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

[Diagram showing the relationship between data, customer, and various interactions through voice, email, chat, and mobile]

- **Privacy Compliance**
- **Data asset**
Scenario 1: Multi-source data mining

- Private inputs of data owners
- Compute servers see nothing
- Get nothing other than the final results
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

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

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 $\phi_p = \{0, 1, ..., p-1\}$

$$u = u_1 + u_2$$

$$\varphi(u) = (u_1, u_2)$$

- $u_1$: uniformly distributed in $\phi_p$
- $u_2$: $u - u_1 \mod p$
Multiplication: Our \( \binom{4}{2} \)-secret sharing scheme

\[
\begin{align*}
\nu_2 \nu'_1 & \quad \nu_1 \nu'_2 \\
\mu_2 & \quad \mu_1
\end{align*}
\]

\[\text{S}_a \quad \text{S}_b\]

\[
\begin{align*}
\nu_1 & \quad \nu'_1 \\
\nu_2 & \quad \nu'_2
\end{align*}
\]

\[\text{S}_1 \quad \text{S}_2\]

\[
\begin{align*}
\mu_1 & \quad \mu'_1 \\
\mu_2 & \quad \mu'_2
\end{align*}
\]

- Two auxiliary servers \( \text{S}_a \) and \( \text{S}_b \) to compute the cross terms
- Benefit: one round of communication only for \( \times \)

\[
w = u \times v = u_1 \nu'_1 t_1 + u_2 \nu'_2 t_2 + u_2 \nu'_1 t_a + u_1 \nu'_2 t_b
\]
Using fixed-point to represent real numbers

\[ \tilde{x} = \begin{cases} 
[kx] & x \geq 0 \\
[kx] + \phi & x < 0 
\end{cases} \]

010010011100100.11011001001...

- Fixed-length \( l - k \) Integer part
- Fixed-length \( k \) decimal part

010010011100100 11011001001

- Use expensive bit-level operations
  - PICCO, Sharemind, SPDZ, etc

- Support built-in fixed-point operations
  - SecureML, ABY3, PrivPy
The PrivPy computation engine

Clients

\[ C_1 \quad \cdots \quad C_k \quad \cdots \quad C_n \]

Servers

\[ C_k \quad \cdots \quad C_n \]

\[ S_1 \quad S_2 \quad S_a \quad S_b \]

SS Store 1

PO Engine

PO Engine

SS Store a

PO Engine

SS Store 2

SS Store b

TASK CONFIG

Python code
Data source addr
Result addr
The PrivPy computation engine

\[
\begin{align*}
\text{Clients:} & \quad C_1, \ldots, C_n \\
\text{Servers:} & \quad S_1, S_2, S_a, S_b
\end{align*}
\]

\[
\begin{align*}
& C_1 \xrightarrow{x_1} S_1 \xrightarrow{x_1} \text{PO Engine} \xrightarrow{\text{Private Ops Protocols}} S_a \text{ PO Engine} \\
& \vdots \quad \vdots \quad \vdots \\
& C_k \xrightarrow{y_1} S_1 \xrightarrow{y_1} \text{PO Engine} \xrightarrow{\text{Private Ops Protocols}} S_a \text{ PO Engine} \\
& \vdots \quad \vdots \quad \vdots \\
& C_n \xrightarrow{z_1} S_2 \xrightarrow{z_1} \text{PO Engine} \xrightarrow{\text{Private Ops Protocols}} S_b \text{ PO Engine}
\end{align*}
\]

\[
\begin{align*}
\text{SS Store} 1 & \xrightarrow{\text{res}_1} \text{PO Engine} \\
\text{SS Store} 2 & \xrightarrow{\text{res}_2} \text{PO Engine} \\
\text{SS Store a} & \\
\text{SS Store b} &
\end{align*}
\]

\[
\text{res}_1 + \text{res}_2 = \text{res}
\]
Python compatible programming front-end

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

```python
# 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

Why? Many pitfalls if written in Python resulting in inefficiency.
AST-level code optimization to avoid pitfalls

Common factor

\[ x \times y_1 + x \times y_2 + \cdots + x \times y_n \]
\[ \downarrow \]
\[ x \times (y_1 + y_2 + \cdots + y_n) \]

Auto vectorization

\[ x_1 \times y_1 + x_2 \times y_2 + \cdots + x_n \times y_n \]
\[ \downarrow \]
\[ \vec{x} = (x_1, x_2, \ldots, x_n) \times \vec{y} = (y_1, y_2, \ldots, y_n) \]

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: $e^x$, $\log(x)$, ...

$$SS(d) = (d_1, d_2)$$

Basic OPs: Add, Mul, Cmp

Derived OPs: Division, Sigmoid, ReLU

Garbled circuit

$$y(x) = \frac{1}{1 + e^{-x}}$$
$$y'(x) = y(x)(1 - y(x))$$
$$y(x_{t+1}) = y(x_t) + y'(x_t)\Delta x$$
$$= y(x_t) + y(x_t)(1 - y(x_t))\Delta x$$
APIS: arrays are first-class citizen

- Array is a built-in type
  - $A = pp.sarr([...]); B = pp.sarr([...])$
  - Both $A \times B$ 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 \( B \)

➢ \( x + A, A \times B \) and \( x > B \) all work

➢ Can even mix plaintext and cipher text

◆ Ndarray methods

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

\[
y = f(w^T \cdot X + b)
\]
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)

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

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)**

<table>
<thead>
<tr>
<th>Batch size</th>
<th>LAN (10Gbps)</th>
<th>WAN (50Mbps)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LR training</td>
<td>MF training</td>
</tr>
<tr>
<td>Single op</td>
<td>5.3e-3</td>
<td>7.1e-3</td>
</tr>
<tr>
<td>Batch (1000 ops)</td>
<td>3.92</td>
<td>5.67</td>
</tr>
</tbody>
</table>
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 – xialixiaoyi@gmail.com