Extension to the Virtually Ideal Functionality Framework:

Passively Secure Fixed Point Arithmetic

Software

Sebastiaan de Hoogh, Berry Schoenmakers

1. Introduction

The Virtually Ideal Functionality Framework, introduced by Martin Geisler in 2007, is a framework that allows programmers to implement secure multiparty protocols in a simple way. That is, the programmer is required to provide a script that is executed by all parties and that specifies what values are protected and what not. The framework will apply interactive protocols when it should, for example when two protected values are multiplied. For a detailed introduction and description we refer to the online documentation of VIFF: http://viff.dk.

A nice feature of VIFF is, therefore, that algorithms can be implemented securely in such a way that the source code looks very similar to the corresponding implementation where nothing is protected. Hence it provides means to allow cryptographically inexperienced programmers to implement algorithms securely without worrying too much about the cryptographic details. Of course, caution is required, especially when choosing the parameters and when to reveal certain information.

A typical restriction of secure multiparty computation is that the values on which the computation is done are required to be integer. Moreover the passively secure protocols that are implemented in the latest version available at http://viff.dk are not up to date. In [Hoo12, CH10a, CH10b] several improvements to the protocols based on Shamir Secret Sharing are introduced as well as basic protocols to allow fixed point arithmetic. These improvements apply to the passive runtime of VIFF as well, since it is based on Shamir Secret Sharing.

In this work, we extended VIFF by replacing basic protocols in the passive runtime with more efficient ones and adding basic tools such as prefix-multiplication. In addition, we introduce a library for secure computation over non integer values. This library has been merged into VIFF in such a way that old programs in VIFF will still work and the new library is capable of mixing old-style shares with new-style shares.

Roughly the library works as follows: the class Share has a new flag called fp, which is by default set to False (this allows original initialization). When it is set to True, then the value of the Share represents a number of which some number of least significant bits are behind the fixed point. The number of bits behind the point is an option given to the runtime. If a multiplication, for example, is going to be executed of two shares, then the interactive multiplication protocol is executed, which is extended to perform a secure truncation if both shares represents a fixed point number.

By design the scheduling of the computations is based on the Shares. Once a Share has been arrived the actual computation will be carried out over its value, which is by construction a field element. So we add the fp-flags to the shares to allow correct scheduling of the protocols. For correct computation we add a *resolution* (r) to each field element, which is by default set to zero (hence it represents by default an integer). The resolution counts the number of bits behind the point. The arithmetic operations on the field elements are changed taking the resolution into account. For example if *x* is a field element representing 1.5 and *y* a field element representing 2. Then *x*.`value`= 3, *x*.`r=1` and *y*.`value=2` and *y*.`r=0`. Let *z=x+y*, then field addition will return *z*, where *z*.`value=7` and *z*.`r=1`. This is indeed correct as 7/2=3.5.

2. Software

Contents:

- `comparisonFP.py,`
- `divisionFP.py,`
- `equalityFP.py,`
- `fieldFP.py`
- `passiveFP.py`
- `runtimeFP.py`
- `boostFP.py`
- `cfieldFP.pyd`
- `cdeferFP.pyd`
- `test_programFP.py`

2.1 Installation

This describes the installation procedure for TUeVIFF on Windows. TUeVIFF is an updated VIFF by Berry Schoenmakers and contains the latest version of VIFF which is also simple to install on a windows machine. To install TUeVIFF perform the following steps (for Windows 7):

1. Install **python 2.7.5** by running python-2.7.5.msi.
2. Add C:\Python27\;C:\Python27\Scripts\ to PATH.
   The Windows environment variable PATH can be accessed via *Computer->Properties->Advanced System Settings->Environment Variables*.
3. Install **easy_install** by running setuptools-0.6c11.win32-py2.7.exe.
   Put the file easy_install.exe in C:\Python27\Scripts\.
4. Install **twisted 13.0.0** by running Twisted-13.0.0.win32-py2.7.msi.
5. Install **zope.interface** by running (from a Command Prompt): easy_install zope.interface
6. Install **gmpy 1.15** by running gmpy-1.15.win32-py2.7.exe.
7. Download gmp-dynamic-vc-4.1.2.zip, and extract gmp.dll to C:\Windows\System32\ (or somewhere else in a directory on PATH).
8. Install **TUeVIFF** by downloading TUeVIFF zip, extracting it to a working directory,
9. Copy the files in to the directory TUeVIFF\viff
10. Run in the root directory of TUeVIFF (setup.py is in the root): python setup.py install --user

2.2 Implementation

In this Section we discuss our updates with respect to VIFF version 1.0 which is basically the same as TUeVIFF. For a detailed description and user manual we refer to the documentation of viff: http://viff.dk/doc/index.html.

2.2.1 RuntimeFP Module

This is where the virtual ideal functionality is hiding. The runtime is responsible for sharing inputs, handling communication, and running the calculations. Each player participating in the protocol will instantiate a **Runtime** object and use it for the calculations.

The Runtime returns **Share** objects for most operations, and these can be added, subtracted, and multiplied as normal thanks to overloaded arithmetic operators. The runtime will take care of scheduling things correctly behind the scenes.

RuntimeFP extends runtime in the following way:

- Class `viff.runtimeFP.Share(`*`runtime, field,Value=None, fp=False`*`)`
  This class represents shared numbers. All arithmetic operations on shares are overloaded by the corresponding protocols. New or changed functions:
  - `__init__(`*`runtime, field, value=None, fp=False`*`)`
    This function initializes a share. If the value is not set, it is a deferred waiting for its value to arrive after communication. If fp is not set, then the share represents an integer value.
  - `clone_nofp()`
    This function clones a share, but will ensure that the fp flag of the clone is disabled. This is used in statically secure protocols, where some *x+r* is opened where *x* is a secret and *r* some random number which is integer. Problems arise, if *x* is a share representing a non-integer value. By construction *x+r* reveals the least significant bits of *x*, i.e., the bits behind the point.
  - `__div__(`*`other`*`)`
  - `__floordiv__(`*`other`*`)`
    Overloaded operators, executing division protocols presented in [CH10, Hoo12].
- Class `viff.runtimeFP.ShareLists(`*`shares`*`)`
  This Class creates a share that waits until all shares in *`shares`* are arrived. This class works the same as `viff.runtime.ShareList` except that it allows all kinds of nested structures of shares (such as n-dimensional matrices of shares or nested lists of shares) as input and it always waits until all shares have been arrived.
- Function `viff.runtimeFP.gather(`*`shares`*`)`
  This function works exactly the same as `viff.runtime.gather_shares` except that it allows nested arrays of shares.
- Additional option:
  - `viff.runtime.options.res`
    The number of bits behind the point. By default it is set to 10 bits.
    The value can be changed by placing `-r` value or `--resolution` value to the command line.
    Example: `python test_programFP.py player-1.ini -r 20`

2.2.2 Passively Secure Protocols

This section discusses the collection of passively secure protocols to be executed by the runtime.

- Class `viff.passiveFP.FPPassiveRuntime`(*`player, threshold, options=None`*)

  This class extends the class `viff.RuntimeFP` with actual protocols that are executed by the overloaded operators. It includes additional protocols that can be called directly. New or changed functions:

  - `Clone_share_nofp`(*`share`*)
    This function clones `shares` by ensuring that the clone has its `fp` flag disabled and such that when the value of `share` arrives, its resolution is set to zero.
  - `fan_in_mulL`(*`shares`*)
    This function returns a share representing the multiplication of all secrets represented by the shares in `shares` in log(*k*) rounds and using *k* interactive multiplications if *k* denotes the length of the list.
  - `fan_in_mulC`(*`shares`*)
    This function returns a share representing the multiplication of all secrets represented by the shares in `shares` in 2 rounds and using *3k-2* interactive multiplications if *k* denotes the length of the list. See Protocol 4.16 in [Hoo12].
  - `gauss`(*`A,d,b,c`*)
    This function returns a matrix of shares representing Gaussian elimination on the secrets represtented by matrix `A`:

    `Ad-bc`.

    This function makes optimal use of scheduling and parallelization. In addition, it performs secure truncations (`TruncPr)` where necessary.
  - `get_value`(*`field_element`*)
    This function returns the value represented by `field_element`. It will retrun a float if necessary. This is different to `viff.field.GFElement.value` which returns only the integer representation of the field value, not taking the resolution or sign into account.
  - `in_prod`(*`shares_a, shares_b`*)
    This function returns a share that represents the value of the inner product of the secrets represented by the input lists. In addition, it performs a secure truncation (`TruncPr)` if necessary. It is based on protocol 4.8 in [Hoo12].
  - `in_prod_public`(*`shares_a, shares_b`*)
    This function returns the value of the inner product of the secrets represented by the input lists. It is based on protocol 4.10 in [Hoo12].
  - `lsb`(*`share`*)
    This function computes a share representing the least significant bit of the secret represented by `share`.
  - `matrix_prod`(*`shares_a, shares_b`*)
    This function returns a matrix of shares that represents the multiplication of the secret matrices represented by `shares_a` and `shares_b`. In addition, it performs secure truncations (`TruncPr)` where necessary.
  - `mul`(*`share_a, share_b`*)
    This function returns a share that represents the value of the multiplication of the secrets of the inputs. In addition, it performs a secure truncation (`TruncPr)` if both shares represents a fixed point number.

- `mul_public(`*`share_a, share_b`*`)`
  This function returns the value of the multiplication of the secrets of the inputs. See Protocol 4.9 of [Hoo12].
- `prefix_mulC(`*`shares`*`)`
  This function returns a list of shares representing the prefix multiplication of all secrets represented by the shares in `shares` in 2 rounds and using *3k-2* interactive multiplications if *k* denotes the length of the list. See Protocol 4.17 in [Hoo12]. In addition, it performs secure truncations (`TruncPr)` where necessary.
- `pow(`*`share_a, exponent`*`)`
  This function returns a share that represents the value of raising the secret represented by `share_a` to the power `exponent`. It performs secure truncations (`TruncPr)` if `share_a` represents a fixed point number.
- `scalar_mul(`*`shares, share_a`*`)`
  This function returns a list of shares representing the result scalar multiplication of the vector of secret represented by `shares` with the secret represented by `share_a`. In addition, it performs secure truncations (`TruncPr)` where necessary.
- `scalar_prod(`*`shares_a, shares_b`*`)`
  This function returns a list of shares representing the result of multiplying each entry of the vector of secret represented by `shares_a` with the corresponding entry of the vector of secret represented by `shares_b.` In addition, it performs secure truncations (`TruncPr)` where necessary.
- `shamir_share(`*`inputters, field, number=None, threshold=None`*`)`
  This function computes and communicates Shamir shares of `number`. It works exactly as `viff.passive.shamir_share` exept that `number` is not required to be an element of `field`. The function accepts `int`'s and `float`'s as well as input. It will communicate whether the shares are representing an `int` or a `float`.
- `truncPR(`*`share, num_bits, result_fp=False`*`)`
  This function returns a share representing the value after truncation of `num_bits` least significant bits of the secret represented by `share`. It is based on Protocol 4.32 of [Hoo12]. The `fp` flag of the returned share will be set to the evaluation of the logical expression `share.fp` or `result_fp`.

- Class `viff.divisionFP.DivisionSH12Mixin`
  This class extends `viff.passiveFP.FPPassiveRuntime` with the following protocols (all used for secure division):
  - `prefix_carry(`*`shared_bits_a,shared_bits_b`*`)`
    This function returns a list of shares representing all carries that occur when adding the bits represented by `shared_bits_a` with the bits represented by `shared_bits_b`. It is based on Protocol 4.23 of [Hoo12].
  - `addBitwise(`*`shared_bits_a,shared_bits_b`*`)`
    This function returns a list of shares representing the result of binary addition of the bits represented by `shared_bits_a` with the bits represented by `shared_bits_b`. It is based on Protocol 4.24 of [Hoo12].
    Note: ordering is little endian.
  - `Bit_decompose(`*`share`*`)`
    This function returns a list of shares representing the binary representation of the secret represented by `share`. The ordering is little endian. It is based on Protocol 4.25 in [Hoo12].

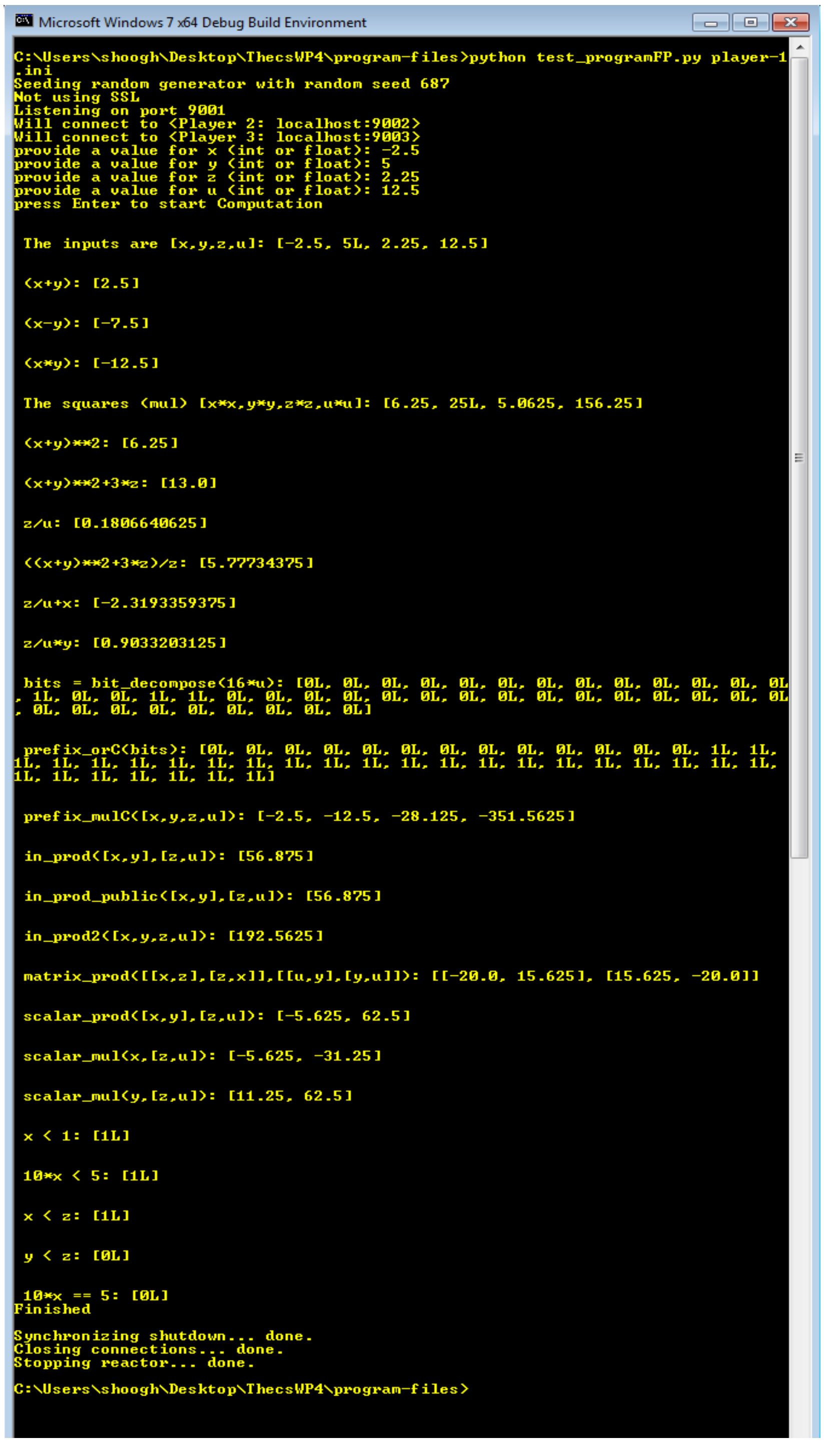
- `prefix_orC(`*`shared_bits`*`)`
  This function returns a list representing evaluation of prefix logical or on the secret bits represented by `shared_bits`. It is based on Protocol 4.18 in [Hoo12].
- `norm(`*`share`*`)`
  This function returns two shares `x` and `v`, where `x` represents the normalized secret and `v` the normalization factor. It is used as a subroutine for division. More precisely, let *a* and *b* be the secrets corresponding to respectively x and v and *s* the secret corresponding to `share`. Then *b* is a power of two and *a*=s/b lies in the interval [1/2, 1).
- `reciprocal(`*`share`*`)`
  This function returns an `fp`–enabled share that corresponds to 1 divided by the secret represented by `share`. It applies the Newton Rhapson Method, see also Protocol 4.35 of [Hoo12].
- `div(`*`share_a, share_b`*`)`
  This function returns a `fp`-enabled share that represents the result of dividing the secret represented by `share_a` by the secret represented by `share_b`. It is based on Protocol 4.34 of [Hoo12].
  `floordiv(`*`share_a, share_b`*`)`
  This function returns an `fp`–disabled share representing the integer part of the result of dividing the secret represented by `share_a` by the secret represented by `share_b`. It is based on Protocol 4.34 of [Hoo12].

- Class `viff.comparisonFP.ConstantRoundRuntime`
  This class adds protocols for secure “greater-then” comparisons based on Protocols 4.28 and 4.29 in [Hoo12] that requires a constant number of interactive rounds.

3. Using the library

Using the fixed point extension of VIFF follows the same procedure as VIFF 1.0. So we refer to the documentation of VIFF for a manual and basic examples of programming in VIFF and using the libraries http://viff.dk.

We demonstrate this fact by demonstrating the program in the appendix (`test_programFP.py`). It has to be executed by 3 parties. An additional set of example programs can be found in the folder `TUeVIFF\apps` after installation.



```
C:\Users\shoogh\Desktop\ThecsWP4\program-files>python test_programFP.py player-1
.ini
Seeding random generator with random seed 687
Not using SSL
Listening on port 9001
Will connect to <Player 2: localhost:9002>
Will connect to <Player 3: localhost:9003>
provide a value for x (int or float): -2.5
provide a value for y (int or float): 5
provide a value for z (int or float): 2.25
provide a value for u (int or float): 12.5
press Enter to start Computation

 The inputs are [x,y,z,u]: [-2.5, 5L, 2.25, 12.5]

 (x+y): [2.5]

 (x-y): [-7.5]

 (x*y): [-12.5]

 The squares (mul) [x*x,y*y,z*z,u*u]: [6.25, 25L, 5.0625, 156.25]

 (x+y)**2: [6.25]

 (x+y)**2+3*z: [13.0]

 z/u: [0.18066406251]

 ((x+y)**2+3*z)/z: [5.77734375]

 z/u+x: [-2.3193359375]

 z/u*y: [0.9033203125]

 bits = bit_decompose(16*u): [0L, 0L, 0L, 0L, 0L, 0L, 0L, 0L, 0L, 0L, 0L, 0L, 0L
, 1L, 0L, 0L, 1L, 1L, 0L, 0L, 0L, 0L, 0L, 0L, 0L, 0L, 0L, 0L, 0L, 0L, 0L, 0L, 0L
, 0L, 0L, 0L, 0L, 0L, 0L, 0L, 0L, 0L]

 prefix_orC(bits): [0L, 0L, 0L, 0L, 0L, 0L, 0L, 0L, 0L, 0L, 0L, 0L, 0L, 1L, 1L,
1L, 1L, 1L, 1L, 1L, 1L, 1L, 1L, 1L, 1L, 1L, 1L, 1L, 1L, 1L, 1L, 1L, 1L, 1L, 1L,
1L, 1L, 1L, 1L, 1L, 1L, 1L]

 prefix_mulC([x,y,z,u]): [-2.5, -12.5, -28.125, -351.5625]

 in_prod([x,y],[z,u]): [56.875]

 in_prod_public([x,y],[z,u]): [56.875]

 in_prod2([x,y,z,u]): [192.5625]

 matrix_prod([[x,z],[z,x]],[[u,y],[y,u]]): [[-20.0, 15.625], [15.625, -20.0]]

 scalar_prod([x,y],[z,u]): [-5.625, 62.5]

 scalar_mul(x,[z,u]): [-5.625, -31.25]

 scalar_mul(y,[z,u]): [11.25, 62.5]

 x < 1: [1L]

 10*x < 5: [1L]

 x < z: [1L]

 y < z: [0L]

 10*x == 5: [0L]
Finished

Synchronizing shutdown... done.
Closing connections... done.
Stopping reactor... done.

C:\Users\shoogh\Desktop\ThecsWP4\program-files>
```

Figure 1. Screen shot of execution of the test program in Appendix

Acknowledgement

This work is funded by the Dutch program COMMIT through the THeCS project.

References

[Hoo12] Sebastiaan de Hoogh. *Design of Large Scale Applications of Secure Multiparty Computation: Secure Linear Programming*. PhD thesis, Eindhoven University of Technology, the Netherlands, July 2012.

[CH10a] Octavian Catrina and Sebastiaan de Hoogh. *Improved Primitives for Secure Multiparty Integer Computation*. In SCN, pages 182--199, 2010.

[CH10b] Octavian Catrina and Sebastiaan de Hoogh. *Secure multiparty linear programming using fixed-point arithmetic.* In Proceedings of the 15th European conference on Research in computer security, ESORICS'10, pages 134--150, 2010.

Appendix: Test Program:

```
import string, operator, sys
from optparse import OptionParser
#enabled boost, this is optional!)
import viff.boostFP                             #optional
viff.boostFP.install()                          #optional
from twisted.internet import reactor

#import the required moduls in the program
from viff.fieldFP import GF
from viff.runtimeFP import create_runtime, gather, gather_shares, Runtime, Share,
FPShare, make_runtime_class
from viff.comparisonFP import Toft07Runtime, ConstantRoundRuntime
from viff.divisionFP import DivisionSH12Mixin
from viff.equalityFP import ProbabilisticEqualityMixin
from viff.config import load_config
from viff.util import rand, find_prime

from twisted.python import log

class Protocol:


    def print_comment(self, dummy, text):
        print text

    def print_vec2(self, array, f, text):
        v = [f(val) for val in array]
        print text + str(v)

    def print_matrix2(self, matrix, f, text):
        v = [[f(val) for val in row] for row in matrix]
        print text + str(v)

    def __init__(self, runtime, inputs=None):
        # Save the Runtime for later use
        self.runtime = runtime
        l = self.runtime.options.bit_length
        r = self.runtime.options.res
        k = self.runtime.options.security_parameter
        self.Zp = GF(find_prime(2**(l + 1) + 2**(4*l + k  + 1), blum=True),
self.runtime.res)

        if self.runtime.id == 1:
            x, y, z, u = [runtime.shamir_share([1], self.Zp, inp) for inp in inputs]
        else:
            x, y, z, u = [runtime.shamir_share([1], self.Zp, None) for i in range(4)]

        inputs = gather([x,y,z,u])
        inputs.addCallback(self.computation)

    def computation(self, inputs):

        x, y, z, u = [Share(self.runtime, self.Zp, entry, entry.fp) for entry in 
inputs]

        open_inputs = gather(map(self.runtime.open, [x,y,z,u]))
        open_inputs.addCallback(self.print_vec2, self.runtime.get_value, "\n\n The 
inputs are [x,y,z,u]: ")

        squares = gather(map(self.runtime.open, [x*x, y*y, z*z, u*u]))
        squares.addCallback(self.print_vec2, self.runtime.get_value, "\n\n The squares 
(mul) [x*x,y*y,z*z,u*u]: ")

        arith1 = gather(map(self.runtime.open,[(x+y)]))
        arith1.addCallback(self.print_vec2, self.runtime.get_value, "\n\n (x+y): ")

        arith0 = gather(map(self.runtime.open,[(x*y)]))
        arith0.addCallback(self.print_vec2, self.runtime.get_value, "\n\n (x*y): ")

        arith = gather(map(self.runtime.open,[(x-y)]))
        arith.addCallback(self.print_vec2, self.runtime.get_value, "\n\n (x-y): ")


        arith2 = gather(map(self.runtime.open,[(x+y)**2]))
        arith2.addCallback(self.print_vec2, self.runtime.get_value, "\n\n (x+y)**2: ")

        arith3 = gather(map(self.runtime.open,[(x+y)**2+3*z]))
        arith3.addCallback(self.print_vec2, self.runtime.get_value, "\n\n 
(x+y)**2+3*z: ")

        arith4 = gather(map(self.runtime.open,[z/u]))
        arith4.addCallback(self.print_vec2, self.runtime.get_value, "\n\n z/u: ")

        arith5 = gather(map(self.runtime.open,[((x+y)**2+3*z)/z]))
        arith5.addCallback(self.print_vec2, self.runtime.get_value, "\n\n 
((x+y)**2+3*z)/z: ")

        arith7 = gather(map(self.runtime.open,[z/u+x]))
        arith7.addCallback(self.print_vec2, self.runtime.get_value, "\n\n z/u+x: ")

        arith8 = gather(map(self.runtime.open,[(z/u)*y]))
        arith8.addCallback(self.print_vec2, self.runtime.get_value, "\n\n z/u*y: ")

        bits = self.runtime.bit_decompose(16*u)
        r_bits = gather(map(self.runtime.open, bits))
        r_bits.addCallback(self.print_vec2, self.runtime.get_value, "\n\n bits = 
bit_decompose(16*u): ")

        preOr = gather(map(self.runtime.open, self.runtime.prefix_orC(bits)))
        preOr.addCallback(self.print_vec2, self.runtime.get_value, "\n\n 
prefix_orC(bits): ")

        preM = gather(map(self.runtime.open, self.runtime.prefix_mulC([x,y,z,u])))
        preM.addCallback(self.print_vec2, self.runtime.get_value, "\n\n 
prefix_mulC([x,y,z,u]): ")

        inp1 = gather(map(self.runtime.open, [self.runtime.in_prod([x,y],[z,u])]))
        inp1.addCallback(self.print_vec2, self.runtime.get_value, "\n\n 
in_prod([x,y],[z,u]): ")

        inp2 = gather(map(self.runtime.open, [self.runtime.in_prod2([x,y,z,u])]))
        inp2.addCallback(self.print_vec2, self.runtime.get_value, "\n\n 
in_prod2([x,y,z,u]): ")

        inpp = gather([self.runtime.in_prod_public([x,y],[z,u])])
        inpp.addCallback(self.print_vec2, self.runtime.get_value, "\n\n 
in_prod_public([x,y],[z,u]): ")

        matp = gather([map(self.runtime.open, row) for row in 
self.runtime.matrix_prod([[x,z],[z,x]],[[u,y],[y,u]])])
```

```
        matp.addCallback(self.print_matrix2, self.runtime.get_value, "\n\n
matrix_prod([[x,z],[z,x]],[[u,y],[y,u]]): ")

        scalp = gather(map(self.runtime.open, self.runtime.scalar_prod([x,y],[z,u])))
        scalp.addCallback(self.print_vec2, self.runtime.get_value, "\n\n
scalar_prod([x,y],[z,u]): ")

        scalm1 = gather(map(self.runtime.open, self.runtime.scalar_mul(x,[z,u])))
        scalm1.addCallback(self.print_vec2, self.runtime.get_value, "\n\n
scalar_mul(x,[z,u]): ")

        scalm2 = gather(map(self.runtime.open, self.runtime.scalar_mul(y,[z,u])))
        scalm2.addCallback(self.print_vec2, self.runtime.get_value, "\n\n
scalar_mul(y,[z,u]): ")

        eq1 = gather(map(self.runtime.open, [x < 1]))
        eq1.addCallback(self.print_vec2, self.runtime.get_value, "\n\n x < 1: ")

        eq2 = gather(map(self.runtime.open, [10*x < 5]))
        eq2.addCallback(self.print_vec2, self.runtime.get_value, "\n\n 10*x < 5: ")

        eq3 = gather(map(self.runtime.open, [x < z]))
        eq3.addCallback(self.print_vec2, self.runtime.get_value, "\n\n x < z: ")

        eq4 = gather(map(self.runtime.open, [y < z]))
        eq4.addCallback(self.print_vec2, self.runtime.get_value, "\n\n y < z: ")

        eq5 = gather(map(self.runtime.open, [10*x == 5]))
        eq5.addCallback(self.print_vec2, self.runtime.get_value, "\n\n 10*x == 5: ")

        result = gather([\
            open_inputs,\
            squares,     \
            arith,       \
            arith0,      \
            arith1,      \
            arith2,      \
            arith3,      \
            arith4,      \
            arith5,      \
            arith7,      \
            arith8,      \
            r_bits,      \
            preOr,       \
            preM,        \
            inp1,        \
            inp2,        \
            inpp,        \
            matp,        \
            scalp,       \
            scalm1,      \
            scalm2,      \
            eq1,         \
            eq2,         \
            eq3,         \
            eq4,         \
            eq5          \
            ])
        result.addCallback(self.print_comment, "Finished\n")


        self.runtime.schedule_callback(result, lambda _: self.runtime.synchronize())
        # The next callback shuts the runtime down, killing the
        # connections between the players.
        self.runtime.schedule_callback(result, lambda _: self.runtime.shutdown())


# Parse command line arguments.
parser = OptionParser()
parser.add_option("--log", action="store_true", dest="log",
                  help="start logging")
Runtime.add_options(parser)
options, args = parser.parse_args()

if len(args) < 1:
    parser.error("you must specify a config file")
else:
    id, players = load_config(args[0])

# Create a deferred Runtime and ask it to run our protocol when ready.
runtime_class = make_runtime_class(mixins=[DivisionSH12Mixin,
ProbabilisticEqualityMixin, Toft07Runtime])

pre_runtime = create_runtime(id, players, options.threshold, options, runtime_class)

if id ==1:
    try:
        inputs = [input("provide a value for %s (int or float): " %val) for val in
['x', 'y', 'z', 'u']]
        assert reduce(operator.and_, [isinstance(x, int) or isinstance(x, float) for x
in inputs]), "Invalid number included"
    except ValueError:
        print 'Invalid Number'

    dummy = raw_input("press Enter to start Computation")
    pre_runtime.addCallback(Protocol, inputs)
else:
    pre_runtime.addCallback(Protocol)


# Start the Twisted event loop.
if options.log:
    log.startLogging(sys.stdout)

reactor.run()
```