NFGen: Automatic Non-linear Function Evaluation Code Generator for General-purpose MPC Platforms

Xiaoyu Fan, Kun Chen, Guosai Wang, Mingchun Zhuang, Yi Li and Wei Xu
ACM CCS 2022
Secure multi-party computation (MPC) offers a promising way to achieve privacy-preserving computation.

Currently, several general-purpose MPC platforms are proposed.

- High efficiency.
- Expressive programming front-end.
- Making the development of complex applications possible.
Secure multi-party computation (MPC) offers a promising way to achieve privacy-preserving computation.

Currently, several general-purpose MPC platforms are proposed.

- High efficiency.
- Expressive programming front-end.
- Making the development of complex applications possible.

### Basic Structure of General-purpose MPC platforms

- **Expressive Programming Front-end**
  - Logistic Regression, K-Means, Neural Network etc.
  - Basic SecureOPs
    - $+,-,\times,1,\sqrt{},e^{-},\sqrt{\cdot}$
- **Cryptographic Back-end**
  - Secret Sharing, Oblivious Transfer, Beaver’s multiplication etc.

E.g., Platforms surveyed in [HHNZ19], MP-SPDZ[Ke120], ABY3[MR18]…
Fixed-point Number and Non-linear Function Evaluation

- Fixed-point (FXP) vs. Floating-point (FLP)

<table>
<thead>
<tr>
<th></th>
<th>FXP</th>
<th>FLP (IEEE74)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Range</td>
<td>$[-2^{n-f-1}, 2^{n-f-1}]$</td>
<td>$[-2^{2^{e-1}}, 2^{2^{e-1}}]$</td>
</tr>
<tr>
<td>Smallest</td>
<td>$2^{-f}$</td>
<td>$2^{1-2^{e-1}}$</td>
</tr>
</tbody>
</table>

### Floating-point Number (IEEE754)

- **s**: Sign bit
- **e**: Exponent (11 bits)
- **m**: Mantissa (52 bits)

### Fixed-point Number

- **s**: Sign bit
- **n - f - 1**: Integer part (16 bits)
- **f**: Fraction part (16 bits)
**Fixed-point Number and Non-linear Function Evaluation**

- **Fixed-point (FXP) vs. Floating-point (FLP)**

<table>
<thead>
<tr>
<th></th>
<th>FXP</th>
<th>FLP (IEEE74)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Range</td>
<td>$[-2^{n-f-1}, 2^{n-f-1}]$</td>
<td>$[-2^{2^{e-1}}, 2^{2^{e-1}}]$</td>
</tr>
<tr>
<td>Smallest</td>
<td>$2^{-f}$</td>
<td>$2^{1-2^{e-1}}$</td>
</tr>
</tbody>
</table>

- **Current non-linear function evaluation**
  - Hand-crafted design a series of basic Ops like $\frac{1}{x}$, $e^x$, $\sqrt{x}$ etc.
  - Express complex functions as sequential combinations of basic Ops.

<table>
<thead>
<tr>
<th>s</th>
<th>Exponent</th>
<th>Mantissa</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$e(11)$</td>
<td>$m(52)$</td>
</tr>
</tbody>
</table>

Floating-point Number (IEEE754)

<table>
<thead>
<tr>
<th>s</th>
<th>Integer part</th>
<th>Fraction part</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$n - f - 1$</td>
<td>$f$</td>
</tr>
</tbody>
</table>

Fixed-point Number

\[
\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}
\]

- 1. compute $e^x$ and $e^{-x}$
- 2. compute the division.
Pitfalls of Current Non-linear Function Evaluation
Pitfalls of Current Non-linear Function Evaluation

- Correctness & Precision

Error Cases in Current MPC Platforms (DE: Direct Evaluation)
Pitfalls of Current Non-linear Function Evaluation

- **Correctness & Precision**
  - Overflow
  - Error Accumulation
  - Max = 5.2e+13
  - Mean = 2e+12

- **Performance**
  - Non-linear building blocks are far expensive than +, x.

- **Generality**
  - Not support hard-to-compute functions like $\gamma(x, z)$, $\Phi(x)$.

- **Portability**
  - Non-linear function design for one platform is hard to transplant to others.

*Error Cases in Current MPC Platforms (DE: Direct Evaluation)*
Our Solution: NFGen (Non-linear Function Code Generator)

Secure Logistic Regression (require sigmoid)

'System desc': {
  \( n, f \): \( \{96, 48\} \)
  OPs: \{+, >, \times \...\}
},

'Function desc': { \( F \): sigmoid \( [a, b] \): \([-10, 10]\), \( 0 \): \( \varepsilon \): \(10^{-6}, 10^{-3}\) \}.
Our Solution: NFGen (Non-linear Function Code Generator)

Secure Logistic Regression (require sigmoid)

Function desc': {
  F: sigmoid
  [a, b]: [−10, 10],
  0: ε: 10⁻⁶, 10⁻³
},

'System desc': {
  (n, f): (96, 48)
  OPs: {+, >, × ...}
},

Iterate k:
FitPiecewise (k, NFD):
Try \( \hat{p}_k \)
Check \( ||d \)
Split if fail

Construct \( \hat{P} \)
Secure Logistic Regression (require sigmoid)

Our Solution: NFGen (Non-linear Function Code Generator)

Function desc': 
- \( F: \text{sigmoid} \)
- \([a, b]: [-10, 10]\)
- \( \theta: \varepsilon: 10^{-6}, 10^{-3} \)

System desc':
- \( (n, f): (96, 48) \)
- OPs: \{+, >, \times\…\}

Iterate \( k: \)
- \( \text{FitPiecewise} (k, \text{NFD}) \):
- Try \( \hat{p}_k \)
- Check \( \|d \)
- Split if fail

Construct \( \hat{p} \)

Load OPPE templet

\( \|\|: 62\text{ms} \)
\( \sqrt{x}: 150\text{ms}, \text{km-Profiler,} \ldots \)
Our Solution: NFGen (Non-linear Function Code Generator)

Secure Logistic Regression (require sigmoid)

End-to-End Workflow of NFGen
Open source: https://github.com/Fannxy/NFGen

@types.vectorize
def sigmoid(x):
    breaks = [-1007.0, ..., 10.0]
    coeffA = [[0.0, ..., 0.0]]
    scaler = [[1.0, ..., 1.0]]
    m = len(coeffA),
    k = len(coeffA[0])
    ...
    comp = sfix.Array(m)
    for i in range(m):
        comp[i] = (x >= breaks[i])
    ...
    return res

Code

Load OPPE templet

Select best plan and generate:

```python
Code
```
Fixed-point Piece-wise Polynomials Construction

- Valid piece-wise polynomial $\hat{p}_k^m$
  - Each term in piece-wise polynomial $\hat{p}_k^m$ can be represented by $\langle n, f \rangle$-FXP.
  
- NP-Complete Integer programming problem.

- $\hat{p}_k^m(x)$ can approximate $F(x)$ satisfying the accuracy requirement.
  
- Best-effort try-split until succeed.
Valid piece-wise polynomial $\hat{p}_k^m$
- Each term in piece-wise polynomial $\hat{p}_k^m$ can be represented by $\langle n, f \rangle$-FXP.

- NP-Complete Integer programming problem.

- $\hat{p}_k^m (x)$ can approximate $F(x)$ satisfying the accuracy requirement.

- Best-effort try-split until succeed.

Workflow of Piece-wise Polynomial Construction
Fixed-point Piece-wise Polynomials Construction

- Valid piece-wise polynomial $\hat{p}_k^m$
  - Each term in piece-wise polynomial $\hat{p}_k^m$ can be represented by $<n, f>$-FXP.
  - NP-Complete Integer programming problem.
  - $\hat{p}_k^m(x)$ can approximate $F(x)$ satisfying the accuracy requirement.
  - Best-effort try-split until succeed.

1) Constrains $\bar{k} \leq k$, avoiding over/under-flow.

Workflow of Piece-wise Polynomial Construction
Fixed-point Piece-wise Polynomials Construction

- Valid piece-wise polynomial $\hat{p}_k^m$
  - Each term in piece-wise polynomial $\hat{p}_k^m$ can be represented by $(n, f)$-FXP.
- NP-Complete Integer programming problem.
- $\hat{p}_k^m(x)$ can approximate $F(x)$ satisfying the accuracy requirement.
- Best-effort try-split until succeed.

1) Constrains $\bar{k} \leq k$, avoiding over/under-flow.
2) Fits $p_k$ in FLP.

Workflow of Piece-wise Polynomial Construction
Fixed-point Piece-wise Polynomials Construction

- Valid piece-wise polynomial $\hat{p}_k^m$
  - Each term in piece-wise polynomial $\hat{p}_k^m$ can be represented by $\langle n, f \rangle$-FXP.
  - NP-Complete Integer programming problem.
  - $\hat{p}_k^m(x)$ can approximate $F(x)$ satisfying the accuracy requirement.
  - Best-effort try-split until succeed.

1) Constrains $\bar{k} \leq k$, avoiding over/under-flow.
2) Fits $p_{\bar{k}}$ in FLP.
3) Converts FLP $p_k$ to FXP $\hat{p}_{\bar{k}}$ with scaling factor.
4) Further reduces error using residual boosting.

Workflow of Piece-wise Polynomial Construction
Fixed-point Piece-wise Polynomials Construction

- Valid piece-wise polynomial $\hat{p}_k^m$
  - Each term in piece-wise polynomial $\hat{p}_k^m$ can be represented by $\langle n, f \rangle$-FXP.
  - NP-Complete Integer programming problem.
  - $\hat{p}_k^m(x)$ can approximate $F(x)$ satisfying the accuracy requirement.
  - Best-effort try-split until succeed.

1) Constrains $\overline{k} \leq k$, avoiding over/under-flow.
2) Fits $p_{\overline{k}}$ in FLP.
3) Converts FLP $p_k$ to FXP $\hat{p}_{\overline{k}}$ with scaling factor.
4) Further reduces error using residual boosting.

Workflow of Piece-wise Polynomial Construction
Fixed-point Piece-wise Polynomials Construction

- Valid piece-wise polynomial \( \hat{p}_k^m \)
  - Each term in piece-wise polynomial \( \hat{p}_k^m \) can be represented by \( (n, f) \)-FXP.
  - NP-Complete Integer programming problem.
  - \( \hat{p}_k^m (x) \) can approximate \( F(x) \) satisfying the accuracy requirement.
  - Best-effort try-split until succeed.

1) Constrains \( \bar{k} \leq k \), avoiding over/under-flow.
2) Fits \( p_{\bar{k}} \) in FLP.
3) Converts FLP \( p_k \) to FXP \( \hat{p}_{\bar{k}} \) with scaling factor.
4) Further reduces error using residual boosting.

Test \( \hat{p}_k \) accuracy

Try generate \( \hat{p}_k \) in \([a, b]\)

Pass
Return valid \( \hat{p}_k \)

Workflow of Piece-wise Polynomial Construction
Fixed-point Piece-wise Polynomials Construction

- Valid piece-wise polynomial $\hat{p}_k^m$
  - Each term in piece-wise polynomial $\hat{p}_k^m$ can be represented by $(n, f)$-FXP.
  - NP-Complete Integer programming problem.
  - $\hat{p}_k^m(x)$ can approximate $F(x)$ satisfying the accuracy requirement.
  - Best-effort try-split until succeed.

Workflow of Piece-wise Polynomial Construction

1) Constrains $\bar{k} \leq k$, avoiding over/under-flow.
2) Fits $p_k$ in FLP.
3) Converts FLP $p_k$ to FXP $\hat{p}_k$ with scaling factor.
4) Further reduces error using residual boosting.

Try generate $\hat{p}_k$ in $[a, b]$

Fail

Binary splits $[a, b]$ and recurse.

Test $\hat{p}_k$ accuracy

Pass

Return valid $\hat{p}_k$
Two Ways to Improve the FXP Polynomial Accuracy

- Severe problem: tiny coefficients in FXP harm the final accuracy.
Two Ways to Improve the FXP Polynomial Accuracy

- Severe problem: tiny coefficients in FXP harm the final accuracy.
- Scaling factor
  - Making use of more significant bits.
Two Ways to Improve the FXP Polynomial Accuracy

- Severe problem: tiny coefficients in FXP harm the final accuracy.

- **Scaling factor**
  - Making use of more significant bits.
    - E.g., computing $7^{th}$ term $(1.044 \times 10^{-11}) \times 10^7$

<table>
<thead>
<tr>
<th>Coefficient</th>
<th>Least Significant</th>
<th>Most Significant</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1012</td>
<td>10435267233655</td>
</tr>
<tr>
<td>0</td>
<td>1026</td>
<td>10435267233655</td>
</tr>
<tr>
<td>0</td>
<td>10 decimal Left Unused</td>
<td>1044</td>
</tr>
<tr>
<td>0</td>
<td>10 decimal Left Unused</td>
<td>1044000000000000</td>
</tr>
</tbody>
</table>

**Scaling Factor**

- Left shift the coefficients as much as possible while avoid overflow.
# Two Ways to Improve the FXP Polynomial Accuracy

- **Severe problem:** tiny coefficients in FXP harm the final accuracy.

## Scaling factor

- **Making use of more significant bits.**
  - E.g., computing $7^{th}$ term $(1.044 \times 10^{-11}) \times 10^7$

<table>
<thead>
<tr>
<th>0</th>
<th>1012</th>
<th>10435267233655</th>
<th>$\times 10^7$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1026</td>
<td>10435267233655</td>
<td>$\times 10^7$</td>
</tr>
<tr>
<td>0</td>
<td>10 decimal Left Unused</td>
<td>1044</td>
<td>$\times 10^7$</td>
</tr>
<tr>
<td>0</td>
<td>14</td>
<td>1044</td>
<td>$\times 10^7$</td>
</tr>
</tbody>
</table>

### Scaling Factor

- **Left shift the coefficients as much as possible while avoid overflow.**

## Residual Boosting

- **Lower-order polynomial tend to have larger coefficients.**

---

25
Two Ways to Improve the FXP Polynomial Accuracy

- **Severe problem:** Tiny coefficients in FXP harm the final accuracy.

### Scaling factor

- Making use of more significant bits.
  - E.g., computing $7^{th}$ term $(1.044\times10^{-11})\times100^7$

| 0 | 1012 | 10435267236655 | $(base_{10})$ |
| 0 | 1026 | 10435267236655 | $(base_{10})$ |

| 0 | 10 decimal Left Unused | 1044 | $(base_{10})$ |
| 0 | 14 | | $(base_{10})$ |

- **Scaling Factor**
  - Left shift the coefficients as much as possible while avoid overflow.

### Residual Boosting

- Lower-order polynomial tend to have larger coefficients.
  - Use a series of lower-order polynomials to fill the residuals.
Automatic Performance Profiler & Code Generation

- Piece-wise polynomial evaluation.
  - **Secure:** Obliviously organize secure +, × and >.
  - **Performance:** $O(m)$ secure $>$ and $O(km + k \log k)$ secure $\times$.
  - Which $\hat{p}^m_k$ has better performance depends on the characters of specific MPC deployment.
Automatic Performance Profiler & Code Generation

- Piece-wise polynomial evaluation.
  - Secure: Obliviously organize secure $+, \times$ and $>$. 
  - Performance: $O(m)$ secure $>$ and $O(km + k \log k)$ secure $\times$.
  - Which $\hat{p}_k^m$ has better performance depends on the characters of specific MPC deployment.

- Train a deployment-specific profiler model $f_S: (k, m) \rightarrow \text{time}(\text{ms})$ and select the most efficient one.

- Generate code into pre-defined code template.

### Performance Characteristic of Different MPC Deployments

<table>
<thead>
<tr>
<th>MPC deploy (#)</th>
<th>$\times$(ms)</th>
<th>$\times:&gt; $</th>
<th>Preference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rep2k(SPDZ)</td>
<td>2</td>
<td>1:4</td>
<td>More prefer less m</td>
</tr>
<tr>
<td>RepF(SPDZ)</td>
<td>32</td>
<td>1:1</td>
<td></td>
</tr>
<tr>
<td>Shamir(SPDZ)</td>
<td>81</td>
<td>1:1</td>
<td>More prefer less k.</td>
</tr>
<tr>
<td>Ps-Rep2k(SPDZ)</td>
<td>851</td>
<td>1:1</td>
<td></td>
</tr>
<tr>
<td>Ps-RepF(SPDZ)</td>
<td>84</td>
<td>1:1</td>
<td></td>
</tr>
<tr>
<td>Rep2k(PrivPy)</td>
<td>1</td>
<td>1:11</td>
<td>Severely prefer less m.</td>
</tr>
</tbody>
</table>
Evaluation: Improved Accuracy

- Overview of 15 common-used functions

- Improved cases

- Baseline: direct evaluation of MP-SPDZ library functions.
- NFGen: generated evaluation code.
Evaluation: Improved Accuracy

- **Overview of 15 common-used functions**
  - Baseline: direct evaluation of MP-SPDZ library functions.
  - NFGen: generated evaluation code.

- **Improved cases**
  - Baseline: direct evaluation of MP-SPDZ library functions.
  - NFGen: generated evaluation code.
Evaluation: Improved Efficiency

<table>
<thead>
<tr>
<th>MPC Sys</th>
<th>Efficiency ratio(%), speedup(×)</th>
<th>Comm ratio(%), save(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Benefit</td>
<td>Mean</td>
</tr>
<tr>
<td>Rep2k(SPDZ)</td>
<td>100%</td>
<td>16.7×</td>
</tr>
<tr>
<td>RepF(SPDZ)</td>
<td>100%</td>
<td>5.3×</td>
</tr>
<tr>
<td>Shamir(SPDZ)</td>
<td>100%</td>
<td>4.0×</td>
</tr>
<tr>
<td>Ps-Rep2k(SPDZ)</td>
<td>67%</td>
<td>1.8×</td>
</tr>
<tr>
<td>Ps-RepF(SPDZ)</td>
<td>87%</td>
<td>2.3×</td>
</tr>
<tr>
<td>Rep2k(PrivPy)</td>
<td>100%</td>
<td>8.6×</td>
</tr>
</tbody>
</table>

- **NFGen** achieves significant improvements.
- 93% achieves benefit in all 15 * 6 cases.
- Average speedup 6.5× and maximum 86.1×.
- Average communication save 39.3% and maximum 93%.

Improved Performance Overview
15 functions for each sys and all achieve the above accuracy requirements.
## Evaluation: Improved Efficiency

<table>
<thead>
<tr>
<th>MPC Sys</th>
<th>Efficiency ratio(%)</th>
<th>speedup(×)</th>
<th>Comm ratio(%)</th>
<th>save(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Benefit</td>
<td>Mean</td>
<td>Max</td>
<td>Benefit</td>
</tr>
<tr>
<td>Rep2k(SPDZ)</td>
<td>100%</td>
<td>16.7×</td>
<td>86.1×</td>
<td>93%</td>
</tr>
<tr>
<td>RepF(SPDZ)</td>
<td>100%</td>
<td>5.3×</td>
<td>16.0×</td>
<td>87%</td>
</tr>
<tr>
<td>Shamir(SPDZ)</td>
<td>100%</td>
<td>4.0×</td>
<td>10.9×</td>
<td>87%</td>
</tr>
<tr>
<td>Ps-Rep2k(SPDZ)</td>
<td>67%</td>
<td>1.8×</td>
<td>6.1×</td>
<td>67%</td>
</tr>
<tr>
<td>Ps-RepF(SPDZ)</td>
<td>87%</td>
<td>2.3×</td>
<td>7.6×</td>
<td>73%</td>
</tr>
<tr>
<td>Rep2k(PrivPy)</td>
<td>100%</td>
<td>8.6×</td>
<td>29.1×</td>
<td>93%</td>
</tr>
</tbody>
</table>

- **NFGGen achieves significant improvements.**
- **93% achieves benefit in all 15 * 6 cases.**
- **Average speedup 6.5× and maximum 86.1×.**
- **Average communication save 39.3% and maximum 93%.**

*Improved Performance Overview*

15 functions for each sys and all achieve the above accuracy requirements.
Support hard-to-compute functions

<table>
<thead>
<tr>
<th>Target function</th>
<th>(k, m)</th>
<th>Fit time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\gamma(x, z) = \int_0^x t^{z-1} e^t , dt$</td>
<td>(6, 4)</td>
<td>1.1s</td>
</tr>
<tr>
<td>$\Gamma(x, z) = \int_x^\infty t^{z-1} e^t , dt$</td>
<td>(6, 6)</td>
<td>1.1s</td>
</tr>
<tr>
<td>$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} , dt$</td>
<td>(4, 6)</td>
<td>0.8s</td>
</tr>
<tr>
<td>$\Phi(x) = \frac{2}{\sqrt{2\pi}} \int_0^x e^{-\frac{t^2}{2}} , dt$</td>
<td>(8, 6)</td>
<td>1.2s</td>
</tr>
</tbody>
</table>
Evaluation: Other Benefits

- Support hard-to-compute functions

<table>
<thead>
<tr>
<th>Target function</th>
<th>(k, m)</th>
<th>Fit time</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y(x, z) = \int_0^x t^{z-1}e^t dt )</td>
<td>(6, 4)</td>
<td>1.1s</td>
</tr>
<tr>
<td>( \Gamma(x, z) = \int_x^\infty t^{z-1}e^t dt )</td>
<td>(6, 6)</td>
<td>1.1s</td>
</tr>
<tr>
<td>( \text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt )</td>
<td>(4, 6)</td>
<td>0.8s</td>
</tr>
<tr>
<td>( \Phi(x) = \frac{2}{\sqrt{2\pi}} \int_0^x e^{\frac{t^2}{2}} dt )</td>
<td>(8, 6)</td>
<td>1.2s</td>
</tr>
</tbody>
</table>

- Accelerate current applications
  - Approximate \( \text{sigmoid}(x) \) and accelerate LR.

![Sigmoid Approximations](image)
Conclusion

- NFGen is our attempt to offer a new way evaluating non-linear functions in MPC,
  - Improved performance from many perspectives (correctness, precision and efficiency).
  - Easy to use: NFGen automatically generate the evaluation code with a simple input config.
  - Support numerous hard-to-compute functions and different bit lengths, making MPC systems more general than before.
- As MPC offers a brand-new architecture, maybe we should explore new algorithm design logic instead of just follow the plaintext development.
Q & A

Thanks for your listening!