

Lessons from the Design and Experimentation with the PICCO Compiler

Marina Blanton
University at Buffalo

SystemPC 2025

The PICCO Compiler

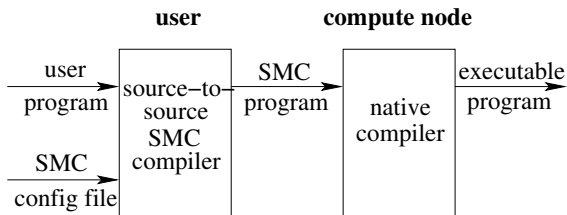
PICCO is one of early compilers for transforming general-purpose programs into MPC protocols

- it was published in 2013 and expanded the capabilities of existing compilers
- it was the first to support operations on all data types (including floating-point arithmetic)
- it was the first to simultaneously support unrestricted loops and branching statements

The PICCO Compiler

PICCO was designed to support **mixed-mode execution** using Shamir secret sharing to protect private data

- variables are annotated as **public** or **private**
- a **configuration file** specifies secret sharing and other parameters
- compilation is a **two-step process**



PICCO's Design

PICCO's design was to balance performance

- the middle ground between sequential execution and running each operation in a separate thread

It has **three mechanisms for parallelizing computation**

- batch execution at the level of arrays, e.g.,

```
C = A * B;
```

- batch execution of parallelizable loops

```
for (i = 0; i < n; i++) [  
    b[i] = a[i] * a[i+n];  
]
```

PICCO's Design

PICCO has **three mechanisms** for parallelizing computation

- parallel execution of arbitrary portions of code

```
[
  code segment 1;
]
[
  code segment 2;
]
code segment 3;
```

PICCO's Design

Compared to more recent automated parallelization techniques, this

- offers higher transparency
- places higher burden on the programmer

PICCO's Design

Compared to more recent automated parallelization techniques, this

- offers higher transparency
- places higher burden on the programmer

There are other optimization aspects such as

- a dot-product operator @
- variable-length variables as in `int<12> a;`

Handling Pointers to Private Data

Pointer support was consequently added (ACM TOPS'18)

- pointers are fundamental to building data structures for efficient data manipulation
- dynamic data allocation and deallocation can be also accomplished using pointers

Adding pointers to private data involved new design choices

- a simple reference to a private object is not enough
- the location a pointer points to can be private

Handling Pointers

A simple example:

```
private int *p;  
if (cond)  
    p = &a;  
else  
    p = &b;
```

Design decisions: [memory pools](#) vs a [collection of locations](#)

Handling Pointers

Our solution

- a pointer's location is initially public
- if more than one location is acquired, the location becomes private

We obtain that

- when working with sorted data, pointers acquire all locations
- offering custom protocols for data structures with specific interfaces is preferred

Further PICCO Work

PICCO's security properties were shown using formal methods by Rathore et al. in “A Formal Model for Secure Multiparty Computation”

- correctness
- non-interference

Protocols based on replicated secret sharing are being added

Effective Uses of PICCO

PICCO uses vary

- research measurements
- starting point for custom implementations of complex functionalities
- quick prototype for simple application deployment
- teaching MPC to students

Challenges

The two-step compilation process makes debugging harder

- an error message thrown during the second compilation step might be about functions the programmer did not write

Challenges

The two-step compilation process makes debugging harder

- an error message thrown during the second compilation step might be about functions the programmer did not write

The more our syntax differs from that of C, the more comprehensive error checking in the first step needs to be

- this increases the amount of work relative to using existing tools

Challenges

Data types in the original program differ from that in the translated implementation

- this prevented some functionality from working as expected
- one example is returning values from a function call
`private int func(private int a)`
- pointers vs arrays representation differences challenges
also creates challenges with function calls
`void func(private int a, private int *ret)`

Hangling Input

Inputs (private and otherwise) are collected through a special interface `smcinput`

- a programmer specifies the variable to read into and which input party contributes that input

A fundamental difference from conventional execution is that **inputs are prepared prior to program execution**

- inputs are assembled and secret shared before the computation starts

Handling Input

Correctly extracting input functionality prior to executing the program is yet another challenge

- our implementation makes a pass through the program
- this constraints the programmer
- a more precise way of collecting this information is executing the public portion of the program at compilation time

Future Directions

Self-optimizing compilers

- choosing an optimal variant from the available options
 - simple example: the use of edaBits
- rewriting user's code to improve parallelism
 - classical example: dot product or matrix multiplication

```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i]*b[i];
```

Intuitive programming interfaces