ASSIGNMENT CRYPTOGRAPHIC PROTOCOLS, SPRING 2017

- Hand-in solutions at the latest at the end of Monday, April 24th, 2017.
- Either by email, or on paper (in my mailbox in room next to my office MF6.105).
- You are allowed to work in pairs, and hand in one copy of the solutions per pair (like for the homework).

INSTALLING TUEVIFF

Note: TUeVIFF is based on VIFF (but not compatible with VIFF anymore at all, differing in many ways).

Follow the instructions in `installation instructions.txt` from `TUeVIFFje.zip`. (Do **NOT** use last year's version on `http://www.win.tue.nl/~berry/TUeVIFF/`).

Synopsis (files in order of importance).

setup.py in `~` (TUeVIFF root directory)

- to install VIFF (after installing Python etc.), or to update an existing installation (after modifying any file in `~/viff`)
- usage: `python setup.py install`

runtime.py in `~/viff`

- base for VIFF framework
- overrides basic operators such as $+, -, *, , <=, <, >, >=, ==, != $ to support secure computation
- combined with passive.py to yield concrete implementation

passive.py in `~/viff`

- extends runtime.py to support secure multiparty computation based on Shamir secret sharing
- provides security against passive adversary

inlinecb.py in `~/viff`

- not available in VIFF
- supports the use of so-called inline callbacks (using Python generators)
- use of '`yield`' allows to wait until exchanges of shares between parties are completed
- `yield` in combination with `declareReturn` allows to declare what a function will compute
- `returnValue` must be used instead of `return`

shamir.py in `~/viff`

- implements Shamir secret sharing (handling of shares for distribution and reconstruction protocol).
- cf. **Section 6.1.1 Shamir Threshold Scheme** of the Lecture Notes Cryptographic Protocols. Note: threshold t **in VIFF** stands for **maximum** number of corrupted parties that can be tolerated, whereas **in the lecture notes** threshold t stands for **minimum** number of honest parties needed. So, a VIFF (1,3)-threshold scheme is the same as a (2,3)-threshold scheme in the lecture notes.

prss.py in `~/viff`

- to support PRSS (Pseudo Random Secret Sharing) which is a variant of Shamir secret sharing that avoids communication for distributing shares. Efficient only for a limited number of parties (work increases exponentially as a function of the number of parties; but very low indeed for 3 parties, say).

field.py in `~/viff`

- implements fields of prime order
- overrides basic operators such as $+, -, *, , <=, <, >, >=, ==, != =$ to support modular arithmetic

millionaires.py in `~/apps`

- sample secure computation with 3 parties. Each party locally generates a number, representing its wealth, and distributes Shamir shares among the three parties (incl. itself). Then these numbers are compared using secure comparisons.

player-*.ini in `~/apps`

- configuration files to run either 3-party computation (default) or 1-party computation (convenient for testing)
- 3-party: use player-1.ini, player-2.ini, player-3.ini respectively
- 1-party (only works with TUeVIFF): use `player-1s.ini`, and also set threshold to 0 by using `-t 0` as option in command line.

LPSolver.py in `~/apps/LP`

- implements secure linear programming (not needed for the assignment)
- see `readme.txt`

ID3-Gini.py in `~/apps/ID3`

- implements secure decision trees (not needed for the assignment)
- see `readme.txt`

Remaining files either perform basic stuff or are irrelevant to us.

QUESTIONS

Your task is to answer the four questions stated below.

- Run the program secretsanta.py from a Command Prompt (DOS box):

  ```
  python secretsanta.py player-1s.ini -t 0
  ```

  Alternatively, enter `run1s secretsanta.py` to use the included DOS Batch file.

  Similarly, enter `run1s-all` to run a couple of VIFF programs from the directory in 1-party mode.

- Inspect the output, consisting of a list of random derangements (running program again gives same output).

- Next, run the program from three DOS boxes, as follows:

  [DOS box 1] `python secretsanta.py player-1.ini`
  [DOS box 2] `python secretsanta.py player-2.ini`
  [DOS box 3] `python secretsanta.py player-3.ini`

  Alternatively, enter `run123a secretsanta.py` to use the included DOS Batch file.

  Similarly, enter `run123a-all` to run a couple of VIFF programs from the directory in 3-party mode.

- Inspect the output, consisting of a list of random derangements (running three programs again gives same output, but note that the derangements are different from the 1-party case above).

- Study the program `secretsanta.py` by inspecting all steps of the three functions `random_unit_vector()`, `random_permutation()`, and `random_derangement()`. The auxiliary functions `tv.equal_zero_public(a)`, `tv.scalar_mul(a,x)`, `tv.in_prod(x,y)`, and `tv.prod(x)` are from `passive.py` and compute, respectively, $a == 0$, ax for scalar a and vector x, dot product $x \cdot y$ for vectors x and y , and product of all entries $\prod_i x_i$ for vector x.

  At first, simply consider the program as an ordinary Python program, ignoring the fact that because of the VIFF framework many functions actually operate on Shamir secret-shared values. Explain why for each $n \geq 2$, a **perfectly uniform** random derangement of length n is generated (see also Wikipedia on derangements).

30% **QUESTION 1** Argue that the program is correct by showing that:

  – Function `random_unit_vector(n)` generates a perfectly uniform random length-n unit vector (exactly one entry equal to 1, other entries equal to 0).
  – Function `random_permutation(n)` generates a perfectly uniform random permutation of $\{0, \ldots, n-1\}$.
  – Function `random_derangement(n)` generates a perfectly uniform random derangement on $\{0, \ldots, n-1\}$.

  To hide the generated random derangements perfectly, executions of the program should not leak anything about the derangements that are finally output. Note that the output

`tv.equal_zero_public()` is public (which is needed for the `if-then-else` statements). The other functions all keep their outputs private (secret-shared). The use of the Python keyword `yield` is required in combination with `tv.equal_zero_public()`, intuitively because the program has to wait until the shares for the value to be revealed have been gathered (for reconstruction).

20% **QUESTION 2** Argue that the program is oblivious, hence no information is leaked about the random derangements (until these are opened of course to facilitate printing the result). To do so, show that for any output $a[]$ of `random_derangement` the execution order of the program has been independent of the value of the entries in $a[]$ (only the length n influences the execution order, but the value of n is available in the clear anyway).

- Next, we take into account that the program actually works on Shamir secret-shared values. For this reason, the basic functions `tv.random_bit()`, `tv.equal_zero_public(a)`, `tv.scalar_mul(a,x)`, `tv.in_prod(x,y)`, and `tv.prod(x)` are actually evaluated by means of protocols.

  For example, a call to function `tv.scalar_mul()` leads to the execution of the corresponding function in `passive.py`, the so-called "passive runtime". And, for example, a line like ‘ `a[i+j] +=d[j]` leads to a call to the + function, which ultimately leads to the execution of function `add()` in `passive.py` (note that both $a[i+j]$ and $d[j]$ are secret-shared values).

- The performance of this kind of VIFF programs is measured by counting the number of times players exchange shares between each other. For instance, a call to function `open()` in `passive.py` (e.g., `tv.open()`) results in exactly one (1) exchange of shares between the players. Another example: a call `scalar_mul(b,v)` for a vector `v` of length m, say, results in exactly m exchanges of shares between the players, as can be seen from the program code in `passive.py`. And, finally, a call to `add()` results in no (0) exchanges of shares, as can be seen from the program code in `passive.py` as well. (Recall from the Lecture Notes Cryptographic Protocols, Section 7.2, that the cost for homomorphic operations $+, -$ is very low, whereas operations like $*$ require the execution of an (interactive) protocol.)

30% **QUESTION 3** Give a performance analysis of the three functions `random_unit_vector()`, `random_permutation()`, and `random_derangement()`, as a function of the length `n`.

- Finally, we consider the following extension. Now we not only want to avoid fixed-points (or, self-loops), but also any 2-cycles. Note that a self-loop or 2-cycle occurs when $a[a[i]] = i$ for some i.

20% **QUESTION 4** Design and implement an oblivious function `random_derangement2()`, which outputs a perfectly uniform permutation without self-loops and without 2-cycles. The function should do so without leaking any further information. Argue why your solution is correct and oblivious, and try to ananlyze its performance.

Run your program to see if it works.

- This concludes your first non-trivial encounter with secure multiparty computation in TUeVIFF.