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

清華大學
Tsinghua University

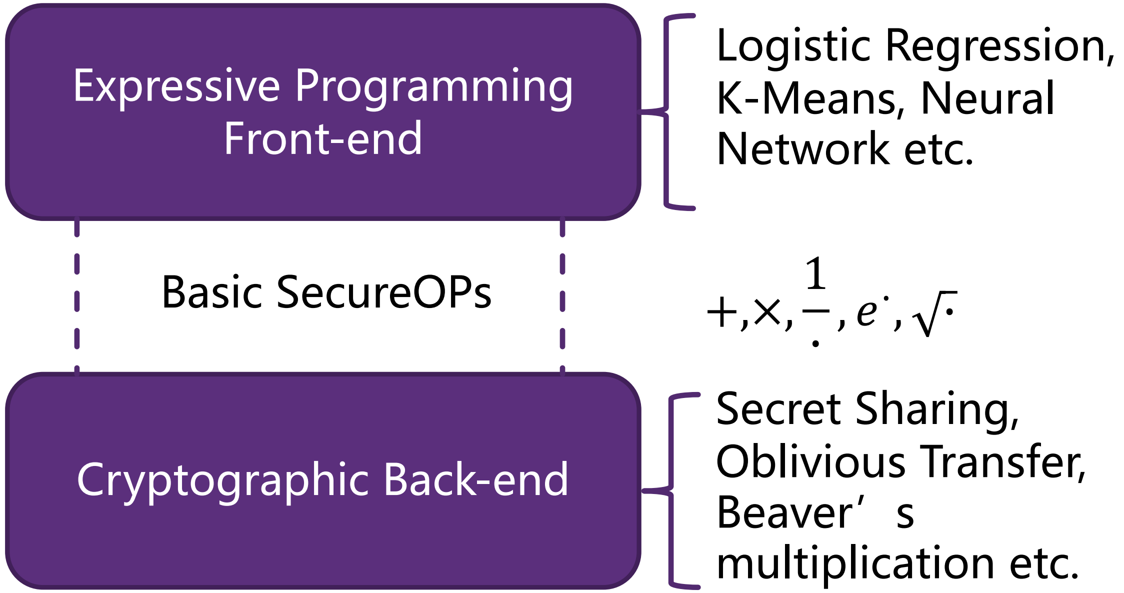
華控清交
HUA KONG
TSINGJIAO

General-purpose Multi-party Computation

- 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.

General-purpose Multi-party Computation

- 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.



Expressive Programming Front-end

Logistic Regression, K-Means, Neural Network etc.

Basic SecureOPs

$+,\times,\frac{1}{\cdot},e^{\cdot},\sqrt{\cdot}$

Cryptographic Back-end

Secret Sharing, Oblivious Transfer, Beaver' s multiplication etc.

Basic Structure of General-purpose MPC platforms

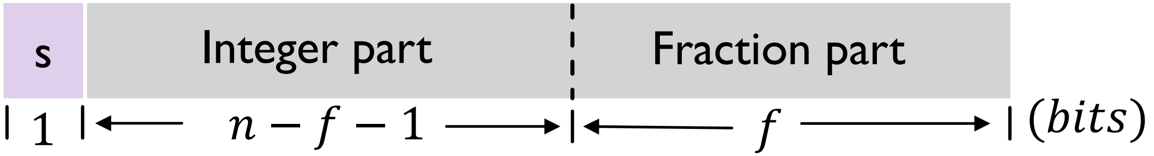
E.g., Platforms surveyed in [HHNZ19], MP-SPDZ[Kel20], ABY3[MR18]…

Fixed-point Number and Non-linear Function Evaluation

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

	FXP	**FLP(IEEE74)**
Range	$[-2^{n-f-1}, 2^{n-f-1}]$	$[-2^{2^{e-1}}, 2^{2^{e-1}}]$
Smallest	2^{-f}	$2^{1-2^{e-1}}$

Floating-point Number (IEEE754)



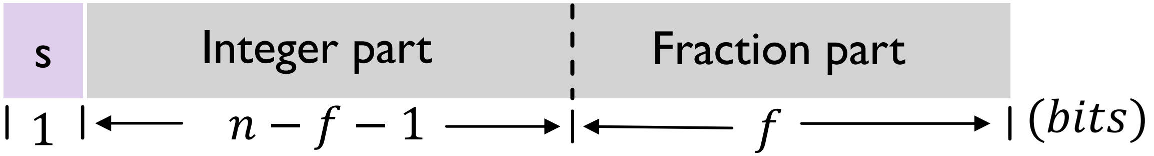
Fixed-point Number

Fixed-point Number and Non-linear Function Evaluation

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

	FXP	FLP(IEEE74)
Range	$[-2^{n-f-1}, 2^{n-f-1}]$	$[-2^{2^{e-1}}, 2^{2^{e-1}}]$
Smallest	2^{-f}	$2^{1-2^{e-1}}$

Floating-point Number (IEEE754)



Fixed-point Number

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

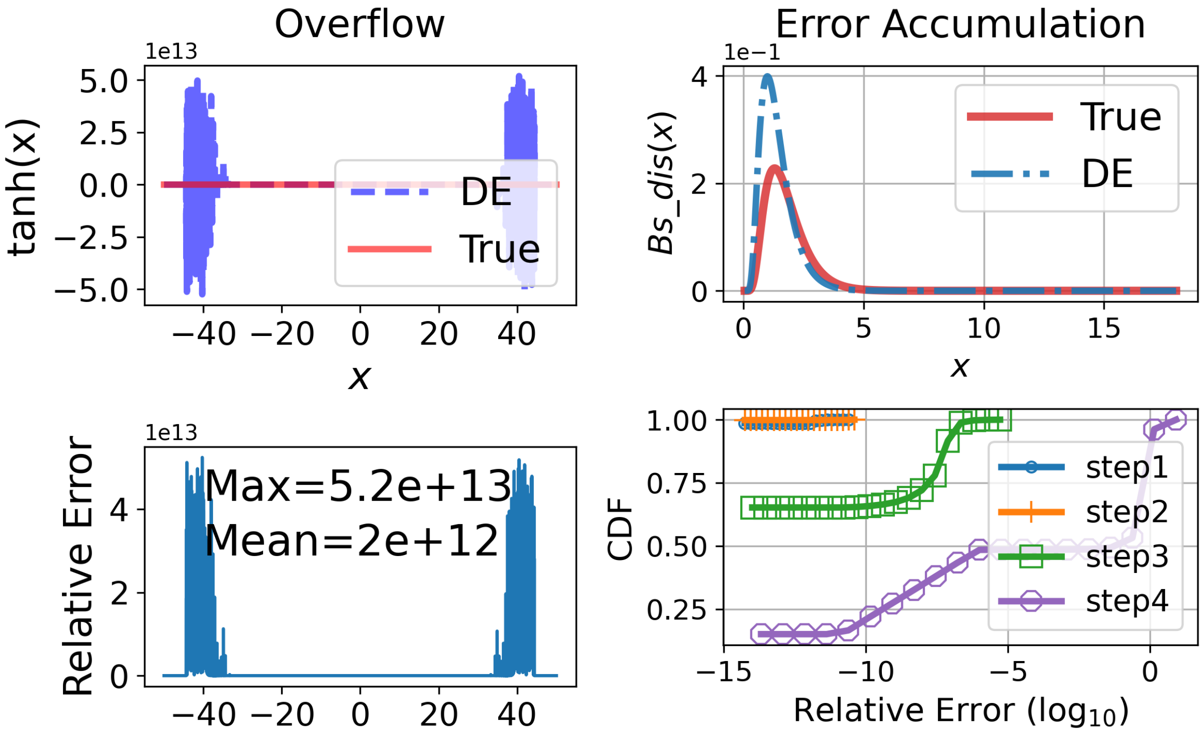
$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

- 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

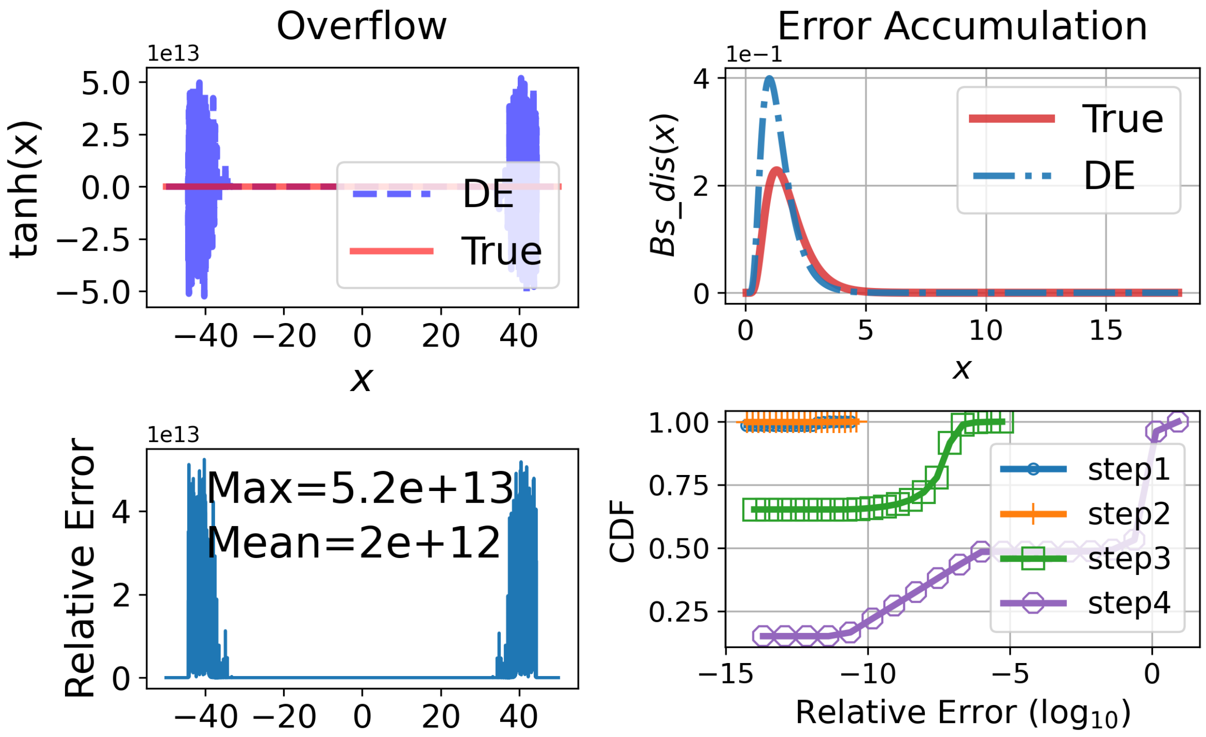
- Correctness & Precision



Error Cases in Current MPC Platforms (DE: Direct Evaluation)

Pitfalls of Current Non-linear Function Evaluation

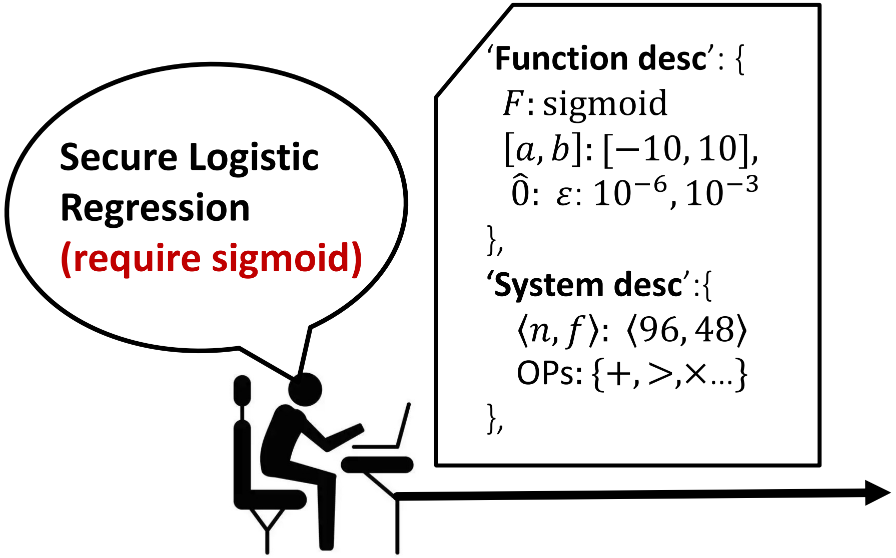
- Correctness & Precision



Error Cases in Current MPC Platforms (DE: Direct Evaluation)

- Performance
  - Non-linear building blocks are far expensive than $+, \times$.
- 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.

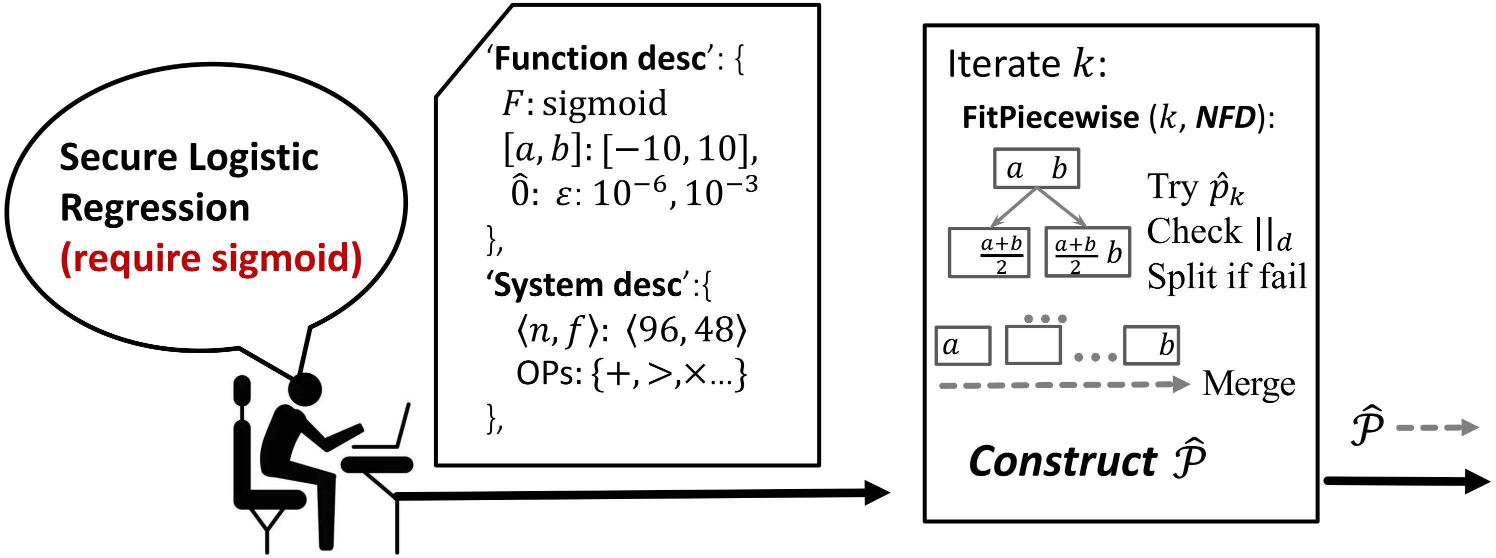
Our Solution: NFGen (Non-linear Function Code Generator)



Secure Logistic Regression **(require sigmoid)**

'**Function desc**': {
F: sigmoid
$[a, b]$: $[-10, 10]$,
$\hat{0}$: ε: $10^{-6}, 10^{-3}$
},
'**System desc**':{
$\langle n, f \rangle$: $\langle 96, 48 \rangle$
OPs: $\{+, >, \times \ldots\}$
},

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



Secure Logistic Regression
(require sigmoid)

'**Function desc**': {
F: sigmoid
$[a, b]$: $[-10, 10]$,
$\hat{0}$: ε: $10^{-6}, 10^{-3}$
},
'**System desc**':{
$\langle n, f \rangle$: $\langle 96, 48 \rangle$
OPs: $\{+, >, \times \ldots\}$
},

Iterate k:

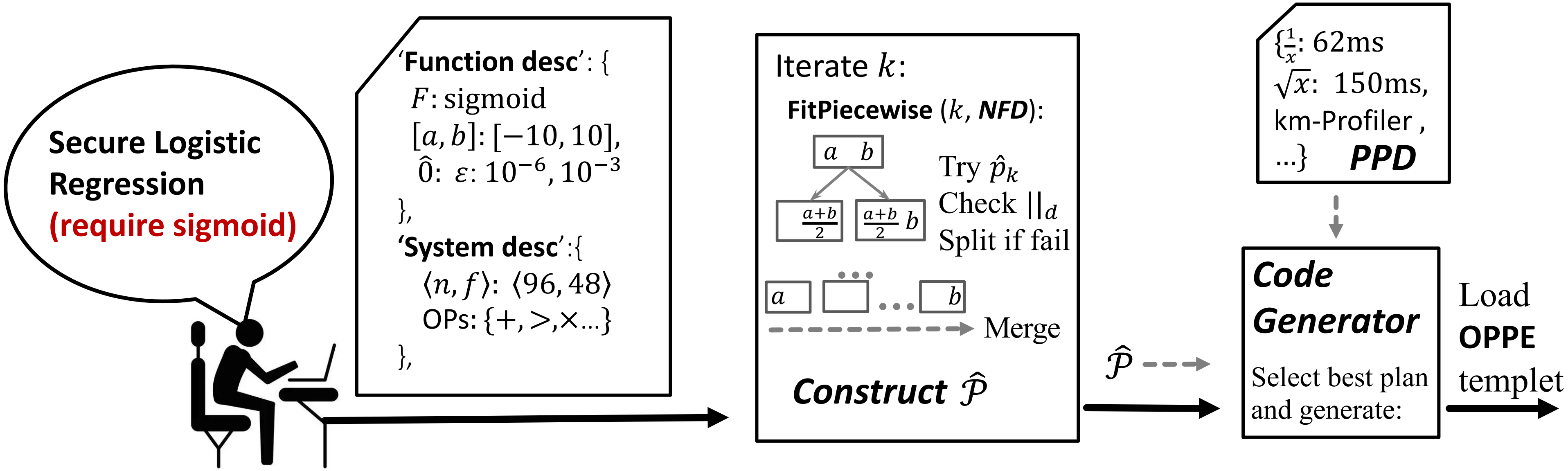
FitPiecewise (k, ***NFD***):

Try $\hat{p}_k$
Check $||_d$
Split if fail

Merge

Construct $\hat{\mathcal{P}}$

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



Secure Logistic Regression
(require sigmoid)

'**Function desc**': {
F: sigmoid
$[a, b]$: $[-10, 10]$,
$\hat{0}$: ε: $10^{-6}, 10^{-3}$
},
'**System desc**':{
$\langle n, f\rangle$: $\langle 96, 48\rangle$
OPs: $\{+, >, \times \ldots\}$
},

Iterate k:
FitPiecewise (k, ***NFD***):
Try $\hat{p}_k$
Check $||_d$
Split if fail
Merge
Construct $\hat{\mathcal{P}}$

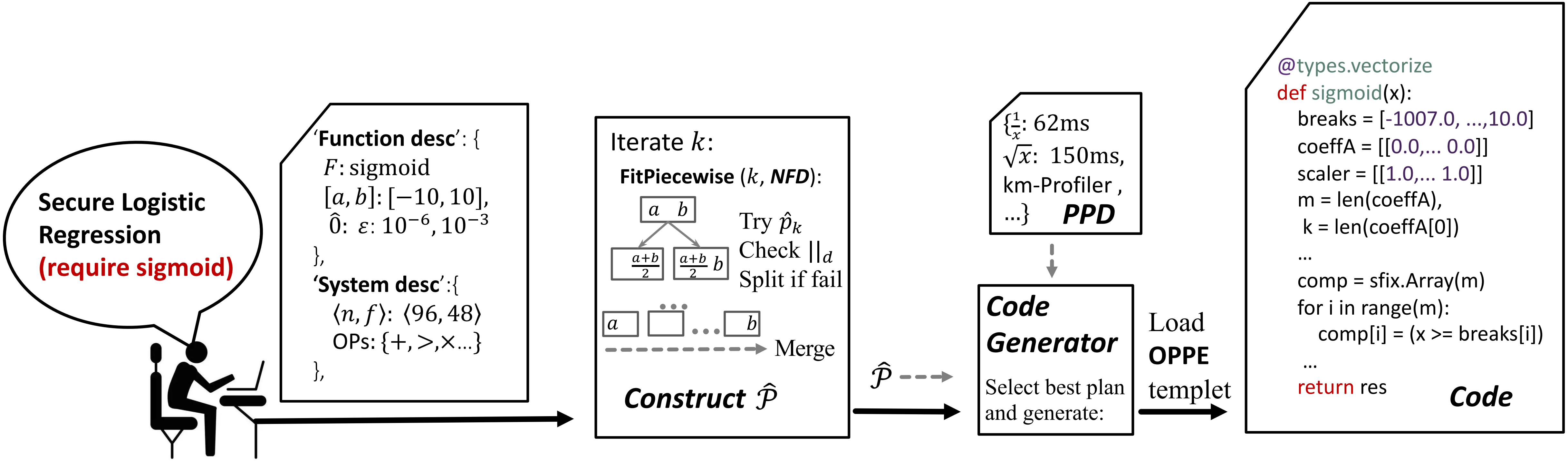
$\hat{\mathcal{P}}$

$\{\frac{1}{x}$: 62ms
$\sqrt{x}$: 150ms,
km-Profiler ,
…} ***PPD***

Code Generator
Select best plan and generate:

Load
OPPE
templet

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



Secure Logistic Regression
(require sigmoid)

'**Function desc**': {
F: sigmoid
$[a, b]$: $[-10, 10]$,
$\hat{0}$: ε: $10^{-6}, 10^{-3}$
},
'**System desc**':{
$\langle n, f\rangle$: $\langle 96, 48\rangle$
OPs: $\{+, >, \times \ldots\}$
},

Iterate k:
FitPiecewise (k, ***NFD***):

Try $\hat{p}_k$
Check $||_d$
Split if fail

Merge

Construct $\hat{\mathcal{P}}$

$\hat{\mathcal{P}}$

$\{\frac{1}{x}$: 62ms
$\sqrt{x}$: 150ms,
km-Profiler ,
…$\}$ ***PPD***

Code Generator

Select best plan and generate:

Load
OPPE
templet

```
@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
```

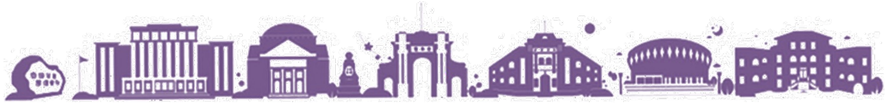
Code

End-to-End Workflow of NFGen

Open source: https://github.com/Fannxy/NFGen

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.



Fixed-point Piece-wise Polynomials Construction

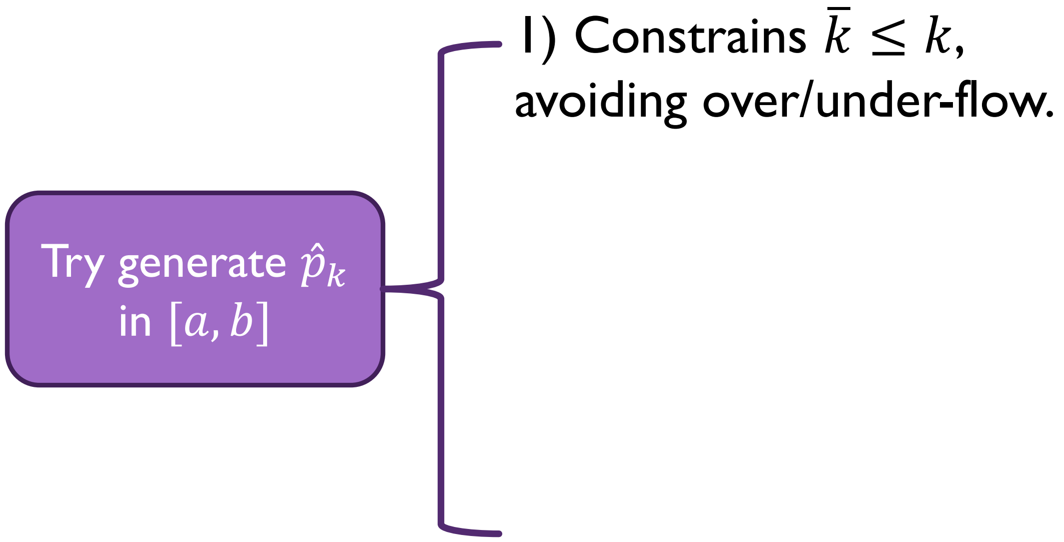
- Valid piece-wise polynomial $\hat{p}_k^m$
  - Each term in piece-wise polynomial $\hat{p}_k^m$ can be represented by $\langle n, f \rangle$-FXP.
  - ➢ NP-Complete Integer programming problem.
  - $\hat{p}_k^m(x)$ can approximate $F(x)$ satisfying the accuracy requirement.
  - ➢ Best-effort try-split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

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

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.

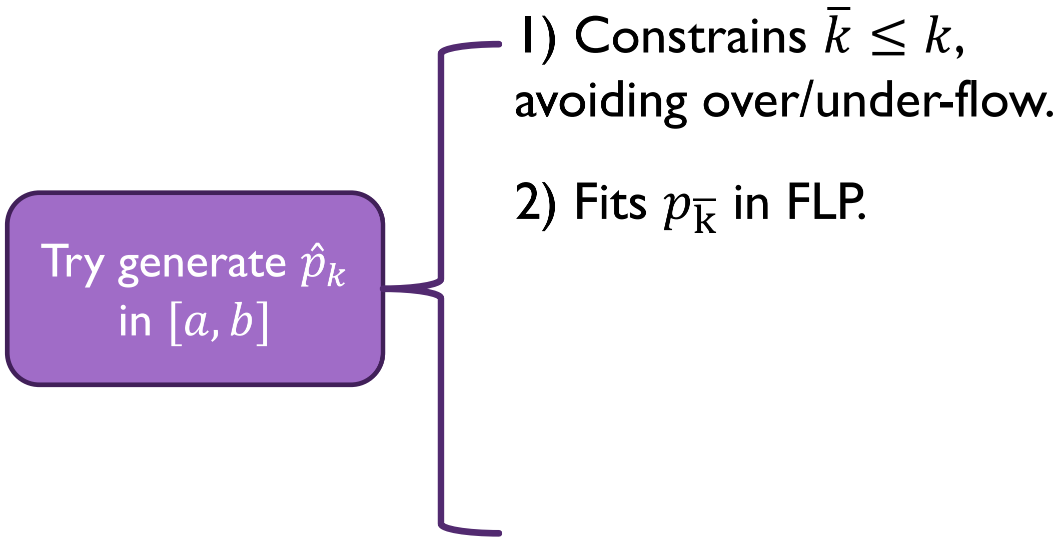

Try generate $\hat{p}_k$ in $[a, b]$

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 $\langle n, f \rangle$-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.



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

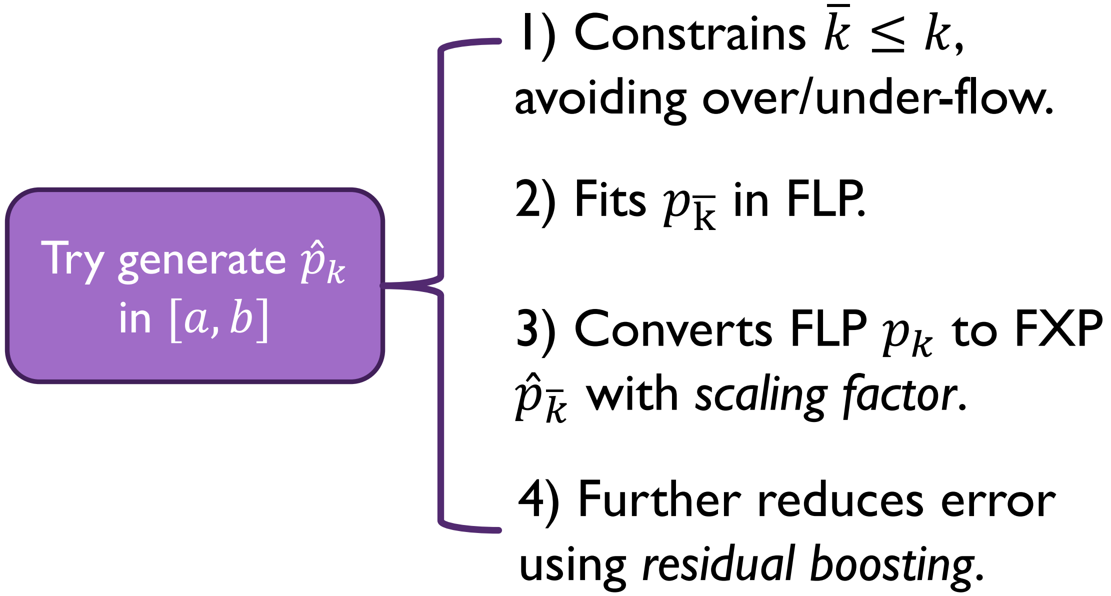
1) Constrains $\bar{k} \leq k$, avoiding over/under-flow.

2) Fits $p_{\overline{\mathrm{k}}}$ 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.


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

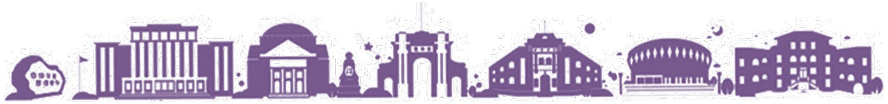
1) Constrains $\bar{k} \leq k$, avoiding over/under-flow.

2) Fits $p_{\bar{k}}$ in FLP.

3) Converts FLP p_k to FXP $\hat{p}_{\bar{k}}$ with *scaling factor*.

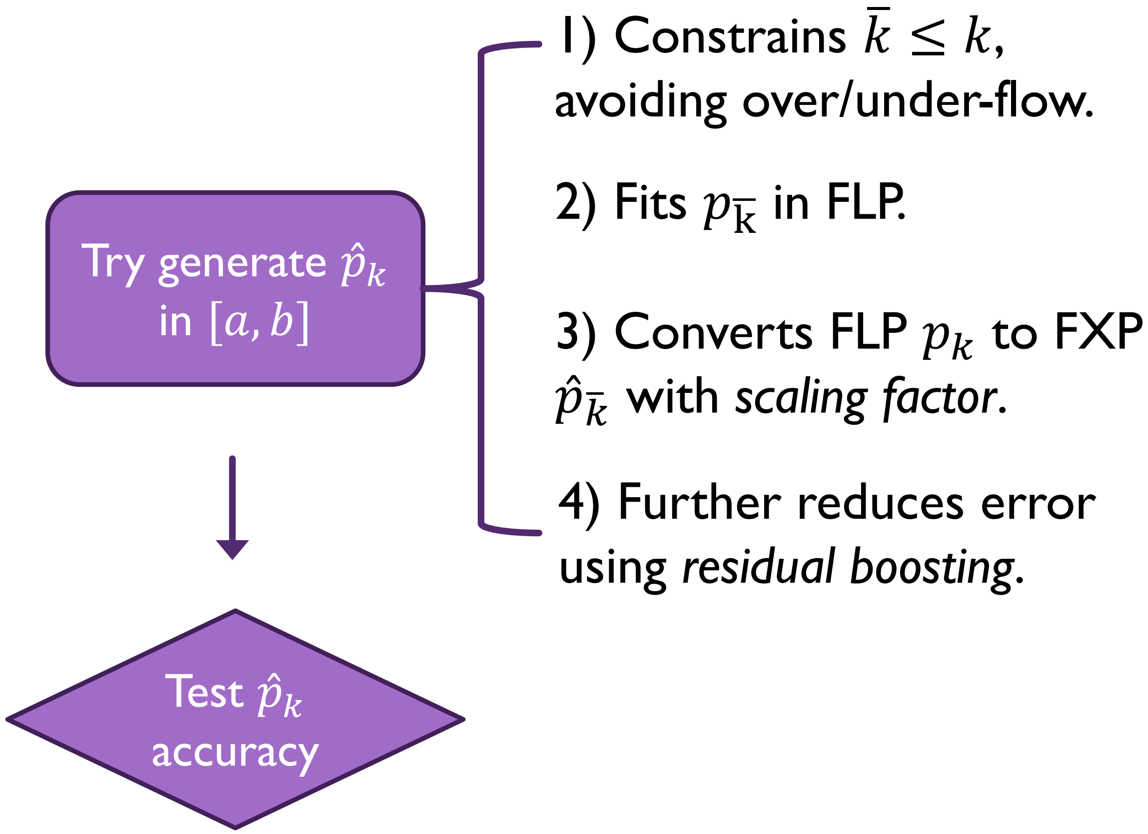
4) Further reduces error using *residual boosting*.

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.

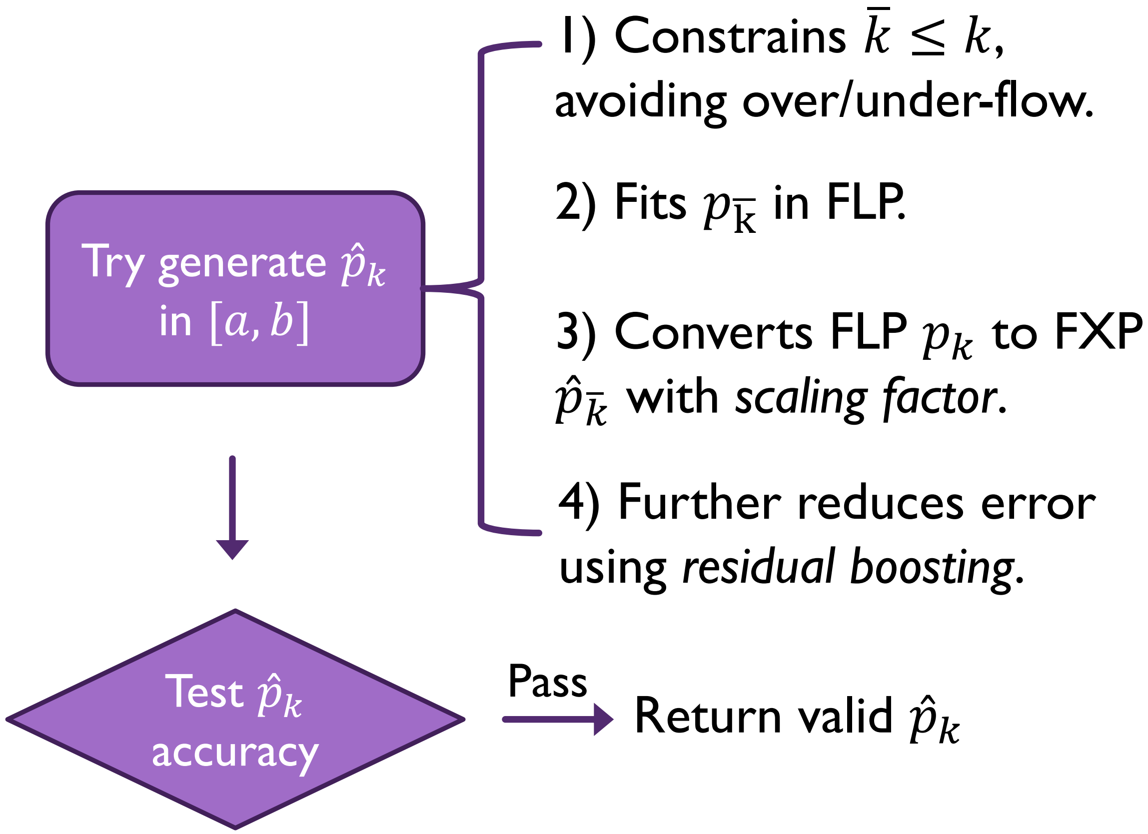
3) Converts FLP p_k to FXP $\hat{p}_{\overline{k}}$ with *scaling factor*.

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 $\langle n, f\rangle$-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.



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

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

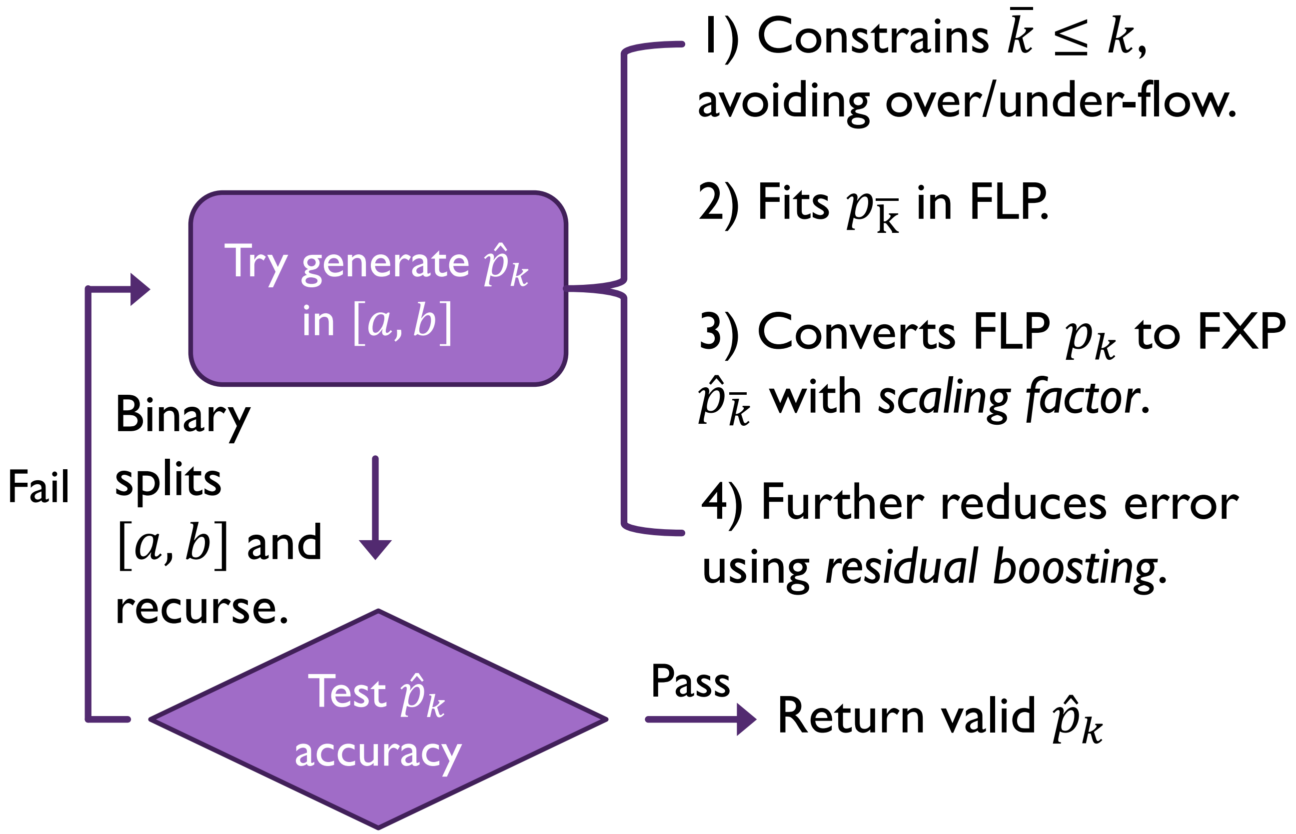
Pass

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 $\langle n, f \rangle$-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.


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

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*.

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

Test $\hat{p}_k$ accuracy

Fail

Pass

Return valid $\hat{p}_k$

Workflow of Piece-wise Polynomial Construction

Two Ways to Improve the FXP Polynomial Accuracy

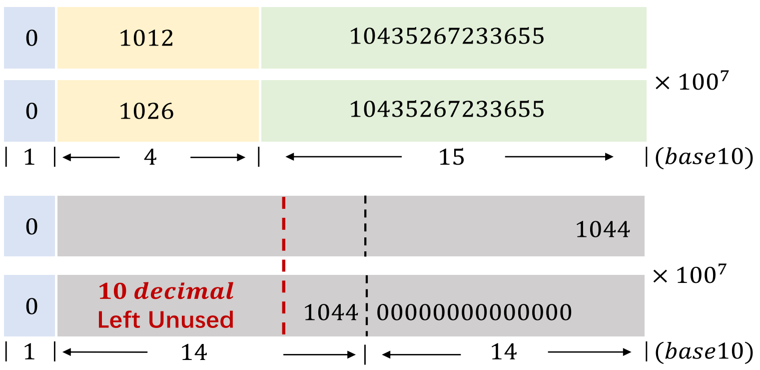
- Severe problem: tiny coefficients in FXP harm the final accuracy.

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 7th term $(1.044\times10^{-11})\times100^7$



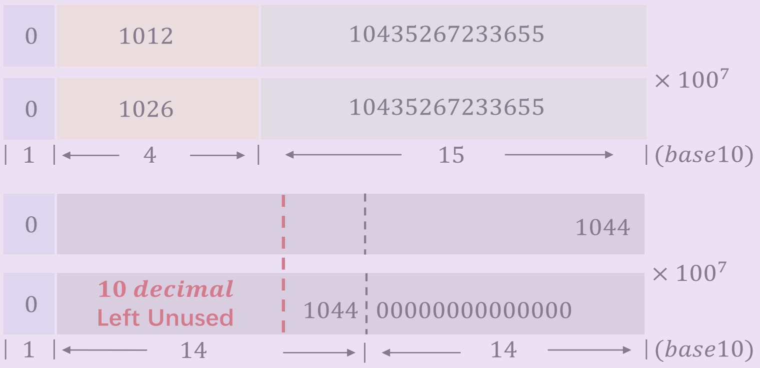
Scaling Factor

  - 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 7th term $(1.044\times10^{-11})\times100^7$



Scaling Factor

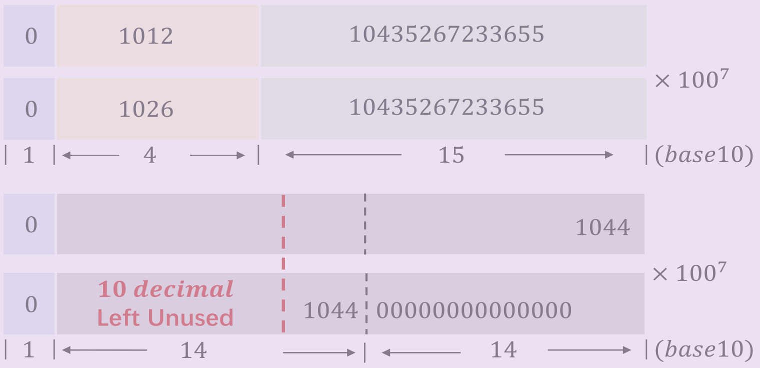
  - Left shift the coefficients as much as possible while avoid overflow.

- Residual Boosting
  - Lower-order polynomial tend to have larger coefficients.

Two Ways to Improve the FXP Polynomial Accuracy

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

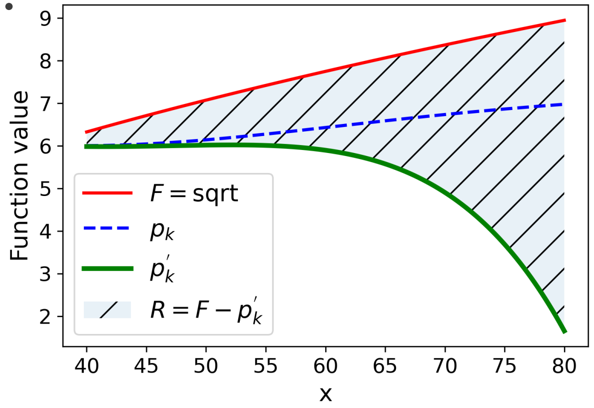
- Scaling factor
  - Making use of more significant bits.
    - E.g., computing 7th term $(1.044\times10^{-11})\times100^{7}$



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.



Residual Function Demonstration

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.

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.

MPC deploy ($\mathcal{S}$)	**$\times$(ms)**	**$\times$:$>$**	**Preference**
Rep2k(SPDZ)	2	1:4	More prefer less m
RepF(SPDZ)	32	1:1	More prefer less k.
Shamir(SPDZ)	81	1:1	
Ps-Rep2k(SPDZ)	851	1:1	
Ps-RepF(SPDZ)	84	1:1	
Rep2k(PrivPy)	1	1:11	Severely prefer less m.

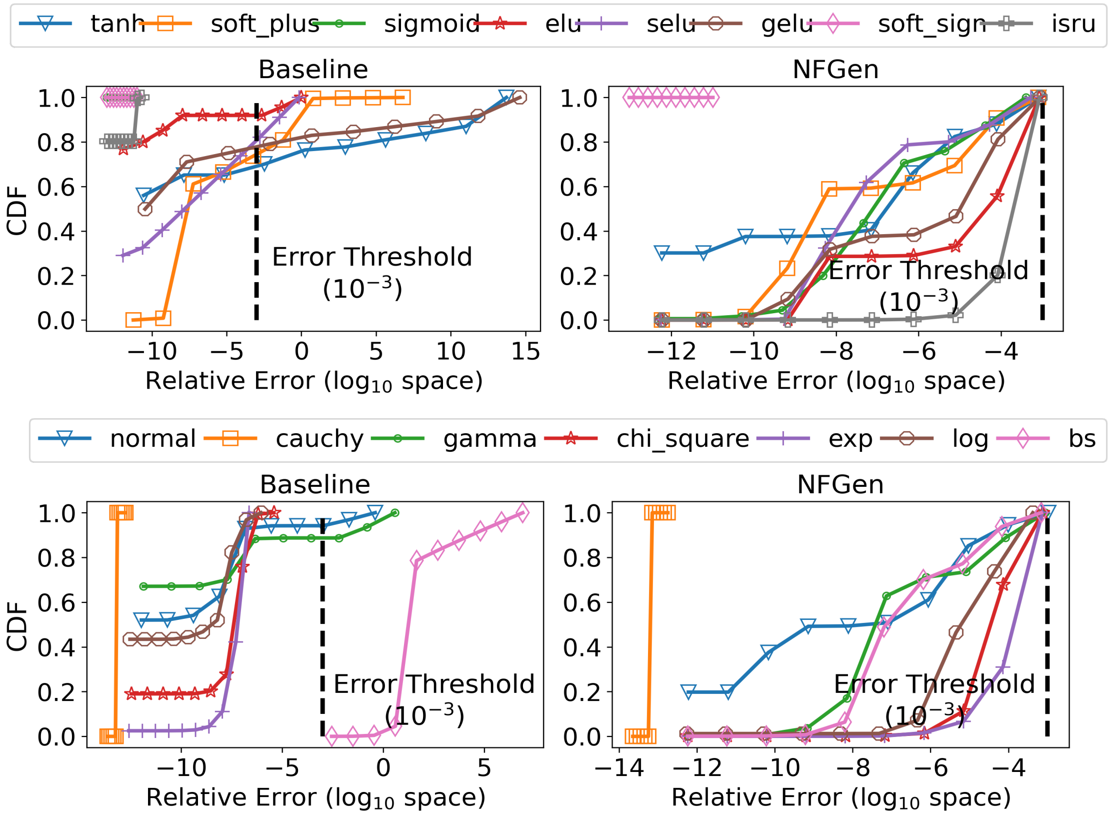
Performance Characteristic of Different MPC Deployments

- Train a deployment-specific profiler model $f_{\mathcal{S}}: (k, m) \rightarrow$ time(ms) and select the most efficient one.
- Generate code into pre-defined code templet.

Evaluation: Improved Accuracy

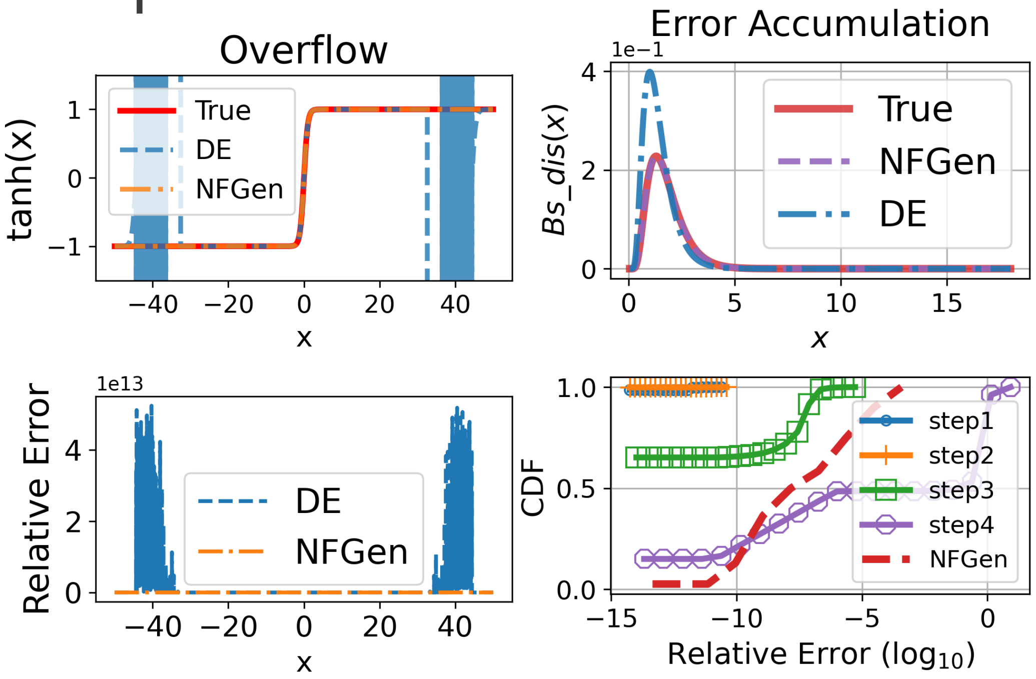
- Baseline: direct evaluation of MP-SPDZ library functions.
- NFGen: generated evaluation code.

- Overview of 15 common-used functions



Overall CDF of Relative Errors

- Improved cases

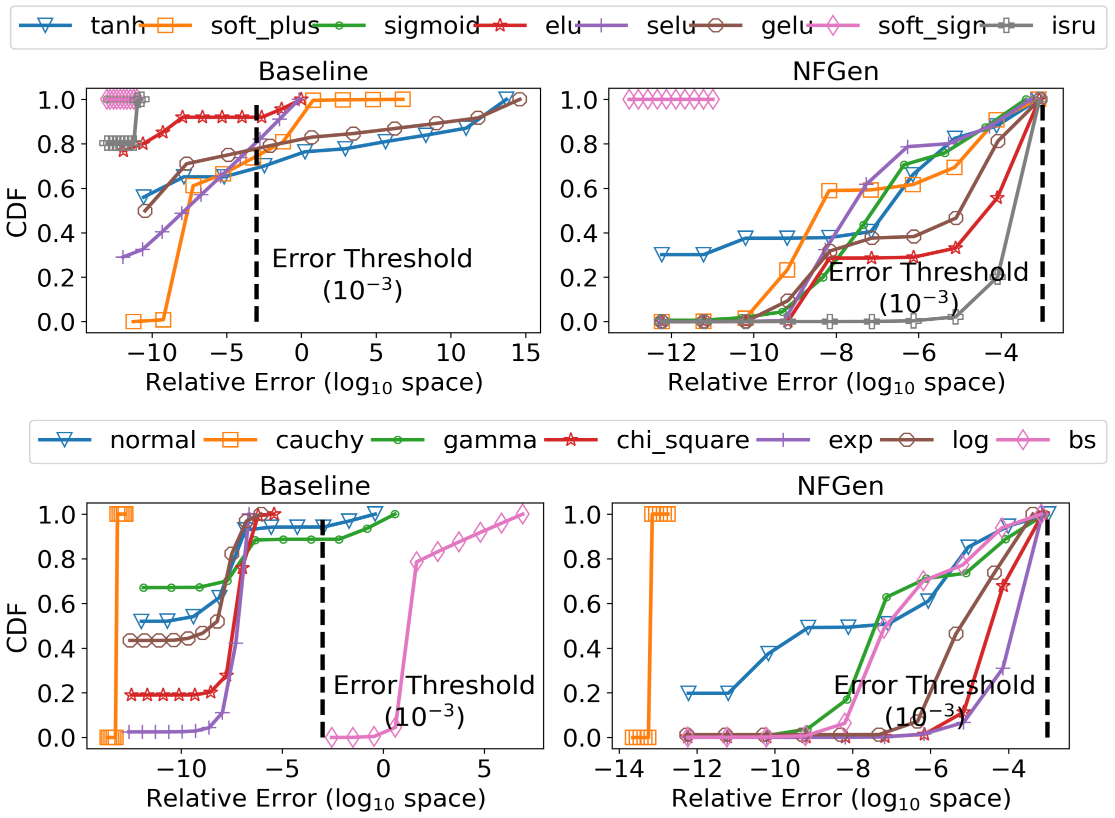


Improved Accuracy Cases

Evaluation: Improved Accuracy

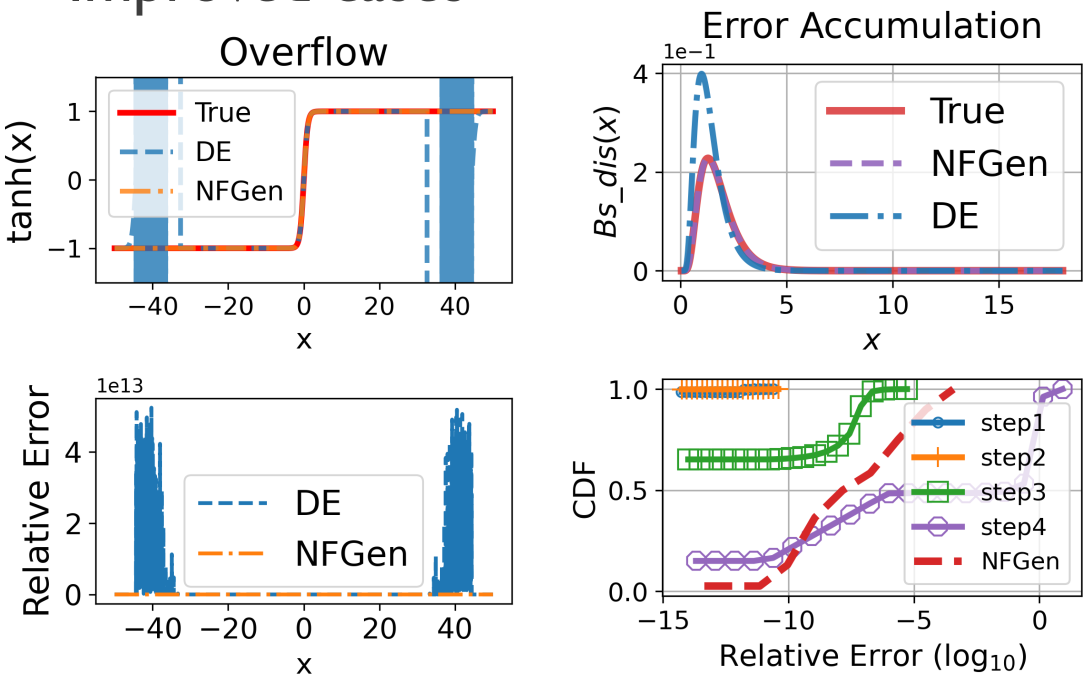
- Baseline: direct evaluation of MP-SPDZ library functions.
- NFGen: generated evaluation code.

- Overview of 15 common-used functions



Overall CDF of Relative Errors

- Improved cases



Improved Accuracy Cases

Evaluation: Improved Efficiency

MPC Sys	**Efficiency ratio(%), speedup(×)**			**Comm ratio(%), save(%)**		
	Benefit	Mean	Max	Benefit	Mean	Max
Rep2k(SPDZ)	100%	16.7×	86.1×	93%	58%	93%
RepF(SPDZ)	100%	5.3×	16.0×	87%	41%	87%
Shamir(SPDZ)	100%	4.0×	10.9×	87%	30%	83%
Ps-Rep2k(SPDZ)	67%	1.8×	6.1×	67%	8%	84%
Ps-RepF(SPDZ)	87%	2.3×	7.6×	73%	27%	87%
Rep2k(PrivPy)	100%	8.6×	29.1×	93%	57%	90%

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

- 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%.

Evaluation: Improved Efficiency

MPC Sys	Efficiency ratio(%), speedup(×)			Comm ratio(%), save(%)		
	Benefit	Mean	Max	Benefit	Mean	Max
Rep2k(SPDZ)	100%	16.7×	86.1×	93%	58%	93%
RepF(SPDZ)	100%	5.3×	16.0×	87%	41%	87%
Shamir(SPDZ)	100%	4.0×	10.9×	87%	30%	83%
Ps-Rep2k(SPDZ)	67%	1.8×	6.1×	67%	8%	84%
Ps-RepF(SPDZ)	87%	2.3×	7.6×	73%	27%	87%
Rep2k(PrivPy)	100%	8.6×	29.1×	93%	57%	90%

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

- 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%.

Evaluation: Other Benefits

- Support hard-to-compute functions

Target function	(k, m)	Fit time
$\gamma(x,z) = \int_0^x t^{z-1} e^t \, dt$	(6, 4)	1.1s
$\Gamma(x,z) = \int_x^\infty t^{z-1} e^t \, dt$	(6, 6)	1.1s
$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} \, dt$	(4, 6)	0.8s
$\Phi(x) = \frac{2}{\sqrt{2\pi}} \int_0^x e^{\frac{t^2}{2}} \, dt$	(8, 6)	1.2s

Hard-to-compute Functions

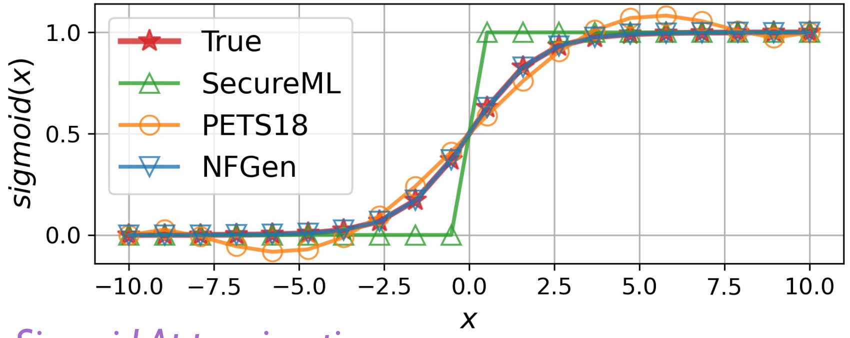
Evaluation: Other Benefits

- Support hard-to-compute functions

Target function	(k, m)	Fit time
$\gamma(x,z) = \int_0^x t^{z-1} e^t \, dt$	(6, 4)	1.1s
$\Gamma(x,z) = \int_x^\infty t^{z-1} e^t \, dt$	(6, 6)	1.1s
$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} \, dt$	(4, 6)	0.8s
$\Phi(x) = \frac{2}{\sqrt{2\pi}} \int_0^x e^{\frac{t^2}{2}} \, dt$	(8, 6)	1.2s

Hard-to-compute Functions

- Accelerate current applications
  - Approximate $sigmoid(x)$ and accelerate LR.



Sigmoid Approximations

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!