



CACE

Computer Aided Cryptography Engineering

Project number: 216499

FP7-ICT-2007-1

D.XXX.Y VIFF Specification

Actual submission date: 17. November 2008

Start date of project: 1. January 2008

Duration: 3 years

Coordinator: Technikon Forschungs- und Planungsgesellschaft mbH (TEC)

Early Draft★ **Fixme:**
Compiled: 17.
November
2008, 11:11.

Project co-funded by the European Commission within the 7th Framework Programme		
Dissemination Level		
PU	Public	X
PP	Restricted to other programme participants (including the Commission services)	
RE	Restricted to a group specified by the consortium (including the Commission services)	
CO	Confidential, only for members of the consortium (including the Commission services)	

VIFF Specification

Martin Geisler

17. November 2008

Early Draft ★

Fixme:
Compiled: 17.
November
2008, 11:11.

The work described in this report has in part been supported by the Commission of the European Communities through the FP7 program under project number 216499. The information in this document is provided as is, and no warranty is given or implied that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

Abstract

This report describes the interface to the virtual machine provided by the VIFF project.

Keywords: multiparty computation, interface.

Contents

1	Introduction	1
1.1	VIFF History	1
2	Twisted	2
3	VIFF	4
3.1	Finite Fields	4
3.2	Secret Shared Values	4
3.3	The Runtime Classes	4
4	Programs using VIFF	5
4.1	Simple VIFF Program	5
4.2	Common Structures	5
4.2.1	Program Outline	6
4.2.2	Simple Calculations	6
4.2.3	Program Phases	7
5	System Overview	8
6	Network Assumptions	10
6.1	Modern Asynchronous Communication Networks	10
6.2	Implementing Protocols on Asynchronous Networks	11
7	Virtual Machine Interface	12
7.1	Primitives for Multiparty Computation	12
7.2	Semantics of Multiparty Primitives	12

List of Figures

1	The language stack.	1
2	Problem with fixed communication rounds	2
3	Using callbacks in Twisted.	3
4	Simple example of a VIFF program.	5
5	Relations between class instances at runtime	9
6	Asymmetric protocol	9
7	VIFF toy-example	11

List of Tables

1 Introduction

This report will describe a platform for building efficient secure multiparty computations. The platform is named VIFF which is short for *Virtual Ideal Functionality Framework*. This framework provides the following notable features:

Asynchronous execution: As described in further detail in Section 6, modern networks are all asynchronous by nature. VIFF is designed to be used on such networks.

Automatic parallel scheduling: Network latencies will typically dominate the execution time. This makes it important to execute many operations in parallel in order to lower the average waiting time.

High degree of modularity: VIFF was designed with a simple core on which more complex protocols can be built.

Easy composability: Combining smaller protocols into larger protocols is an essential feature. Protocols written for VIFF can automatically be run in parallel with other protocols. This applies to both new primitive operations and complex protocols.

The framework provides a Python library for writing MPC protocols. We see this library as a bytecode language for a higher-level language. The high-level language can provide different kinds of static security analysis of the programs, something which is not possible for VIFF itself to do. Using Python as the target language for a compiler may seem like a strange thing to do, but it is no different from implementing a new language by compiling it to, say, C. Figure 1 illustrates how VIFF can be used as an intermediate language.

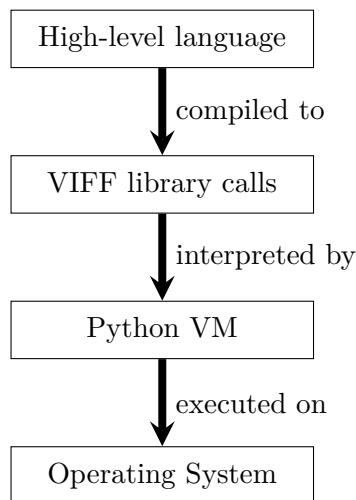


Figure 1: The language stack.

1.1 VIFF History

VIFF was created in the spring of 2007 as a test-bed for alternative MPC implementations and to show that it is possible to make a simple and light-weight framework for MPC. The

ideas in VIFF has since shown themselves to be solid and VIFF has grown from being a toy to a full and efficient MPC framework.

VIFF is a spin-off project from the Secure Information Management and Processing (SIMAP) project, which in turn is a successor to the Secure Computing Economy and Trust (SCET) project, both at the University of Aarhus. The SCET project set out to implement a platform for multiparty computations with a focus on economic applications such as auctions and benchmarking. The project implemented a prototype of a secure double auction [1]. The SIMAP project is more general and will be designing a dedicated programming language in which high-level protocol descriptions can be specified [5].

The biggest difference between VIFF and SIMAP is that VIFF automatically makes operations run in parallel, whereas this must be done explicitly with the SIMAP runtime. This makes VIFF much simpler since one does not need to specify how a new primitive interacts with every other primitive. Also, the automatic parallelism can potentially yield a faster execution since it will adapt better to changing network conditions: With a static schedule based on rounds, the execution stalls if a round takes longer than expected. VIFF would begin executing the next available operation immediately, please see Figure 2.

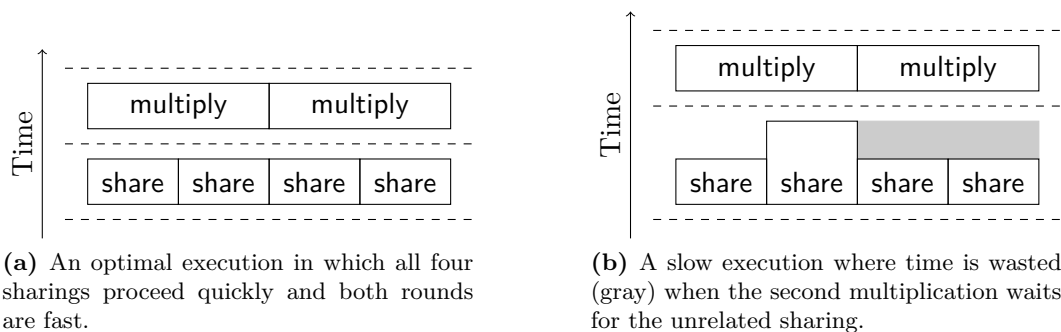


Figure 2: An execution where fixed communication rounds leads to wasted time. The dashed lines represent the synchronization done between each round and multiple operations in the same line represent parallel execution.

2 Twisted

Like many languages, Python comes with a standard library that gives access to sockets for doing network communication. The Python standard library presents a thin wrapper over the standard BSD socket interface. Twisted [3] is a Python framework that abstracts the low-level socket communication away and allows the programmer to easily build efficient network applications with asynchronous communication.

A key functionality provided by Twisted is *asynchronous* communication. With synchronous communication a call like `s.recv(4096)` will block until some data (up to 4096 bytes) is available in the socket `s`. With asynchronous communication one would instead create a function and arrange for this to be called when data is available. Such a function is denoted a *callback* in Twisted.

If a program uses just a single socket the asynchronous programming style provides no benefit. However, when several sockets are used, the `select` function makes it easy can be used

to wait on input from any of them. A single threaded program can thus efficiently process input from several connections. An event loop is created in which the program repeatedly sleeps waiting for input events from its list of open sockets. When data arrives it is dispatched to the corresponding event handler (callback function). In Twisted, a `reactor` object maintains the event loop and must be started with `reactor.run()` before events are processed. The program shuts down when `reactor.stop()` is executed.

When a call is made to an asynchronous function for which no data is available yet, a `Deferred` instance is returned instead of the real data when the data. Given a `Deferred` instance one can basically do just one thing: add callbacks to it by invoking its `addCallback` method. The instance keeps a list of callback functions which are called in sequence when the `Deferred` gets a result. If the functions `f` and `g` have been added as callbacks to a `Deferred` `d`, then a call to `d.callback(10)` will result in `g(f(10))` being executed. In other words, the first callback function (`f`) is executed with the data passed to the `callback` method. The return value from `f` is then passed onto `g`.

An example of a function returning a `Deferred` is the `getPage` function defined in Twisted. Figure 3 shows how it can be used to download a web page, compute the length of the page, print this number and finally stop the event loop.

```
from twisted.web.client import getPage
from twisted.internet import reactor

def count_lines(content):
    return content.count("\n")

def print_count(count):
    print "Lines:", count

def stop_reactor(ignored):
    reactor.stop()

page = getPage("http://example.net/")
page.addCallback(count_lines)
page.addCallback(print_count)
page.addCallback(stop_reactor)

reactor.run()
```

Figure 3: Using callbacks in Twisted.

The third callback, `stop_reactor`, has an argument which is not used. The argument is necessary because the callback is passed the result from `print_count` – functions without a `return` statement will implicitly return `None` in Python.

3 VIFF

3.1 Finite Fields

VIFF provides classes for modeling Galois (finite) fields. The `GF` function creates classes which implements Galois fields of prime order whereas the `GF256` class implements the $\mathbf{GF}(2^8)$ field with characteristic 2.

All fields work the same: instantiate an object from a field to get hold of an element of that field. Field elements implement the normal arithmetic one would expect: addition, multiplication, etc. This is provided via overloaded operators allowing one to write:

```
Zp = GF(19)
a = Zp(10)
b = Zp(5)
c = 2 * a + b
```

After this `c` is a `FieldElement` object with a value of six.

3.2 Secret Shared Values

In Twisted the `Deferred` class represents a generic deferred value. In VIFF the subclass `Share` does the same, but it is specialized to always hold a deferred `FieldElement`.

Furthermore, the `Share` class overloads the arithmetic operators. If `a` and `b` are `Share` objects, then the expression `x = a + b` will create a new `Share` object `x` which will eventually contain the correct sum of `a` and `b`.

The calculation of the arithmetic operations is delegated to a `Runtime` instance. All shares are associated with the runtime used to create them, and in the example `a + b` will result in the call `a.runtime.add(a, b)`. Similar calls will be made for subtraction, multiplication, exclusive-or, comparisons, etc.

3.3 The Runtime Classes

Any VIFF program will have an object which we call a “runtime” since it provides the basis for the protocol execution. There are several different runtime classes in VIFF:

- `PassiveRuntime` from the `viff.passive` module,
- `ActiveRuntime` from the `viff.active` module,
- `PaillierRuntime` from the `viff.paillier` module.

They all provide a common API which will be described in Section 7.

The `viff.runtime` module provides `BasicRuntime`, which is a common super class for the other runtime classes. This class is responsible for the basic infrastructure that allows the players to know one another and to communicate securely. Twisted provides support for secure communication using SSL and the runtime maintains pairwise SSL connections to the other parties.

4 Programs using VIFF

4.1 Simple VIFF Program

A simple program that uses VIFF is given in Figure 4. The program is for three players who each provide a private input. The three input numbers are multiplied and the opened product is published to everybody.

```

import sys
2 from twisted.internet import reactor

4 from viff.field import GF
from viff.runtime import create_runtime
6 from viff.config import load_config

8 id, players = load_config(sys.argv[1])
input = int(sys.argv[2])
10
def protocol(rt):
12     def got_result(result):
        print "Product:", result
14         rt.shutdown()

16     Zp = GF(1031)
        x, y, z = rt.shamir_share([1, 2, 3], Zp, input)
18     product = x * y * z
        opened_product = rt.open(product)
20     opened_product.addCallback(got_result)

22 pre_runtime = create_runtime(id, players, 1)
pre_runtime.addCallback(protocol)
24 reactor.run()

```

Figure 4: Simple example of a VIFF program.

The program is executed on three machines. If the program is saved in a file called `multiply.py`, then P_i executes

```
python multiply.py player-i.ini vi
```

where i is replaced with the player number and v_i is replaced by the value P_i wants to contribute.★

Fixme:
Describe
config files.

4.2 Common Structures

The program in Figure 4 is about as simple as it can be with VIFF, and in this section we will look more carefully at the idioms used when programming with VIFF.

4.2.1 Program Outline

Like any Python program the one in Figure 4 starts by importing any needed modules. The standard `sys` module gives access to the command line arguments, and `twisted.internet` contains the `reactor` object from Twisted. Following that a number of functions are imported from VIFF.

VIFF uses a simple configuration file to store information about the other players. This file is loaded from the first command line argument with the `load_config` function. It returns the ID of this player and a list with information about the other players. The second command line argument is used as the input value.

Next comes the definition of a function which will be used to execute the protocol proper, we will examine the function shortly. On line 22 the `create_runtime` function is used. It is given the ID of this player, the information about other players and the threshold that should be used for Shamir secret sharing. The result is a `Deferred` which will be fired with a `PassiveRuntime` object. The `pre_runtime` will only fire when the connections have been established to the other two players. We add our `protocol` function as a callback to `pre_runtime` – this will ensure that `protocol` is run with an initialized runtime as its first argument.

Finally we must start the Twisted reactor. The `create_runtime` has scheduled the opening of TCP connections between the parties, and it is necessary to start the reactor to process the events associated with this.

The outline of the simple program can be condensed into:

- Import needed functions from Python, Twisted and VIFF.
- Parse command line arguments.
- Define a protocol callback.
- Create a deferred runtime.
- Initiate the Twisted event-loop.

4.2.2 Simple Calculations

After defining the finite field \mathbb{Z}_p over which the calculations will be done, the `protocol` function runs three MPC protocols:

1. It invokes the `shamir_share` method on the runtime `rt` in line 17. All players contribute their inputs which are then Shamir secret shared and the shares are distributed among the players.

The result is that each player holds three variables: `x` is the player's share of the input number from P_1 , `y` is the share from P_2 's input number and `z` is the share from P_3 .

2. Two secure multiplication are done in line 18. The `x`, `y` and `z` variables are `Share` instances, and so the overloaded `*` operator will take care of calling the `mul` method on `rt`. This means that this line is equivalent to the longer:

```
product = rt.mul(rt.mul(x, y), z)
```

The `mul` method does a local multiplication followed by a resharing step in order to obtain shares with the correct threshold.

3. Finally the product is opened on line 20. The `open` method simply sends the shares to the designated receivers (all players by default) and Shamir recombines them.

The important thing to note is that most VIFF functions take `Share` arguments and return new `Share` arguments. The `shamir_share` method takes a normal integer since it is used at the beginning of a calculation.

The `opened_product` is a `Share`. This means that we cannot directly print its value. We must instead add the callback we defined in the beginning of the function. The `got_result` callback prints the result and stops the runtime. The `shutdown` method will make the players synchronize, close the TCP connections and stop the reactor.

4.2.3 Program Phases

The program in Figure 4 does its computation in a single phase, but more complex programs might need several phases. An example is programs that employ the `ActiveRuntime` class which has a pre-processing phase followed by the actual computation phase.

Another example is the following simple game for three players. We want the players to secret share a number each. When everybody has shared their input we open the shares to reveal the numbers. The player with the highest number wins the game.

A naïve implementation would be this:

```
def announce_winner(results):
    m, i = max(zip(results, [1, 2, 3]))
    print "The winner was Player", i, "with the number", m

def protocol(rt):
    x, y, z = rt.shamir_share([1, 2, 3], Zp, input)
    results = gather_shares([rt.open(x), rt.open(y), rt.open(z)])
    results.addCallback(announce_winner)
```

In this program P_1 will share x into x_1, x_2, x_3 , send x_2 to P_2 and x_3 to P_3 . Likewise for P_2 and P_3 . To open the shared values all players broadcast their shares to everybody else, i.e., P_1 will send x_1 to P_2 and P_3 and do the same with y_1 and z_1 .

Unfortunately this won't be secure, not even against a passive adversary. ★ The problem is that the calls to `rt.open` will schedule the broadcast of shares immediately. This means that P_1 will send out x_1 right after having computed it from x , and will send out y_1 right after having received it from P_2 ! This allows a corrupt P_3 to sit quiet and wait until the honest P_1 and P_2 sends him their shares – and so P_3 can compute x and y before having to share z . That makes it very easy for him to win the game! Please note that P_3 did not deviate from the protocol, he only waited a little which is allowed in an asynchronous setting. ★

The solution is to put in an explicit synchronization point after the sharing phase and then only proceed to the opening phase after everybody has reached this point. The `synchronize` method is used for this: it gives back a `Deferred` which will fire when everybody has executed the same call to `synchronize`. We wish to synchronize after having received all our shares. To do this we use another tool, `gather_shares`, which takes a list of shares and waits on all of them.

The revised program is:

```
def announce_winner(results):
```

Fixme: Add a drawing.

Fixme: Describe why the program behaves like this (automatic parallelism).

```

2     m, i = max(zip(results, [1, 2, 3]))
       print "The winner was Player", i, "with the number", m
4
   def open(ignored, rt, shares):
6       x, y, z = shares
       results = gather_shares([rt.open(x), rt.open(y), rt.open(z)])
8       results.addCallback(announce_winner)

10  def synchronize(ignored, rt, shares):
       sync = rt.synchronize()
12  sync.addCallback(open, rt, shares)

14  def protocol(rt):
       x, y, z = rt.shamir_share([1, 2, 3], Zp, input)
16  shares = gather_shares([x, y, z])
       shares.addCallback(synchronize, rt, [x, y, z])

```

The first argument to the `synchronize` callback (line 10) is a list with the three field elements in `x`, `y` and `z`. We ignore those since we need to call the `open` method on the `Share` instances (`x`, `y` and `z`), not the `FieldElement` instances. In the `open` callback (line 5) the first argument is `None` since this is the “result” of a call to the `synchronize` method (line 11). We therefore ignore it as well.

5 System Overview

Fixme:
Merge with
previous
sections?

★ VIFF is a software library which provides the necessary infrastructure to conduct cryptographic protocols. The protocol is executed between n parties. The parties will typically be run on different hosts and they are connected to each other with pairwise SSL connections. We assume that the necessary certificates have been distributed correctly before the protocol starts.★

Fixme:
Describe
certificate
generation.

Each party runs a program which uses VIFF as a library. The program will communicate with other parties through a central VIFF class: the `Runtime` class. We will describe the interface for this class in Section 7. Using the `Runtime` class a party can secret share inputs to the calculation, do computations with secret shared input, and open secret shared results again. The variables used for most computations are `Share` objects. Please see Figure 5 for an overview of how the different objects relate to each other at runtime.

Most operations in VIFF are symmetric: all parties play an equal role. This is true for binary operations like addition and multiplication where all n parties jointly share x and y and wish to compute shares of, e.g., $x * y$. An example of an asymmetric operation is secret sharing of input values: one party provides the input and the other parties receive shares.

Because the majority of operations are symmetric, each party will execute the same code. The idea is that though the execution is done in parallel on n machines, one should not have to worry too much about this. In this code one writes $z = x * y$ to multiply x and y and store the result in z . The fact that x and z are `Share` objects and that this operation requires a network round trip is hidden. Also, the same code is used regardless of which party did the original secret sharing. As an example, consider the code in Figure 6 which shows an example of an asymmetric protocol.

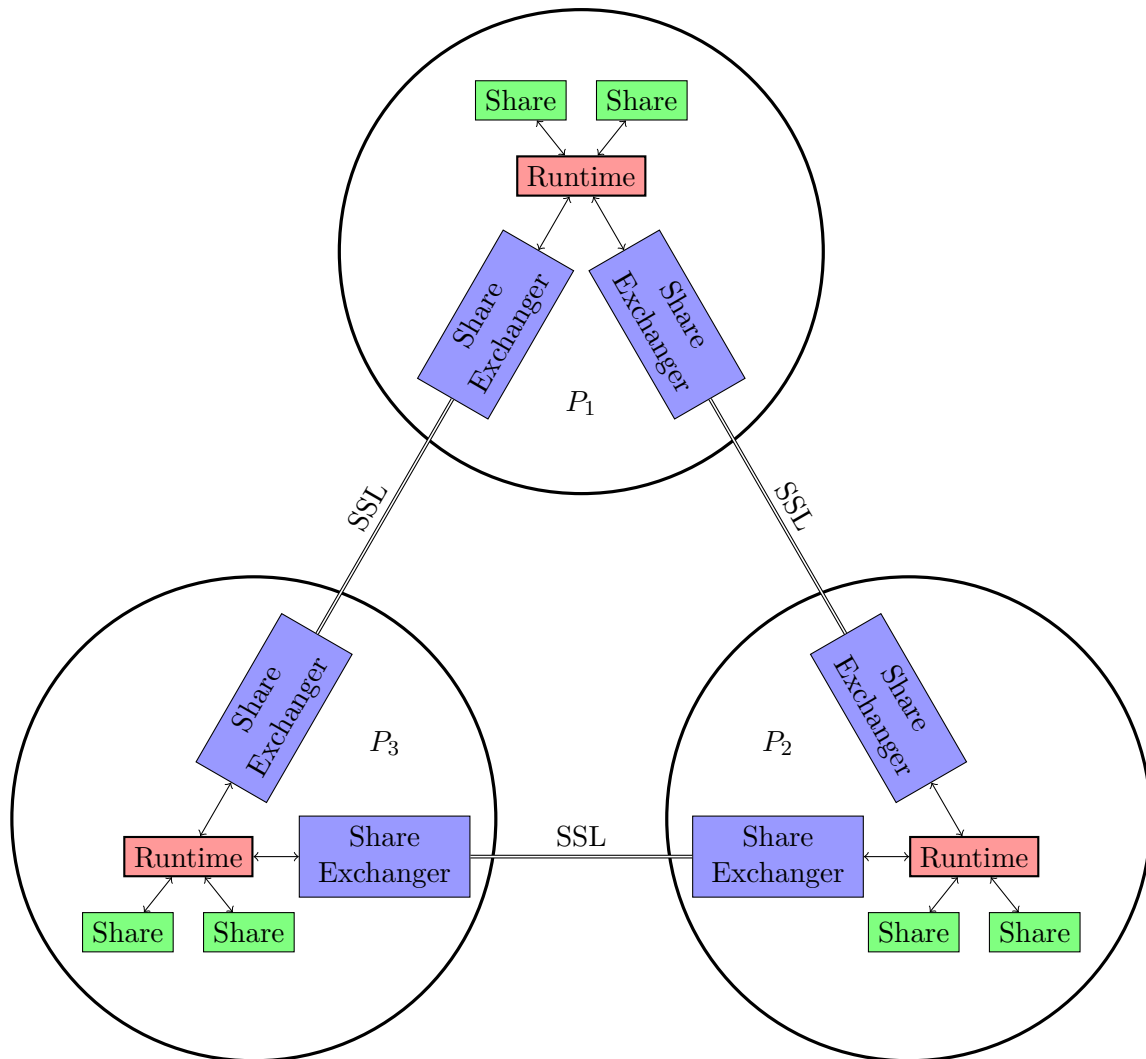


Figure 5: Relations between class instances at runtime. Each party is represented by a big circle. The Runtime objects are connected to each other via ShareExchanger objects, which maintain SSL connections between the parties. A number of Share objects exist on each party, they use the Runtime object when asked to perform calculations such as addition, multiplication, etc.

```

if rt.id in (1, 2):
    input = int(raw_input("Enter a value: "))
    x, y = rt.shamir_share([1, 2], input)
else:
    x, y = rt.shamir_share([1, 2])
z = x + y

```

Figure 6: Asymmetric protocol where only P_1 and P_2 provide input. The `rt` variable is a Runtime object which has an `id` attribute. This makes it easy to only ask for user input when the code is executed as P_1 or P_2 . The parties P_i for $i > 2$ must still participate in the Shamir sharing in order to receive their shares. In the final line all n parties compute the sum based on their shares.

The correct way to view a VIFF program is to think of it as an abstract, opaque “machine” which can do arithmetic. The machine can take inputs via methods like `shamir_share` and `prss_share` and one can extract values with the `open` method. When a value is stored in the machine one can only manipulate it with the arithmetic operations provided. We will describe this machine further in Section 7.

6 Network Assumptions

The classic results on secure multiparty computation are made under the assumption of a synchronous setting where the communication takes place on a network with known packet delays (latency) and where the parties are equipped with clocks with a known maximum drift rate. In this setting it is easy to divide the protocols into logical units called *rounds*. A round begins with the delivery of all messages sent in the previous round. Each party is then asked to specify a number of new messages which will be delivered at the beginning of the next round. It is natural to take the number of rounds needed as a measure of the protocol execution time. The total number of bits transmitted (communication complexity) is often counted as well. The time used for local computation by the parties is assumed to be negligible in comparison to the network delays and is typically not counted.

6.1 Modern Asynchronous Communication Networks

The synchronous model does not match communication networks or computers as we know them today. Modern networks are *asynchronous* and computers do normally not have access to precise clocks.

When sending packets over the Internet, the Internet Protocol (IP) [6] has the responsibility of getting the packets to the correct destination. But the IP gives very few guarantees: Intermediate routers might drop packets at any time (due problems like congestion and transmission errors) and packets may be reordered or duplicated. In particular, the IP gives no guarantees about delivery time (if the packet even reaches the destination).

The Transmission Control Protocol (TCP) [7] is normally used to create a virtual connection on top of the connection-less IP network. Because packets can be lost on the IP level, TCP must be prepared to ask for retransmission of data. This means that the delivery can be delayed further. A sender and receiver communicating over TCP are reading and writing a *stream* of bytes – there are no messages at the TCP level. As bytes are written to the stream, TCP will take care of buffering and will send out IP packets as they are filled or when it has been too long since the last packet was sent. Such buffering introduces further unpredictable delays in the protocol.

Normal computers also have no access to a globally correct clock. Computers are typically built with an on-board oscillator used to keep track of the time. Even if initially synchronized, clocks will drift away from each other since frequencies of oscillators vary with temperature. The Network Time Protocol (NTP) is widely used to keep computers synchronized to a standard time [4]. Roughly speaking, this is done by exchanging packets containing timestamps, from which the network delay can be estimated and the local clock adjusted accordingly. But the NTP server is a trusted third party and we would rather design our protocols without relying on such a service.

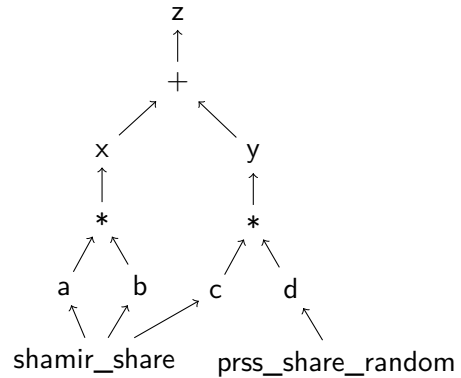
```

# (Standard program setup not shown.)
Zp = GF(1031)

input = int(raw_input("Your input: "))
a, b, c = rt.shamir_share([1, 2, 3], Zp, input)
d = rt.prss_share_random(Zp)

x = a * b
y = c * d
z = x + y

```

(a) VIFF program, `rt` is a Runtime object.

(b) Expression tree.

Figure 7: A small toy-example written for VIFF and the corresponding expression tree.

6.2 Implementing Protocols on Asynchronous Networks

To cope with the asynchronous setting the VIFF runtime system tries to avoid waiting unless it is explicitly asked to do so. In a synchronous setting all parties wait for each other at the end of each round, but VIFF has no notion of “rounds”. What determines the order of execution is solely the inherent dependencies in a given program. If two parts of a program have no mutual dependencies, then their relative ordering in the execution is unpredictable. This assumes that the calculations remain secure when executed out-of-order. Protocols written for asynchronous networks naturally enjoy this property since the adversary can delay packets arbitrarily, which makes the reordering done by VIFF a special case.

As an example, consider the simple program in Figure 7a for three parties, $n = 3$. It starts by reading some input from the user (an integer) and then defines the field \mathbb{Z}_{1031} where the toy-calculation will take place. All three parties then take part in a Shamir sharing of their respective inputs, this results in three `Share` objects being defined. A fourth `Share` object is generated using pseudorandom secret sharing [2].

Here all variables represent secret-shared values – VIFF supports Shamir secret sharing for when $n \geq 3$ and additive secret shares for when $n = 2$. The execution of the above calculation is best understood as the evaluation of a tree, please see Figure 7b. Arrows denote dependencies between the expressions that result in the calculation of the variable `z`.

The two variables `x` and `y` are mutually independent, and so one cannot reliably say which will be calculated first. But more importantly: We may calculate `x` and `y` in *parallel*. It is in fact very important for efficiency reasons that we calculate `x` and `y` in parallel. The execution time of a multiparty computation is limited by the speed of the CPUs engaged in the local computations and by the delays in the network. Network latencies can reach several hundred milliseconds, and will typically dominate the running time. So when we say *parallel* we mean that when the calculation of `x` waits on network communication from other parties, then it is important that the calculation of `y` gets a chance to begin its network communication. This puts maximum load on both the CPU and the network.

7 Virtual Machine Interface

We will now describe the high-level interface provided by the VIFF runtime system. In Section 5 it was briefly described that a VIFF program should be seen as a series of manipulations of values stored in an abstract “machine”, which can be formally modelled as an ideal functionality. We will describe the operations available to each party, but the *semantics* of each command is best understood with a global view of the computation. This view will correspond to a description of the ideal functionality.

7.1 Primitives for Multiparty Computation

Any computable function can in principle be described as a set of Boolean operations. Boolean operations can be further decomposed into a universal gate, like the NAND gate. However, even though all other logical gates can be constructed from two or more of these universal gates, we seldom want to limit ourselves to just Boolean operations when writing computer programs. Instead, we consider operations such as addition, multiplication, and comparison as our fundamental building blocks. These primitives can be executed in a single clock cycle on a modern CPU. In fact many such operations can normally be executed at once due to the use of several parallel pipelines in the CPU.

When doing MPC we consider addition and multiplication of values from a finite field as primitive operations. As mentioned, implementing NAND or NOR on bit-values would have been enough, but evaluating things at such a fine granularity is very expensive, just as it would be expensive if a CPU only did a few NAND operations per clock cycle instead of adding or multiplying whole 32- or 64-bit integers. Input to the computation in the form of (secret) sharing and output in the form of (secure) opening of sharings are considered primitive too.

We will allow several primitive operations to be started at once – not due to CPU pipelines, but due to the inherent delays of network traffic which makes it possible to send out several packets before getting a reply to the first. Executing several operations in parallel like this leads to a need for a way to specify synchronization points in programs.

This gives us the following primitives:

- Secure input and output.
- Secure arithmetic (addition, subtraction and multiplication).
- Synchronization.

In the following, we will be described the API for invoking these operations.

7.2 Semantics of Multiparty Primitives

A program using VIFF works on data shared between a number of parties. Even though the data is physically stored on distinct machines, the semantics of the operations is as if the data was stored in one machine. We call this machine the ideal functionality, \mathcal{F} . The functionality has a memory M in which data can be stored associated with a variable name. The value of the variable x is denoted $M(x)$.

In the following we will let `rt` denote a `Runtime` instance and `Zp` a `FiniteField` instance representing \mathbb{Z}_p for some large prime p . The available commands and their semantics are:

Input Data can be stored by letting all parties execute

```
x = rt.shamir_share([i], Zp, v)
```

where v is an integer from \mathbb{Z}_p for P_i and `None` for P_j with $j \neq i$. This stores $x \mapsto v$ in the memory M of \mathcal{F} .

As a shorthand one can let several parties contribute input in a single call to the `shamir_share` method:

```
x1, x2, ..., xm = rt.shamir_share([i1, i2, ..., im], Zp, vi)
```

Here v_i is an integer from \mathbb{Z}_p for P_i with $i \in \{i_1, \dots, i_m\}$ and `None` otherwise. This is equivalent to inputting m single numbers.

Output Variables can be output to reveal their value to a particular party. Letting all parties execute

```
open_x = rt.open(x, [i1, i2, ..., im])
```

gives `open_x` the value $M(x)$ for all P_i with $i \in \{i_1, \dots, i_m\}$ and `None` for the other parties.

Linear combination To store a linear combination of previously defined variables x_1, \dots, x_j with constants c_1, \dots, c_j in x , all parties execute

```
x = c1 * x1 + c2 * x2 + ... + cj * xj
```

This makes \mathcal{F} store the assignment $x \mapsto \sum_{i=1}^j c_i \cdot M(x_i)$ in its memory. Please note that this command covers simple addition of variables when all $c_i = 1$.

Multiplication To store $x \mapsto M(y) \cdot M(z)$ in the memory M the parties execute

```
x = y * z
```

Synchronization Executing

```
sync = rt.synchronize()
```

makes `sync` a `Deferred` which will trigger when all P_i have made the same call to `synchronize`.

This is a tool which can be used to create well-defined rendezvous points in a program by scheduling the rest of the computation to take place when `sync` triggers.

References

- [1] Peter Bogetoft, Ivan Damgård, Thomas Jakobsen, Kurt Nielsen, Jakob Pagter, and Tomas Toft. A practical implementation of secure auctions based on multiparty integer computation. In Giovanni Di Crescenzo and Avi Rubin, editors, *Financial Cryptography*, volume 4107 of *Lecture Notes in Computer Science*, pages 142–147. Springer, 2006.
- [2] Ronald Cramer, Ivan Damgård, and Yuval Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In Joe Kilian, editor, *TCC*, volume 3378 of *Lecture Notes in Computer Science*, pages 342–362. Springer, 2005.

- [3] Glyph Lefkowitz, Itamar Shtull-Trauring, et al. Twisted. Release 2.5.0, Twisted Matrix Laboratories, January 2007. Homepage: <http://twistedmatrix.com/>.
- [4] David L. Mills. A brief history of NTP time: Memoirs of an Internet timekeeper. *Computer Communication Review*, 33(2):9–21, 2003.
- [5] Janus Dam Nielsen and Michael I. Schwartzbach. A domain-specific programming language for secure multiparty computation. In *PLAS*, pages 21–30. ACM, 2007.
- [6] Jonathan B. Postel, editor. *Internet Protocol*, RFC 791. Internet Engineering Task Force, September 1981. Available on-line: <http://ietf.org/rfc/rfc791.txt>.
- [7] Jonathan B. Postel, editor. *Transmission Control Protocol*, RFC 793. Internet Engineering Task Force, September 1981. Available on-line: <http://ietf.org/rfc/rfc0793.txt>.