Imperial College London

Department of Computing

Secure ("Private") Function Evaluation Coursework

https://www.doc.ic.ac.uk/~nd/peng/coursework.pdf

01 The goal of this coursework is to gain a better understanding of secure multi-party computation by writing a program to perform two-party secure function evaluation using Yao's garbled-circuit protocol.

02 In particular, you will implement the method described in section 22.2 of Nigel Smart's ***Cryptography Made Simple*** for garbled circuits and the method described in section 20.3 for oblivious transfers (Bellare and Micali with optimisations from Naor and Pinkas).

03 A typical Python solution will be around 500 lines including about 200 lines of provided code.

04 You will be provided several classes and functions to help you. In particular, (i) to send messages between Alice and Bob, (ii) to carry out the core operations for oblivious transfer (generate primes, `xor` byte sequences, generate variable-length cryptographic hashes, do modular exponentiation on prime number groups), (iii) skeleton main code for Alice and Bob. You will also be provided a `Makefile`, several test circuits plus their outputs.

05 The recommended programming language for the coursework is **Python3**. You can use you own computer if you have (or if you install) a recent version of Python3. The lab machines have version 3.6.6 installed. The very latest version of Python3 is 3.7.1

06 If you really want to use a different programming language (e.g. *Python2*, *Java*, *C++*, *Go*, *Javascript*, *Scala*, *Ruby*, *Rust*, *Haskell*, *Elixir*, *OCaml*) you need to get my permission first. Be warned, coursework support is only offered for Python3. Writing in another language means writing your own support code e.g. for sending messages and doing cryptographic operations. This is not difficult if you have some familiarity with these topics and your preferred programming language has libraries to help you. If you have my permission to use another language, you will need to include full instructions on how to compile and run your program on a CSG Linux machine (or on a Mac running High Sierra or Mojave) including installation instructions if the language is not installed by default on those machines, stating the version of the compiler and runtime you used.

07 **You can work on and submit the coursework either individually or jointly with one classmate.**
Because there are many details to get right to successfully develop a solution, it is recommended that you work jointly. One way of dividing the initial workload would be for one person to work on oblivious transfer and the other to work on garbled circuit construction and evaluation. Each should help the other to debug their code and to check that it corresponds to the methods described in Smart.

GETTING STARTED

08 Download the e-book version of ***Cryptography Made Simple*** from the College library. Review relevant parts of chapter 22 and section 20.3 before starting on the implementation. If you're more technically minded, you may also wish to read the ***Fairplay*** paper[1].

09 Download, unzip and review the files in `https://www.doc.ic.ac.uk/~nd/peng/cwfiles.zip`. These include a simple `Makefile`, JSON files of test circuits, a sample output file (`outputs_given`) that you can use for `diff`-ing your output with, `util.py` which includes various functions and classes that you will find useful, and a skeleton for `main.py`. Optional and unassessed: For an extra challenge, complete the coursework without using my implementations in util.py and main.py.

10 Develop and test your program step-by-step. The protocol is detailed. It's easy to make mistakes and get in a muddle. So, don't add too much functionality in one go. Test small circuits first. Test first with no-encryption, then with some simple encryption scheme of your own, then with a secure encryption scheme using the recommended cryptographic library. Similarly skip oblivious transfers at first, then add a simple non-oblivious transfer (e.g. Bob sends a selection bit and Alice replies - no security), then finally add the secure oblivious transfer protocol described in Smart. Use the testing and debugging techniques that you've learned and version control so that you can compare versions or rollback when things go wrong.

11 Before coding think carefully about how you want to represent a garbled circuit. Use classes and dictionaries. Having a good representation will save you time later. Refactor if you're unhappy but don't aim for perfection (whatever that is).

12 Use the classes provided in `util.py` for sending messages between Alice and Bob. The classes use **ZeroMQ** (ØMQ), an elegant message-passing library that has bindings for numerous programming languages (`http://zguide.zeromq.org/page:all`). The classes will send and receive any Python data object. If you're using your own Python3 installation you should be able to install it with `pip3 install pyzmq`

13 Use the **Fernet** class from `cryptography.fernet` to encrypt truth table entries. Fernet is a "recipe" that performs all the steps needed to securely encrypt data (using AES-128, HMAC with SHA-256 for authentication, PKCS#7 for padding). Its super easy to use. See `https://cryptography.io/en/latest/fernet` for how. If you're using your own Python3 installation you should be able to install it with `pip3 install cryptography`

14 The cryptographic functionality you need for performing oblivious transfers is provided in `util.py`. Bellare-Micali OT is based on a particular Diffie-Hellman computational assumption about discrete logs in cyclic groups. Note: division, '/' in Smart is multiplication with the modular inverse of the denominator. The group is public so Alice should send it to Bob.

15 The symbolic mathematics library **SymPy** is used to generate primes and find prime factors. If you're using your own Python3 installation you should be able install it with `pip3 install sympy`

16 The functions `pickle.loads` and `pickle.dumps` can be used to serialise Python objects if you need to do so.

[1] *Fairplay - A Secure Two-Party Computation System*, https://www.cs.huji.ac.il/~noam/fairplay.pdf

OVERVIEW

17 Alice is the garbled circuit constructor, Bob the garbled circuit evaluator.

18 Your program should be runnable either as Alice (`make alice`) or as Bob (`make bob`). Typically, you'll run Bob first, then Alice in another terminal window. Note: with ZeroMQ you can run Alice before Bob! If you want to run Bob (the server) on a different computer, change `SERVER_HOST` (in `util.py`) from `localhost` to Bob's domain name or IP address and enable the incoming connection if you have a firewall running.

19 Alice reads a JSON file of a non-garbled circuit and generates the garbled circuit.

20 Alice then transfers to Bob a redacted copy with (1) the garbled tables, (2) the key plus the p-bit `xor`'ed with Alice's input bit for each of Alice's input wires, and (3) the p-bit values of the circuit's output wires. In Smart this is done in several messages. If you wish, you can send the three items as a single message. Alternatively you can send a garbled circuit with those parts of the circuit that Bob shouldn't have access to removed (you can use the function `copy.deepcopy` to copy complex Python objects).

21 Bob then perform oblivious transfers with Alice to get the key plus `xor`'ed p-bit for each of his input bits. Finally, Bob evaluates the garbled circuit and sends the results to Alice.

22 See section 22.2.3 of Smart for further details.

JSON CIRCUIT SPECIFICATIONS

23 Several JSON files are provided for testing. Additional files may be used for marking. If you devise an interesting or large circuit of your own, please submit your JSON file for it.

24 Here is an example of a JSON file for the circuit in Smart (figure 22.2, page 443):

```

{ "+": "yao garbled circuit evaluation, naranker dulay 2018",

  "circuits" : [
    { "name": "circuit from Smart, figure 22.2, page 443",
      "alice": [1, 2],
      "bob":   [3, 4],
      "out":   [7],
      "gates": [
        { "id":5, "type":"AND",  "in":[1, 3] },
        { "id":6, "type":"XOR",  "in":[2, 4] },
        { "id":7, "type":"OR",   "in":[5, 6] }
      ]
    }
  ]
}

```

25 The JSON files are mostly self-explanatory. Below are some additional notes.

26 Each JSON file can have several circuits for testing.

27 The gates allowed are: 1-input: `NOT`. 2-inputs: `OR`, `AND`, `XOR`, `NOR`, `NAND`, `XNOR`

28 Circuits will have at least 1 input wire, 1 output wire and 1 gate.

29 Wire id’s and gate id’s are integers and don't need to be consecutive, e.g. a circuit could use gate id’s 1, 10, 20. The gate id is also the id of the gate’s output.

30 The inputs of a circuit and the output of a gate can fan-out i.e. be connected to more than one gate input.

31 The outputs of lower numbered gates will always be wired to higher numbered gates and/or be defined as circuit outputs. This means that you can evaluate the circuit in gate number order i.e. you don't need to perform a *topological sort.*

32 In order to fully test a circuit, your program should perform garbled circuit evaluation for all inputs and print the results in a truth table (ascending values for the Alice-Bob combination) For the circuit above, the printed output should be:

```
circuit from Smart, figure 22.2, page 443
  Alice[1, 2] = 0 0  Bob[3, 4] = 0 0   Outputs[7] = 0
  Alice[1, 2] = 0 0  Bob[3, 4] = 0 1   Outputs[7] = 1
  Alice[1, 2] = 0 0  Bob[3, 4] = 1 0   Outputs[7] = 0
  Alice[1, 2] = 0 0  Bob[3, 4] = 1 1   Outputs[7] = 1
  Alice[1, 2] = 0 1  Bob[3, 4] = 0 0   Outputs[7] = 1
  Alice[1, 2] = 0 1  Bob[3, 4] = 0 1   Outputs[7] = 0
  Alice[1, 2] = 0 1  Bob[3, 4] = 1 0   Outputs[7] = 1
  Alice[1, 2] = 0 1  Bob[3, 4] = 1 1   Outputs[7] = 0
  Alice[1, 2] = 1 0  Bob[3, 4] = 0 0   Outputs[7] = 0
  Alice[1, 2] = 1 0  Bob[3, 4] = 0 1   Outputs[7] = 1
  Alice[1, 2] = 1 0  Bob[3, 4] = 1 0   Outputs[7] = 1
  Alice[1, 2] = 1 0  Bob[3, 4] = 1 1   Outputs[7] = 1
  Alice[1, 2] = 1 1  Bob[3, 4] = 0 0   Outputs[7] = 1
  Alice[1, 2] = 1 1  Bob[3, 4] = 0 1   Outputs[7] = 0
  Alice[1, 2] = 1 1  Bob[3, 4] = 1 0   Outputs[7] = 1
  Alice[1, 2] = 1 1  Bob[3, 4] = 1 1   Outputs[7] = 1
```

33 Alice should print the results. Strictly, Alice must not know what Bob's inputs are (since they’re private to Bob!). However, for the purposes of printing truth tables only, your code for Alice can assume that Bob will use inputs corresponding to the truth table order shown above - your oblivious transfer and garbled circuit evaluation **must not** use this knowledge however.

34 You can assume circuits are valid.

35 Optional and unassessed: Write a function to check that a circuit is valid – it’s interesting to think about and easy to implement the checks that are needed.

36 Optional and unassessed. Extend to more types of gates, e.g. 3 and 4 input ANDs and ORs.

37 Optional and unassessed. Find a tool to generate circuits from high-level language functions, add a file reader for its output format and test your program for some big circuits.

38 Optional and unassessed. Research about the numerous circuit optimisations that have been used in the past 10 years and implement one or two.

SUBMISSION

39 Submit your solution as a single zip file (smc.zip) using **CATE**. I will be marking electronically, so please do not use very long lines. Python recommends 4 spaces for indents and lines no longer than 79 chars (I use 2 spaces for indents and aim for less than 80 chars on a line).

40 Include in your submission:

(a) Your program plus any additional instructions for running it. Ideally `make bob` then `make alice` in a separate terminal window should be enough for me to see the truth table outputs for all circuits. I will run your program either on a CSG lab machine or on my Mac. Your code must have your name(s) and login(s) at the beginning of your Python files and JSON files.

(b) A JSON circuit file (`f.million.json`) for the Millionaires problem for 2-bit Alice values and 2-bit Bob values. The output of the circuit should be 1 if Alice>=Bob otherwise 0. Hint: *Karnaugh maps* are a very easy way of producing a circuit from a truth table.

(d) Optional: Any other interesting circuits (in `f.others.json`)

(e) Your truth-table outputs for my test circuits (in `outputs_tests`) plus for your Millionaire's problem circuit (in `outputs_million`) plus for any other interesting circuits (in `outputs_others`). Use the same output formatting as in `outputs_given` so that I can use `diff` to compare.

(f) Optional: Any feedback you have about the coursework, good or bad, including how it could be improved. Submit this as a `feedback` text file or `feedback.md` markdown file. Your feedback will not affect your mark either way!

41 The deadline for submission is **Monday 26th November 2018**.

42 Use **Piazza** if you have questions about the coursework. **Do not post your solutions or share them with others.** Email me directly if you have a specific question about your solution. The most up-to-date version of this specification will be on my course webpage (see URL above).

MARKING SCHEME

43 The coursework will be marked out of 20 using the following scheme:

	Marks	My solution Approx. lines
Garbled circuit construction and evaluation	10	200
Oblivious transfer protocol	6	80
Other code e.g. in main	2	40
JSON circuit for Millionaires problem	2	20 (9 gates)

44 I will be looking for correct implementation, clean design, readable code, helpful comments, evidence of testing/debugging. Don't worry if your solution is bigger than mine - your coding style and test/debug code will invariably be more modern than mine. Do leave any testing/debugging code in your submission but turn it off for the submitted version. If you wish you can submit a sample of your testing/debugging output in a text file called `debugging_output`

45 If you don't manage to complete all parts of the coursework, clearly state what's incomplete or not working e.g. '*our solution is not working for oblivious transfers*'. If you have some idea where the problem lies in the code, indicate it in your submission.