PrivPy: Scalable and General Privacy-Preserving Data Mining

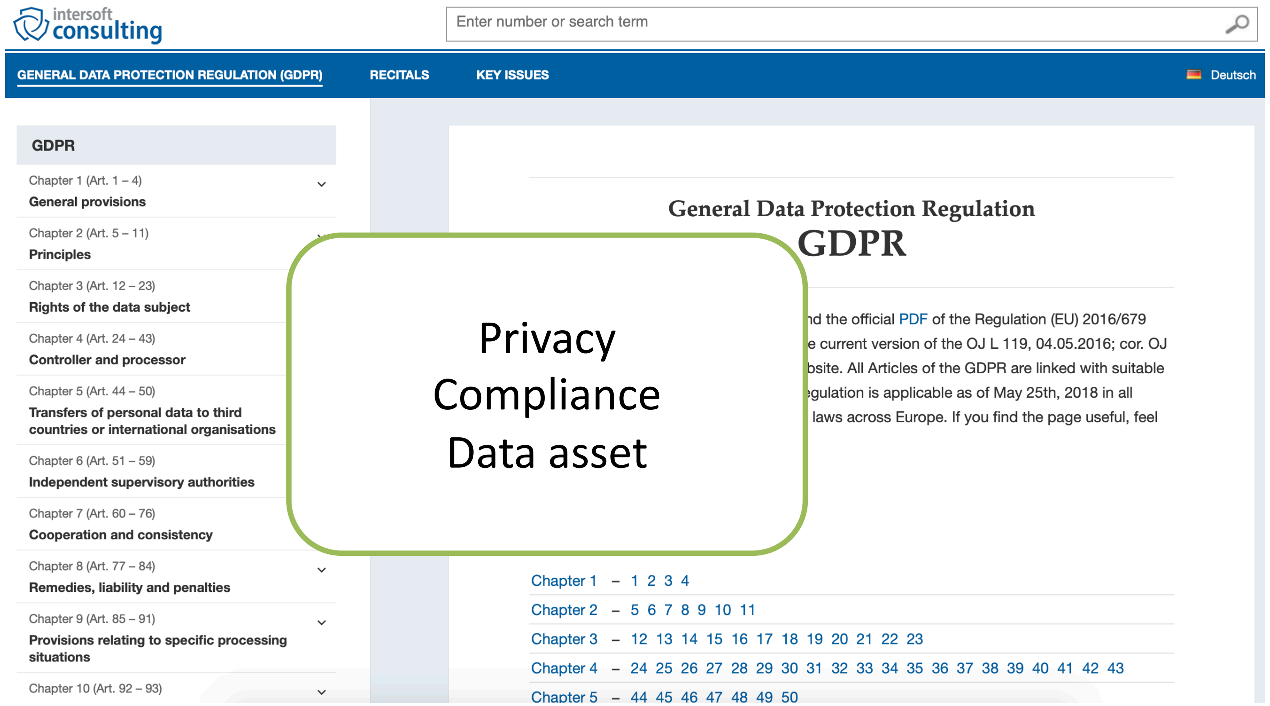
Yi Li*, Yitao Duan†, Yu Yu §, Shouyao Zhao §, Wei Xu*

* Institute for Interdisciplinary Information Sciences, Tsinghua University

† NetEase Youdao

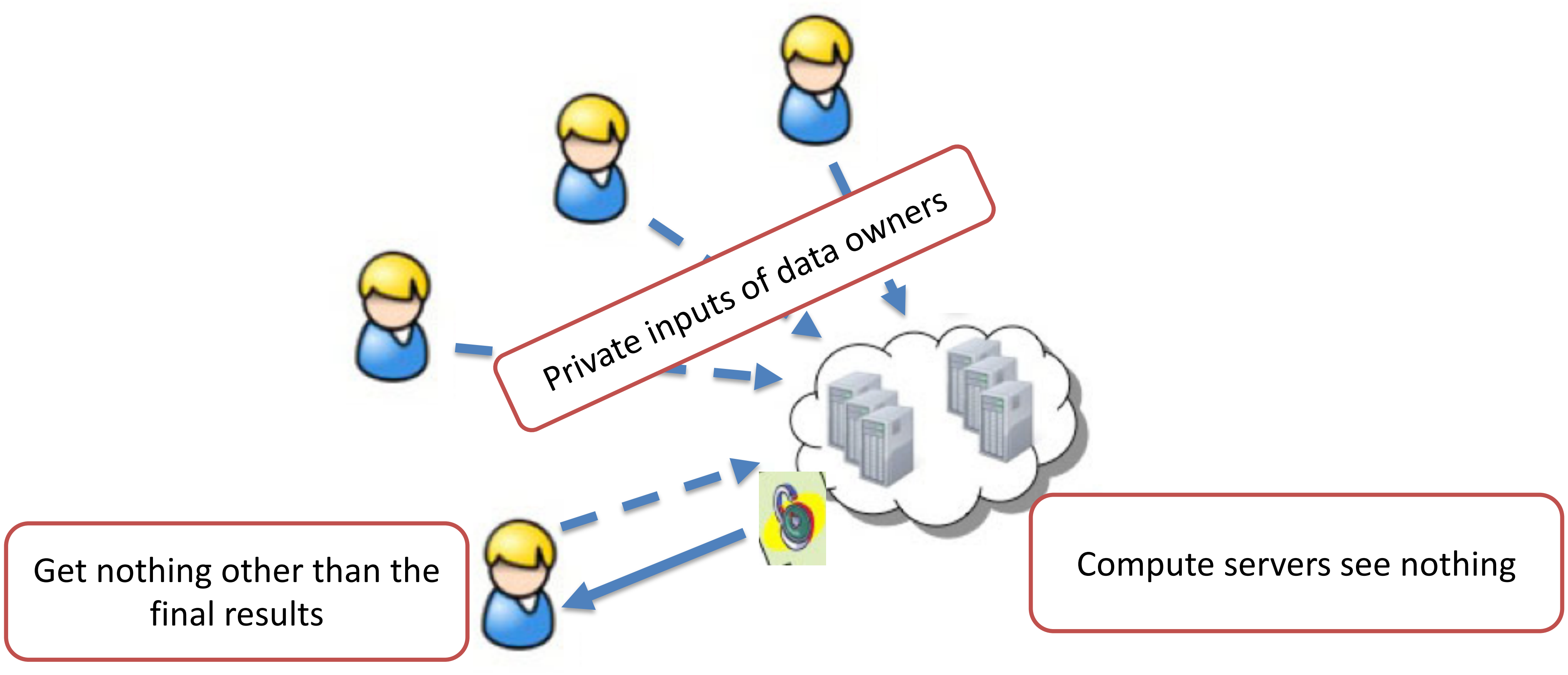
§ Shanghai Jiaotong University

Making use of data vs. data privacy

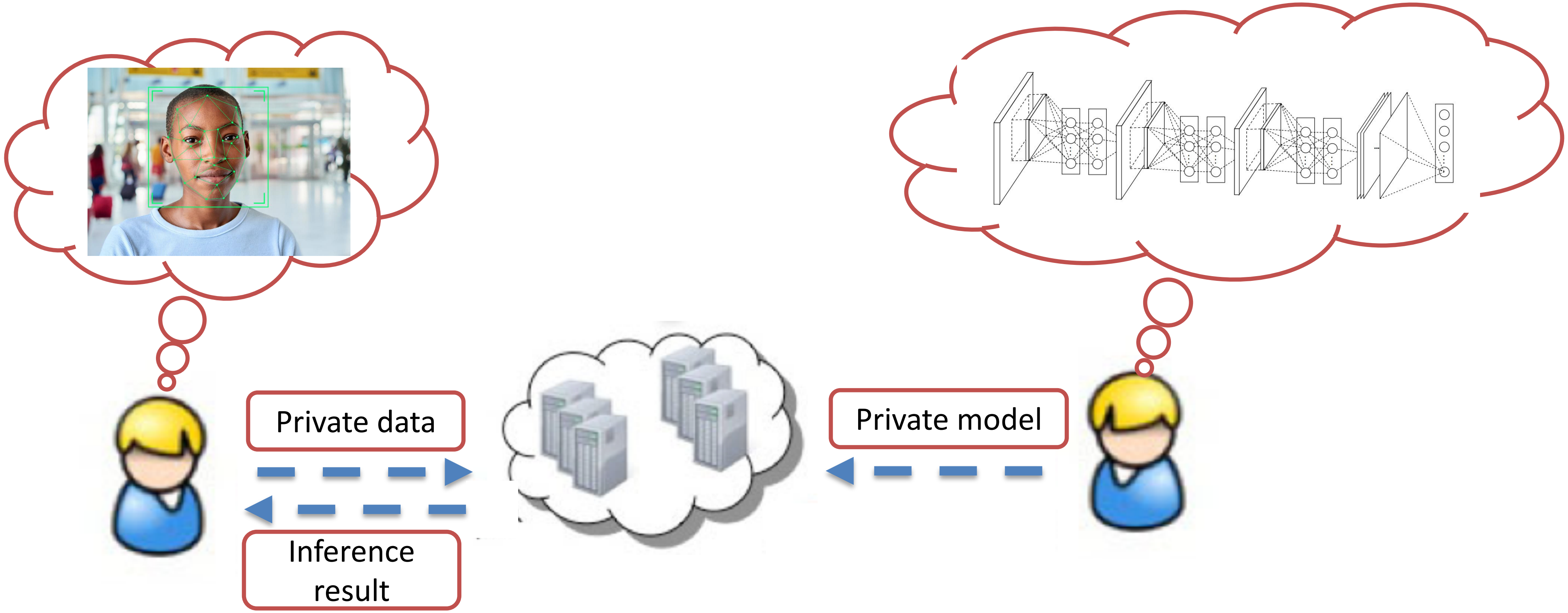


Privacy
Compliance
Data asset

Scenario 1: Multi-source data mining



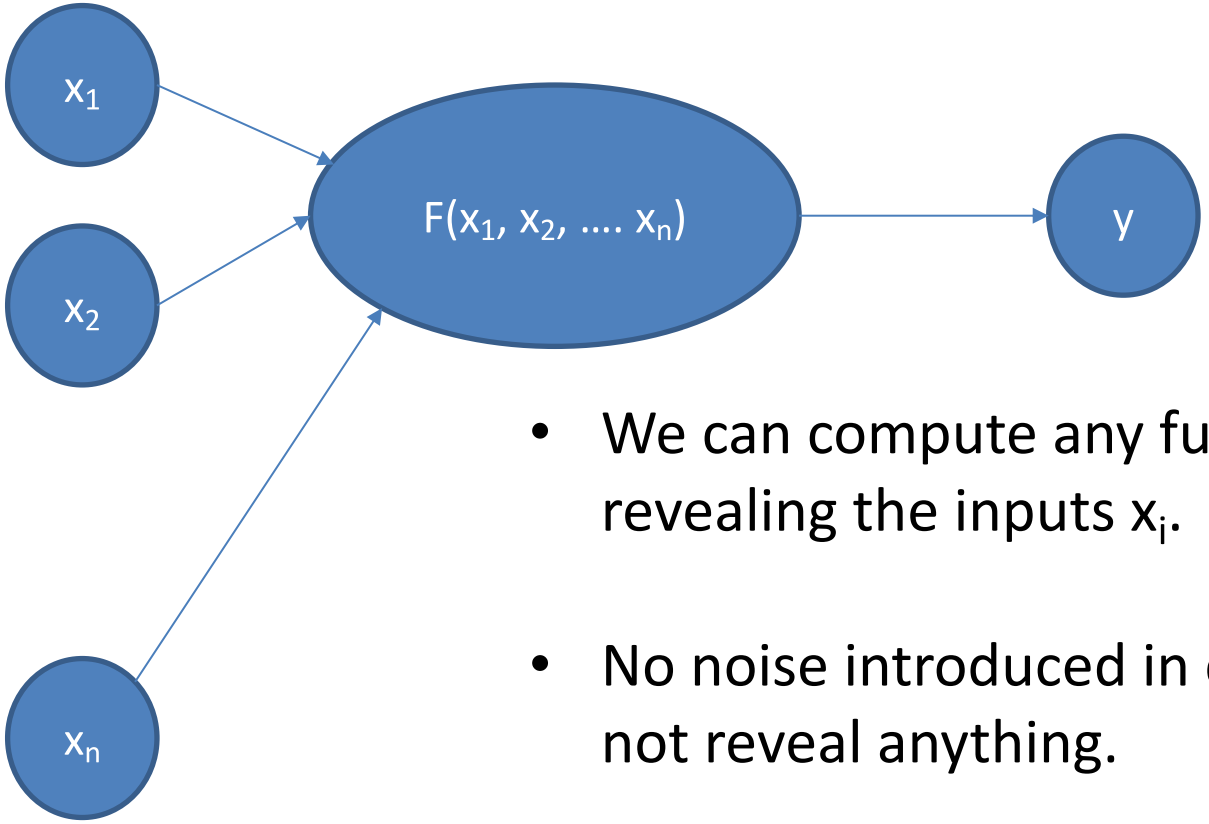
Scenario 2: Inference w/ secret models and data



Similar setting: federated learning, but want to protect the model itself.

A nice theory provide solution

◆ Secure multi-party computation (MPC)



- We can compute any function F() without revealing the inputs x_i.
- No noise introduced in computation, and do not reveal anything.

Tons of cryptography-based solutions tell us …

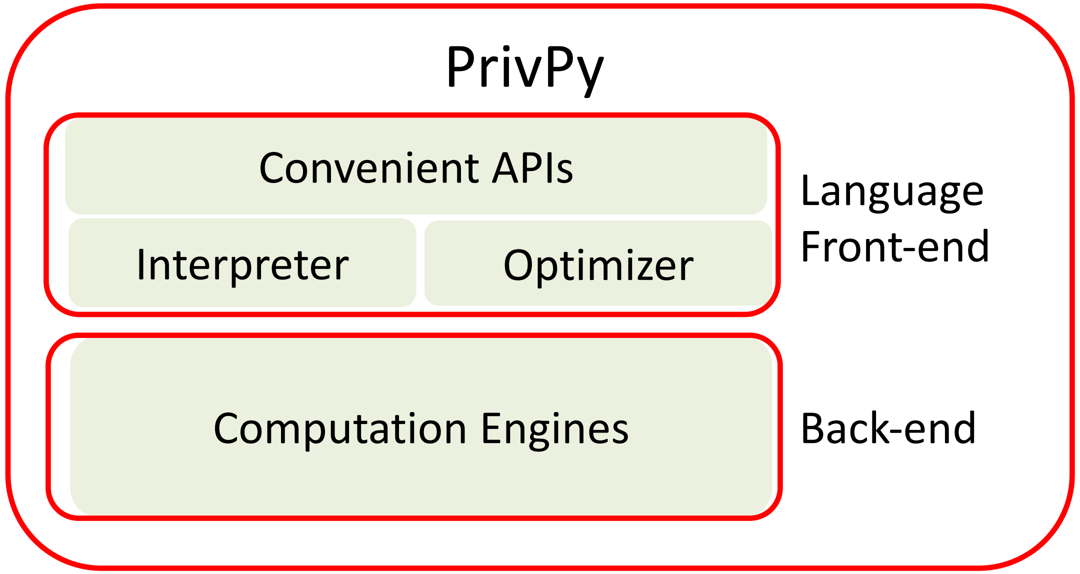
- Many novel theoretical solutions
  - Secret Sharing (Shamir 1979)
  - Garbled Circuit (Yao 1986)
  - Fully Homomorphic Encryption (Gentry 2009)
- Even many “practical” solutions exist
  - Sharemind (2008)
  - TASTY (2010)
  - PICCO (2013)
  - SPDZ (2008)
  - SecureML(2017)
  - ABY3(2018)
- But, why people still not using it to mine real world data?

The gap between cryptography and data science

The Cryptography World	The Data Science World
• Efficient bit-wise and integer operations	• Efficient operations on real numbers
• Fast single number arithmetic	• Fast vector and array operations
• Theoretically innovative	• Scalable system implementation
• A custom and beautiful programming language	• Familiar language with rich algorithm libraries

The gap is like
a set of data structures v.s. a relational database

PrivPy attempts to bridge the gap



- A fast (4,2)-secret-sharing protocol and engine
- Python language with automatic code optimizer
- NumPy types and libraries
- Runs non-trivial algorithms on real data

Crypto preliminary: basic secret sharing

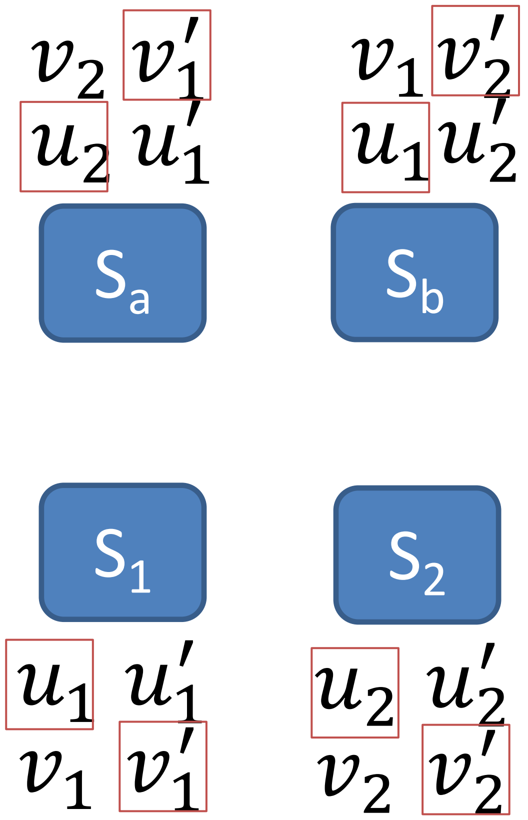
- Two semi-honest servers: S_1 and S_2
- A large (e.g. 256 bits) number p
- Computation in the field $\phi_p = \{0, 1, \ldots, p\text{-}1\}$

S_1 S_2

$$u = u_1 + u_2$$

$\varphi(u) = (u_1, u_2)$
u_1: uniformly distributed in ϕ_p
$u_2 := u - u_1 \pmod{p}$

Multiplication: Our $\binom{4}{2}$-secret sharing scheme



- Two auxiliary servers S_a and S_b to compute the cross terms
- Benefit: one round of communication only for $\times$

$$w = u \times v = \underbrace{\boxed{u_1 v_1'}}_{t_1} + \underbrace{\boxed{u_2 v_2'}}_{t_2} + \underbrace{\boxed{u_2 v_1'}}_{t_a} + \underbrace{\boxed{u_1 v_2'}}_{t_b}$$

Using fixed-point to represent real numbers

$$\tilde{x} = \begin{cases} \lfloor kx \rfloor & x \geq 0 \\ \lfloor kx \rfloor + \phi & x < 0 \end{cases}$$

$\boxed{010010011100100}.\boxed{11011001001}\ldots$

Fixed-length $l - k$ Integer part

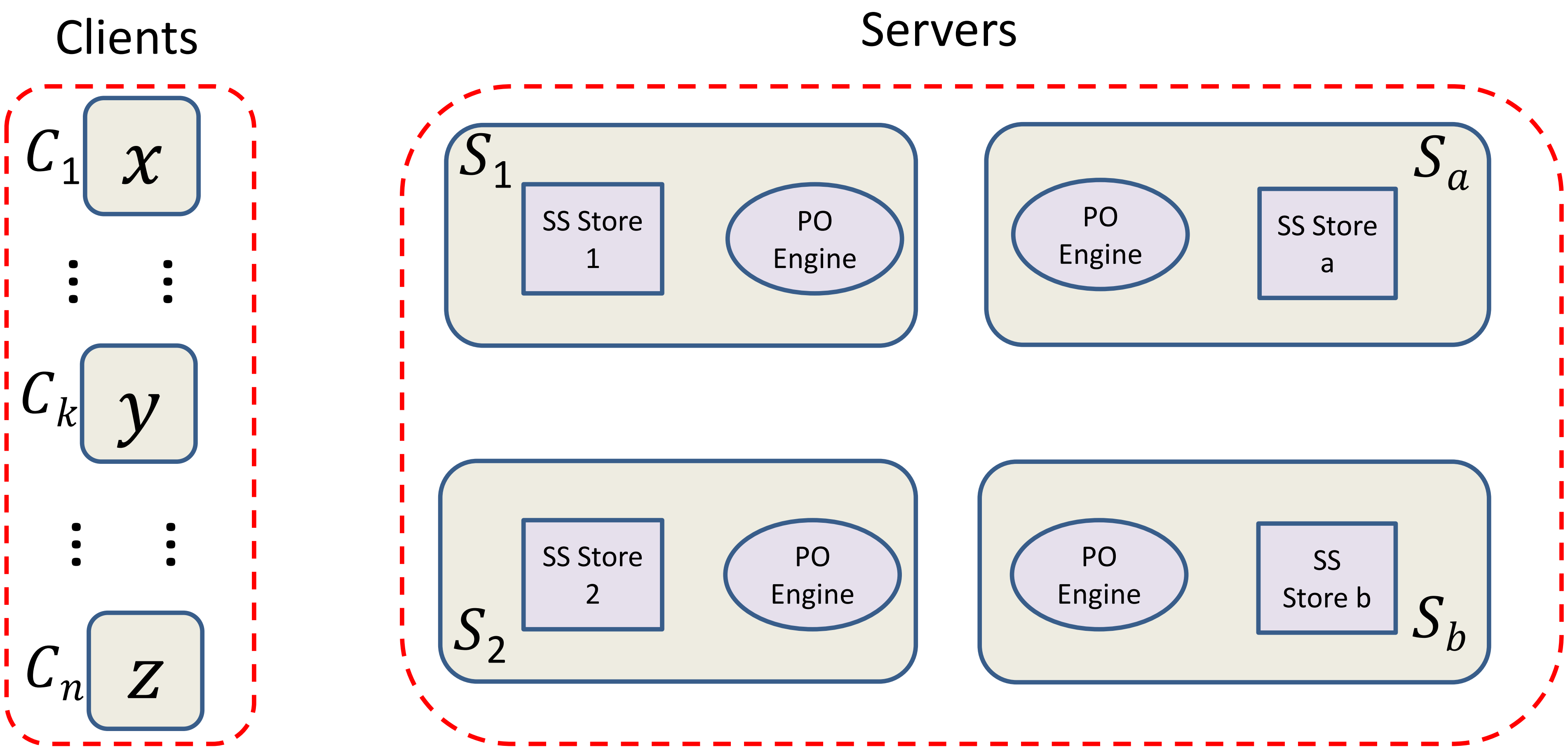
Fixed-length k decimal part

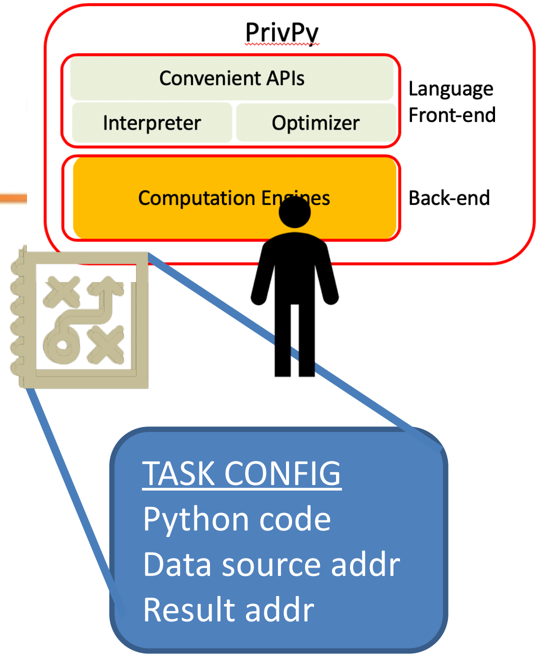
$\Downarrow$

010010011100100 11011001001

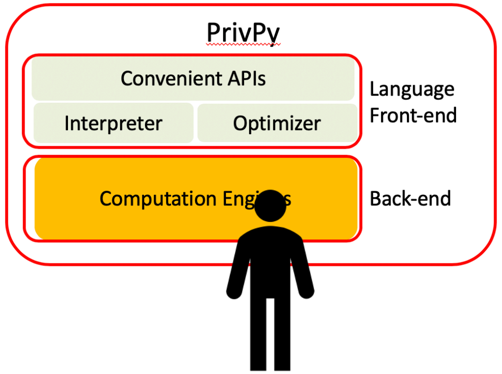
- Use expensive bit-level operations
  - PICCO, Sharemind, SPDZ, etc
- Support built-in fixed-point operations
  - SecureML, ABY3, PrivPy

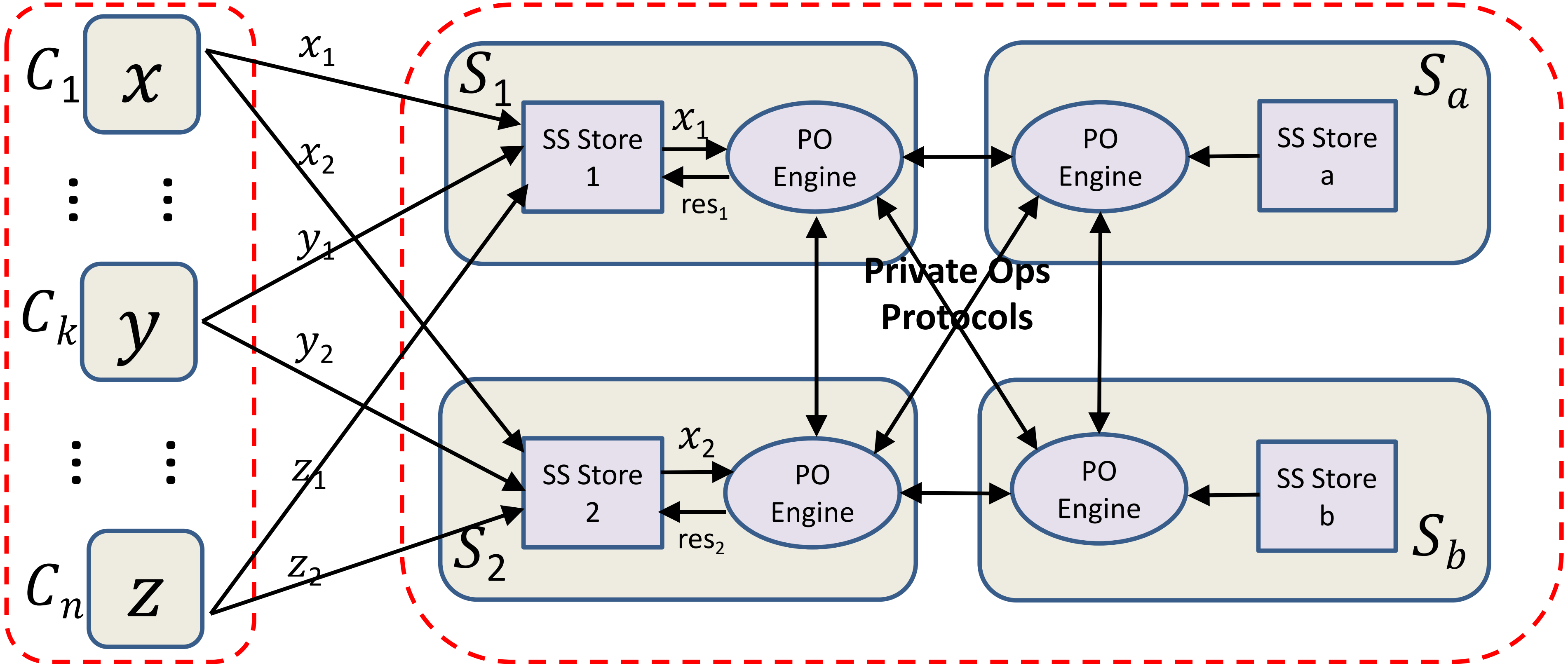
The PrivPy computation engine





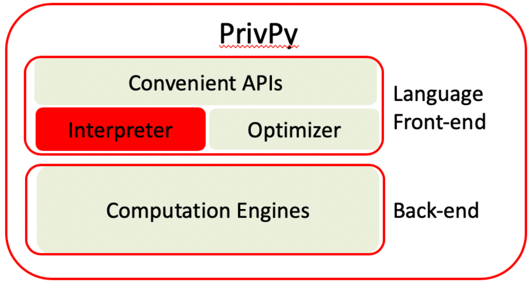
The PrivPy computation engine





$res_1 + res_2 = res$

Python compatible programming front-end



- Overload basic operations for private variables: +, -, ×, >, etc

```
# private data declaration
x = privpy.ss(clientID)
# the code to execute on servers
def logistic(x, start, iter_cnt):
  result = 1.0 / (1 + math.exp(-start))
  deltaX = (x - start) / iter_cnt
  for i in range(iter_cnt):
    derivate = result * (1 - result)
    result += deltaX * derivate
  return result
result = logistic(x, 0, 100) # main()
#reveal results on clients
result.reveal()
```

Most existing solutions define their own language

```
public int main() {
   public int i, M;
   smcinput(M, 1, 1);
   private int<1> A[M], B[M];
   private int<10> dist = 0;

   smcinput(A, 1, M);
   smcinput(B, 1, M);
   for (i = 0; i < M; i++)
     dist += A[i] ^ B[i];

   smcoutput(dist, 1);
   return 0;
}
```

PICCO

```
#include <million.h>
#include <obliv.oh>

void millionaire (void *args) {
    ProtocolIO *io = args;
    obliv int a, b;
    obliv bool res = false;
    a = feedOblivInt (io->myinput, 1)
    b = feedOblivInt (io->myinput, 2)
    obliv if (a < b) res = true;
    revealOblivBool (&io->result, res, 0);
}
```

OblivC

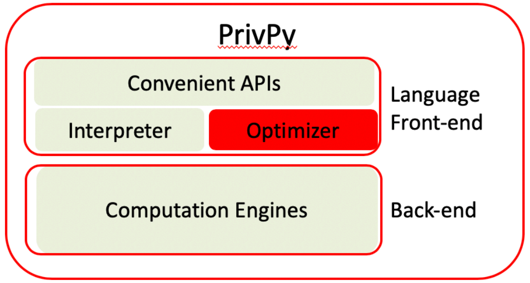
```
intersection = Array(n, sint)
is_match_at = Array(n, sint)

@for_range(n)
def _(i):
    @for_range(n)
    def _(j):
        match = a[i] == b[j]
        is_match_at[i] += match
        intersection[i] = if_else(match, a[i], intersection[i])
```

SPDZ

Why? Many pitfalls if written in Python resulting in inefficiency.

AST-level code optimization to avoid pitfalls

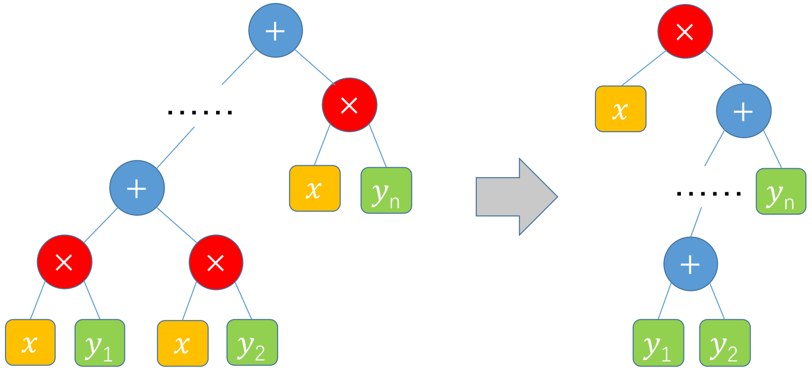


Common factor

$$x * y_1 + x * y_2 + \cdots + x * y_n$$

$$\Downarrow$$

$$x * (y_1 + y_2 + \cdots + y_n)$$



Auto vectorization

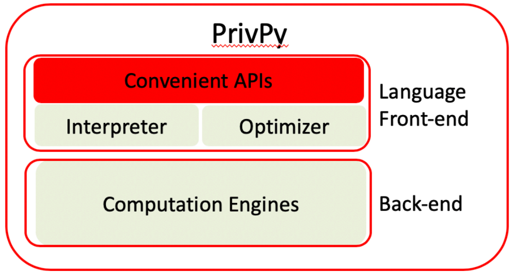
$$x_1 * y_1 + x_2 * y_2 + \cdots + x_n * y_n$$

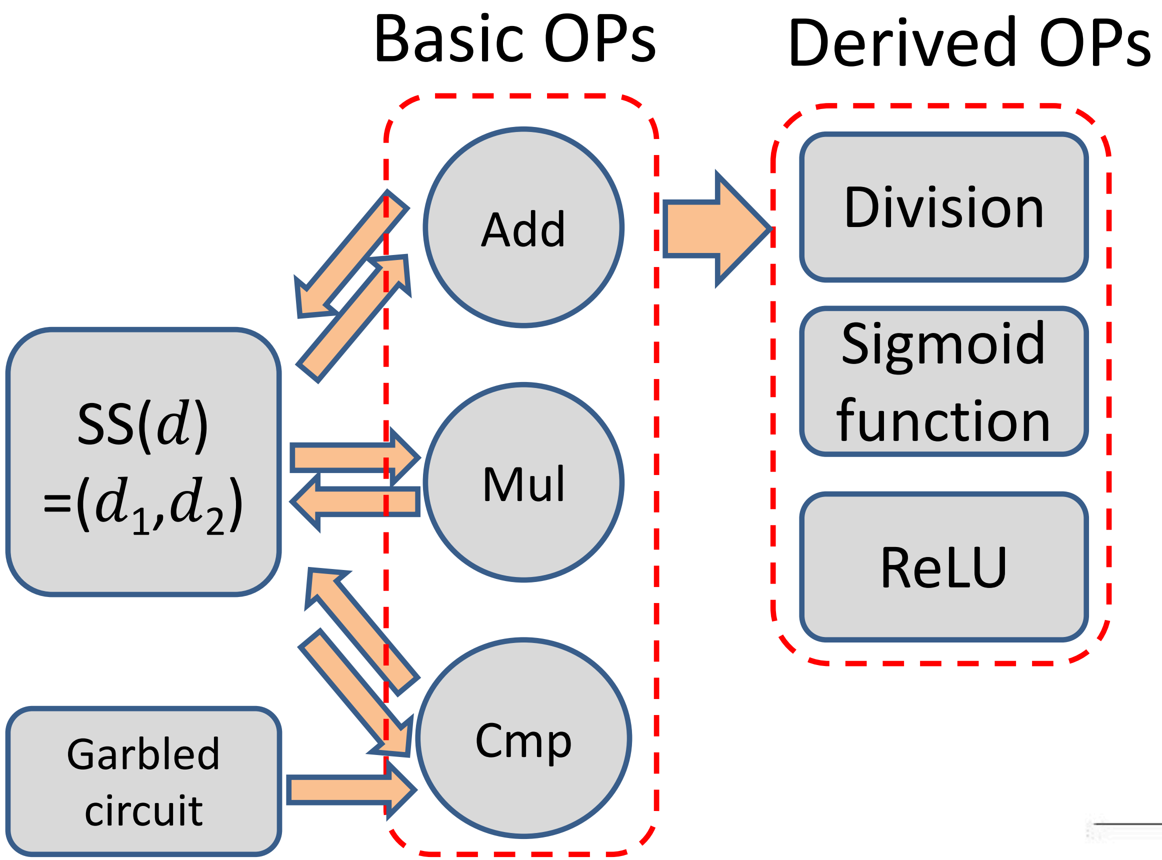
$$\Downarrow$$

$$\vec{x} = (x_1, x_2, \ldots, x_n) \times \vec{y} = (y_1, y_2, \ldots, y_n)$$

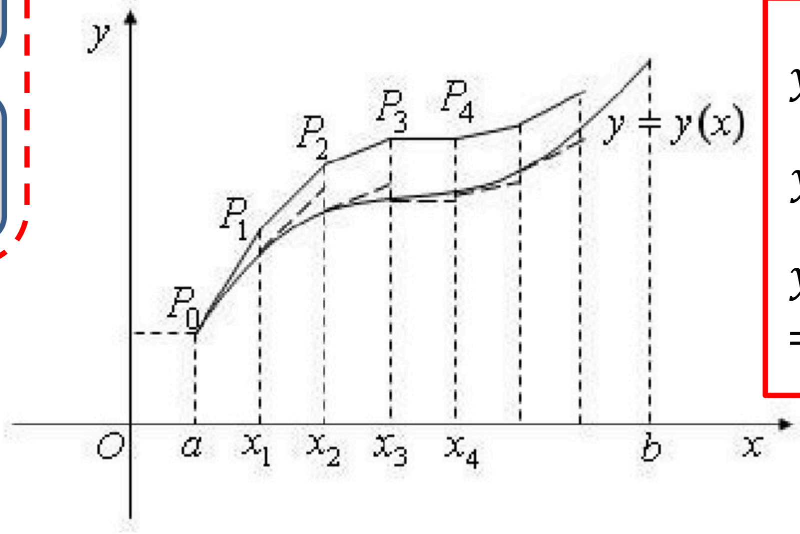
Still adding more optimizations to the language frontend.

APIs: from basic OPs to algorithms





- Division: Newton-Raphson method
- Sigmoid: Euler Method
- ReLu: comparison
- Other functions: e^x, log(x), …

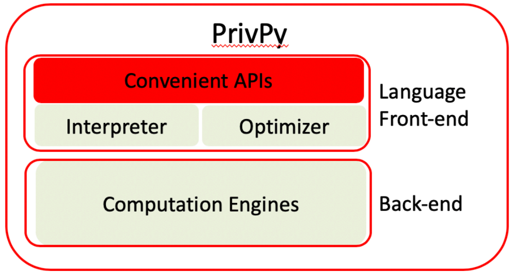


$$y(x) = \frac{1}{1 + e^{-x}}$$

$$y'(x) = y(x)(1 - y(x))$$

$$y(x_{t+1}) = y(x_t) + y'(x_t)\Delta x$$
$$= y(x_t) + y(x_t)\big(1 - y(x_t)\big)\Delta x$$

APIs: arrays are first-class citizen

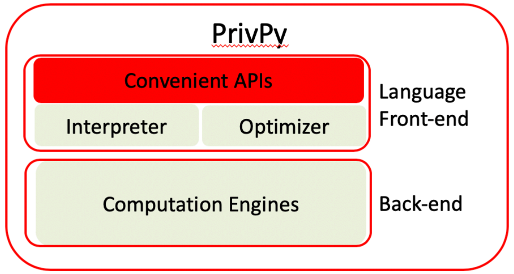


- Array is a built-in type
  - $A = pp.sarr([\ldots]); B = pp.sarr([\ldots])$
  - Both $A * B$ and $A + B$ work

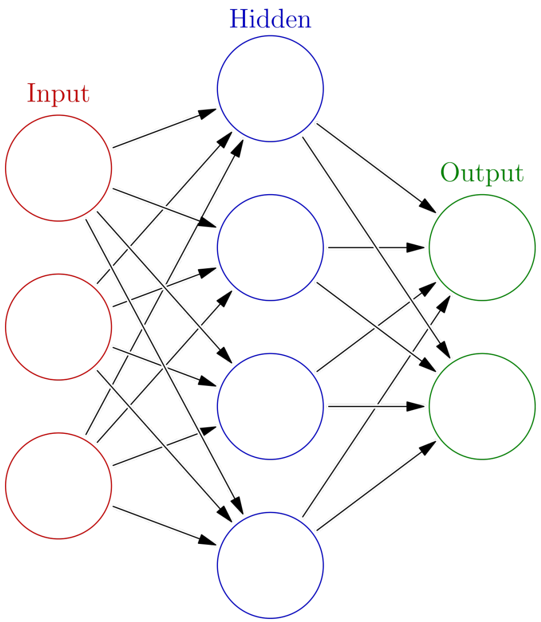
- Array type is essential for data mining: reduces # of ops, thus # of rounds

- Support **large arrays** (e.g. 1 million $\times$ 5000, ~200GB) using automatic disk buffer management

Beyond arrays: *NumPy's* broadcasting and ndarray



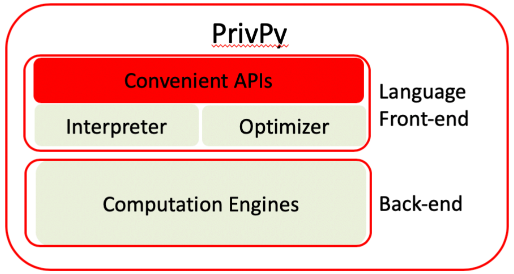
- Allow operations between arrays of different shapes
  - E.g.
    - 12d-scalar x, a 3 * 4 array A and a 2 * 3 * 4 array B
    - $x + A$, $A * B$ and $x > B$ all work
    - Can even mix plaintext and cipher text



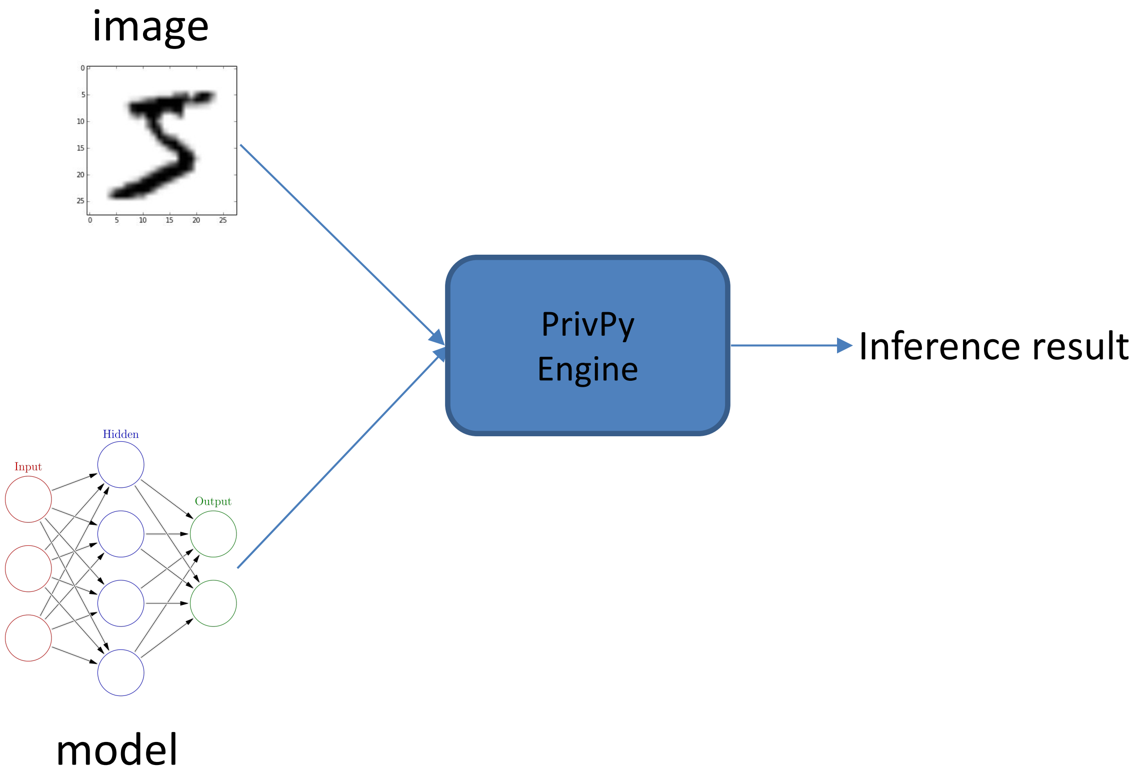
$$y = f(w^{\intercal} \cdot X + b)$$

- Ndarray methods

all	any	append	argmax
argmin	argparition	argsort	clip
compress	copy	cumprod	cumsum
diag	dot	fill	flatten
item	itemset	max	mean
min	ones	outer	partition
prod	ptp	put	ravel
repeat	reshape	resize	searchsorted
sort	squeeze	std	sum
swapaxes	take	tile	trace
transpose	var	zeros	



API example: neural network inference




```python
import privpy as pp
x = ... # read data using ss()
W, b = ... # read model using ss()
for i in range(len(W)):
  x = pp.dot(W.T, x) + b
  x = pp.relu(x)
res = pp.argmax(x, axis=1)
res.reveal()
```

Basic operation performance

Throughput of basic operations (ops per second)

Engine	Approach	LAN (10Gbps)	
		decimal multiplication	comparison
PrivPy	SS	10,473,532	1,282,027
Helib	FHE	258	-
Obliv-C	GC	3,930	78,431
P4P+HE	SS+HE	4,344	-
SPDZ	SS with active security	83,073	20,472
SPDZ+PrivPy	SS with active security	83,229	20,320

Our thin wrapper

Real world algorithm performance

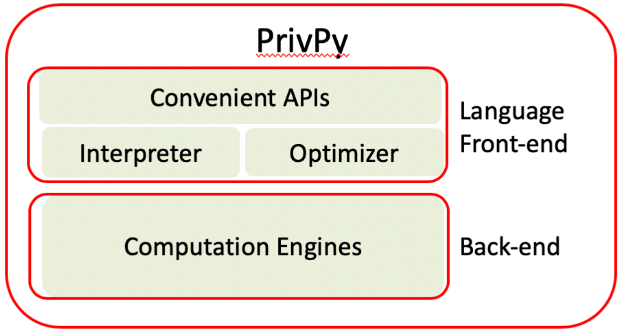
Dataset: MNIST with 70,000 labeled handwritten digits
Algorithm:
- **Logistic Regression (LR)**: trained using SGD
- **Matrix Factorization (MF)**: decomposes a $m \times n$ matrix to a $m \times 5$ matrix and a $5 \times n$ matrix
- **CNN**: LeNet-5

Time of training/inference for 1 iteration (seconds)

Batch size	**LAN (10Gbps)**			**WAN (50Mbps)**		
	LR training	MF training	CNN inference	LR training	MF training	CNN inference
Single op	5.3e-3	7.1e-3	9.6e-2	2.61	0.37	7.64
Batch (1000 ops)	3.92	5.67	12.02	7.3	13.2	56.3

Conclusion and future work

- MPC can be useful in data mining, but big gap to bridge
- PrivPy is an early attempt to make MPC practical for large datasets
  - Language, data types, function libraries
  - Scalable and efficient system implementation
  - Heavily rely on language-level optimizations



- PrivPy is an on-going effort
  - Integrating with other privacy-preserving techniques – differential privacy, federated learning, trusted execution etc.
  - More libraries, algorithms and compiler optimizations

Wei Xu –http://iiis.tsinghua.edu.cn/~weixu
Yi Li – xiaolixiaoyi@gmail.com