CACE

Computer Aided Cryptography Engineering

Project number: 216499

FP7-ICT-2007-1

D4.3
MPC Virtual Machine Specification

Due date of deliverable: 31. December 2008
Actual submission date: 9. January 2009

WP contributing to the deliverable: WP4

Start date of project: 1. January 2008 Duration: 3 years

Coordinator:
Technikon Forschungs- und Planungsgesellschaft mbH
Burgplatz 3a, 9500 Villach, Austria
Phone: +43 4242 233 550
Email: coordination@cace-project.eu
www.cace-project.eu

1.0

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)	

MPC Virtual Machine Specification

Editor
Martin Geisler (AU)

Contributors
Ivan Damgård (AU)
Benny Pinkas (Haifa)

9. January 2009
1.0

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.

Abstract

This report describes the interface to the virtual machine provided by the VIFF framework

Keywords: multiparty computation, interface.

Contents

List of Figures

List of Tables

1 Introduction

This deliverable should be read in connection with other CACE deliverables: D4.2 which describes PySMCL, a domain specific high-level language for multiparty computation (MPC). PySMCL programs will be compiled to Python programs that can be run using the virtual machine described here. D4.4 describes different low-level protocols that we intend to implement (or in some cases have already implemented). By including different run-time classes (a notion described in details below), a PySMCL program can be run using any of the implemented protocols.

This deliverable serves as documentation showing that the project has successfully passed milestones M4.3 Benchmark Requirements and M4.4 MPC virtual machine specification.

We assume the reader is familiar with the MPC notion, see D4.1 for an introduction.

The platform, or virtual machine, described in this document is named VIFF which is short for *Virtual Ideal Functionality Framework*. The first prototype of VIFF was created in the spring of 2007. At the start of the CACE project, it was decided to build the CACE virtual machine based on VIFF, since this would give a shorter time to a full implementation. As a result, in the first year of CACE, we have completed not only the design, but also the the implementation of a more advanced prototype. In the coming period, a full version implementing the protocols described in D4.4 is planned. As agreed between the CACE partners, VIFF is publicly available under the LGPL license.

VIFF 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. 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.

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.

In parallel with the work on the current prototype of VIFF, we have implemented the Fairplay set of secure multi-party computation protocols [1, 4], which is based on a different set of cryptographic protocols than those currently implemented in VIFF. Each of these two sets of cryptographic protocols are better suited for implementing different operations (in short, the protocols implemented in Fairplay are better for performing bit-wise operations and for comparing numbers, while the current VIFF protocols are better at performing arithmetic operations like addition and multiplication). The current Fairplay package translates programs written in a high-level language called SFDL into implementations of cryptographic protocols written in Java or C.

The final version of VIFF will be extended so that it integrates both types of implemented protocols in a single framework which will allow users to access the protocols in a unified way. This extension has been designed (but not yet implemented) and it is described in Section 7.3.

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. Figure 1 illustrates how VIFF can be used as an intermediate language.

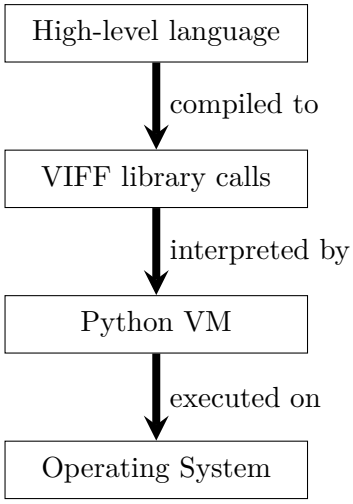


Figure 1: The language stack.

2 Asynchronous Communication

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 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 2 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 2: 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 The Runtime Classes

Any VIFF program will have an object which we call a "runtime" since it provides the basis for the protocol execution. The current prototype includes three different runtime classes. These correspond directly to three of the protocols from our implementation suite, as described in D.4.4, as follows:

- PassiveRuntime from the viff.passive module, this is the *Asynchronous Protocol* from D4.4, except that so far we only guarantee passive security.

- ActiveRuntime from the viff.active module, this is the *Hybrid protocol* from D4.4.
- PaillierRuntime from the viff.paillier module, this is the *2-party Self-trust Protocol* from D4.4, except that so far, we only guarantee passive security.

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

In addition, in the fully implemented version of VIFF, we plan that programs will have access to the following runtime classes:

- Fairplay2P, which implements a Fairplay version of a 2-party self-trust protocol based on the implementation described in [4].
- FairplayMP, which implements a Fairplay version of an asynchronous multi-party protocol based on the implementation described in [1].
- MSTP, implementing a Multi-Party Self-trust protocol with properties as described in D4.4.

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 so in the current prototype players are connected to each other with pairwise SSL connections, where we currently assume that the necessary certificates have been distributed correctly before the protocol starts. In the final version, all secure communication will be done using the NaCl library from CACE WP2.

3.2 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.

3.3 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.

4 System Overview

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, connected by secure lines as described above.

Each party runs a program which uses VIFF as a library. The program will communicate with other parties through the Runtime class explained above. We will describe the detailed interface for this class in Section 7. Using the Runtime class a party can proved secret inputs to the calculation, do computations with secret values, and open the intended results. The variables used for most computations are Share objects, which store values that are secret: all players hold data related to the value, but it can only be reconstructed if the players cooperate. Please see Figure 3 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 a shared representation 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 4 which shows an example of an asymmetric protocol.

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.

5 Programs using VIFF

5.1 Simple VIFF Program

A simple program that uses VIFF is given in Figure 5. 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.

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 v_i

where i is replaced with the player number and v_i is replaced by the value P_i wants to contribute. Each player has a configuration file with information about the other players. An

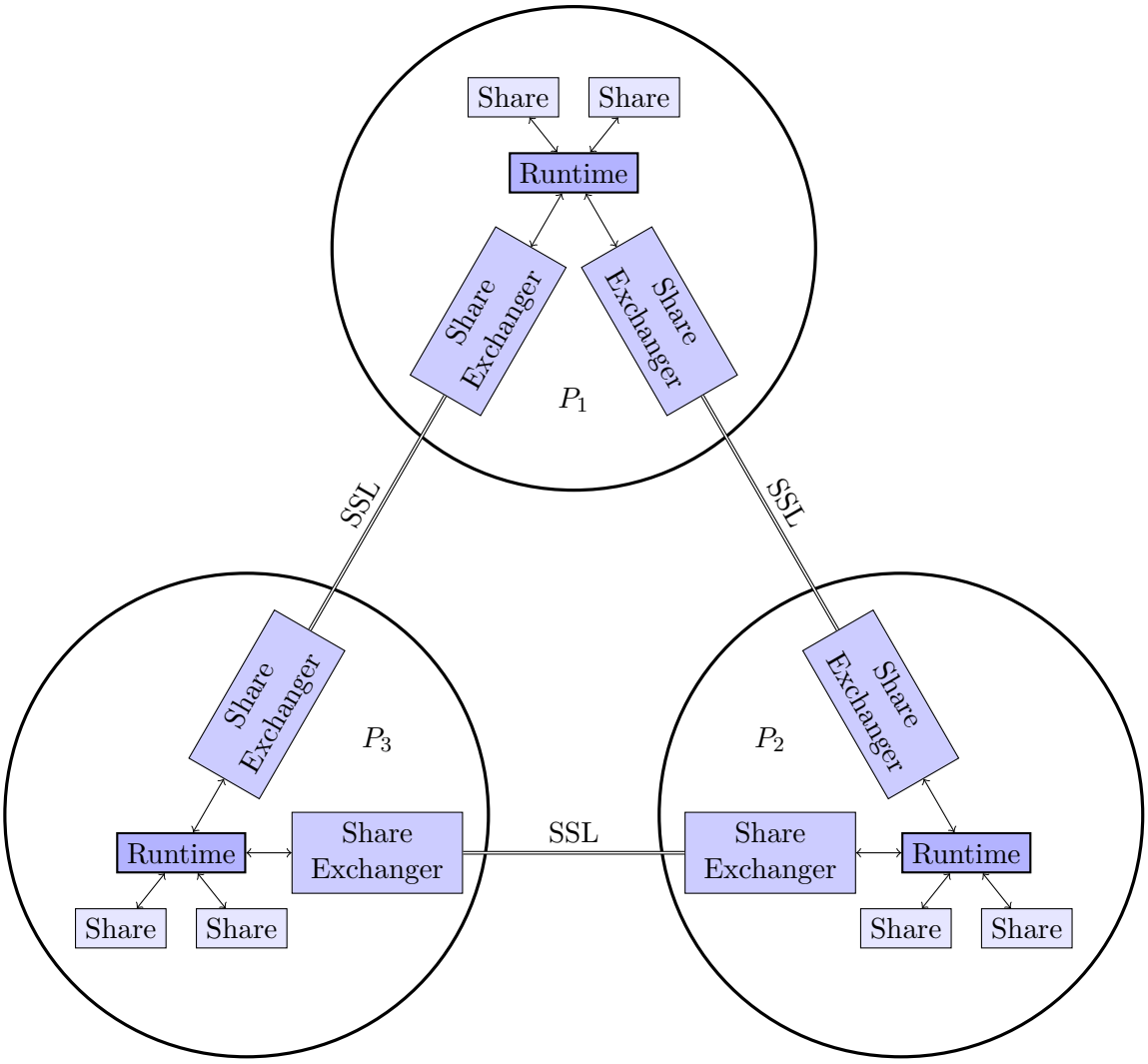


Figure 3: Relations between class instances at runtime. Each party is represented by a big circle. The Runtime objects are connected to each other via ShareExchanger objects, which maintain SSL connections between the parties. A number of Share objects exist on each party, they use the Runtime object 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 4: 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.

```
import sys
from twisted.internet import reactor

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

id, players = load_config(sys.arg[1])
input = int(sys.argv[2])

def protocol(rt):
    def got_result(result):
        print "Product:", result
        rt.shutdown()

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

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

Figure 5: Simple example of a VIFF program.

example is given in Figure 6 which shows the file for P_1 with information about two other players: the host and port entries specify the Internet location of each player, the pubkey and seckey entries hold Paillier [6] keys (tuples and triples) used for homomorphic encryptions and the prss_keys entries hold shared keys used used for pseudo-random secret sharing [2] amont different subsets of the players. Please note that P_1 has full information about its own keys, but only knows the public keys of the other players. The files for P_2 and P_3 look similar, except that they have only public knowledge about P_1.

```
# VIFF config file for Player 1

[Player 1]
  host = localhost
  port = 6001
  pubkey = 4943726044...59, 1868523483...68
  seckey = 4943726044...59, 1868523483...68, 8239543407...36
  [[prss_keys]]
    1 3 = 0x3466d8a4ddb2a805e7caeac6998ecaa3a2d1a8fbL
    1 2 = 0xd3b7d0af5088bc5f457b4eb9baeb27cb801011d1L
  [[prss_dealer_keys]]
    [[[Dealer 1]]]
      1 3 = 0xa71d201802aed3716a28d6fd48a54c6d810d1900L
      1 2 = 0xb233934c5de67ca31273f146e6357ca360a0f29fL
      2 3 = 0x205ab3a2d78a23a3e4e1f5af65cf589186363a9cL
    [[[Dealer 2]]]
      1 3 = 0x6dd7345218b80d6782934fa8fcf93c1d1af48944L
      1 2 = 0xf8bdc0812a92900279a03d8d79d464ef72df087fL
    [[[Dealer 3]]]
      1 3 = 0xc8c8cff9861336cb64725b5d987af831b2192f21L
      1 2 = 0x5325dbe156b886a35bad876e769ced7d27f85c04L

[Player 2]
  host = localhost
  port = 6002
  pubkey = 1104608088...83, 2803958346...63

[Player 3]
  host = localhost
  port = 6003
  pubkey = 9928137166...03, 8881390462...28

# End of config
```

Figure 6: Example configuration file for player 1. The integer literals for the public and secret Paillier keys have been abbreviated to fit on the page.

Currently, the configuration files for all players are generated by a trusted party in a setup phase by using a script called generate−certificates.py. Given a suitable PKI it should

be simple to let each player generate his own keys in private and then distribute the (signed) public keys as needed. However, for the simple testing done so far, a central solution suffices.

5.2 Common Structures

The program in Figure 5 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.

5.2.1 Program Outline

Like any Python program the one in Figure 5 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.

Then follows 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.

5.2.2 Simple Calculations

After defining the finite field Zp 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.

5.2.3 Program Phases

The program in Figure 5 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 wont be secure, not even against a passive adversary. The problem is that the calls to rt.open will schedule the broadcast of shares immediatedly. 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.

This "trigger-happy" behaviour is an unfortunate artifact of the automatic parallel scheduling done in VIFF – if a value is ready, it is processed as quick as possible. In this case P_1 knows that is needs to open y and to do so it will send y_1 to everybody immediately, even though it destroys the security in the example.

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

def open(ignored, rt, shares):
    x, y, z = shares
    results = gather_shares([rt.open(x), rt.open(y), rt.open(z)])
    results.addCallback(announce_winner)

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

def protocol(rt):
    x, y, z = rt.shamir_share([1, 2, 3], Zp, input)
    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.

The above code is somewhat verbose in its form. As mentioned previously, we plan to let the programmer work in a higher-level language, which will then be compiled into code like the above. We therefore find the complexity acceptable.

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 often does not match communication networks or computers as we know them today. Modern networks are often better thought of as being *asynchronous*.

For instance, when sending packets over the Internet, the Internet Protocol (IP) [7] 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) [8] 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 [5]. 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.

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 pseudo-random secret sharing [2].

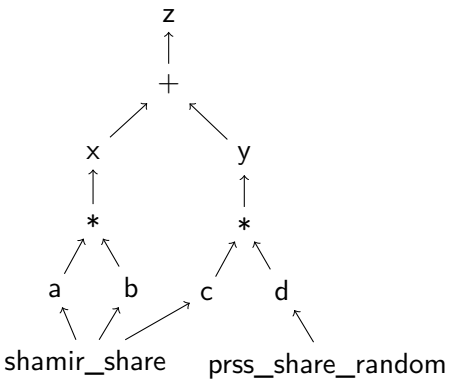
Here all variables represent secret-shared values – VIFF supports Shamir secret sharing for when $n \geq 3$ and additive secret shares for when $n = 2$. The execution of the above

```
# (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.

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 4 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 often do not 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 only such a fine granularity can be very expensive. Giving input to the computation and producing output are considered primitives too.

We will allow several primitive operations to be started at once – not due to CPU pipelines, but due to the inherit 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.input([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 $x \longmapsto v$ in the memory M of $\mathcal{F}$.

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

$x_1, x_2, \ldots, x_m$ = rt.input([$i_1, i_2, \ldots, i_m$], Zp, v_i)

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

The input method is implemented as Shamir secret sharing [9] in PassiveRuntime and as additive secret sharing in PaillierRuntime. The PassiveRuntime provides an alternative input method, prss_share, which does pseudo-random secret sharing [2].

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

opened_x = rt.output(x, [$i_1, i_2, \ldots, i_m$])

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

The output method recombines the Shamir shares in the case of a PassiveRuntime or the additive shares in case of a PaillierRuntime.

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

$$\mathsf{x} = c_1 * x_1 + c_2 * x_2 + \cdots + c_j * x_j$$

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

The addition and multiplication operators are overloaded for Share objects and the above code could also be written as

$$\mathsf{x} = \mathsf{rt.add}(\ldots(\mathsf{rt.add}(\mathsf{rt.mul}(c_1,\ x_1),\ \mathsf{rt.mul}(c_2,\ x_2)),\ \ldots,\ \mathsf{rt.mul}(c_j,\ x_j))\ldots)$$

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

$$\mathsf{x} = \mathsf{y} * \mathsf{z}$$

The Share objects overload the multiplication operator and make the above equivalent to, but more convenient than, this code:

$$\mathsf{x} = \mathsf{rt.mul}(\mathsf{y},\ \mathsf{z})$$

Synchronization Executing

$$\mathsf{sync} = \mathsf{rt.synchronize}()$$

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.

These commands are common to all three available Runtime subclasses.

7.3 Extending VIFF with Fairplay Primitives

We will extend the VIFF package so it can incorporate the functionality that is provided by the Fairplay package. This extension, however, has to start at the PySMCL level, since this is the level at which users are expected to write programs, see D4.2. The combination of the two packages must done in a way which is as transparent as possible, and which provides the user with a unified programming framework.

Recall that a Fairplay program is written by users in a high-level language called SFDL. This program is translated by the Fairplay compiler into a representation as a Binary circuit which computes the same functionality as the SFDL program. The Fairplay package contains programs which perform a secure computation of this circuit.

We will define PySFDL, which will be a high-level language in Python syntax, and with the same expressive power as SFDL. In other words, a PySFDL program is valid Python code, and it can be translated to an equivalent SFDL program.

PySMCL code can contain functions written in PySFDL as long as they begin with a `@Fairplay` decorator. A decorator is a special Python notion, in short, a decorator is a second-order function which takes a function and outputs a new one with the same name. Such a decorator can therefore modify Python code before it is actually run. The PySMCL preprocessor will identify this decorator and translate the PySFDL code into SFDL. It will then call the Fairplay compiler to translate the SFDL code into a Binary circuit. The original PySFDL code will be replaced with a reference to the resulting Binary circuit.

The VIFF runtime environment will include a Fairplay runtime class (or possibly two classes – a Fairplay2P class for secure two-party computation, and a FairplayMP class for secure multi-party computation). During execution, the parts of the code written in PySFDL will be executed by calling the Fairplay runtime class. This class will locate the relevant Binary circuit representation of the code, and evaluate it by calling the relevant Fairplay function (implemented in C or Java).

8 Benchmarking

To evaluate the performance of our protocol implementations a benchmarking program has been implemented in VIFF. The program has several parameters:

- Type of operation. Currently we support benchmarking of secure multiplications, comparisons and equality testing.
- Number of operations.
- Parallel or sequential scheduling.
- Security against passive or active adversaries.
- Two- or multiparty computation. In the case of a two-party computation, only passive security is currently implemented.

The goal is to test each of the protocols along several different axis by varying one parameter while keeping the others fixed.

The motivation for the choice of operations to benchmark is that in virtually all the protocols we consider, the only basic arithmetic operation that requires communication is multiplication. The efficiency of this therefore determines the speed of most non-trivial calculations. However, comparison is equally important as it is essential in applications such as auctions, secure databases and benchmarking. Its efficiency does not only follow from the speed of multiplications, but depends also on the overall protocol used, and it is therefore be benchmarked separately.

To run a benchmark one must first generate the necessary configuration files and then start the benchmarking script on each host. The players will connect to each other and generate random values needed in the test. Following that, any protocol-specific preprocessing is done (and timed). The benchmark is then executed and the players terminate. To ensure proper timings the players synchronize before the preprocessing and the benchmark itself.

Collecting benchmark results is currently done with ad-hoc scripts, but we are working on a unified solution suitable for running daily benchmarks. There the results will be collected

Protocol	Players	Multiplication	Compare
Two-party Self-Trust	2	0.3 s	10s
Asynchronous Multiparty, passive security	3	2 ms	350 ms
Asynchronous Multiparty, active security	4	4 ms	700 ms
Hybrid, active security	4	4 ms	700 ms

Figure 8: Goals for execution time for MPC protocols. Protocol types as defined in D4.4

in a database and proper statistics can be calculated. We expect this to be very helpful in judging the precise impact of the daily code changes.

As for the performance we require from a fully implemented virtual machine, we should consider the applications we target, but also we have to be realistic in our expectations. For instance, protocols based on homomorphic encryption such as the existing 2-party runtime class, cannot be as efficient as corresponding multiparty protocols based on secret sharing. This is simply because the best known algorithms for public-key operations are cubic time, whereas the operations for secret sharing are quadratic at worst.

It is our estimate that most of the applications we target, such as many auctions, data-mining, benchmarking, voting and surveys, are actually not very performance-critical. This is because the inputs are submitted up to a certain deadline and then the actual secure computation may take minutes or in some cases even hours without this being a problem. In a public procurement, for instance, it usually takes several days before the result is announced using current solutions. There are exceptions, however: in on-line auctions and gambling, on-line performance is very critical. In these cases, however, efficiency can be gained by providing the input in special form and/or doing preprocessing. We therefore believe that the performance targeted in Figure 8 will allow us to cover most of the relevant applications. Note, for instance, that even though we expect comparison for 2-party self-trust to be quite slow, with preprocessing, we expect to be able to do comparison in at most 500 ms.

The performance we list here as our goals in Figure 8 is for a case where the players are connected by a fast LAN (otherwise, unpredictable network delays makes it very difficult to say something precise). We list times for the minimal number of players for which the protocol can work. The times for comparison are for 32 bit numbers.

References

[1] Assaf Ben-David, Noam Nisan, and Benny Pinkas. Fairplaymp: a system for secure multi-party computation. In Peng Ning, Paul F. Syverson, and Somesh Jha, editors, *ACM Conference on Computer and Communications Security*, pages 257–266. ACM, 2008. ISBN 978-1-59593-810-7.

[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] Yehuda Lindell, Benny Pinkas, and Nigel P. Smart. Implementing two-party computation efficiently with security against malicious adversaries. In Rafail Ostrovsky, Roberto De Prisco, and Ivan Visconti, editors, *SCN*, volume 5229 of *Lecture Notes in Computer Science*, pages 2–20. Springer, 2008. ISBN 978-3-540-85854-6.

[5] David L. Mills. A brief history of NTP time: Memoirs of an Internet timekeeper. *Computer Communication Review*, 33(2):9–21, 2003.

[6] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In Jacques Stern, editor, *EUROCRYPT*, volume 1592 of *Lecture Notes in Computer Science*, pages 223–238. Springer, 1999.

[7] Jonathan B. Postel, editor. *Internet Protocol*, RFC 791. Internet Engineering Task Force, September 1981. Available on-line: `http://ietf.org/rfc/rfc791.txt`.

[8] Jonathan B. Postel, editor. *Transmission Control Protocol*, RFC 793. Internet Engineering Task Force, September 1981. Available on-line: `http://ietf.org/rfc/rfc0793.txt`.

[9] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, November 1979.