of the runtime config file. The third argument stores the private key of the public-private key pair of the computational party running the computation. `M` and `K` are the number of input and output parties, respectively. After the first five arguments, the next `M` arguments list the names of the files containing input shares of input parties 1 through `M`. The `K` arguments that follow will be used for storing the output of the execution. These arguments specify prefixes of output files for each of the output parties. The program will store shares of the output for the output party `i` in a file named "`<output i>ID`" using the ID of the computational party. The same prefixes for the output filenames need to be used across all computational parties. This is because the output reconstruction program expects consistent naming of the output files.
  **Our current implementation requires that the computational parties start the execution in a particular order:** the program has to be started by the parties in the decreasing order of their IDs, i.e., party $N$ first, then by party $N - 1$, etc. with party 1 starting the program last. This is because the machines connect to each other in a specific order. After all computational parties start, the $(N + 1)$th machine needs to run:
  `picco-seed <runtime config> <utility config>`.

Upon computation completion, each program outputs the program running time and stores the result of computation (output using `smcoutput`) in a file for each output party. If the output for some output party contains private variables, that party will need to use the the utility program to reconstruct the result.

## RECONSTRUCTION OF PROGRAM RESULTS

The procedure of reconstructing program results is very similar to that of generating program inputs using the utility program. Each output party needs to the execute the following command:
`picco-utility -O <output party ID> <shares filename> <utility config> <result filename>`.
Here the flag `-O` indicates that the utility program will be used to reconstruct the program result. The third argument is the name prefix of output files containing values (e.g., shares for private variables) of program results (the program will read files "`<shares filename>i`" for each computational party `i`, and the last argument is the name of the file that will store the result of data reconstruction. Other arguments are self-explanatory. The utility program stores the plaintext output data in the same format as the plaintext input was stored in the input files.

## RESTRICTIONS ON USER PROGRAMS

In the current implementation, not all features of C are supported in user programs written our extension of C. We tested a rather small subset of C reserved words and the rest are commented out (and may not go past the parser). Thus, if your program does not compile, please contact us and we will examine the code and add the necessary functionalities to the PICCO compiler. The list below provides a more detailed information about restrictions on user programs in the current implementation.

- The current implementation supports private arrays with at most two dimensions.
- Built-in I/O statements `smcinput` and `smcoutput` are not allowed to appear within the body of a loop or iteration (such as a repeatedly called function).
- If `smcinput` or `smcoutput` is used for an array, the number of elements to read/write needs to be given as either a constant or a variable initialized with a constant. More complex ways of specifying the size (such as arbitrary expressions) are currently not supported.
- There are restrictions on arithmetic or comparison statements used within the body of a parallel loop: If a statement contains more than a single operation, it needs to be rewritten into multiple statements that execute one operation at a time. This applies to type casting of private variables as well.
  Also, any assignment statement should store the result of the computation into an element of an array because it is not meaningful to simultaneously store multiple values from different loop iterations in a single non-array type variable.
- Our current implementation does not allow code that uses private variables to be located in multiple files, e.g., in header files. Thus, all code to be translated needs to be placed in a single file.
- Our current implementation does not allow for global private variable declaration.
- During program translation, the PICCO compiler places a number of temporary variables in the translated user program. Thus, if the user program contains variables with the same names, they might result in conflicts and it is the best to avoid declaring variables with the same names in the user program. The variables created by the compiler are:
  - `tmp`
  - `ftmp`
  - `priv_ind`
  - `priv_tmp`
  - `priv_ftmp`

  The last three variables on the list are used only if the user program contains at least one access to an array with a private index.
- Due to the implementation specifics of the `mpz_t` data type used for all private variables in translated programs, functions cannot return variables of type `mpz_t`. For that reason, all user-declared functions with private return values should be modified to include an extra argument passed by reference which corresponds to the return value of the function and the return type of the function should be set to void.