Playing games via secure two-party computation

Computer Security and Cryptography – Final homework

Luca Di Stefano
Gran Sasso Science Institute (GSSI), L'Aquila, Italy
luca.distefano@gssi.it

Abstract. Secure two-party computation (2PC) allows two individuals to compute a function $f(X, Y)$ on their respective private data, X and Y, without revealing them to the other party.

Many two-player games require the strategies of each player to be revealed at the same time. 2PC protocols make it possible to play such games over a network without the need for a third party.

We review how a basic Garbled Circuit protocol works and we use a toolchain of free software to implement and play one such game on the TinyGarble implementation.

1 Introduction

Yao's protocol [1] is a well-known solution to the secure two-party computation (2PC) problem.

The paper proves that the protocol can be applied to any 0-1 valued function f, and that the time to execute the protocol is proportional to the size of a boolean circuit that computes f.

Practical implementations of 2PC protocols are becoming more and more efficient thanks to algorithmic breakthroughs such as Free XOR [2] and the use of fixed-key block ciphers instead of cryptographic hash functions [3].

We demonstrate how 2PC can be used to solve the simple task of playing a two-player game over a network: to achieve this goal, we will use a toolchain of free and open-source software.

This work is structured as follows: Section 2 provides an overview of Garbled Circuit protocols. Section 3 describes a two-player game that will be used as a case study.

Section 4 describes the tools used to realize a boolean circuit that implements the rules of the game, while in Section 5 we show how to execute the actual two-party computation.

2 Overview of Garbled Circuit protocols

The concept of Garbled Circuits (GCs) is central to current implementations of Yao's protocol, such as Fairplay [4].

In these frameworks, the function f is described by a publicly-known boolean circuit C_f. The two parties in the computation (say, Alice and Bob) are respectively known as the *garbler* and the *evaluator*.

Let us first describe the garbling procedure for a single gate $g(A, B)$ that has two input wires A, B and one output wire Z. Assume that each wire carries one bit.

Let Enc_k be a symmetric encryption function with key k, and let $X \oplus Y$ be the concatenation of binary strings X, Y.

1. Choose two random strings L_0^A, L_1^A that encode 0 and 1 on wire A. These strings will be called *labels*.
2. Do the same for wires B and Z, with different labels $L_0^B, L_1^B, L_0^Z, L_1^Z$.
3. Construct a truth table: for each i, j, k such that $g(i, j) = k$, store $Enc_{L_i^A \oplus L_j^B}(L_k^Z)$.
4. Shuffle (*garble*) the rows of the truth table.

A GC protocol can now be summarized as follows:

1. Alice garbles (encrypts) C_f into a GC, C'_f. She does that by garbling all gates in the circuit as described above.
2. Alice sends C'_f to Bob, along with the labels corresponding to her input X. Notice that the labels do not reveal any information about X.
3. Bob also obtains the labels corresponding to his input, Y, from Alice. They use Oblivious Transfer (OT) so that Y is also kept secret.
4. Bob uses the labels to evaluate each garbled gate. He recurses on the other gates by using the output labels he obtains by evaluating the previous ones.
5. At the end, Bob finds out a set of n labels L^{OUT_i}, where n is the size of the output. Then he can either send them to Alice or ask her for the mapping of each label to a bit. Either way, the two parties can learn the output by communicating with each other.

Step 5 sounds quite tricky at first. How can we avoid cheating when one party has to communicate the outcome to the other one? However, [1] proves that the only way one can cheat without being "caught" is by using a fake secret input from the start of the computation.

3 Case study: Rock-Paper-Scissors-Spock-Lizard

Rock-Paper-Scissors-Spock-Lizard (RPSSL) was originally developed by Sam Kass and Karen Bryla,[1] and popularized by the sitcom "The Big Bang Theory". [5]

It is a two-player, zero-sum game where each player has five available strategies. In this work we encode them as a set $S = \{0, 1, \ldots, 4\}$, where 0 = "Rock", 1 = "Paper", ..., 4 = "Lizard".

The players reveal their chosen strategies at the same time and verify their payoff following Table 1.

Table 1: Payoff matrix for the RPSSL game.

	Rock	**Paper**	**Scissors**	**Spock**	**Lizard**
Rock	0,0	-1,1	1,-1	-1,1	1,-1
Paper	1,-1	0,0	-1,1	1,-1	-1,1
Scissors	-1, 1	1,-1	0,0	-1,1	1,-1
Spock	1, -1	-1,1	1,-1	0,0	-1,1
Lizard	-1,1	1,-1	-1,1	1,-1	0,0

Playing a game of RPS over a network, in the absence of a trusted third party, is not trivial: there is no guarantee that both strategies are revealed at the same time, so one of the player could just wait until the other reveals her move and choose one of the winning strategies. We can solve this problem by formulating the game as a 2PC problem: namely, we can define a function $RPSSL(X, Y)$ with two 3-bit inputs and one two-bit output. X (resp. Y) is the strategy of player 1 (resp. 2).

The output represents the payoff of player 1 as a signed 2-bit integer, with negative zero for errors:

- `00`: the game is a draw;
- `01`: Player 1 wins;
- `10`: Error (X or Y is an illegal input);
- `11`: Player 2 wins.

4 Verilog synthesis with MyHDL and Yosys

4.1 MyHDL

We first need to create a boolean circuit that computes $f(X, Y)$. This circuit has to be represented in the Verilog Hardware Description Language (HDL) for compatibility with

[1] `http://www.samkass.com/theories/RPSSL.html`

the tools we will use in Section 5.

Rather than writing Verilog code by hand, which can be difficult for developers that have little HDL knowledge, we use MyHDL[2] to generate it from a description of the logic written in Python.

MyHDL blocks are Python higher-order functions. Each block takes as input a set of input/output `intbv` objects (the class `intbv` represents "bit vectors") and returns one or more functions containing the logic of the block itself.

For instance, this block sets the `OUT` wire to 1 if `X` is a winning move against `Y` and to 0 otherwise:

```
ROCK, PAPER, SCISSORS, SPOCK, LIZARD = range(5)
def win(X, Y, OUT):
    @always_comb    # Decorator. Declares this is a combinatorial block
    def logic():
        if X == ROCK and (Y == SCISSORS or Y == LIZARD): OUT.next = 1
        elif X == PAPER and (Y == ROCK or Y == SPOCK): OUT.next = 1
        elif X == SCISSORS and (Y == PAPER or Y == LIZARD): OUT.next = 1
        elif X == SPOCK and (Y == SCISSORS or Y == ROCK): OUT.next = 1
        elif X == LIZARD and (Y == PAPER or Y == SPOCK): OUT.next = 1
        else: OUT.next = 0
    return logic
```

The outcome of the game is computed by another block, which takes the output of `win` as an input:

```
def rpssl(X, Y, XWIN, OUT):
    @always_comb
    def logic():
        if X > 4 or Y > 4:
            OUT.next = 0b10
        elif X == Y:
            OUT.next = 0b00
        elif XWIN:
            OUT.next = 0b01
        else:
            OUT.next = 0b11
    return logic
```

To wire the blocks together we need to define an additional function, which we call `top`. Notice that the GC implementation we will use in Section 5 requires standard names for the wires:[3]

- `g_input`: Garbler's input;
- `e_input`: Evaluator's input;

[2] `http://www.myhdl.org/`

[3] cf. `https://github.com/esonghori/TinyGarble/tree/master/scd`.

- `clk, rst`: Clock and reset wires (we have to add them even though they are not used);
- `o`: Output.

We will then generate the Verilog code by passing `top` to the `toVerilog` MyHDL function.

```
def top(clk, rst, g_input, e_input, o):
    OUT_win = Signal(intbv(0)[1:])
    win_instance = win(g_input, e_input, OUT_win)
    rpssl_instance = rpssl(g_input, e_input, OUT_win, o)
    return win_instance, rpssl_instance

# Verilog code generation.
# First, we init the wires with their bit width.
g_input = Signal(intbv(0)[3:])
e_input = Signal(intbv(0)[3:])
clk = Signal(intbv(0)[1:])
rst = Signal(intbv(0)[1:])
o = Signal(intbv(0)[2:])

# The result will be written to a file named top.v
toVerilog(top, clk, rst, g_input, e_input, o)
```

4.2 Yosys

The result file contains a *behavioral* description of the circuit that contains constructs such as arithmetic/logic expressions and conditional branches. We now leverage the Yosys Open Synthesis Suite[4] to translate this description to a *netlist*, i.e. a network of logic gates.

The following Yosys script reads the description of the circuit (`top.v`) and a basic cell library (`stdcells_S.v`), and synthesizes the netlist.

```
1 read_verilog top.v
2 read_verilog -lib stdcells_S.v
3 hierarchy -check -top top
4 proc; opt; fsm; opt; memory; opt;
5 techmap; opt
6 abc -liberty asic_cell_yosys_extended.lib
7 clean ;;
8 write_verilog -noattr rpssl_netlist.v
```

The commands at lines 3-4 break down behavioral Verilog elements into simpler, abstract components (“cells”).

Line 5 performs the “technology mapping” step, where cells are replaced by logic gates.

[4] `http://www.clifford.at/yosys/`

Finally, the `abc` command on Line 6 maps such gates to the architecture described by the library `asic_cell_yosys_extended.lib`.

5 Two-party computation with TinyGarble and JustGarble

JustGarble [3] is an efficient circuit-garbling library that exploits AES hardware instructions and the "Free XOR" technique described in [2]. Free XOR allows the evaluation of XOR gates by simply computing the XOR of the input labels, without the need for a decryption step.

As the name suggests, JustGarble only provides garbling primitives to an external 2PC framework: an example of such frameworks is TinyGarble [6].

In this section we will use TinyGarble to execute a 2PC between two processes running on the same machine, using the circuit we have synthesized in Section 4. All the communication happens through the TCP protocol, therefore the same procedure can be in principle carried out on the Internet.

First of all we have to translate the netlist to a format compatible with JustGarble, called Simple Circuit Description (SCD) [3]. This is done through a TinyGarble utility program:

```
# From the main TinyGarble directory
$ bin/scd/V2SCD_Main -i rpssl_netlist.v -o rpssl.scd --log2std
```

Now we can compute the function by creating two processes, the *garbler* and the *evaluator*. Notice that all inputs and outputs are in hexadecimal format.

```
# Alice (the garbler, i.e. Player one) plays "Rock"
$ bin/garbled_circuit/TinyGarble --alice --scd_file rpssl.scd --input 0
# On another terminal, Bob (the evaluator) plays "Scissors"
$ bin/garbled_circuit/TinyGarble --bob --scd_file rpssl.scd --input 2
01 # Alice wins
```

6 Other tools and source code

TinyGarble has been compiled and executed on a VirtualBox virtual machine running the Xubuntu 16.04 operating system. The behavior of the `rpssl_netlist.v` synthesized circuit has been verified with the online tool EDA Playground.[5]

The full source code related to this report is available on GitHub.[6]

[5] The circuit and the testbench program are available at `https://www.edaplayground.com/x/6DZS`. Running the testbench online requires an account; alternatively, one can download it and use the free tool Icarus Verilog (`http://iverilog.icarus.com/`) to execute it.

[6] `https://github.com/lou1306/gssi/tree/master/2pc`

7 Conclusions and future work

We quickly reviewed the basics of a Garbled Circuit protocol. We used TinyGarble, a GC implementation based on JustGarble, to play a simple two-player game over a network and in the absence of a trusted third party.

We showed that the realization of the boolean circuit that computes the desired function can be eased by a toolchain of free and open-source tools: developers can express the logic in a quite high-level language (Python 3 in this case) and obtain a circuit representation through automated steps. We believe that this possibility can lead to a wider adoption of secure two-party computation.

Future work should address whether the proposed approach can be generalized to more complex functions: for instance, those that require flip-flops or whose computation is split across multiple clock cycles.

References

[1] A. C. Yao, "Protocols for secure computations," in *23rd annual symposium on foundations of computer science (sfcs 1982)*, 1982, pp. 160–164.

[2] V. Kolesnikov and T. Schneider, "Improved Garbled Circuit: Free XOR Gates and Applications," in *Automata, languages and programming*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2008.

[3] M. Bellare, Viet Tung Hoang, S. Keelveedhi, and P. Rogaway, "Efficient Garbling from a Fixed-Key Blockcipher," in *2013 ieee symposium on security and privacy*, 2013, pp. 478–492.

[4] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella, "Fairplay - Secure Two-Party Computation System," *{USENIX} Security Symposium*, pp. 287–302, 2004.

[5] D. Goetsch and J. Glickman, "The Lizard–Spock Expansion." CBS, 2008.

[6] E. M. Songhori, S. U. Hussain, A.-R. Sadeghi, T. Schneider, and F. Koushanfar, "TinyGarble: Highly Compressed and Scalable Sequential Garbled Circuits," in *2015 ieee symposium on security and privacy*, 2015, vol. 2, pp. 411–428.