Implementing Multiparty Computation

A VIFF Case Study

http://viff.dk/

Martin Geisler
⟨mg@cs.au.dk⟩

University of Aarhus
Denmark

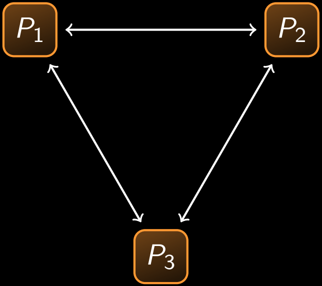
October 12, 2009
SPEED-CC

Outline

Outline

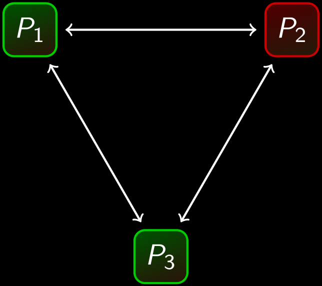
- Overview
  - Multiparty Computation
  - Virtual Ideal Functionality Framework
- Design
  - Network
  - Environment
  - Asynchronicity
  - Program Counters
- Conclusion

Quick Recap of Multiparty Computation



- n players
- wish to jointly compute f
- player P_i has input x_i
- players learn $y = f(x_1, x_2, \ldots, x_n)$

Quick Recap of Multiparty Computation



- n players
- wish to jointly compute f
- player P_i has input x_i
- players learn $y = f(x_1, x_2, \ldots, x_n)$
- up to t players are corrupt
- must keep inputs private
- must ensure correct output
- players only learn y

Requirements

We need fast local operations:

- fast cryptosystems
- fast hash functions
- and so on. . .

Requirements

We need fast local operations:

- fast cryptosystems
- fast hash functions
- and so on. . .

But we also need:

- fast cryptographic protocols
- flexible protocol description language
- efficient usage of network resources

VIFF Overview

- VIFF: Virtual Ideal Functionality Framework
- Python library for MPC

VIFF Overview

- VIFF: Virtual Ideal Functionality Framework
- Python library for MPC
- we wanted to write:

```
i = int(sys.argv[1])              # read commandline argument
(a, b, c) = shamir_share(i)       # Shamir secret share input
x = a * b + c                     # secure multiparty computation
print open(x)                     # broadcast and recombine
```

(we almost got there)

VIFF Overview

- VIFF: Virtual Ideal Functionality Framework
- Python library for MPC
- we wanted to write:

```
i = int(sys.argv[1])                 # read commandline argument
(a, b, c) = shamir_share(i)          # Shamir secret share input
x = a * b + c                        # secure multiparty computation
print open(x)                        # broadcast and recombine
```

  (we almost got there)
- we also wanted this code to execute in one round:

```
x = a * b
y = b * c
z = c * a
```

- we wanted to do MPC over real networks, i.e., the Internet

Applications

We have implemented a number of applications in VIFF:

- Distributed AES
- Distributed RSA
- Double Auction
- Voting
- Poker

Related Projects

SIMAP — `http://simap.dk/`

- general multiparty computations
- Java implementation
- some work done on a domain specific language

Related Projects

SIMAP — `http://simap.dk/`

- general multiparty computations
- Java implementation
- some work done on a domain specific language

FairPlay — `http://fairplayproject.net/`

- Yao-garbled circuits for 2 or more parties
- Java implementation
- own language for MPC programs

Related Projects

SIMAP — `http://simap.dk/`

- general multiparty computations
- Java implementation
- some work done on a domain specific language

FairPlay — `http://fairplayproject.net/`

- Yao-garbled circuits for 2 or more parties
- Java implementation
- own language for MPC programs

Sharemind — `http://sharemind.cs.ut.ee/`

- computation over the ring $\mathbb{Z}_{2^{32}}$
- C++ implementation
- scalable to very large data sets
- own MPC assembler language and compiler

Outline

Asynchronous vs. Synchronous Network

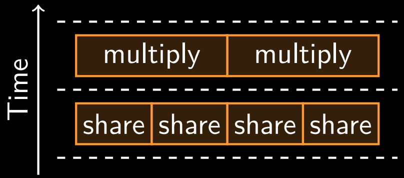
VIFF assumes an asynchronous network:

- real-world networks are asynchronous
- it is the most flexible choice

Asynchronous vs. Synchronous Network

VIFF assumes an asynchronous network:

- real-world networks are asynchronous
- it is the most flexible choice

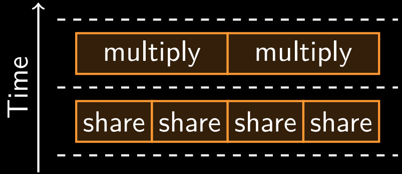


- all rounds equally fast
- optimal execution

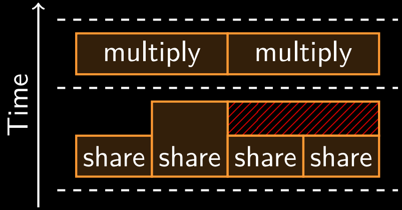
Asynchronous vs. Synchronous Network

VIFF assumes an asynchronous network:

- real-world networks are asynchronous
- it is the most flexible choice



- all rounds equally fast
- optimal execution



- processing stalls
- wasted time!

Transport Protocol

We currently use SSL over TCP:

- gives reliable, authenticated point-to-point channels
- litterature generally wants exactly this

Transport Protocol

We currently use SSL over TCP:

- gives reliable, authenticated point-to-point channels
- litterature generally wants exactly this

UDP would be an interesting alternative:

- discrete packets — send one share per packet
- we do not care about reordering
- most protocols can handle some dropped packets!

Network Architecture

We use a peer-to-peer architecture:

- parties are symmetric
- very general architecture

Network Architecture

We use a peer-to-peer architecture:

- parties are symmetric
- very general architecture

SIMAP used a central coordinator:

- forwards packets only
- makes NAT-traversal simple
- a potential bottle-neck

Programming Language

VIFF is written in Python:

- flexible language, well suited for rapid prototyping
- Twisted library for asynchronous network communication

Programming Language

VIFF is written in Python:

- flexible language, well suited for rapid prototyping
- Twisted library for asynchronous network communication
- anonymous functions:

```
share_x.addCallback(lambda x: x * x)
```

Programming Language

VIFF is written in Python:

- flexible language, well suited for rapid prototyping
- Twisted library for asynchronous network communication
- anonymous functions:

```
share_x.addCallback(lambda x: x * x)
```

- operator overloading:

```
a.add(b).sub(a.mul(b).mul(2))
```

$\rightsquigarrow$

```
a + b - 2 * a * b
```

Programming Language

VIFF is written in Python:

- flexible language, well suited for rapid prototyping
- Twisted library for asynchronous network communication
- anonymous functions:

```
share_x.addCallback(lambda x: x * x)
```

- operator overloading:

```
a.add(b).sub(a.mul(b).mul(2))
```

$\rightsquigarrow$

```
a + b - 2 * a * b
```

- absolutely everything is interpreted
- lack of static types enables stupid mistakes

Programming Environment

VIFF provides the a framework in the form of a library:

- makes “VIFF programs” regular Python programs
- provides full access to Python standard library

Programming Environment

VIFF provides the a framework in the form of a library:

- makes "VIFF programs" regular Python programs
- provides full access to Python standard library
- however, we cannot use control structures directly:

```
if rt.open(a < b and b < c):
    print "Wow, monotone!"
```

Must rewrite as:

```
def check_monotone(result):
    if result:
        print "Wow, monotone!"

x = rt.open(a < b and b < c)
x.addCallback(check_monotone)
```

- long-term solution: put a DSL on top of VIFF

Programming Paradigm

Asynchronous communication via callbacks:

- “don’t call us, we’ll call you”
- uses a network library called Twisted

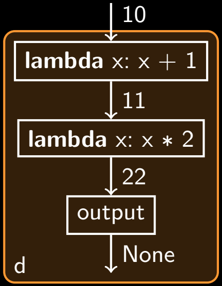
Programming Paradigm

Asynchronous communication via callbacks:

- "don't call us, we'll call you"
- uses a network library called Twisted
- Twisted's fundamental abstraction is the Deferred:

```
def output(x): print x

d = Deferred()
d.addCallback(lambda x: x + 1)
d.addCallback(lambda x: x * 2)
d.addCallback(output)
d.callback(10)
```



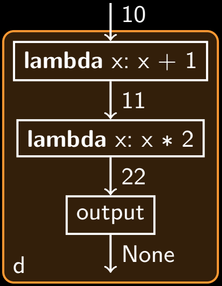
Programming Paradigm

Asynchronous communication via callbacks:

- “don’t call us, we’ll call you”
- uses a network library called Twisted
- Twisted’s fundamental abstraction is the Deferred:

```
def output(x): print x

d = Deferred()
d.addCallback(lambda x: x + 1)
d.addCallback(lambda x: x * 2)
d.addCallback(output)
d.callback(10)
```



- this can lead to an unnatural way of programming
- completely single-threaded — no blocking the event loop!

More on Deferreds

We use Deferreds heavily:

- subclass Share provides operator overloading:

```
x = a * b + c * 10
```

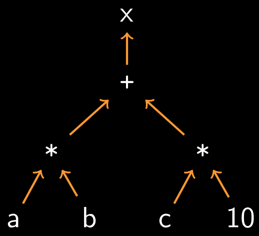
More on Deferreds

We use Deferreds heavily:

- subclass Share provides operator overloading:

```
x = a * b + c * 10
```

- Share objects are created and combined:



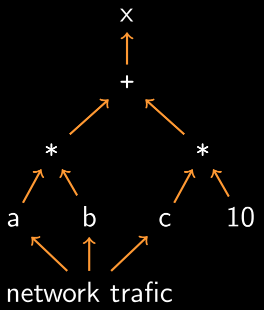
More on Deferreds

We use Deferreds heavily:

- subclass Share provides operator overloading:

```
x = a * b + c * 10
```

- Share objects are created and combined:



Dangers of Deferreds

Deferreds are not free:

- a single, empty Deferred is about 200 bytes
- adding a callback costs at least 300 bytes more

Dangers of Deferreds

Deferreds are not free:

- a single, empty Deferred is about 200 bytes
- adding a callback costs at least 300 bytes more
- it is easy to allocate lots of Deferreds:

```
for i in range(10000):
    x = x * x
```

- all 10,000 multiplications are scheduled immediately:

What About Threads?

Threads are the main alternative to callbacks:

- can use multiple cores!
- normal program flow, you can block when you want

What About Threads?

Threads are the main alternative to callbacks:

- can use multiple cores!
- normal program flow, you can block when you want
- thread-switches supposedly have some overhead
- must synchronize threads (and avoid dead-locks. . .)
- need a way to specify future tasks (callbacks. . .)

Pipelining

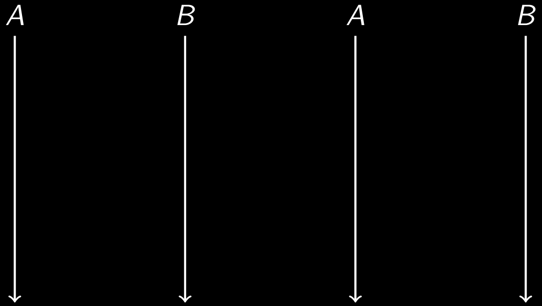
Network delay kills throughput unless we run things in parallel:

- like a CPU, we pipeline many operations in parallel

Pipelining

Network delay kills throughput unless we run things in parallel:

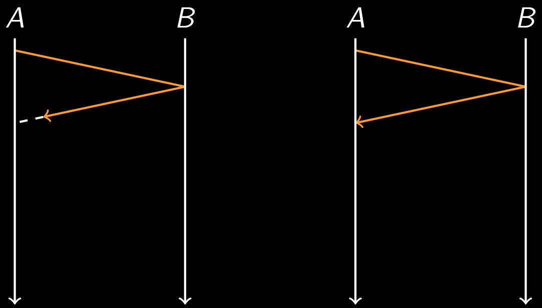
- like a CPU, we pipeline many operations in parallel
- can potentially remove idle time:



Pipelining

Network delay kills throughput unless we run things in parallel:

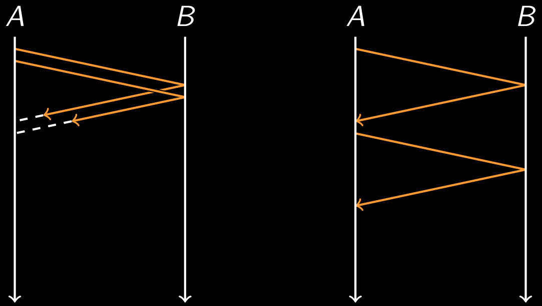
- like a CPU, we pipeline many operations in parallel
- can potentially remove idle time:



Pipelining

Network delay kills throughput unless we run things in parallel:

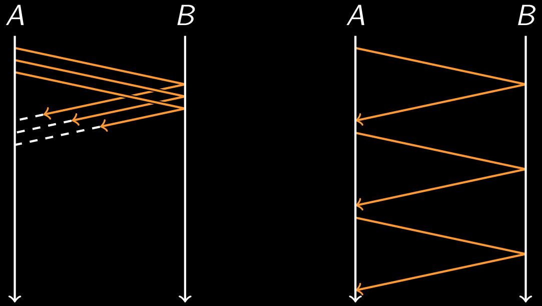
- like a CPU, we pipeline many operations in parallel
- can potentially remove idle time:



Pipelining

Network delay kills throughput unless we run things in parallel:

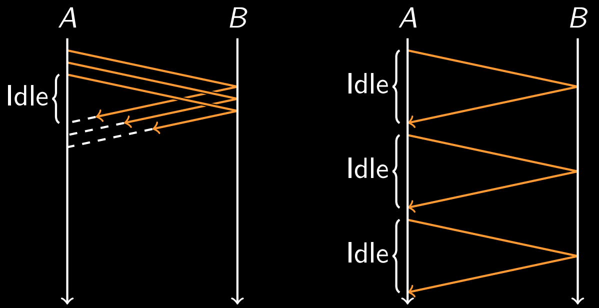
- like a CPU, we pipeline many operations in parallel
- can potentially remove idle time:



Pipelining

Network delay kills throughput unless we run things in parallel:

- like a CPU, we pipeline many operations in parallel
- can potentially remove idle time:



Automatic pipelining

VIFF will automatically pipeline everything:

- network traffic begins upon return to event loop
- no notion of rounds
- fits naturally with asynchronous execution

Why We Must Keep Track of Things

Consider this very high-level code for multiplication:

```
def mul(share_a, share_b):
    result = gather_shares([share_a, share_b])
    result.addCallback(finish_mul)
    return result
```

Why We Must Keep Track of Things

Consider this very high-level code for multiplication:

```
def mul(share_a, share_b):
    result = gather_shares([share_a, share_b])
    result.addCallback(finish_mul)
    return result
```

It is used twice like this:

```
x = a * b
y = c * d
```

Why We Must Keep Track of Things

Consider this very high-level code for multiplication:

```
def mul(share_a, share_b):
    result = gather_shares([share_a, share_b])
    result.addCallback(finish_mul)
    return result
```

It is used twice like this:

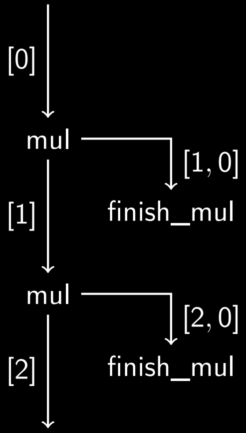
```
x = a * b
y = c * d
```

We now have a problem:

- finish_mul is executed when a and b arrives
- finish_mul is executed when c and d arrives
- other players cannot know which pair arrives first!

Program Counters

VIFF use program counters to track operations:



Program Counter Properties

- assignment depends on program structure
- ensures deterministic assignments
- unique labels for each operation

Preprocessing

Many protocols can be divided into two phases:

- an off-line phase which is independent of actual input
- an on-line phase which do depend on the input

Preprocessing

Many protocols can be divided into two phases:

- an off-line phase which is independent of actual input
- an on-line phase which do depend on the input

A good example is an actively secure multiplication:

- generate a random triple $([a], [b], [ab])$ off-line
- use it to multiply $[x]$ and $[y]$:

$$d = \text{open}\big([x] - [a]\big)$$
$$e = \text{open}\big([y] - [b]\big)$$
$$[xy] = de + d[y] + e[x] + [ab]$$

But how to implement this?

Program Counters Strikes Again!

We have an unique label for each operation:

- run program without any preprocessed data
- record program counters for missing data
- start next run with a preprocessing phase

Program Counters Strikes Again!

We have an unique label for each operation:

- run program without any preprocessed data
- record program counters for missing data
- start next run with a preprocessing phase

Will the program always use the same program counters?

- yes! — otherwise it would leak information on the inputs

Outline

Conclusion

Experiences with VIFF:

- asynchronous design works well
- flexible design pays off

Conclusion

Experiences with VIFF:

- asynchronous design works well
- flexible design pays off

Thank you!