PICCO Programmer's Guide

PICCO Team

September 22, 2025

Contents

1 PICCO Overview

PICCO is a suite of tools that transforms a program written in an extension of conventional programming language into a secure computation protocol and comes with supplemental libraries and tools. It supports all features of a general-purpose language and produces an interactive program to be executed by n computational nodes in the semi-honest setting (where the parties are expected to follow the computation, i.e., don't modify the program). The currently supported protocols assume the honest majority, i.e., $t < n/2$ corrupt parties and it is assumed that $n = 2t + 1$ for $t \geq 1$. Supported techniques include Shamir secret sharing and replicated secret sharing over a ring $\mathbb{Z}_{2^k}$.

The instructions for installing PICCO, compiling and executing user programs can be found at `https://github.com/applied-crypto-lab/picco`, while this guide focuses on program specification.

User programs are mixed-mode programs that combine *public* (i.e., observable during the execution) variables and constants and *private* variables, which are protected (available in the form of secret shares during the execution) and reveal no information about their true values.

Portions of the code that operate solely on public values are not modified by the compiler. Computation on private values, on the other hand, is transformed into a secure multi-party computation protocol by combining secure protocols for individual operations using a protocol library that comes with the compiler. If an operation combines public and private values, the result is private. Data flow from private to public values is not allowed since in that case it is not possible to generate a secure protocol. All other data flow types (public to private, public to public, and private to private) can be used.

Most of the operations are transformed into interactive secure protocols. This means that evaluating that operation involves communication between the parties and incurs network delay. Because of that, a crucial factor that impacts performance is whether or not certain operations can be combined to be performed at the same time, as to lower the number of interaction rounds. For that reason, PICCO provides provisions to combine multiple operations together or execute different portions of the program in parallel. For example, it is possible to specify operations at the level of vectors (arrays), where communication associated with multiple operations of the same type will be performed in a batch. That is, local computation associated with the operations is performed sequentially, while communication is combined and transmitted in a batch. This may not decrease the volume of communication, but rather substantially lowers the number of interaction rounds.

Another significant difference from conventional program execution is that the (private) inputs are likely coming from sources different from the nodes running the computation. This means that they need to be prepared, secret-shared, and communicated to the computational nodes before the computation starts. This places certain restrictions on the inputs, e.g., their size cannot be computed during program execution but rather needs to be known before the computation starts. PICCO has special provisions for I/O handling as described later in this document, which allows the computational nodes to load a variable from the shares they received. To achieve that, all data owners (or input parties) are numbered and the interface specifies the origin of an input variable. The same holds for the output: all output recipients are numbered and the program can be written to create shares of an output variable for one or more output recipients.

2 Data Types

PICCO supports conventional data types, which we detail in this section.

2.1 Simple Types

The same numeric data types as in conventional C are supported. Each variable needs to be declared as either public or private using the corresponding qualifier. For example, variables are declared as

```
private int a;
```

or

```
public float b;
```

If the qualifier is omitted, as in

```
int a;
```

the code conservatively defaults to a private variable. This is to ensure that we protect data by default, and removing the protection requires an explicit programmer action.

Modern CPUs are capable of executing a single numeric operation on integers or floating-point real numbers in a single CPU cycle. This is very different for secure protocols on private numbers, where the cost (communication and even the number of rounds) depend on the bitlength of the numbers. For that reason, declaring variables of custom bitlength, just large enough to hold the values the computation handles, can have a significant impact on the program's runtime.

To permit programmers to utilize this optimization, PICCO allows private variables to be declared of arbitrary bitlengths. This is done using the angle bracket notation with an integer bitlength. For example, we can declare a 20-bit private integer as

```
private int<20> a;
```

Then a Boolean value can be declared as a 1-bit integer, a character (or a byte) can be declared as an 8-bit integer, etc. If the bitlength is omitted, the integer defaults to 32 bits.

The following shortcuts can be used:

- `char` – 8-bit integer
- `short` – 16-bit integer
- `long` – 64-bit integer

All integers are signed, and currently `signed` and `unsigned` quantifiers cannot be used.

Floating-point values have two different bitlengths: for the mantissa and the exponent, respectively. For that reason, floating-point variables are declared using two bitlengths inside the angle bracket notation as in

```
private float<40, 12> b;
```

The first number specifies the bitlength of the mantissa (including the leading 1) and the second number specifies the bitlength of the exponent. Omitting the sizes defaults to `float<32, 9>`, which is between a single and double floating-point precisions.

2.2 Composite Types

In addition to simple data types, PICCO permits declaration and use of composite data types such as structs. The syntax is the same as in C except that we include a private/public qualifier with each field. For example, we can define

```
struct node {
  private int key;
  private int data;
  public int index;
};

struct node a;
```

2.3 Arrays

Arrays that operate only over public values are declared and used in the same way as in conventional C. Arrays consisting of private elements can currently only be one-dimensional or two-dimensional. Otherwise, declaration proceeds as in C, by specifying the array size or sizes using constants, variables, or expressions. For example,

```
public int size = 100;

private int A[2][size];

private float B[2*size];
```

Note that memory allocation is a publicly observable event which requires that the size of any array is public. If the amount of memory the program will use in the form of an array can depend on a private value, the array should be allocated to the size that will be sufficient regardless of what value the private data takes.

2.4 Pointers

Pointers to both public and private data are supported. Pointers to private data are declared and used similar to conventional pointers, but what happens under the hood can substantially differ. Examples of pointer declarations that use private values are:

```
private int *p;

struct node {
   int data;
   struct node *next;
};
struct node *head;
```

In these examples, if a pointer is consequently initialized to point to a specific memory location, there is no fundamental difference between working with pointers to public or private values (i.e., operations on the values will differ, but not pointer operation).

However, if a pointer is for a data type consisting of only private fields, it is also possible for the pointer's location itself to become private. That is, instead of pointing to a known location, a pointer can inquire multiple locations, in which case it is not known what the true location of the pointer is. Then, for example, dereferencing the pointer does not simply retrieve the value from the pointer's location. Instead, we need to read the (private) values at all locations associated with the pointer and privately select one of them marked as the true location. The largest implication of the above is that the cost of some operations significantly differs now. That is, instead of a quick memory read, dereferencing a pointer with a private location requires interactive communication to select the value at the true location. This will be further discussed in Section 3.

3 Language Constructs

This section discusses various language constructs and how working with private values affects their usage in a program and what impact this has on the resulting program.

3.1 Declaration, Initialization, and Assignment

Data types and variable declaration were discussed in the previous section. We only note that qualifiers such as `const` and `static` cannot be currently applied to private variables.

Private data is typically initialized by reading its value from the input (i.e., reading secret shares communicated by the input owners prior to program execution). This will be discussed in Section 4. However, initializing private variables to other values can be done as well in a conventional way. For example,

```
private int a = 0;

public int b = 0;
private int<8> c = b;
```

Similarly, arrays of private variables are expected to be read from the input. Static assignment of arrays as in

```
int<1> a[3] = {0, 0, 1};
```

is supported as well (and can be useful, e.g., during program development).

Any variable assignment must preserve the security requirement that information cannot flow from private to public variables (as this would lead to information disclosure about private values). This means that the assignment of kind

```
public int a = b;
```

where `b` is a private variable will be blocked by the compiler. All other types of assignments – public-to-public, private-to-private, and public-to-private – are permitted.

Similar to C, PICCO invokes type conversion at the time of assignment for numeric data types. For example, executing the code

```
private float a = 4.1;
private int b = a;
```

will result in variable `b` storing the (integer) value 4. The conversion procedure between private floating-point and integer values is, however, expensive and it is best to use type conversion sparingly.

Pointer assignment uses standard syntax such as

```
p = ptr;
p = &a;
p = ptr + 3;
```

where `p` and `ptr` are pointers of the same type, `a` is anon-pointer variable, and `ptr` has memory allocated for at least 4 elements. In expressions of the kind

```
p = ptr + exp;
p = &ptr[exp];
```

The expression `exp` must evaluate to a public integer.

Assignments that access `struct` elements or perform pointer dereferencing also use conventional syntax:

```
p->x = 3;
t.x = b;
```

3.2 Dynamic Memory Allocation and Deallocation

Memory allocation for arrays consisting of private values is as was specified above in Section 2.3. Dynamic memory allocation for pointers that point to private data uses a special interface, `pmalloc`. The internal representation of private data types is hidden from the programmer and does not follow the standard compiler types. For that reason, it is not sufficient to simply specify the size of the data type in bytes when allocating memory and our interface takes both the data types and the number of items of that data type to allocate. For example,

```
private int *p = pmalloc(size, private int);
private float *ptr = pmalloc(100, private float);
```

Memory deallocation uses a similar interface, `pfree`, with the same syntax as that of `free`:

```
pfree(ptr);
```

3.3 Operators and Operations

Arithmetic operations are performed on private variables or on a mix of private and public variables in the same way as in conventional programs. The arithmetic operators are

```
+, -, *, /
```

When dividing by a private variable as in `a/b`, the program behavior must be the same regardless of the value of the divisor `b`. This means that no error will be triggered if the value of `b` happens to be 0. Thus, it becomes the programmer's responsibility to check for such cases if it is possible to the divisor to take on a 0 value.

The operations are supported for integer and floating-point values. However, when using private integer and floating-point values in a single operation, the compiler will throw an error as it is not clear whether the programmer wants to proceed with integer or floating-point arithmetic (either can introduce a numerical error depending on the values). In that case, the program should be modified to use casting to force the operands to be of the same type.

The same *relational operators* as in regular C are supported:

```
<, <=, >, >=, ==, !=
```

which produce a single-bit output. Similar to the above, some ambiguity arises when comparing an integer and an floating-point value for equality. For that reason, it is expected that the user will remove the ambiguity by converting one of the values to the desired type and comparing the values using the same representation. In the current implementation, the compiler will trigger an error when comparing an integer and a floating-point value for equality, where at least one of the operands is private.

Logical operators

```
&&, ||, !
```

can only be applied to private 1-bit integers.

Shift operators

```
<<, >>
```

expect the operands to be an integer. Shifts by both a private amount and a public amount are supported.

Other bitwise operations

`&, |, ^`

are currently supported only on 1-bit private integers, while ~ is not currently implemented.

Unary operators ++ and −− can be used with private integers and the remainder operation `%` can be used with private integers as well. Assignment shortcuts such as +=, −=, ⋆=, /=, and %= are currently not implemented.

3.4 Type Casting

Numeric values can be cast from one type to another. For private variables, values can be converted between integer and floating-point types and between values of the same type, but different bitlength. For example, in the code

```
float a;
int<8> b;
int c = (int)a;
b = (int<8>)c;
```

both types of conversion are used: floating-point `a` is converted to an integer and 32-bit integer `c` is converted to an 8-bit integer.

In addition to converting a single value, an array of values can be converted to a different type by casting a pointer to be a pointer an array of values of a different type. Similar to C, instead of simply casting the memory content to be treated as a value of a different type, we perform conversion of the value itself and the actual value is preserved to the extend permissible by the new representation.

3.5 Array Operations

In addition to accessing an element of an array at a public location, as in `A[i]`, where `i` is a public integer and `A` is an array of public or private values, one might now want to access an element of an array at a private location. The syntax for the private variant is the same as before:

```
b = A[a];
B[a] = c;
```

where `a` is a private integer. Variable `b` is required to private as it depends on private `a`, while `A` can be an array of public or private values. Similarly, writing `c` to a private location of `B` requires that the elements of `B` are private, while `c` can be public or private.

Because it is not know which location is to be read or updated, the secure computation accesses all of the array locations. For that reason, the size of the array must be known. This is easily achieved for properly allocated arrays and pointers, the memory for which is allocated using the `pmalloc` interface. Note that if the private index `a` is outside the boundaries of the array, reading the element at position `a` will return 0. In addition, writing any value to the element at position `a` outside of the valid range will result in no element being updated.

Another important topic is operations at the level of arrays. Most operations in translated programs correspond to interactive secure protocols. For that reason, it is important to reduce the number of sequential protocol invocations, or rounds. One easy way to reduce the number of rounds is to perform a number of identical operations at the same time in a batch. One mechanism that PICCO uses for this purpose is batching at the level of arrays. For that reason, many operations can be performed at the level of arrays. For example, if we declare `A` and `B` to be arrays of the same size, the following operations can be executed

```
C = A * B;
C = A / B;
D = A > B;
```

Each operation will be executed element-wise. For example, in the case of `D = A > B`, each element of array `A` will be compared to the corresponding element of `B` and the result will be stored in the same element of `D`. The functionality inherits the implementation and any limitations from the corresponding operations on single values.

Operations between an array and a singe values, as in, e.g.,

```
C = 5 * A;
```

are not supported. Also, the shortcuts `A++` and `A--` currently cannot be used.

One important special operation at the level of arrays is the dot product. It has a corresponding secure protocol of much lower cost than individual invocation of addition and multiplication operations. To enable the use of that protocol in programs, there is a special operator, `@`, for the dot product operation. For example, one might execute

```
int c = A @ B;
```

where, as before, the arrays `A` and `B` need to be of the same size. The dot product operation is supported only for array of integers.

3.6 Private Value Declassification

Recall that data flow from private to public variables is not permitted. If there are special circumstances when a private intermediate value is to be declassified, there is a special mechanism for doing so that invokes the `smcopen` functionality. In particular, one can execute

```
c = smcopen(a);
```

where `a` and `c` are variables of the same type, with `a` being private and `c` being public.

3.7 Conditional Statements

Conditional statements with public conditions proceed as before. Private conditional statements are also supported, but their execution fundamentally differs from that of statements with public conditions. In particular, in order to protect information about the condition, the execution cannot reveal which branch is executed (or whether the only branch is executed if there is no the `else` clause). For that reason, all branches are always executed, but the result of only the true branch is applied. For example, if the program has

```
if (a < 0) {
  b = b + c;
  a += -1;
}
else
  b = b - c;
```

with private `a`, `b`, and `c`, the instructions associated with both branches are executed, but the result of only one of them is privately applied. Then is `a` is negative, both `b` and `a` are updated according to the `if` branch.

If a is not negative, both a and b are updated, but only the value of b changes according to the computation in the else branch.

It is important to note that the body of any conditional statement with a private condition cannot have public side effects. Otherwise, such public side effects can reveal information about the private condition. Side effects are changes visible after the execution of the conditional statement and the most common case of public side effects is modifying a public variable. It also includes declarations and memory allocation. If a function is called from the body of a conditional statement with a public condition, the compiler will verify that the function does not have public side effects.

3.8 Loops

All types of loops (for, while, do-while) are supported. Because the number of loop iterations is an observable event, the loop terminating condition must be available as a public value at runtime.

This means that loops can have public terminating conditions, as is typical for for loops that iterate over each element in a data structure of a known size. If it also possible to have a loop that evaluates an expression over private variables to determine if the terminating condition is met, discloses the resulting Boolean decision by means of smcopen, and uses the resulting public information to either continue or terminate the loop. In circumstances when it is undesirable to disclose the terminating condition, a loop can be written to iterate the number of times up to a publicly known bound. The content of the loop can stop applying the updates once the (private) terminating condition is met by using if-else constructs with a private condition.

Loops are also an important mechanism for reducing the round complexity of secure computation execution. There can be many circumstances when loop iterations are independent of each other and thus can be executed in a batch, significantly aiding in reducing the runtime of secure protocols. PICCO has provisions for specifying loops, all iterations of which can be executed at the same time. In addition to executing sequential loop iterations as in

```
for (i = 0; i < k; i++) {
  if (a[i] < 0)
    a[i] = -1*a[i];
  a[k+i] = a[i]*b[i];
}
```

one can use the parallel loop construct for(;;)[ ] to execute all iterations at the same time:

```
for (i = 0; i < k; i++) [
  if (a[i] < 0)
    a[i] = -1*a[i];
  a[k+i] = a[i]*b[i];
]
```

Compared to the original loop that executes all comparisons and multiplications sequentially, the modified loop executes a batch of k comparisons followed by multiplication batches.

It is important to note that currently an expression being evaluated in the body of a batch loop can consist of only one operation at a time (e.g., an arithmetic operation, comparison, etc). 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.

3.9 Functions

Functions largely follow the conventional C syntax. One exception is that, due to the specifics of the internal representation of private data types in translated programs, the current implementation does allow a function to return a private value. One way to get around this limitation is to pass the address of a variable as one of the arguments to the function and have the function write the result into that variable. Another way is to store the result in a global variable.

Only functions that do not have any public side effects can be called from within the body of a conditional statement with a private condition.

3.10 Parallelization

In addition to the two previously described mechanisms for reducing the number of communication rounds – batching array operations and batching operations inside loops – there is a third mechanism for executing operations in parallel. The last category permits execution of arbitrary portions of code in parallel by means of different threads. The syntax for using this option is;

```
[ ops; ]
[ ops; ]
  ...
[ ops; ]
ops;
```

With this option, the compiler will create a number of threads equal to the number of `[ ]` clauses, each containing a sequence of arbitrary instruction between the square brackets. The threads run in parallel, and once the longest running thread terminates, the execution proceeds single-threaded with other instructions in the program.

Note that there is overhead associated with thread handling, and this option does not always result in improved performance.

With recent changes to the compiler to expand the set of supported techniques, parallel execution is presently unlikely to work.

4 I/O Handling

Input and output handling is performed through a special interface. All input may be supplied by participants who differ from the nodes running the computation and the input needs to be prepared prior to computation initiation. This means that each input owner will split all of it input into shares and provide each share to a different computational party as described in PICCO README.

As far as program content is concerned, entering input from a given input party is performed by means of `smcinput`. The interface takes the ID of the input party contributing the input, all input parties should be numbered sequentially from 1. The number of input parties is also specified in `smc-config` at program compilation time.

To read a single integer or real number supplied by input party 1 into variable `a`, one would specify the following:

```
smcinput(a, 1);
```

The interface is designed for entering private values into the computation, but can be used for public values as well (which are expected to be constrained to information such as the size of private input read afterwards).

The smcinput interface can also be used to read multiple values into a one- or two-dimensional array. To do so, the size of the input to read must be specified. For example, in the code below

```
private int A[size1];
private float B[size1][size2];

smcinput(A, 1, size1);
smcinput(B, 2, size1, size2);
```

specifies to read `size1` private integer values into one-dimensional array `A`, with the input coming from input party 1, and `size1*size2` private real values into two-dimensional array `B`, with the input being contributed by input party 2. With two-dimensional arrays, the last two arguments specify the number of rows and columns, respectively.

Note that PICCO currently does not support private arrays of dimensionality higher than 2, and `smcinput` can only be called on one- or two-dimensional arrays.

A similar interface, called `smcoutput`, is used to produce output to a specific output party. The output parties similarly need to be sequentially ordered by their IDs and can be different from the input parties. The number of output parties is specified in `smc-config`. For example, to have output parties 1 and 2 receive the content of private variable `a`, we write

```
smcoutput(a, 1);
smcoutput(a, 2);
```

Receiving multiple values in the form of arrays uses the same syntax as `smcinput`.

Both `smcinput` and `smcoutput` can be called on private and public variables at any point in the program, but the current implementation has some limitations. In particular, `smcinput` and `smcoutput` cannot be called from the body of a loop or iteration (such as a repeatedly called function) or branching statement (e.g., if-else block). (Only a single call will be captured in that case.) In addition, 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.