April 19-20, 2019
Computer History Museum
Mountain View, CA

Secure and Privacy-preserving Computation

Karim Eldefrawy

karim@csl.sri.com

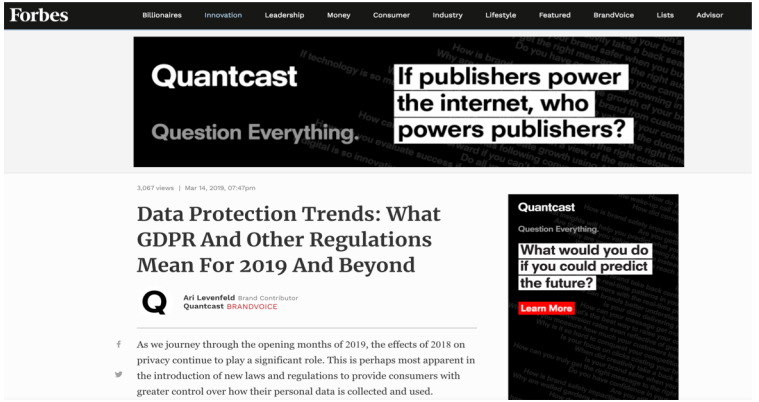
Recent Breaches

500 million records containing personal information of guests

143 million records containing personal information of consumers

Privacy Related Regulations

- Europe:
  - GDPR: General Data Protection Regulation
- North America:
  - FCRA: Fair Credit Reporting Act
  - ECPA: Electronic Communications Privacy Act
  - HIPPA: Health Insurance Portability and Accountability Act



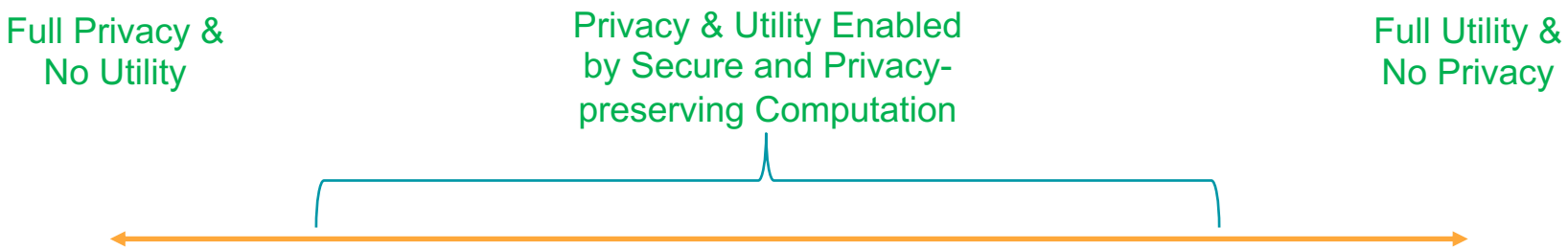
https://www.forbes.com/sites/quantcast/2019/03/14/data-protection-trends-what-gdpr-and-other-regulations-mean-for-2019-and-beyond/#6b018d22728d

https://healthitsecurity.com/news/how-the-federal-data-privacy-debate-regulations-may-impact-healthcare

Privacy vs Utility

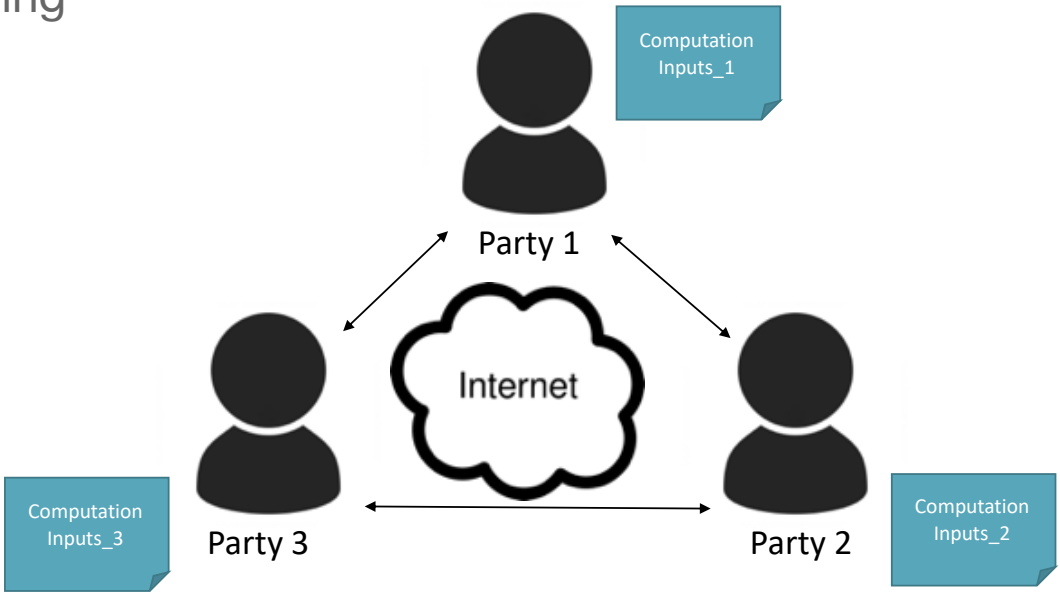
- Do we have to give up our data to get utility online (e.g., personalized recommendations, credit scores, loyalty points)?

- Secure and privacy-preserving computation techniques give us the best of both worlds (simultaneously), privacy and security AND utility



Secure Multi-Party Computation (MPC)

- MPC enables two or more (distrusting) parties to compute without revealing their data.



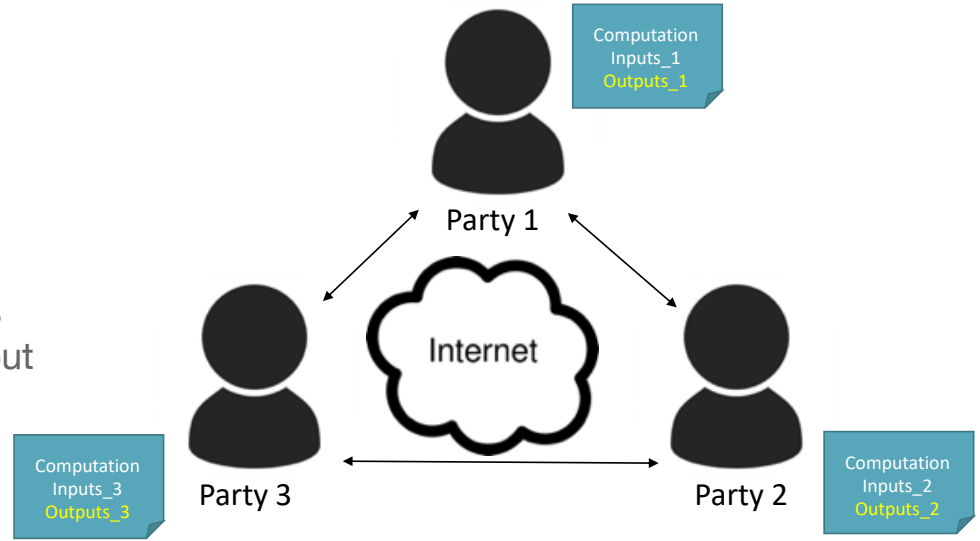
Secure Multi-Party Computation (MPC)

- MPC enables two or more (distrusting) parties to compute without revealing their data.

- Two security models:

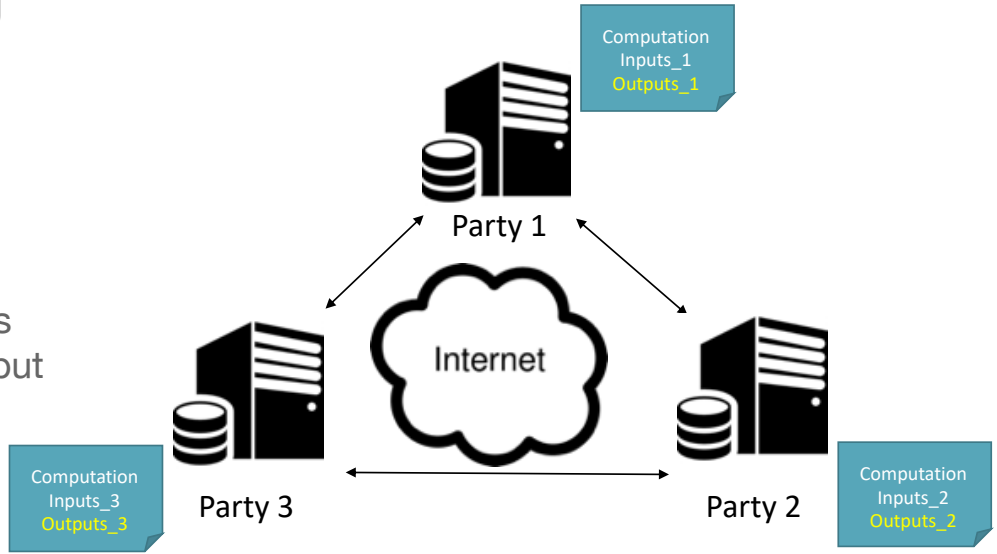
  - Semi-honest (honest-but-curious): parties do not modify software (follow protocol) but try to learn more information and violate privacy

  - Malicious: parties modify software (do not follow protocol) and can misbehave and cheat



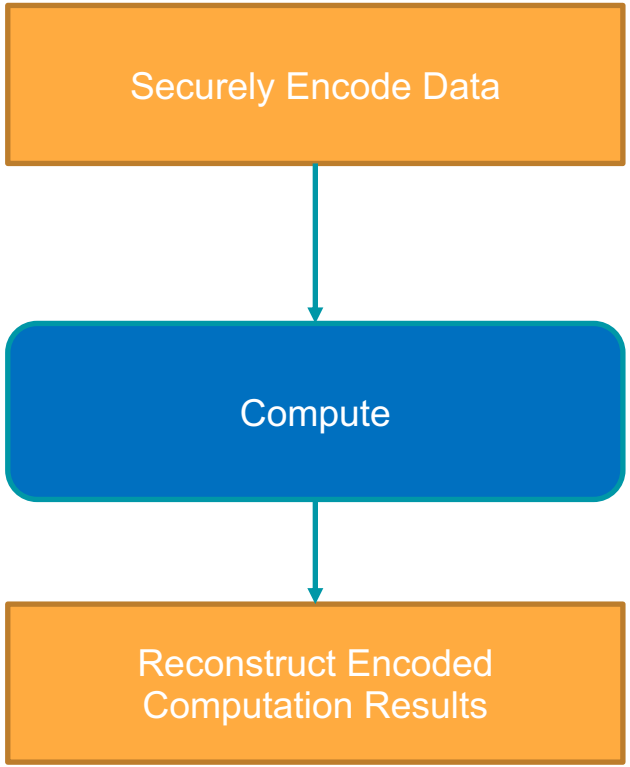
Secure Multi-Party Computation (MPC)

- MPC enables two or more (distrusting) parties to compute without revealing their data.
- Two security models:
  - Semi-honest (honest-but-curious): parties do not modify software (follow protocol) but try to learn more information and violate privacy
  - Malicious: parties modify software (do not follow protocol) and can misbehave and cheat



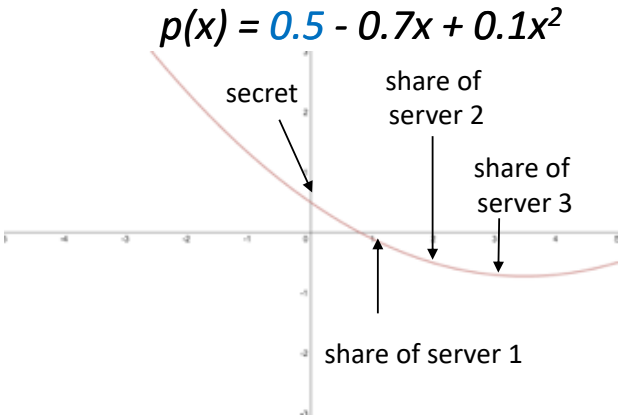
Blueprint for MPC

- Encode data into a (secure) distributed format
  - e.g., using the share algorithm in a secret sharing scheme
- Perform computation on securely encoded distributed data
- Reconstruct encoded results
  - e.g., using the reconstruct algorithm in a secret sharing scheme



Encoding Data via Secret Sharing

- **Shamir's Technique:** assuming n parties store secret in constant term of degree *t* polynomial to tolerate up to *t* leaked shares (called *t+1* out of *n*)



- Need two algorithms:

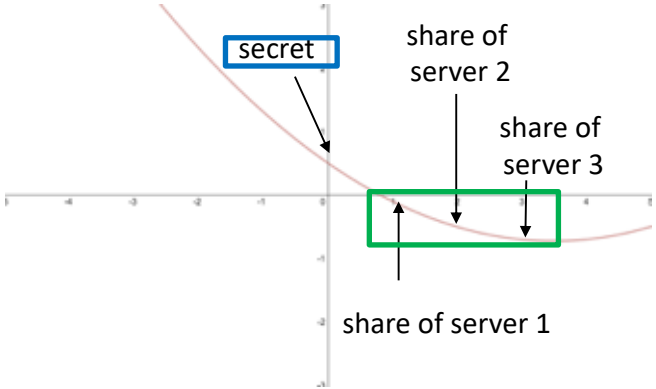
  i. ***Share:*** for secret *s*, pick random coefficients $a_1 \ldots a_t$ & set $a_0 = s$ $p(x) = a_0 + a_1x + a_2x^2 + \ldots a_tx^t$ distribute shares as *p(1), p(2) … p(n)* to the *n* servers

  ii. ***Reconstruct:*** from *p(1), p(2) … p(t+1)* interpolate *p(x)* and recover secret as $p(0) = a_0 = s$

Secret Sharing: Share Algorithm

```python
import string

from six import integer_types
from utilitybelt import int_to_charset, charset_to_int, base58_chars, \
    base32_chars, zbase32_chars
from .primes import get_large_enough_prime
from .polynomials import random_polynomial, \
get_polynomial_points, modular_lagrange_interpolation



def share_secret(secret_int, point_threshold, num_points, prime=None):
    """ Split a secret (integer) into shares (pair of integers / x,y coords).
        Sample the points of a random polynomial with the y intercept equal to
        the secret int.
    """
    if point_threshold < 2:
        raise ValueError("Threshold must be >= 2.")
    if point_threshold > num_points:
        raise ValueError("Threshold must be < the total number of points.")
    if not prime:
        prime = get_large_enough_prime([secret_int, num_points])
    if not prime:
        raise ValueError("Error! Secret is too long for share calculation!")
    coefficients = random_polynomial(point_threshold-1, secret_int, prime)
    points = get_polynomial_points(coefficients, num_points, prime)
return points

```

$p(x) = 0.5 - 0.7x + 0.1x^2$

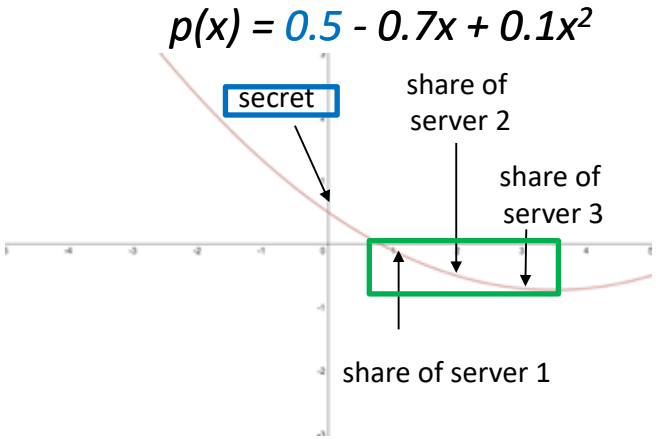


Share algorithm in code sample called:
share_secret(…)

Source code sample form: https://github.com/blockstack/secret-sharing

Secret Sharing: Reconstruct Algorithm

```
32
33
34  def reconstruct_secret(points, prime=None):
35      """ Join int points into a secret int.
36          Get the intercept of a random polynomial defined by the given points.
37      """
38      if not isinstance(points, list):
39          raise ValueError("Points must be in list form.")
40      for point in points:
41          if not isinstance(point, tuple) and len(point) == 2:
42              raise ValueError("Each point must be a tuple of two values.")
43          if not (isinstance(point[0], integer_types) and
44                  isinstance(point[1], integer_types)):
45              raise ValueError("Each value in the point must be an int.")
46      x_values, y_values = zip(*points)
47      if not prime:
48          prime = get_large_enough_prime(y_values)
49      free_coefficient = modular_lagrange_interpolation(0, points, prime)
50      secret_int = free_coefficient  # the secret int is the free coefficient
51  return secret_int
52
```

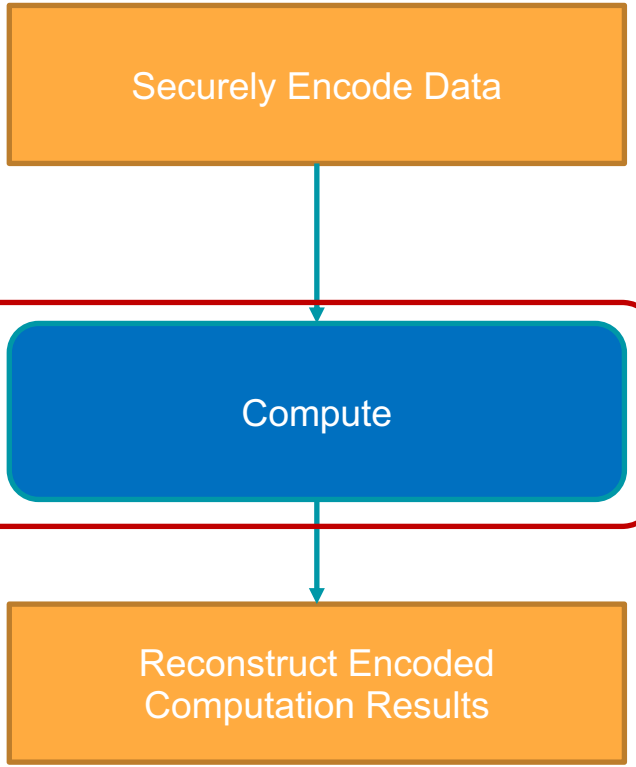


Reconstruct algorithm in code sample called:
reconstruct_secret(…)

Source code sample form: https://github.com/blockstack/secret-sharing

Blueprint for MPC

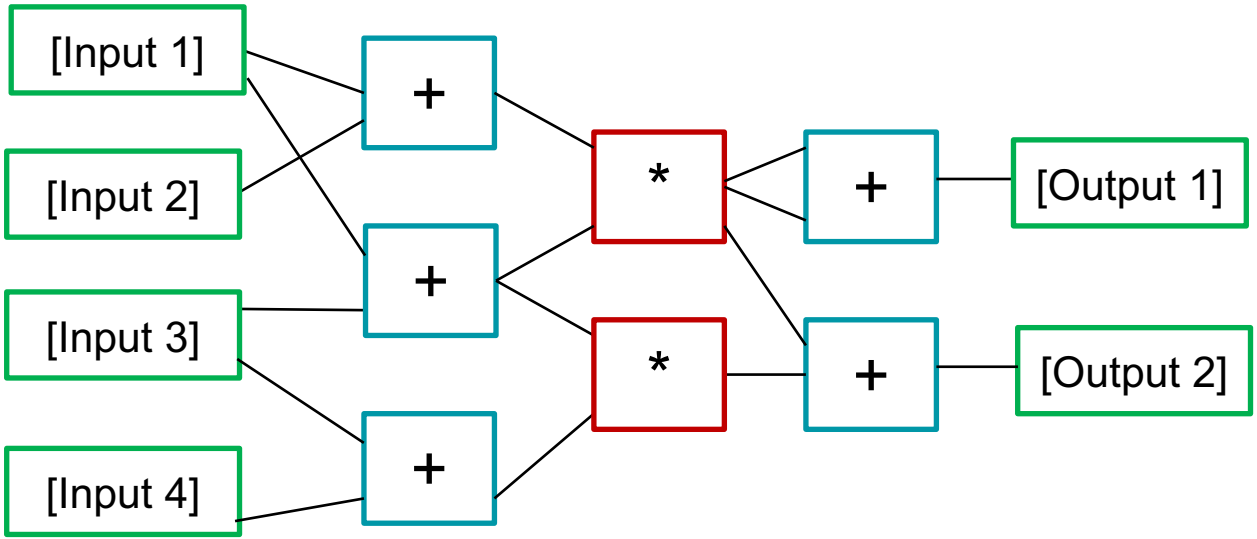
- Encode data into a (secure) distributed format
  - e.g., using the share algorithm in a secret sharing scheme
- Perform computation on securely encoded distributed data
- Reconstruct encoded results
  - e.g., using the reconstruct algorithm in a secret sharing scheme



NEXT

Computing Using Secret Shared Data

- One way to implement MPC
- Computation represented as an arithmetic circuit
- Circuit consists of addition and multiplication of secret shares
- [Input X] denotes secret sharing of Input X



Realizing Computation Gates

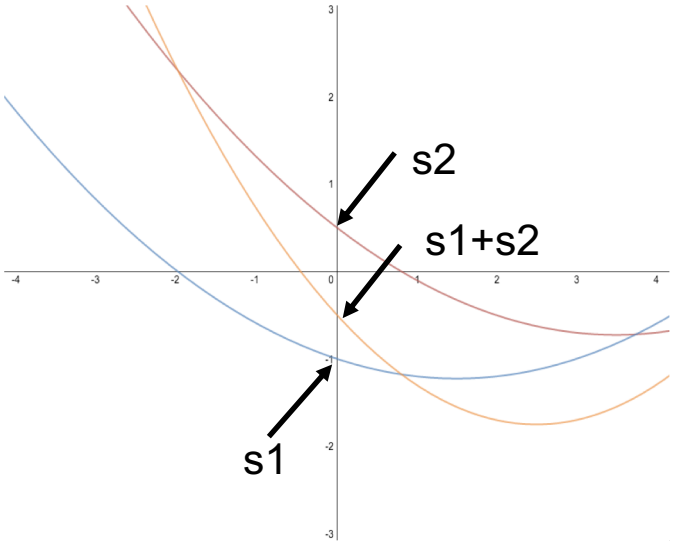
- **Addition:** $[s_1] + [s_2] = [s_1+s_2]$

if $f_1(x) = a_0 + a_1x + a_2x^2 + \ldots a_tx^t$

and $f_2(x) = b_0 + b_1x + b_2x^2 + \ldots b_tx^t$

then

$f_1(x) + f_2(x) = (a_0 + b_0) + (a_1 + b_1)\, x + \ldots + (a_t + b_t)\, x^t$



- **Multiplication:** resulting polynomial has double degree!

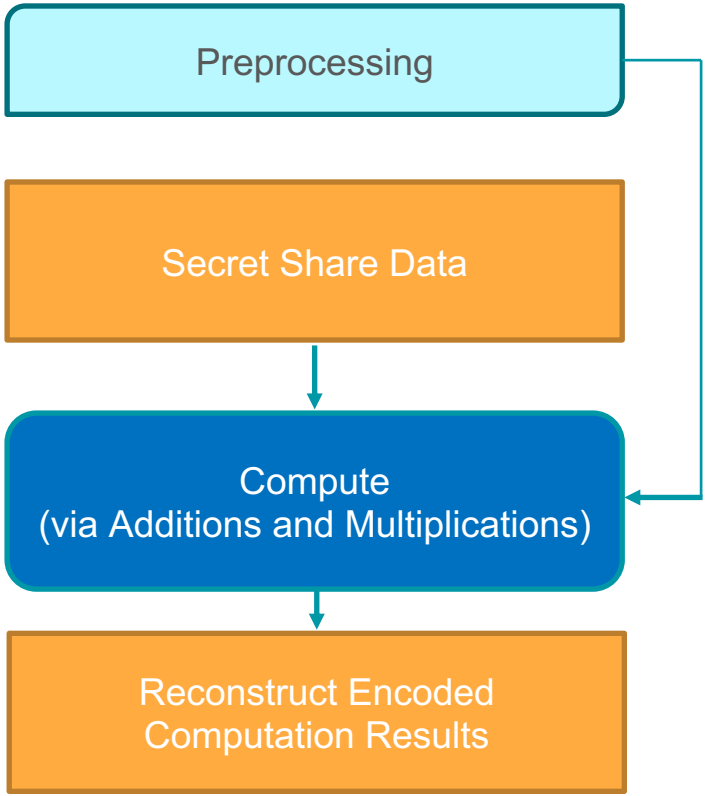
$f_1(x) * f_2(x) = (a_0\, b_0) + (b_0\, a_1 + a_0\, b_1)\, x + \ldots + (a_t\, b_t)\, x^{2t}$

How to Multiply Using Pre-computation

- Generate random triples *(a,b,c)* such that *c=a*b* and share them *([a],[b],[c])*
- To multiply two shares $[s_1]$ and $[s_2]$ each server computes:
  - $[s_1 - a]$ and reveals it — $i = reconstruct([s_1 - a])$
  - $[s_2 - b]$ and reveals it — $k = reconstruct([s_2 - b])$
  - Servers compute their share of the product $[s_1 x s_2] = i*k + i*[b] + k*[a] + [c]$
  - Why it works:
    $(s_1 - a)*(s_2 - b) + (s_1 - a)*b + (s_2 - b)*a + a*b =$
    $s_1*s_2 - a*s_2 - b*s_1 + a*b + b*s_1 - a*b + a*s_2 - a*b + a*b =$
    s_1*s_2

Recap of MPC's Operation

- Preprocessing to set up some parameters (e.g., multiplication triples)
- Parties secret share their data
- Parties compute (via additions and multiplications) on the secret shared data
- Parties reconstruct the result of the computation



Putting It All Together

- Implementing a basic MPC class using the Charm library
- Charm is a Python framework for rapidly prototyping advanced cryptographic protocols and primitives
- https://github.com/JHUISI/charm

Note: sending shares to other parties in mpcReconstruct() is left out, this code assumes all shares are already received.

```python
from charm.toolbox.pairinggroup import PairingGroup,ZR,order
from charm.toolbox.secretshare import *

class MPC:

    def __init__(self, threshold, num_parties, field_size):
        self.num_parties = int(num_parties)
        self.threshold = int(threshold)
        self.field_size = field_size # for now will only use 512 bits
        self.field = PairingGroup('SS512')
        self.ssObject = SecretShare(self.field, verbose_status=True)


    def mpcShare(self, ip_to_share):
        print(self.threshold)
        print(type(self.num_parties))
        shares = self.ssObject.genShares(ip_to_share, self.threshold, self.num_parties)
        return shares


    # Function to add shares (assumed to be given in a list)
    def mpcAdd(self, inputs):
        sum = self.field.init(ZR, 0)
        print('Placeholder for mpcAdd function.\n')
        for i in range(0, len(inputs)):
            sum += inputs[i]

        return sum


    def mpcMult(self, shares):
        print('Placeholder for mpcMult function.\n')
        return


    def mpcReconstruct(self, shares):
        print('Placeholder for recon function.\n')
        print(shares)
        dictionary_points={}
        for i in range(1,self.num_parties+1):
            dictionary_points[self.field.init(ZR, i)]=shares[i]

        recoveredSecret=self.ssObject.recoverSecret(dictionary_points)

        return recoveredSecret
```

Multiplication Algorithm Pseudocode

- Assuming that a random triple (a, b ,c) where c=a*b is generated as part of the setup
- Each party gets a share of the triple (a, b ,c), i.e., ([a], [b] ,[c])
- Recall that the product is:

$(s_1 - a)*(s_2 - b) + (s_1 - a)*b + (s_2 - b)*a + a*b$

```
31
32      # Function to multiply shares (assumed to be given in a list)
33      def mpcMult(self, shares, triple):
34          print('Placeholder for mpcMult function.\n')
35
36          s1 = shares[0]
37          s2 = shares[1]
38
39          a = triple[0]
40          b = triple[1]
41          c = triple[2]
42
43          tmp1 = self.mpcAdd([s1, -a])
44          tmp2 = self.mpcAdd([s2, -b])
45
46          i = self.mpcReconstruct(tmp1)
47          k = self.mpcReconstruct(tmp2)
48
49          product_share = (i*k) + (i*b) + (k*a) + c
50
51          return product_share
52
53
```

Note: sending shares to other parties in mpcReconstruct() is left out, this code assumes all shares are already received.

Example Open Source MPC Software

- SCALE-MAMBA: https://github.com/KULeuven-COSIC/SCALE-MAMBA

- Two main components to the system
  - a run time system called Secure Computation Algorithms from LEuven (SCALE)
  - a Python-like programming language called Multiparty AlgorithMs Basic Argot* (MAMBA**)

- Example on right from page 18 in documentation: https://homes.esat.kuleuven.be/~nsmart/SCALE/Documentation.pdf

```
# (C) 2017 University of Bristol. See License.txt
# (C) 2018 KU Leuven. See License.txt

def test(actual, expected):

actual = actual.reveal()

print_ln('expected %s, got %s', expected, actual)
# cint: clear integers modulo p
# sint: secret integers modulo p

a = sint(1)
b = cint(2)

test(a + b, 3)
test(a + a, 2)
test(a * b, 2)
test(a * a, 1)
test(a - b, -1)
test(a < b, 1)
test(a <= b, 1)
test(a >= b, 0)
test(a > b, 0)
test(a == b, 0)
test(a != b, 1)
clear_a = a.reveal()

# arrays and loops
a = Array(100, sint)

@for_range(100)
def f(i):
a[i] = sint(i)**2
test(a[99], 99**2)

# conditional
if_then(cint(0))
a[0] = 123
else_then()
a[0] = 789
end_if()

test(a[0], 789)
```

* Argot means secret language
** MAMBA is a snake (similar to Python)

Advanced Topics

- Dealing with misbehaving parties
  - Semi-honest (honest-but-curious): do not modify software but want to violate privacy
  - Malicious model: modify software and can cheat
- Scalability:
  - Large datasets: 100s of GBs or more
  - Complex functions: machine learning, database queries, numerical analysis
  - Larger number of parties: 10s or even 100s of parties

Other Techniques

- Secure Two-party Computation
  - Specialized techniques if computation is only between two parties (e.g., two companies)
  - Suitable for cloud settings
- Fully Homomorphic Encryption
  - Suitable for cloud settings (offloading computation to the cloud), or when interaction is not desired
- Differential Privacy
  - For computing and revealing statistics over databases while preserving privacy

Summary

- Never invent your own cryptography or privacy techniques!
- We do not have to reveal data to use it in (online) computation
- Advanced cryptographic techniques are now becoming increasingly practical
- Open source frameworks and libraries are becoming mature
- More work required to improve usability for non-experts
- Starting to be used in live practical applications

Resources

- Further reading:
  - Free book overviewing different MPC techniques (the protocols): https://www.cs.virginia.edu/~evans/pragmaticmpc/pragmaticmpc.pdf
  - More books, tutorials, articles and videos: https://github.com/rdragos/awesome-mpc

- Some mature and usable open source software for MPC:
  - Python based - SCALE-MAMBA: https://github.com/KULeuven-COSIC/SCALE-MAMBA
  - Java based - JIFF: https://github.com/multiparty/jiff

Questions


codechix
DevPulseCon