Privacy Engineering Coursework

Joon-Ho Son <js6317>
William Burr <wb2117>

November 2020

1 Getting Started

The GitLab repository can be found at `https://gitlab.doc.ic.ac.uk/js6317/bgw/`.

On CSG machines, Go 1.15 is available at `/vol/linux/apps/go/bin/go`. If, for some reason, this doesn't work then Go 1.14.3 should be installed at `/usr/lib/go-1.14/bin/go`. Below we will assume the `go` binary is on your path already. Run with

```
go run cmd/mpc/mpc.go
```

We recommend sorting the output for readability. To do this, run:

```
go run cmd/mpc/mpc.go | sort
```

If the protocol finishes successfully, you should see that the program finishes with something like:

```
[...]
MPC: 17:59:37.362092 Expected output: 7
MPC: 17:59:37.362104 Actual output:   7
MPC: 17:59:37.362112 Protocol succeeded (:
```

The protocol can be configured using command line arguments. For example:

```
go run cmd/mpc/mpc.go -circuit 5 -degree 2 -prime 1003 -seed 4
```

The various circuit definitions can be found in `pkg/config/config.go`. The full usage is detailed below:

```
Usage of mpc:
  -circuit int
        Circuit to run. (default 1)
  -degree int
        Degree of polynomial. If unset, it is set to (N-1)/2
  -prime int
        Prime number to use for modular arithmetic. (default 101)
  -seed int
        Seed for pseudorandom number generation. If unset, the current time is used.
```

2 Details and Implementation

2.1 Logging

Upon running the program and piping the output into `sort`, the log output for each party should look something like this:

```
MPC: 18:50:03.988454 Circuit Configuration
MPC: 18:50:03.988464 ===================================
MPC: 18:50:03.988481   Circuit number:    5
MPC: 18:50:03.988497   Number of parties: 6
MPC: 18:50:03.988525   Secrets:           [20 40 21 31 1 71]
MPC: 18:50:03.988539   Polynomial degree: 2
[...]
005: 18:50:03.989304 Running party 5 with secret 71
005: 18:50:03.989451 ===================================
005: 18:50:03.989623   [0 | IN0 ] received share 21 from party 0
005: 18:50:03.989673   [1 | IN1 ] received share 54 from party 1
005: 18:50:03.989719     [2 | MUL ] 21 × 54 mod 101 = 23
005: 18:50:03.989775     [2 | MUL ] using polynomial 23 + 11x^1 + 39x^2
```

```
005: 18:50:03.989812      [2 | MUL ] sent shares [73 100 3 85 43 79]
005: 18:50:03.989933      [2 | MUL ] received shares [21 63 29 38 68 79]
005: 18:50:03.990330      [2 | MUL ] (21 × 6) + (63 × -15) + ...
005: 18:50:03.990362    [3 | IN2 ] received share 93 from party 2
005: 18:50:03.990395    [4 | IN3 ] received share 68 from party 3
005: 18:50:03.990761      [5 | MUL ] 93 × 68 mod 101 = 62
005: 18:50:03.990807      [5 | MUL ] using polynomial 62 + 60x^1 + 84x^2
005: 18:50:03.990843      [5 | MUL ] sent shares [4 13 89 30 38 12]
005: 18:50:03.990878      [5 | MUL ] received shares [48 27 96 96 92 12]
005: 18:50:03.991023      [5 | MUL ] (48 × 6) + (27 × -15) + ...
005: 18:50:03.991179        [6 | ADD ] 25 + 95 mod 101 = 19
005: 18:50:03.991492    [7 | IN4 ] received share 2 from party 4
005: 18:50:03.991582    [8 | IN5 ] using polynomial 71 + 85x^1 + 74x^2
005: 18:50:03.991626    [8 | IN5 ] sent shares [28 32 83 80 23 13]
005: 18:50:03.991660      [9 | MUL ] 2 × 13 mod 101 = 26
005: 18:50:03.991702      [9 | MUL ] using polynomial 26 + 3x^1 + 92x^2
005: 18:50:03.991871      [9 | MUL ] sent shares [20 97 55 96 18 23]
005: 18:50:03.992037      [9 | MUL ] received shares [72 0 98 47 98 23]
005: 18:50:03.992179      [9 | MUL ] (72 × 6) + (0 × -15) + ...
005: 18:50:03.992295        [10| ADD ] 19 + 30 mod 101 = 49
005: 18:50:03.992330          [10| OUT ] sent shares [49 49 49 49 49 49]
005: 18:50:03.992376          [10| OUT ] received shares [23 96 24 9 51 49]
005: 18:50:03.992601          [10| OUT ] (23 × 6) + (96 × -15) + ...
005: 18:50:03.992614    Party 5 finished with output 7
[...]
```

As above, log lines for each party's computation follows the format:

```
<PARTY_NUMBER>: <TIMESTAMP>  [<GATE_NUMBER> | <GATE_TYPE>] <LOG_MESSAGE>
```

Gates are indented according to their level in the tree.

2.2 Party Communication

The main protocol is implemented in `party.Run`. It first traverses the circuit "tree" and processes each gate in turn. After computing the output of a gate, its `Output` value is set so that other gates that depend on it can access its value. The traversal is done in post-order so that a gate's dependencies are always available.

Parties communicate via their `msg` channel, through which *all* inter-party communication is done. Each party is initialised with a copy of the circuit, to prevent any accidental shared memory.

Parties IDs are indexed from 0, although they are indexed from 1 for the purpose of calculations (e.g. computing the recombination vector).

2.3 Circuit Definition

Circuits are represented using the struct `circuit.Circuit`. They are defined using a tree-like structure, with `gate.Gate`s as nodes.

```
&circuit.Circuit{
    NParties: 6,
    Root: gate.NewAdd(
        gate.NewAdd(
            gate.NewMul(
                &gate.Input{Party: 0},
                &gate.Input{Party: 1},
            ),
            gate.NewMul(
                &gate.Input{Party: 2},
                &gate.Input{Party: 3},
            ),
        ),
        gate.NewMul(
            &gate.Input{Party: 4},
            &gate.Input{Party: 5},
        ),
    ),
}
```

Circuits that have multiple inputs into the circuit are supported. For example:

```
&circuit.Circuit{
    NParties: 2,
```

```
    Root: gate.NewMul(
        gate.NewMul(
            &gate.Input{Party: 0},
            &gate.Input{Party: 1},
        ),
        &gate.Input{Party: 0},
    ),
}
```

See `pkg/config/config.go` for the full list of hardcoded circuit configurations.

2.4 Packages

2.4.1 `main`

This package is the main entry point into the program. It parses command-line arguments, selects a configuration, and spawns the appropriate number of parties, each in its own Goroutine.

2.4.2 `circuit`

This package contains the representation of a circuit. A circuit is defined by a `Root` node (containing child nodes) and `NParties`. It also provides methods to evaluate the actual value of the expression, which is used to determine the correctness of the protocol output.

2.4.3 `config`

This package contains various configurations that can be selected via the command line. Configurations 1 and 2 correspond to the circuits provided in the original skeleton code.

2.4.4 `field`

This package implements a representation of a finite field, and relevant modular arithmetic operations. We chose not to use `big.Int` for simplicity, opting for the standard `int` type. Instead, all modular arithmetic functions are implemented in this package. This does limit the size of input/prime that can be used, however, this should not pose a major issue as long as a reasonably-sized prime is used.

2.4.5 `gate`

This package contains the implementation of circuit gates, namely `Add`, `Mul`, and `Input`. They all implement the `Gate` interface.

2.4.6 `party`

This package contains the main functionality for the protocol. Each party evaluates each gate in its copy of the circuit in turn:

```
for gIdx, g := range gates {
        switch v := g.(type) {
        case *gate.Input:
                p.processInput(gIdx, v)
        case *gate.Add:
                p.processAdd(gIdx, v)
        case *gate.Mul:
                p.processMul(gIdx, v)
        }
}
```

Once the primary evaluation loop finishes, the party performs output reconstruction with the shares from all other parties.

2.4.7 `poly`

This package implements functionality for creating and manipulating a basic representation of polynomials.

3 Evaluation

Each of the following circuits below are represented as configs in `config.go`.

3.1 Circuit 1 (*Smart* example)

Circuit 1, the example in *Smart*, is represented as `config1` and can be run with the same configuration used in `circuit.py` as follows:

```
go run cmd/mpc/mpc.go -circuit 1 -degree 2 -prime 101
```

The final result is given by the logs:

```
MPC: 00:35:49.856857 Expected output: 7
MPC: 00:35:49.856859 Actual output:   7
MPC: 00:35:49.856861 Protocol succeeded (:
```

3.2 Circuit 2 (Factorial Tree)

Circuit 2 for generating a factorial is represented as `config2`, and can be run as follows:

```
go run cmd/mpc/mpc.go -circuit 2 -degree 2 -prime 100_003
```

The final result is given below:

```
MPC: 00:47:45.866532 Expected output: 40320
MPC: 00:47:45.866533 Actual output:   40320
MPC: 00:47:45.866535 Protocol succeeded (:
```

3.3 Circuit 3 (Nth Fibonacci Number)

For our own circuit, we wrote a method for generating a circuit that computes the Nth Fibonacci number where $N > 1$. The generated circuit always uses two `Input` gates for the first two Fibonacci numbers (0 and 1) and generates the required number of `Add` gates to compute the Nth Fibonacci.

Since Fibonacci sequences are recursively defined with overlapping sub-problems, we are able to leverage our design which allows for the output of a gate to be used in multiple follow-up gates.

When N=10, the generated circuit can be run as follows:

```
go run cmd/mpc/mpc.go -circuit 3 -prime 1_000_000_007
```

The final result is given below:

```
MPC: 01:57:31.181590 Expected output: 55
MPC: 01:57:31.181592 Actual output:   55
MPC: 01:57:31.181594 Protocol succeeded (:
```