USENIX

THE ADVANCED COMPUTING
SYSTEMS ASSOCIATION

SLOTHE : Lazy Approximation of Non-Arithmetic Neural Network Functions over Encrypted Data

Kevin Nam, Youyeon Joo, Seungjin Ha, and
Yunheung Paek, *Seoul National University*

https://www.usenix.org/conference/usenixsecurity25/presentation/nam-slothe

This paper is included in the Proceedings of the
34th USENIX Security Symposium.

August 13–15, 2025 • Seattle, WA, USA

978-1-939133-52-6

Open access to the Proceedings of the
34th USENIX Security Symposium is sponsored by USENIX.

SLOTHE : Lazy Approximation of Non-Arithmetic Neural Network Functions over Encrypted Data

Kevin Nam[1,*], Youyeon Joo[1,*], Seungjin Ha[1], and Yunheung Paek[1]

[1]Dept. of Electrical and Computer Engineering & ISRC, Seoul National University
{kvnam, yyjoo, sjha}@sor.snu.ac.kr, ypaek@snu.ac.kr

Abstract

Machine Learning as a Service (MLaaS) introduces strong privacy concerns for both clients and model providers. Fully Homomorphic Encryption (FHE) offers a promising solution by enabling inference over encrypted data, but its limited expressiveness requires approximating non-arithmetic functions (NAFs) with polynomials, often leading to significant accuracy and performance trade-offs. Existing works adopt an eager approximation (EA) strategy, which statically replaces each NAF with a fixed polynomial, locking in computational errors and limiting optimization opportunities. We propose SLOTHE, a lazy approximation (LA) solution that recursively decomposes NAF codes into arithmetic and non-arithmetic sub-functions, selectively approximating only the non-arithmetic components when required. SLOTHE introduces a tunable cost model to balance accuracy and latency, and incorporates FHE-aware optimizations to eliminate redundant computation. Implemented using CKKS, SLOTHE achieves up to 42,378× lower maximum error than EA-based works, with improved inference accuracy and latency across BERT-based transformers. SLOTHE can also be adapted for MPC-based protocols, making it a flexible tool for secure neural network inference.

1 Introduction

Neural Networks (NNs), such as the rapidly growing generative AI models, lies at the heart of many applications and services such as language understanding and recommendation systems. To support these services, providers invest substantial effort in training high-quality models tailored to their use cases. To safeguard their models, often considered valuable intellectual property, service providers typically deploy inference as a remote service, hosted on their own cloud infrastructure, rather than distributing models directly to client devices. However, this paradigm raises concerns for clients, who must send their input data—often containing sensitive information (*e.g.*, voice recordings, personal records)—to the cloud for processing. To fill the gap between the privacy of the input queries and benefits of such Machine Learning as a Service (MLaaS), many researchers put efforts towards integrating privacy-preserving techniques into NN inference.

Fully homomorphic encryption (FHE) is one promising technique that can be used to realize privacy-preserving NNs thanks to its ability to perform computations over encrypted data. To enable NN inferences using FHE, the NN's code must first be translated into an FHE program that produces outputs identical to those of the original, unencrypted NN. However, this process faces a significant challenge due to the limited expressiveness of FHE. That is, typical FHE schemes [1–3] natively support only additions and multiplications, but a large portion of functions in NNs include operations such as exponentials in GeLU or Sigmoid, which cannot be directly expressed with the arithmetics. Such functions, which we hereafter note as *non-arithmetic functions* (NAFs), play a dominant role in determining the overall accuracy of NNs [4, 5], thus require to be accurately computed. A common solution to tackle this problem is *polynomial approximation* which simplifies complex functions like NAFs by approximating their mathematical formulas into polynomials, composed entirely of basic arithmetic operations that can be expressed within the framework of FHE [6–11].

We deem that all existing works [9–16] perform polynomial approximation in an *eager* manner where each NAF encountered during NN execution is immediately replaced with a polynomial that corresponds to an approximation of their mathematical formula for FHE computation. This *eager approximation* (EA) strategy has proven effective for small-scale NNs [13], however, for larger-scale NNs with a substantial number of complicated NAFs, the impact of approximation error becomes significantly more pronounced. For instance, LOHEN [17] reports -4.6% accuracy drop for MobileNet inference. We also observed from our priliminary experiments that the inference of RoBERTa-L [18] using the RTE dataset [19] induced -6.9% accuracy drop even when using a state-of-the-art strategy [10]. Moreover, the compu-

* co-first authors

tational error introduced by EA is locked in from the outset, as each NAF is replaced with a specific polynomial at once, leaving little room for further error mitigation.

In this regard, we propose a new strategy, named *lazy approximation* (LA). Our strategy is grounded in the rationale that the NAFs in NN programs are executed with codes that are implemented using C/C++ or CUDA standard libraries. Each function code is not a simple formula but a composition of various components (*e.g.*, sub-function calls), comprising both arithmetic and non-arithmetic components. This insight opens the door for a new perspective on NAF evaluation, where, instead of approximating the entire NAF at once, we may selectively approximate non-arithmetic sub-functions with polynomials while computing arithmetic sub-functions directly using FHE without introducing errors. It is noted here that not all non-arithmetic sub-functions need to be approximated instantly. Since a sub-function may itself be a composition of smaller sub-functions, it can be further decomposed into sub-components, within which new arithmetic operations may be discovered for direct FHE execution.

In our solution, dubbed *SLOTHE*, we repeat this approximation process by recursively identifying arithmetic and non-arithmetic sub-functions within their codes. The primary goal of SLOTHE is to uncover as many arithmetic components as possible for direct FHE computation, and thus to perform approximations only on *pure* non-arithmetic sub-functions that consist solely of non-arithmetic components. We have observed that SLOTHE incurs lower errors compared to traditional EA-based solutions. However, we have also found that it may suffer from a setback in terms of performance when the LA strategy is applied in a brute-force manner. The amount of computation increases as we repeatedly decompose sub-functions, which can lead to an excessive number of polynomials and computations.

Based on our findings, we have designed SLOTHE with a mechanism aimed at harnessing the accuracy benefits of the LA strategy while mitigating the performance overhead. To achieve this, we developed a heuristic cost evaluation method that allows us to control the laziness of our approximation. This design enables us to estimate the results of each recursive step, allowing us to decide which sub-function to approximate and replace with polynomials. To further enhance the performance, we adopted a method to eliminate paths in the codes that are redundant when computed with the given FHE parameter, thus reducing latency without inducing additional error. Finally, we designed SLOTHE to carry out approximations with regard to the specific needs of the developers. Specifically, SLOTHE can be programmed to find a result that minimizes the error while keeping the latency lower than a pre-defined maximum value. Conversely, it can also be programmed to find a result with an error just enough to satisfy an error tolerance while achieving competitive latency.

To demonstrate the effectiveness of our work, we have implemented SLOTHE with the state-of-the-art FHE scheme, *CKKS*, and LLVM infrastructures along with an SMT solver to process the code structures. We used SLOTHE to produce approximiations of several NAFs used in NNs and evaluated the errors, latencies, and their impact on full NN inference accuracy. In most cases, SLOTHE exhibits better results for both maximum error and latency. Our contributions can be highlighted as following :

- SLOTHE is an NAF approximation strategy based on LA. We desgined an automatic mechanism that analyzes the function bodies (FBA) to control the laziness of our strategy.
- SLOTHE can be configured to produce outputs that satisfies a set of requirements including error and latency thresholds.
- We devised an FHE-aware optimization with the goal of eliminating paths that are redundant over FHE parameters.
- SLOTHE achieves up to a 42,378× reduction in maximum error than state-of-the-art EA works. It also exhibits superiority for both error and latency.
- SLOTHE enables BERT inferences to achieve up to 6% higher accuracy and 1.18× speedup than NEXUS [10]. It also obtained zero-accuracy drop for several cases with small additional latency.

Tbl. 1: FHE Parameter Description and values used in this paper. The parameter adheres to $\lambda = 128$ security.

Parameter	Description	In this paper
N	Polynomial ring degree	2^{16}
$log(QP)$	Maximum ciphertext modulus	1630
$log(Q)$	Ciphertext modulus	1330
$log(q_i)$	Moduli in the chain	50
Δ	Scaling factor	2^{50}
σ	Standard deviation for χ_e	3.19
L_{max}	Maximum level of ciphertext	24
L	Level after bootstrapping	12

2 Background and Related Works

2.1 CKKS

CKKS [1] encodes and encrypts multiple fixed-point data into a single RLWE ciphertext. We define a polynomial ring with modulus Q as $R_{Q,N} = \mathbb{Z}_Q[X]/(X^N+1) = R_N/QR_N$ for a power-of-two integer $N \geq 2$. The encoding of CKKS is defined by the inverse of the canonical embedding (denoted as τ^{-1}). The message vector $\mathbf{m} \in \mathbb{C}^{\frac{N}{2}}$, which can pack at most 2^{N-1} values, is mapped into a polynomial $m = \mathsf{Ecd}(\mathbf{m}, \Delta) \in R_N$ by τ^{-1}, where Δ is the scaling factor that is used to encrypt lower precision into integers. Ecd returns $\lceil \Delta \cdot \tau^{-1}(\mathbf{m}) \rfloor$. The encryption of m using the secret key $s \in R_N$ is computed as $ct = (b,a) = (m + e - a \cdot s, a) \in R^2_{Q,N}$, where $a \in R_{Q,N}$ and $e \in R_N$ are random samples from $\mathcal{U}$ and χ_e, which are random uniform and gaussian distributions, respectively. Specifically, the Gaussian distribution χ_e is centered at zero with standard deviation σ, commonly chosen as

Tbl. 2: CKKS Operation latencies on GPU with parameters of Tbl. 1.

level (μs)	4	6	8	10	12
PAdd	12	18	24	32	39
CAdd	25	36	48	59	72
PMult	120	136	160	191	238
CMult	1000	1222	1351	1496	1649
Rot(<<1)	899	1182	1211	1339	1578
BTS	$\approx 93,000$				

$\sigma \approx 3.19$ [20]. $\langle ct, s \rangle$ denotes the decryption which is computed as $(b,a) \cdot (1,s) = m + e$.

To enhance computational efficiency, a recent variant of CKKS called *RNS-CKKS* adopts a method to decompose the ciphertext into multiple polynomials with smaller coefficients that can be computed independently in parallel. Specifically, the ciphertext modulus Q is decomposed into multiple smaller moduli using Residue Number System (RNS), which satisfies $Q = \Pi_{i=0}^{l} q_i$, where each modulus q_i is a prime number (q_0 denotes the base modulus). The ciphertext space $\mathbb{Z}_Q$ is mapped to $\mathbb{Z}_{q_0} \times \mathbb{Z}_{q_1} \times ... \times \mathbb{Z}_{q_i}$, based on the Chinese Remainder Theorem (CRT). The number of q_i is referred to as the maximum level (lvl) of a ciphertext, indicating the maximum number of successive multiplications available. Each q_i is typically chosen to be close to Δ [21, 22]. We refer to prior literature [1, 23] for further cryptographic details. The parameter set of CKKS should be carefully designed considering performance, data precision, and security level (λ). To adhere to a certain security level, Q should be below an upper-bound, which is determined according to N. Previous works [10, 11, 14, 20, 24, 25] usually select parameters with a security level of $\lambda = 128$-bit, which is considered as the standard [20] of FHE application. Our work uses the FHE parameter set Tbl. 1, which also adheres to $\lambda = 128$-bit security level. It is an efficient parameter adopted from HEaaN [1], a library made by the authors of CKKS. Below are representative RNS-CKKS operations :

Addition/Multiplication. The arithmetics between two ciphertexts ct_1 and ct_2 which encrypt the messages $\mathbf{m}_1$ and $\mathbf{m}_2$ results in a ciphertext that encrypts the element-wise arithmetic operation between $\mathbf{m}_1$ and $\mathbf{m}_2$. Tbl. 2 depicts the latency of CKKS operations run on GPU. The latency of multiplication grows linearly with the level (lvl) because more RNS polynomials need to be computed.

Rescale. Multiplications also accumulates the scale, which should not exceed its upper bound Q. Rescaling is thus applied which equals dividing Q by $q_l \approx \Delta$, removing the last polynomial in $\mathbb{Z}_{q_l}$ (thus reducing the current lvl).

Rotation (Rot). Given an RLWE ciphertext ct that encrypts m, the rotation of ct by an offset k results in the encryption of m' where $\mathbf{m}' = \mathbf{m} \ll k$.

Bootstrapping (BTS). When a ciphertext reaches the limit level, BTS should be performed to recover Q and lvl. Among many variants, we use the one that offers a modest precision [26], which requires a minimum of three lvl left in a ciphertext in order to initiate. BTS is generally known to take longer than a single multiplication on a scale of 2 to 3 orders of magnitude (See Tbl. 2).

Data Precision and Inherent Errors. The data precision of a CKKS ciphertext (*i.e.*, the error of the decrypted value) can be defined according to FHE parameters [20]. Without considering the injected noise, a CKKS ciphertext can be considered to have a data precision that can hold data worth the bit-length of the scaling factor. The injected noise however affects the encoded message ($m\Delta + e$), and the precision of CKKS (analogously) is around $log(q_i) - log(e)$ bits [20] (*e.g.*, the precision is reported to be around 24-bits when $log(q_i) = 42$). Multiplications accumulate the errors, but the bitwidth of the noise is then controlled through rescaling. For example, squaring a ciphertext with $m \cdot \Delta + e$ gives $m^2\Delta^2 + 2me\Delta + e^2$, rescaled to $m^2\Delta + 2me$, where the error scale is the same as the initial bit-length.

2.2 Threat Model

Our FHE (CKKS) implementation used to evaluate the approximations generated by SLOTHE follows the same threat model as numerous works on FHE-based 2PC privacy-preserving MLaaS [9–11, 14, 17, 25, 27–30] where both parties, the client and the cloud server, are semi-honest adversaries with computationally bounded computing capabilities. They follow the specified protocols without deviation, but may try to learn additional information about the data or model beyond what is explicitly allowed. Specifically, the client encrypts the input data and sends it to the cloud, which performs the computation over encrypted data. The cloud sends the final encrypted result to the client, the only one who can decrypt and see the unencrypted result. Note that this threat model is also commonly adopted by solutions adopting Multi-Party Computation (MPC) or Hybrid(FHE+MPC) protocols [15, 16, 31–34], which are alternative cryptographic backends to FHE. We later demonstrate how SLOTHE can also be applied to them, and discuss the possibility of extending the applicability to protocols that tolerate malicious adversaries.

2.3 NAF Computation over FHE

To compute an application with FHE, we first need to translate the original code into an FHE program, composed of operations that FHE supports. Arithmetic vector operations can easily be translated into FHE operations, and previous works introduce efficient packing mechanisms to maximize the benefits of SIMD computation and mitigate rotation costs [40, 41]. However, NAFs cannot be directly expressed with arithmetics. Therefore, when an NAF is called, previous works replace it with a pre-defined FHE routine corresponding to a polynomial approximation. CKKS-based methods replace NAFs

Tbl. 3: Comparison of SLOTHE and related works on NAF approximation strategies. Highlighted works are strategies from works using MPC/Hybrid protocols. We note that the underlying strategies themselves are generally not tied to a specific privacy-preserving technique.

Strategy	Related Works	Replacement Method	Automatic(dynamic) Generation?	(FHE)Param. Aware Perf. Optim.?	User-Defined Constraints?
LUT	HEIR [27], PBSNN [28], SIGMA [35], LOHEN [17]	Eager	✗	✗	None
Boolean Circuit	Gazelle [31], SHE [36], XHEC [37]	Eager	✗	✗	None
LUT&Poly	Sirnn [32], Iron [33]	Eager	✗	✗	None
Poly	Nexus [10], Bumblebee [16], Bolt [15], Park et al. [29], Crypten [38], Cheon et al. [14], Lee et al. [39], Jang et al. [9]	Eager	✗	✗	None
	NFGen [34]	Eager	✓	✗	Error
	SLOTHE	**Lazy**	✓	✓	**Error & Latency**

with pre-made polynomial approximations, just as how manually programmed FHE codes are designed.

Some other works [27, 42] adopt TFHE, another FHE scheme which has different characteristics with CKKS. TFHE offers an operation called programmable bootstrapping (PBS), which enables us to compute arbitrary functions during bootstrapping in a LUT fashion. However, the precision of TFHE tends to be much lower compared to CKKS when using similar bit-length for q. For example, using a TFHE parameter with q being around 45-bits gives us a precision of only around 5-bit integer [28], and using such parameters, the LUT computation for NAFs induces errors that are sometimes even higher than the polynomial approximations [43]. For example, Park et al. [29] reports that the multi-bit LUT-based method induces a maximum error which exceeds 1.5×10^{-2}, while requiring 17,438× more latency than the polynomial method for the logistic function. It is possible to minimize the error by adopting a bit-wise encryption (*i.e.*, single bit in each ciphertext) and use PBS to evaluate boolean gates, from which we can evaluate arbitrary boolean circuits [36, 37, 44, 45]. However, this induces massive computational overhead as bootstrapping is called as much as the number of boolean gates in the circuit, drastically increasing the overhead. For example, computing ReLU for one MNIST image (28×28 indices) takes 59.6 seconds when computed with boolean circuits on GPU [37], which only takes 1.2 seconds using LUTs [17]. Results worsen for complex NAFs. We pre-emptively note that SLOTHE enables us to compute a full transformer within 50 seconds without accuracy loss (§5.2).

2.4 Polynomial Approximation Strategies

Tbl. 3 shows similarities and differences between SLOTHE and previous works on NAF approximation strategies. Strategies from FHE/MPC/Hybrid techniques often share common characteristics. In particular, the underlying polynomial, LUT, or circuit used in the strategies are generally not tied to the specific privacy-preserving technique. The approximation itself does not depend on the cryptographic backend but rather on the functional behavior being approximated. For example, NEXUS [10], which is an FHE-based work, adopts the piecewise-polynomial approach of Bumblebee [16], which uses a Hybrid protocol. The implemented techniques are indeed different – FHE requires computing a high-degree polynomial [11] for comparison to predicatively compute the polynomials, while Bumblebee uses Oblivious Transfers (OTs) for the segment selection. However, there is no restriction in computing the same piece-wise polynomials with the other approach. Same holds for LUTs and boolean circuits, computed with programmable bootstrapping in FHE works while they can also be computed with MPC protocols as Millionaire protocol, Oblivious Transfer, or Garbled Circuits [15, 31–33].

All previous works simply replace NAFs eagerly, as discussed in §2.3. Preliminary EA works adopt the well-known numerical approximation algorithms [6–8], but they were not effective enough to afford the level of preciseness and value range that highly sophisticated NNs require [12, 46]. Crypten [38] identifies the most appropriate numerical algorithm for each non-arithmetic they investigate, but the results were not accurate enough as comparatively shown in following works [35]. Other works that adopt LUTs and boolean circuits are scalable when data precision increases, but their latency grows proportionally as they lack of parameter aware performance optimizations. Later studies adopted additional efforts in designing the polynomials. Several works [10, 11, 15, 16, 32, 33] addressed the issue by adopting manually crafted approximations to make forms that are retrofitted to their specific application. Sirnn [32] uses both LUTs and polynomials as building blocks to design approximations for NAFs. IRON [33] builds on Sirnn, improving its performance. Bolt [15], Bumblebee [16], and NEXUS [10] use customized piece-wise polynomials, resulting in improved latency and error than IRON and Sirnn.

NFGen [34] offers an automatic approximation mechanism. It first generates a single polynomial approximation, then if the result does not fit the error requirement, it splits the input domain into half, produces two distinct polynomial approximations for each, and repeats the process until the piece-wise polynomials meet the requirement. One limitation of such polynomial EA works is that there are no concrete grounds on which basis polynomial or piece-wise breakdown works better than others, thus requiring long empirical efforts. Also,

Tbl. 4: Accuracy of unencrypted backbone transformers and their corresponding FHE programs using state-of-the-art EA [10] for NAFs. † denotes reported results from previous literature [10]. The other results are measured 10,000 queries using pre-trained models.

Dataset	NN type	Unencrypted Backbone Acc.	Acc. w. HE (EA from [10])	Accuracy Drop
RTE	† BERT-base [47]	70.04%	69.88%	-0.16%
	ALBERT [48]	89.2%	83.4%	-5.8%
	RoBERTa-L [49]	86.6%	79.7%	-6.9%
CoLA	BERT-base	56.7%	53.2%	-3.5%

the candidate pool is almost infinite (as much as we can split a range into half) while NFGen only splits domains into halves thus does not cover other possible cases. It also does not consider latency as a constraint, which is a crucial requirement for developers. To make things worse, piece-wise polynomials increase the overhead whereas they do not have any means to address it. Whereas piece-wise polynomial design requires conditional branches between them to select one of the pieces, FHE cannot perform such behavior of choosing from options. Under such constraint, FHE adopts predicated execution [10, 27], which results in increasing the overhead as much piece-wise polynomials as we have.

Dominance of Polynomial Approximation Errors. Polynomial approximation accounts for the majority of the computational errors when computing NNs using FHE. The parameter sets commonly used by previous works [5, 20, 25] provide decent data precision able to accurately decrypt values below 2^{-20} ($\approx 10^{-6}$). Ju et al. [5] report that the mean absolute error from the FHE inherent nature (introduced in §2.1) during NN computation is merely around 1.4×10^{-6}. When it comes to polynomial approximation, the error was 1.5×10^{-4}, which is about 100× bigger than the inherent error of FHE. Similarly, Park et al. [29] and NEXUS [10] report the maximum errors due to Tanh and GeLU approximation to be $4.0\times\ 10^{-2}$ and $2.0\times\ 10^{-2}$, respectively, which is around four orders of magnitude greater than the inherent error of FHE.

3 Motivation

EA Induces Significant Accuracy Drops. The impacts are significantly bigger on those performing comparatively complicated tasks as well as those that are fine-tuned or trained with specific methods to obtain high accuracy. Tbl. 4 shows the accuracy drops of several transformer inference tasks over CKKS using state-of-the-art EA for NAFs [10], which manually crafted polynomial approximations for NAFs in BERT-base. ALBERT [48] and RoBERTa [49] are based on BERT [47] architecture, fine-tuned and trained with avant-garde methods [18, 50] to achieve higher accuracy (*e.g.*, the accuracy of the RTE inference of the BERT-base is 70.4%, while ALBERT is 89.2%). According to results from previous literature [10], the manually crafted EA was enough to keep the accuracy drop of the BERT-base under -0.5%. However, for the avant-garde ones, the accuracy drops are much bigger, up to -6.9%, suggesting that previous EA works are not fit as general solutions. We also report that BERT-base inference for the CoLA dataset induced -3.5% accuracy, showing that the approximations used in NEXUS have inconsistent effects when running different tasks on the same NN. Our core narrative is that, the goal of MLaaS is to offer NN inferences with high accuracy, which results from numerous trials and errors in designing and fine-tuning the models retrofitted for specific needs. Thus, a method to design NAF approximations that minimizes accuracy drops for *arbitrary* NNs is crucial.

Tbl. 5: Number of operations in the function body of Tanh(x) [51].

# of Blocks.	Arith.	Non-Arith.	Condition	Function Call
Tanh(x)	11	4	7	4

Potential Benefits of LA. It goes without saying that there are more codes below a function call. Tbl. 5 lists the number of operations in the function body of Tanh(x) from GNU C [51] library analyzed with LLVM IR [52]. We exclude the non-essential operations (that can be omitted when running with FHE), leaving 11 arithmetic operations which can be computed with FHE without inducing errors. The non-arithmetics are divisions and comparisons, which are known to be able to be approximated with minimal errors with a well-known numerical method [11]. This suggests that instead of approximating Tanh(x) as a whole, translating its function body may lead us to a set of FHE operations with less computational errors. Also, the 4 additional subsequent function calls inside Tanh(x) can also be analyzed in the same way, recursively performing the code analysis.

The Necessity of Controlling the Laziness. FHE cannot perform decision-making tasks, so to compute conditional branches, the custom wisdom is to use predicated execution to compute the branches, inducing colossal computational overhead [27,45]. Accordingly, blindly pursuing LA induces a tremendous overhead as much as the amounts of code increase by expanding the sub-function calls with their function body codes. For example, fully pursuing LA with the code of GeLU results in a FHE program worth 88.25 seconds of latency, which is 307.49× slower than the EA polynomial used in a previous work [10] (our evaluation show more quantitative results). To address this, we devise a method to control the laziness and selectively process FBAs only when required. We also implement FHE-aware optimizations that reduces paths that are redundant when computed with FHE.

4 SLOTHE Design

4.1 Overview

As stated in §1, NAF refers to the functions that include non-arithmetic computations called during NN inference com-

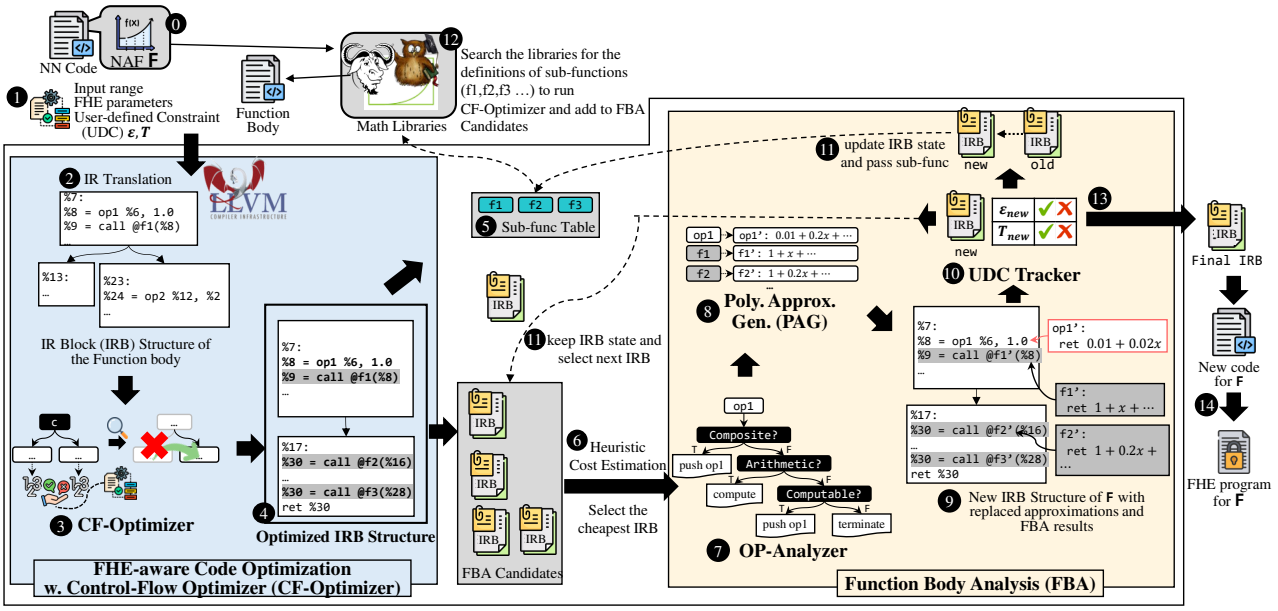


Fig. 1: SLOTHE Overview. IRB stands for Intermediate Representation Block.

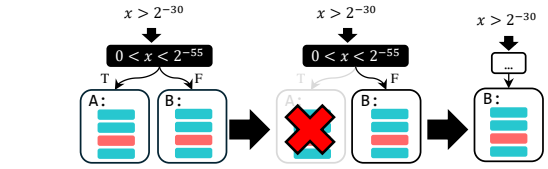
putation (*e.g.*, Tanh, GeLU). Fig. 1 depicts the overview of SLOTHE. Let F be an NAF called during inference (❶). When F is called, SLOTHE reads its source code of their definitions from the libraries and the user-defined constraints (UDC) including error tolerance ε, performance threshold T, and the FHE parameters to be used (❶). To facilitate the management of the code structure, SLOTHE uses LLVM to translate the code into Intermediate Representations Block (IRB) structure (❷). Each line in IRBs corresponds to an instruction-level operation (*e.g.*, add, mul) including sub-function calls directing other codes. For example, the function code $f(x) = x + 1$ would be translated into $\%1 = \%0 + 1.0$ in the IRB where %0 refers to the input and %1 to an intermediate register.

SLOTHE consists of two main phases, the FHE-Aware Code Optimization by the Control Flow Optimizer (CF-Optimizer), and the Function Body Analysis (FBA). CF-Optimizer (❸) serves the role of reducing the number of paths (thus code blocks) in the IRBs that are redundant under the given FHE parameters, in order to reduce the amount of predicated computation while mitigating notable increase in error. We describe its details in §4.2. From the optimized IRB of F (❹), SLOTHE now analyzes its sub-components to decide the portions to approximate, which corresponds to the second main phase of FBA, which we will describe the details in §4.3. Initially, SLOTHE takes the output IRB from CF-Optimizer and reads each line to classify the operations into elementary and composite ones with the OP-Analyzer (❼, §4.3.1). The non-arithmetic ones are sent to the Polynoial Approximation Generator (PAG) which produces basic polynomial approximations for each of them (❽, §4.3.2). SLOTHE then updates the IRB by replacing the non-arithmetics with the code corresponding to the generated polynomial computation. In the overview example, op1, f1, and f2 are approximated into polynomials in ❽, which replace each corresponding line in the new IRB as ❾. The, the User-Defined Constraint (UDC) Tracker determines if the requirements are met with the new (updated) IRB (❿, §4.3.3).

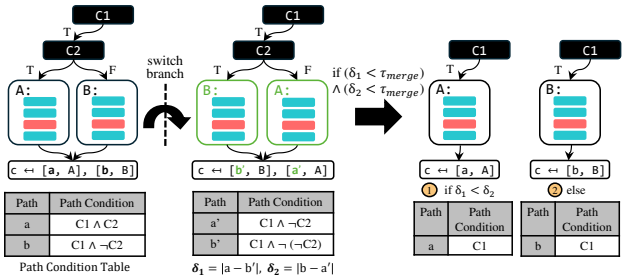
If there are any budget left, SLOTHE saves the current IRB and tries additional FBAs, diving into the code of a sub-function (⓫). To facilitate this step, we implement a structure called the sub-function table (❺), which refers to the list of sub-functions identified by the function call instructions in the current IRB structure. In the example, note that there are three call instructions, each pointing to f1, f2, and f3. SLOTHE repeats the first phase for the sub-functions in the table, starting from reading their function bodies from the code libraries (⓬) the creating optimized IRBs for each sub-functions, which are now the candidates for the next FBA. From the candidates, we perform a heuristic cost estimation to select one as the input of the next FBA (❻, §4.3). If the resulting IRB of the new FBA also meets the requirements, SLOTHE, again, performs an additional FBA. This process is recursively executed until the resulting IRB exceeds the UDC thresholds. If so, UDC-Tracker requests to stop, and sends the previous IRB as the final output (⓭), which is now composed of arithmetic operations only, thus can be directly translated into an FHE program (⓮). The details of each process will be detailed in the following subsections.

4.2 FHE-Aware Code Optimization

FHE cannot make decisions in choosing conditional paths since the values are encrypted, thus they are computed with



(a) Unreachable path elimination.



(b) Equivalent path merging. τ denotes the difference threshold for merging.

Fig. 2: Examples of FHE-aware code optimization.

predicated execution. The more paths an IRB has, the more FHE computational overhead is induced. To address such issues, CF-Optimizer looks for opportunities in reducing the number of paths considering the characteristics of the input FHE ciphertexts.

4.2.1 Unreachable Paths Elimination

NAF codes in libraries such as `glibc` and `Eigen` are generally implemented to create precise results for (double) floating-point data precision. Our intuition is that the FHE parameters have different data precision and constraints, which makes a significant number of paths unreachable when computed with FHE. Fig. 2a shows an example of the unreachable path using an FHE parameter set with a 50-bit q_i, which gives around 30-bit precision under the radix point. With this parameter, the condition `if` $(0 < x < 2^{-55})$ in the original code should always be false, thus we can eliminate the redundant path as well as the conditional branch. We also add criteria related to the encrypted input value range. In an NN, let us assume that the input of an NAF layer is the output of a normalization layer, where the outputs are normalized between $[-1, 1]$. Although we do not know the actual value, it is possible to distinguish paths that cannot be reached (*e.g.*, `if` $|x| < 22$ will always be true). To identify redundant paths, the CF-Optimizer first lists the path conditions in a symbolic execution-like manner and translate them into input related clauses using the dependency analysis results from LLVM. For example, when the initial input is x and a given condition is "$y < 2$?" while "$x = 2y$", the condition is translated into "$x < 4$?". We then assess the condition value range or results range for each path and check if the condition is reacheable with the given FHE parameter. We used an SMT solver [53] with the help of a gradient optimizer as our numerical analyzer. If the conditions are met, the solver notifies SLOTHE

Algorithm 1: Function body analysis (FBA) flow.

Input: NAF F, IRB IRB_{in}, Error tolerance ε, Performance threshold T, FHE parameters p, UDC mode M
Output: IRB IRB_{new}

1 **Initialize:** $IRB_{old} = IRB_{NAF}$, $AQ = []$, $\varepsilon_{prev} = 0$;
`// run OP-Analyzer (§4.3.1)`
2 AQ = **OP-Analyzer**(IRB_{in}) ;
3 **if** $AQ == \perp$ **then**
4 | **return** IRB_{old} ;
5 $IRB_{tmp} = IRB_{old}$;
6 **for** *operation* op_i *in* AQ **do**
7 | pag_res = **PAG**(op_i) `// run PAG (§4.3.2)`
8 | IRB_{tmp}= IRB_{tmp}.replace(op_i, pag_res) ;
`// run UDC-Tracker (§4.3.3)`
9 signal = **UDC-Tracker**($F, IRB_{tmp}, \varepsilon_{prev}, \varepsilon, T, p, M$) ;
10 **do** signal ;
11 `wait ; // until next FBA call then goto #Line 2`

to eliminate the paths and simplify the IRB.

4.2.2 Equivalent Paths Merging

We further reduce the number of conditional branches by merging paths that return equivalent results (with negligible differences) when computed over FHE. Fig. 2b depicts the method, which is similar to the elimination of the unreachable paths. For the branch `if` $(C2?\ A : B)$ in the figure, we compute the difference between the current result and that of when we negate the condition to `if` $((\sim C2)?\ A : B)$, and check if the two blocks are functionally equivalent under the FHE parameters. This process is executed using the same method in §4.2.1. Whereas the SMT solver was used to check if the value range in the condition clause was recheable, it now also needs to check if the threshold conditions are met. Specifically, we check if $\delta_1 = |a - b'|$ and $\delta_2 = |b - a'|$ are smaller than a threshold (τ_{merge}) which we initially set as the minimal representation fraction of the used FHE parameter (*e.g.*, 2^{-30} in the previous example). If both δ_1 and δ_2 are smaller, it means that the results generate the same results using FHE, thus we can eliminate the conditional branch and merge the two paths without inducing any FHE computational error. Therefore, SLOTHE eliminates one of the two blocks, resulting in merging the two paths into one. Note that developers can set τ_{merge} to a higher value, reducing the constraint required for merging the paths in the trade of some additional error. Allowing this trade-off indicates that SLOTHE considers another source of error, which not only can be allowed since the solution is effective as long as the overall error adheres to the total error tolerance (ε) but also has the benefit of enlarging the candidate pool for the final solution, potentially leading us to a result that better fits the developers' needs.

4.3 Function Body Analysis (FBA)

Algo. 1 shows an algorithm corresponding to the FBA process illustrated in Fig. 1. The overall flow is as follows. First, we

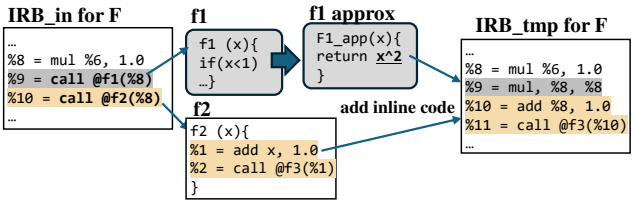


Fig. 3: Example of IRB update during FBA.

produce an approximation of NAF (IRB_{tmp}) by replacing its non-arithmetic operations with polynomials, and evaluate if the current version satisfies the UDC given by the developers. This procedure includes classifying the operation types (§4.3.1) and generating an approximation queue (AQ), a list of the sub-components that are approximated by PAG (§4.3.2). Then, the UDC-Tracker (§4.3.3) checks if we have a surplus latency budget or if the error is still higher than requested. If so, we select one of the non-arithmetic sub-functions that are called in the NAF and perform FBA. In other words, instead of replacing it with a polynomial, we analyze its internal code with the expectation of reducing the error in trade for some additional latency. Note that after analyzing the IRB of the selected sub-function (IRB_{in}), we modify the IRB of NAF (IRB_{tmp}) by expanding the sub-function call with an inline code for IRB_{in}. Also, the next FBA target can be selected not only from the other sub-functions of NAF but also the subsequent sub-functions that are called by the sub-function currently being analyzed. We recursively repeat this process until the UDC requirements are met.

Fig. 3 shows an example of IRB update, where an NAF F has two sub-function calls. If f1 is determined to be approximated, thus the call instruction is replaced with inline code that corresponds to the polynomial computation. For f2, we decide to look deeper, thus we replace the function call for f2 with inline codes from the function definition of f2. From the modified IRB, we repeat the process to deal with the new function call of f3 imported from the inline code of f2. In a sense, we gradually postpone approximation until we find a result that adheres to the UDC requirements. After iterations, the final IRB will consist of only arithmetics that can be directly translated into an FHE program.

FBA Target Selection. Initially, there is only one candidate since we only have the IRB of NAF. However, to carry out another round of FBA (*e.g.*, when we have a latency budget left), we need to choose the next FBA target from the sub-functions called in the IRB of NAF. For example, let us assume the case of Fig. 1 where NAF has three sub-functions, f1, f2, and f3, which are the potential targets of the next FBA. From the candidates, we aim to choose the one that is likely to induce the least amount of overhead with a heuristic estimation of the latency by counting the number of condition clauses and sub-functions. This is possible because both conditional clauses and sub-functions are going to be approximated into polynomials with multiplication depth alike, from which we can deduce the number of required BTS, and the two operations usually dominate the FHE overhead. SLOTHE searches the definitions of the sub-functions from the math libraries, translates them into IRB, and runs CF-Optimizer to generate optimized IR blocks for the three sub-functions. From the optimized IRBs, SLOTHE chooses the one with the minimal number of condition clauses and sub-functions as the target of the next FBA. After the FBA (assume f1), if another FBA is called, SLOTHE repeats the process, but now chooses from a candidate pool that not only includes the ones that FBA did not receive (f2 and f3) but also the subsequent sub-functions of the one that was analyzed by the previous FBA for f1.

Algorithm 2: OP-Analyzer flow.

```
   Input: IRB IRB,
   Output: Approximation queue AQ
1  Initialize: ops ↤ list of operations in IRB ;
2  for operation op in ops do
3      if op is composite operation then
4          AQ.push(op) ;
5      else
6          if op is arithmetic operation then
7              compute op ;
8          else
9              if op is computable then
10                 AQ.push(op) ;
11             else
                   // alert to abort the FBA;
12                 return ⊥ ;
13 return AQ ;
```

4.3.1 OP-Analyzer

Algo. 2 shows the algorithm of OP-Analyzer. OP-Analyzer has the role of classifying the operations by reading the operator type of each line of the input IRB (IRB_{in}). If an operation is a composite one, we add it to the AQ. If it is an elementary and arithmetic operation such as multiplication, we simply replace it with the corresponding FHE operation. If it is an elementary but non-arithmetic operation that can be approximated (*e.g.*, division), we also add it to AQ. If the operation is a non-arithmetic that cannot be expressed nor approximated with FHE such as typecast operators, it means that we cannot perform FBA for IRB_{in}. Therefore, we abort the iteration, rollback to the previous IRB state of NAF (IRB_{old} in Algo. 1), where IRB_{in} is not analyzed, but rather have the sub-function call of IRB_{in} is replaced with a polynomial as a whole.

4.3.2 Polynomial Approximation Generator (PAG)

PAG is merely a component of SLOTHE that generates polynomial approximations for the operations (non-arithmetics and composite) in AQ sent by OP-Analyzer. In order to highlight the strength of our primary contributions of the laziness by the recursive FBA and other optimizations, we do not implement PAG with powerful approximation methods such as the manual or piece-wise polynomial methods introduced in

Algorithm 3: UDC-Tracker algorithm.

Input: NAF F, IRB IRB_{tmp}, Previous IRB's error ε_{prev}, Error tolerance ε, Performance threshold T, FHE parameters p, UDC mode M, Input range $[a,b]$
Output: signal to FBA

1 **Initialize:** $\varepsilon_{new} = 0$, $T_{new} = 0$;
// calculate ε_{new}, default $[a,b]$: $[-8,8]$
2 $\varepsilon_{new} \leftarrow$ the maximum of $|F(x)\text{-}IRB_{tmp}(x)|$ in $[a,b]$;
// calculate T_{new}
3 $T_{new} \leftarrow T_{CKKS}(IRB_{tmp}, p)$;
// satisfaction condition of given UDC
4 **if** $M ==$ *minErr* **then**
5 | $cond = (T_{new} > T)$;
6 **if** $M ==$ *minTime* **then**
7 | $cond = (\varepsilon_{new} < \varepsilon)$;
8 **if** $\varepsilon_{new} > \varepsilon_{prev}$ **then**
9 | signal $\leftarrow$ "keep IRB_{old} and select next IRB" ;
10 **else**
11 | **if** !$cond$ **then**
12 | | signal $\leftarrow$ "IRB_{old}=IRB_{tmp} and pass sub-func" ;
13 | **else**
14 | | **if** *minErr* **then**
15 | | | signal $\leftarrow$ "terminate FBA and return IRB_{old}" ;
16 | | **if** *minTime* **then**
17 | | | signal $\leftarrow$ "terminate FBA and return IRB_{tmp}" ;
18 **return** signal ;

§2. Rather, we simply use an open-source program which generates one-pieced polynomials based on the naive Remez algorithm [54]. That is, we aim to show that we do not necessarily need the sophisticated EA mechanisms of previous works to generate efficient LA results. For each operation, we generate remez polynomials with degrees between 10-27, which require 4 or 5 multiplicative depths to generate the maximum degree ($4 \leq \lceil log(degrees) \rceil \leq 5$), and accordingly, each generated polynomial would induce similar overhead, for which reason we practically count only the number of sub-functions in the heuristic cost estimation §4.3. We chose this upper bound following the wisdom of prior work [55], where the authors note that while remez polynomials with higher orders may be more accurate in the unencrypted code, the precision of the FHE parameters is not able to hold the minimal increment required for the coefficients, thus leads to more errors due to precision loss. From the generated polynomials, we evaluate the maximum/average error of 10,000 samples and select the best option. Only for divisions and comparison we use Goldshmidt polynomial [8] and minimax algorithm [24] because they are very well-known methods for the two frequently used operations.

4.3.3 UDC-Tracker

UDC-Tracker determines if the current IRB of NAF (IRB_{tmp}) satisfies the UDC requirements and decides whether to perform additional FBA or not. Its process is illustrated in Algo. 3, starting by calculating the maximum error and latency of IRB_{tmp} (line 2-3). The maximum error is calculated by measuring the difference between the actual return value of the NAF function and that of IRB_{tmp} for 10,000 random samples from the input range. Unless specified otherwise by a precedent condition (like §4.2.1), we use $[-8,8]$, following the observation of previous works that the input of NAFs in NNs are generally within such range [10, 11]. The latency is heuristically estimated by accumulating the costs of the FHE operations required for IRB_{tmp} from the pre-measurements of each one of them (*e.g.*, Tbl. 2). As previously introduced, SLOTHE has two specific modes. minErr targets minimizing the error while keeping the latency below a threshold T (line 5). If the condition is not met, it means that there is a latency budget left from which we can exploit for additional FBA with the expectation of further error reduction. Accordingly, we update IRB_{old} with the result of the current FBA (IRB_{tmp}), add the subsequent sub-functions to the FBA candidates, from where we select one and begin the next FBA (line 12). If the condition is met, it means that the latency exceeds the threshold. Thus, we abort the current FBA and send the previous IRB of NAF (IRB_{old}) as the final output (line 15). The other mode, minTime, targets generating a result with a small latency while inducing an error that is below the error tolerance (line 7). If the condition is not met, it means that the current error is still bigger than the tolerance and thus requires additional FBA for further error reduction (line 12). If the condition is met, where the error is now below the tolerated value, we end the recursive FBAs and return the current IRB of NAF (IRB_{new}) as the final output (line 17). For both modes, if the measured error is higher than that of the state before the current FBA, we abort, roll back the step, and start again by selecting another sub-function as the FBA target.

5 Evaluation

We answer the following about the effectiveness of SLOTHE.
- Are SLOTHE approximations competitive compared to previous works? (§5.1, §5.3)
- Are SLOTHE approximations effective in reducing NN accuracy loss? (§5.2, §5.3)
- Does SLOTHE provide the appropriate result according to the user's requirements? (§5.1, §5.2, §5.3)
- Is SLOTHE applicable to MPC/Hybrid protocols with minimal modifications? (§5.4)

NAF and NN Benchmarks. NAFs such as GeLU called in PyTorch NN framework [56] in turn calls C codes from Eigen [57] which in turn calls glibc [51], for which reason we target their codes. For traditional ones such as ReLU, we use the auto-generated C codes Pytorch creates to compute. We evaluated 7 NAFs that are used in the trending gen AI transformers (*e.g.*, GeLU, Swish), or evaluated in previous works, including traditional ones. For the NNs, we use 3 transformers, BERT-B, RoBERTa-L, and ALBERT-xxl [18, 47, 50]. We evaluate the accuracy of the QNLI, RTE, and

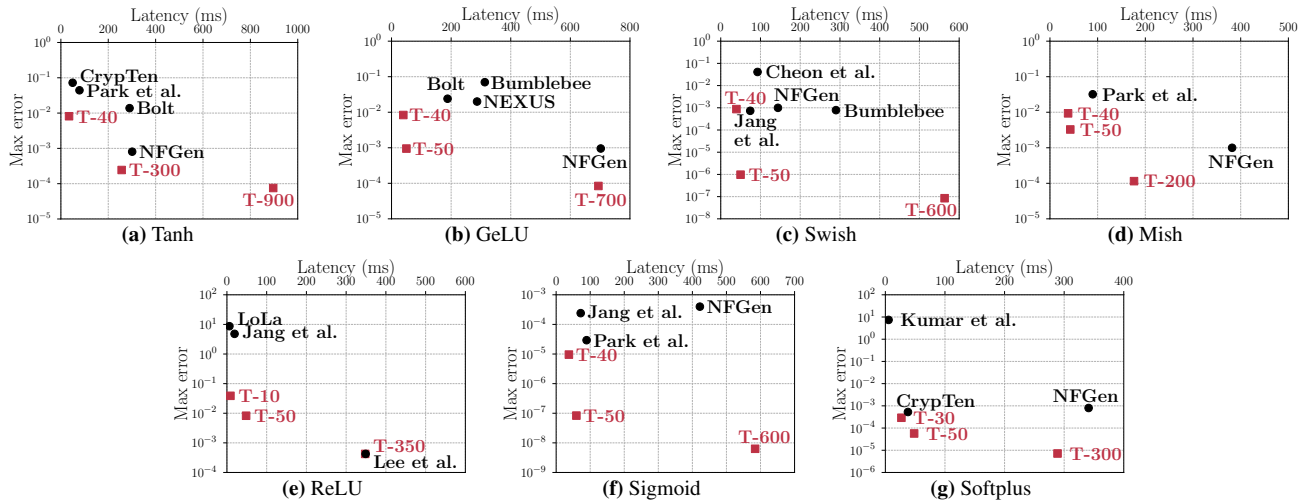


Fig. 4: Maximum error and latency of polynomial approximations of SLOTHE and previous works run with CKKS (left bottom is better).

RACE dataset [19, 58]. We use open-sourced pre-trained NN codes [48, 49] to demonstrate the effectiveness in preserving the accuracy of arbitrary NNs without the need of additional training adapted to HE which previous works rely on [59–61].

Error and Accuracy Measurements. Unless otherwise specified, we measure errors and accuracy as follows. For NAFs, we measure the difference between the decryption of the FHE computation result of the generated LA approximation and that of the *unencrypted* computation for the original NAF. We measure the maximum and average error of 100,000 randomly sampled input cases in $[-8, 8]$, a range that accommodates almost all NAF inputs of BERT, according to the previous work [10]. The NN accuracy refers to the Top-1 accuracy of 1,000 queries, which is a widely adopted method.

Experimental Environment. We run all experiments using a machine with Intel Xeon Gold 6326 CPU, 1TB of main memory, and one NVIDIA A6000 GPU with 40GB of memory using unified virtual memory (UVM). We used the Liberate-FHE GPU library [62] as the underlying RNS-CKKS library to generate FHE programs. We chose Liberate-FHE over other options used in several previous works [5, 9, 14] because they either lack of GPU support [22] or are not fully programmable [21]. Furthermore, BTS implementation in some works [10, 11] exhibit relatively sluggish latency [21, 62]. We note that the end-to-end latency of NEXUS [10] measured with our setting was highly competitive compared to their literature (BERT-base is reported to be 37.34 seconds with 4× A100 GPUs, while our measurement achieves similar latency with only one A6000 GPU) making it a more appropriate baseline for our study. For the previous works on RNS-CKKS based EA using polynomial approximations, we make sure to compare ours with previous works fairly by running them using the same version of Liberate-FHE. As for the LUT which require TFHE PBS, we measure their performance using the official TFHE library [43]. We implemented the shell code of SLOTHE and its internal functionalities with C/C++ language, using the infrastructures of LLVM 16.0.0 (*e.g.*, IR, dependency analysis, condition table generation).

Compared EA Works. For each NAFs, we compared SLOTHE against the state-of-the-art approximation strategies. Note that some NAF approximation strategies are from works that use MPC/Hybrid protocols. We compute the underlying polynomials and LUTs with FHE, as discussed in §2.4, using their open-source [10, 34] if they are available. If not, we refer to their literature and reproduce the polynomials. For the polynomial approximation strategies, we run with the same RNS-CKKS parameters as SLOTHE for fair comparison. For the LUT evaluation, we use TFHE-rs to design programmable bootstrapping that adheres to the precision in their literature. The specific EA works we compare with are as below :

- For each NAF, we select multiple manual EA approximation formats used by state-of-the-art works from both FHE [9–11, 13, 14, 29, 63] and MPC/Hybrid [15, 16, 38].
- NEXUS presents transformer inferences over FHE, which we use as our main comparison baseline for the full NNs and evaluate the impact of replacing their NAF approximations with those of SLOTHE.
- We also compare against NFGen [34], an open-source framework that produces piece-wise polynomial approximations. We configure NFGen to produce results with errors lower than 10^{-3}, because NFGen failed to generate results with a smaller threshold such as 10^{-4}.
- We compare against HEIR [27] as our comparison baseline for LUT strategy. We also measure the performance of an LUT+poly approach from MPC/Hybrid work of IRON [33]. We generate the codes with HEIR [27] using different TFHE parameters, and implemented with TFHE-rs.

Notations. We use notations to indicate the different results that SLOTHE generates. 'T-k' refers to the result when SLOTHE is in minErr mode with k milliseconds as the la-

Tbl. 6: Max./Avg. Error (in 10^{-3}) and amortized latency (μs) of GeLU approximations of SLOTHE and LUT strategy.

μs, 'k'=10^3	SLOTHE T-50	Poly. EA [10]	LUT EA [27]		LUT+Poly. EA [33] (q=55)
			q=55	q=16	
Max. Err. (10^{-3})	**0.95**	20.0 (21.0×)	22.2 (23.4×)	153.2 (161.3×)	7.46 (7.85×)
Avg. Err. (10^{-3})	**0.30**	1.1 (3.7×)	4.1 (13.7×)	81.0 (270.0×)	0.83 (2.76×)
Amortized Latency (us)	**1.52**	8.76 (5.8×)	28k (18k×)	4.1k (2.7k×)	28k (18k×)

Tbl. 7: GeLU Latency Breakdown (ms) and Max. error of SLOTHE.

Latency (ms)		Padd	Cadd	Pmult	Cmult	BTS	Total	Max. Err.
GeLU	T40	0.11	1.33	11.42	26.64	0.00	39.50	8.35×10^{-3}
	T50	0.15	2.48	18.93	28.34	0.00	49.90	9.49×10^{-4}
	T700	0.18	0.70	4.87	139.99	548.36	694.10	8.41×10^{-5}

tency upper-bound threshold. 'E-k' refers to the result when SLOTHE is in minTime mode with 10^{-k} as the error tolerance.

5.1 NAF Evaluation over FHE

Error and Latency. Fig. 4 depicts the maximum error and latency of SLOTHE approximations in minErr mode and polynomial approximation strategies from prior works (referred by their names). We measure the latency of the end-to-end computation of a fully packed RNS-CKKS ciphertext. The results show that not only SLOTHE is able to generate approximations with both lower error and better latency (or same) compared to previous works. For example, compared with BOLT for GeLU, T-50 reduces the error by a factor of 25.29× while achieving 2.65× speedup. Compared with Cheon et al. for Swish, T-50 achieves 42,138× lower error and 1.86× speedup at the same time. It is also noteworthy that SLOTHE offers diverse results according to the different UDC requirements, giving developers various options for designing FHE programs. Tbl. 7 shows the latency breakdown and the max. error of SLOTHE results for GeLU. Setting a higher latency upper-bound reduced the errors, thanks to SLOTHE utilizing the latency budget with active recursive FBAs. The increased latency is mostly due to BTS, which is usually caused due to the high cost of the comparisons for predicated executions.

Comparison Against NFGen. The results of SLOTHE demonstrate to be favorable over the piece-wise polynomials generated by NFGen. For example, in the GeLU case, T-50 achieves 14.07× speedup than NFGen while inducing an almost identical error. Some of them even outperform NFGen in terms of both performance and errors (*e.g.*, T-200 in Mish).

Comparison with LUT-based EA. Tbl. 6 shows the results of the GeLU approximations generated by SLOTHE compared to polynomial EA [10], and LUT EA [27, 28], and a mix of them [33]. We measure the amortized latency (*i.e.*, average latency per input), a common metric used by previous works [10, 11, 26, 29]. T-50 produced by SLOTHE exhibits a smaller error and a faster performance than the previous works.

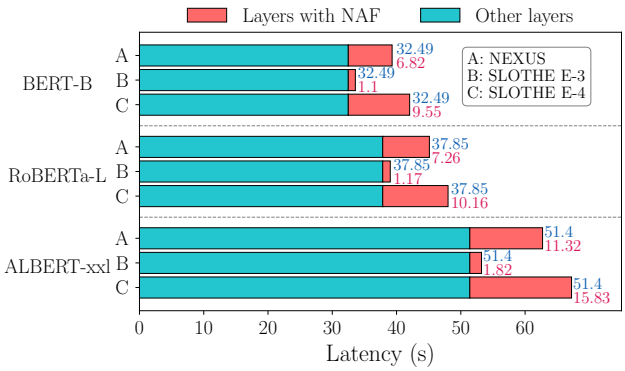


Fig. 5: The inference latency (in seconds) per input of BERT-Base, RoBERTa-L, and ALBERT-xxl with NEXUS and SLOTHE.

Tbl. 8: Inference Accuracy of SLOTHE and NEXUS. We used the pre-trained NNs from their open-sources [48, 49].

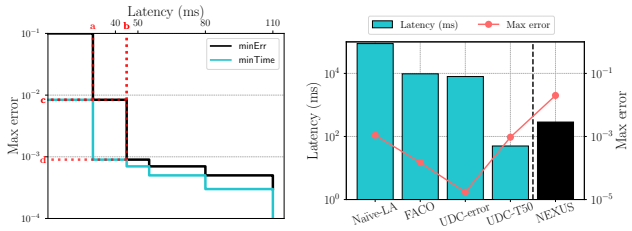
Data-set	NN	Unencrypted Acc. (%)	Inference over FHE		
			NEXUS	SLOTHE (E-3)	SLOTHE (E-4)
QNLI	ALBERT-xxl	95.30%	90.1% (-5.2)	94% (-1.3)	95.2% (-0.1)
	RoBERTa-L	94.70%	90% (-4.7)	93.7% (-1)	**94.7% (0)**
	BERT-base	89.4%	88.8% (-0.6)	89.3% (-0.1)	89.4% (0)
RTE	ALBERT-xxl	89.20%	83.4% (-5.8)	88% (-1.2)	88.9% (-0.3)
	RoBERTa-L	86.60%	79.7% (-6.9)	85.7% (-0.9)	**86.4% (-0.2)**
	BERT-base	70.0%	69.8% (-0.2)	70.0% (0)	70.0% (0)
RACE	ALBERT-xxl	86.50%	80.1% (-6.4)	85.2% (-1.3)	**86.4% (-0.1)**
	RoBERTa-L	83.20%	78.1% (-5.1)	82.3% (-0.9)	82.8% (-0.4)
	BERT-base	64.3%	63.1% (-1.2)	64.1% (-0.2)	64.2% (-0.1)

Compared to the polynomial EA [10], the maximum error and the average error are smaller by a factor of 21.1× and 3.7×, respectively, while achieving 5.8× speedup. Compared to the high-cost LUT evaluation over FHE with $q = 16$, SLOTHE achieves 2697× speedup with 161.3×/270× smaller maximum/average error. Increasing q in the LUT reduces its error, but the already high computational overhead increases even further. IRON [33] adopted a mixture of the two, computing certain building blocks with LUT and using them to compute polynomials. This reduced the amount of errors but since the LUT computation is still there with additional polynomial computation, the latency is still very high. Overall, the results demonstrate that the polynomials generated by SLOTHE is more favorable for both error and performance.

5.2 NN Evaluation over FHE

We now evaluate how the results of SLOTHE affect the latency and the inference accuracy. We use NEXUS [10] as our baseline, a state-of-the-art FHE-based transformer inference using manually crafted piece-wise polynomial approximations for the NAFs in the NNs. In order to assess the exact impacts, we only replace the NAF computation with the results of SLOTHE running in minTime mode and keep the others intact (*e.g.*, MatMul) from the baselines.

Latency. Fig. 5 and Tbl. 8 depicts the inference latency and accuracy of NEXUS and when applying SLOTHE to it. We



(a) Results of minErr and minTime. **(b)** Impacts on latency and max. err.

Fig. 6: Impact of our design choices on GeLU. In (b), the effects are cumulative built upon the previous one, from left to right.

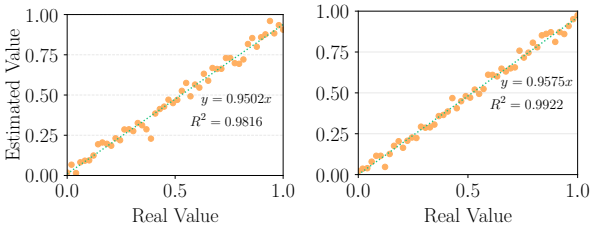
use E-3 and E-4 as the UDC requirements. The results were as expected. In average, E-3 had the effect of 6.2× speedup for the NAFs, which led to an overall speedup of 1.2× compared to NEXUS. When it comes to E-4, the latency was increased, with 6% overall additional overhead.

Accuracy. For BERT-base, NEXUS is enough to keep the accuracy drop less than 1.2%. SLOTHE keeps the accuracy loss under -0.2% for the same NN. RoBERTa-L and ALBERT-xxl serves the same inference tasks with much higher accuracy than BERT-base for all datasets, exhibiting more competitiveness. For the two NNs, the accuracy drop of NEXUS is severe, ranging between 4.7%–7.9%. On the other hand, SLOTHE w. E-3 only suffered from accuracy drops between 0.9%–1.3%. E-3 also achives better latency, which means that SLOTHE outperforms NEXUS for both accuracy and latency. SLOTHE w. E-4, which induces even less error per NAF, restored most of the accuracy drop of NEXUS with only a minimal additional latency of 6%. We also highlight that for several cases, SLOTHE E-4 did not suffer from any accuracy loss, thus producing the exact same outputs as the unencrypted state. The results demostrates that SLOTHE successfully avoids severe the accuracy losses while achieving competitive latency for arbitrary NNs including the highly fine-tuned NNs.

5.3 Impact of Design Choices

Fig. 6-(b) shows the impact of our design choices on top of `Naive-LA`, which refers to when we blindly perform LA by executing all the inline codes called by NAF unless an approximation is unavoidable. As expected, FHE-Aware code optimization, denoted as `FACO` resulted in 89% latency reduction. Note that 87% of the error was also reduced because reducing the number of conditions led to less number of comparisons as well as other non-arithmetics which would otherwise be approximated. We then let our UDC-Tracker first avoid FBAs of those that increase the error, denoted as `UDC-err`, from which we could reduce 18% of the latency and 88% of errors. If we set the latency threshold to 50 ms (UDC-T50), SLOTHE produces a result that satisfies the threshold. The error is evidently bigger than `UDC-err` since we reduced the latency dramatically. Nevertheless, the result demonstrates both better performance and smaller errors compared to Nexus.

minErr and min Time. Fig. 6-(a) shows how the two modes, minErr and minTime are related. The black and blue graphs refer to minErr (y=Err(T_x)) and minTime (x=Time($E_{(-log(y))}$)), respectively. Note that the graphs are stair-shaped because FBAs can only be executed when there is enough budget. The two graphs always meet at each increasing step because the FBA target selection order is always identical for all runs of the same NAF. The sub-function being selected for the second FBA will always be the one that has the sub-function of NAF, which has the smallest number of paths and function calls in its function body, no matter what constraints are given. As a result, the set of the outputs of minErr and minTime are identical without superiority of one over the other (T-k = E-j when $k \in [a,b]$ and $j \in [d,c]$), and their difference only lies in how fit would the result be to the UDC.



(a) FBA Target Selection **(b)** UDC-Tracker

Fig. 7: Normalized Geomean. of heuristic cost analysis of SLOTHE.

Tbl. 9: MPC/Hybrid GeLU evaluation. We run SLOTHE with cryptographic backends from Bumblebee. Dagger† is explained in §5.4.

2^{22} indices		Latency (s) LAN	Latency (s) WAN	Max. Err. ($\times 10^{-4}$)	Comm. (GBs)
IRON [33]		126	4118	6.1	93.3
BOLT [15]		14	744	9.7	17.3
Bumblebee [16]		22	321	13.0	3.4
SLOTHE	E-3	12	254	8.5	1.6
	E-4†	28	398	2.5	3.9

Heuristic Cost Estimation. We evaluate the precision of SLOTHE's heuristic latency estimation in FBA Target Selection and UDC-Tracker by repeatedly performing the analysis 50 times for the geometric mean of all FBAs and compare it with the geomean of the actual latency. The results in Fig. 7 demonstrate that the estimated latencies are linearly correlated with the actual ones, with correlations of 0.9816 and 0.9922. This suggests that the heuristic latency estimation of SLOTHE provides reliable grounds in performing FBA and controlling the laziness correctly.

5.4 Applying SLOTHE for MPC

As previously discussed in §2.4, approximation strategies are frequently able to be applied for both FHE and MPC/Hybrid techniques. Accordingly, we demonstrate how SLOTHE can be applied for MPC/Hybrid protocols by changing the cryptographic backends used for the computation. We adopt the cryptographic backends from Bumblebee (*e.g.*, Ferret OT) and compute the approximations SLOTHE generates. Since

the computing cryptographic backend changes, we modify some gadget configurations within SLOTHE. For example, instead of assessing the cost of comparison for paths with high-degree minimax polynomials, we use the cost of comparison based on Ferret OT [16]. The UDC are also adapted but only at a minimal scale, since the parameters for MPC and FHE are similar, where we only need to additionally consider the noise bit-width in the FHE parameter. We use two identical machines with Intel Xeon Gold CPU over two networks, LAN with 3Gbps bandwidth and 0.8ms RTT and WAN with 100Mbps and 80ms RTT, similar environments and assumptions used in the compared works [10, 15, 16]. We also adopt their MPC data precision ($f = 12, l = 37$) as UDC.

Tbl. 9 shows the results of end-to-end evaluation of computing 2^{22} indices of GeLU compared with three state-of-the-art works MPC/Hybrid works. Note that the errors are different with when computed with FHE since the parameters are different. The minimal error with $f = 12$ is 2.4×10^{-4} [10] and inherently cannot go under 10^{-5}. As a result, E-4 selects the approximation with minimal error from the candidates it can find. The results show that running the approximations generated with SLOTHE is also competitive when computed with MPC/Hybrid protocols even without the additional manual efforts. It is also noteworthy that similar with the evaluations over FHE, SLOTHE achieved the results without benefitting from the manual optimizations from previous works (*e.g.*, designing the polynomials to be symmetric to reduce the computational overhead). E-3 is faster than the compared works with lower error than BOLT and Bumblebee. E-4 exhibits similar latency with Bumblebee with 5.2× lower error.

6 Discussion

Enhancing PAG Capabilities. PAG has the role of generating single polynomial approximations using remez algorithm, goldshmidt polynomial, and minimax algorithm. Whereas we restricted the capabilities of PAG as above in order to highlight the strength of FBA and FHE-aware code optimizations, we note that it is possible to enhance PAG by implementing more numerical approximation options such as least square algorithm. We can also consider implementing PAG as a database, including the EA results of previous works as well as the results that SLOTHE created for NAFs that it met previously. When the same NAF is called later, SLOTHE can simply select the approximation from the database and apply when the same NAFs are called during the next runs.

SLOTHE on Shallow Codes. SLOTHE can be applied to arbitrary NAFs, but its effect is less efficient on codes with shallow depth and small number of sub-functions. For example, ReLU merely consists of a single comparison without additional depth. There is no extra FBA available, so the result is identical to what PAG generates for comparison. PAG uses minimax algorithm to generate results such as T-350 (See Fig. 4), which is identical to the state-of-the-art approximation from Lee et al. since it also generates polynomials based on the same algorithm, and this specific form is widely adopted by numerous works requiring accurate ReLU for deep neural networks [**?**, 5, 14, 64, 65]. However, we emphasize that there are much diverse types of NAFs used in other NNs, including the in-trend transformers (*e.g.*,GeLU, Tanh in our evaluation), and those NAF codes are much more complicated with significant depth, making EA inefficient, inducing high errors and sluggish latency. Also, restricting the developers to use only a certain type of activation is against our motivation of targeting arbitrary neural networks. For such reason, instead of delving into manually crafting retrofitted approximations for specific NAFs, SLOTHE is designed to offer programmability and automation for arbitrary NAFs.

SLOTHE with Other FHE Schemes. We envision SLOTHE could be extended to accomodate computation with other FHE schemes such as BGV [3], BFV [2], and TFHE [45] as they share similar traits. While the specific latencies may differ, the functionalities of BGV and BFV are similar to CKKS, supporting packing, rotations, and BTS. It is also feasible to extend SLOTHE with a multi-scheme purpose as many works including CHIMERA [66] suggests, by adopting the cryptographic scheme conversion. Although LUT evaluations are slower than SLOTHE, such extension will open doors to a new programmability domain.

7 Other Related Work

RLWE-based NN Inference. The scope of SLOTHE is the non-arithmetic layers focusing on their impacts on accuracy and latency, but we also present works targeting other orthogonal topics, such as such as matrix-vector multiplication. Lee et al. [11], HCNN [67], and multiple prior works [5, 10, 13] introduced specific packing strategies that leads to better matrix computation. Our work used the state-of-the-art work of NEXUS [10] as the baseline for the arithmetic layer designs. HECO [41], COYOTE [40], and CHET [68] introduced automated methods in packing for arbitrary operations, while ELASM [25], HECATE [30], and DaCapo [14] focused on scheduling certain FHE operations for performance enhancements. However, they only affect the performance and not the accuracy as they do not change the approximation strategy. Applying SLOTHE on top of their works will help achieving further speedup in computing the NNs with higher accuracy.

Post-Processing of NNs with Approximated NAFs. AESPA [59] and AutoFHE [60] introduced methods to adapt Convolutional NNs with re-training or fine-tuning. Their goal is to find the polynomials to replace ReLU that are a good fit with their target NNs from a very large set of possible EA results. Their methods enabled them to use polynomials with comparatively low degree, however, the required post-processing time was worth 20-100 hours. On the other hand, SLOTHE does not require such post-processing of NNs. SLOTHE uses approximations produced by LA, and can be

applied for arbitrary NAFs and NNs beyond ReLU and CNNs, even to transformers which are more sensitive to errors, as shown in our evaluation.

TFHE for NAF Computation. PEGASUS [42], HEIR [27], and CHLOE [69] introduced and optimized NAF computation using TFHE-based LUT evaluation. They all make use of a multi-scheme, using CKKS for arithmetics and TFHE for non-arithmetics in a code. Their methods are also based on eager approximation in that they replace non-arithmetic functions into LUT operations as soon as they appear in the code, which induces more errors and computations as shown in our evaluation. On the contrary, SLOTHE performs an in-depth code analysis, diving into the subsequent codes of the function calls in NAFs adaptively in order to apply approximations only when required. Its unique ability makes SLOTHE possible to achieve both higher performance and lower error even if it only uses CKKS.

Other Cryptographic Backends. while FHE suffers from severe computational overhead, MPC/Hybrid requires significant communication and the memory overhead can become a bottleneck [70]. NEXUS [10] reports MPC/Hybrid to be faster than FHE over LAN but the opposite for WAN. They also note that FHE computation can be economically more beneficial when run over cloud services. On top of the above, we also envision other uses cases where the client prefers a fire-and-forget manner without getting involved in the computation, where FHE is the sole option. Nevertheless, while SLOTHE is currently retrofitted for FHE performance and error reduction, we have demonstrated in §5.4 that SLOTHE can be adapted for MPC/Hybrid by modifying the code optimization with MPC awareness.

There have been massive amounts of works related to MLaaS using MPC, and we present some state-of-the-art works that we did not yet introduce. Cheetah [71] introduces optimized building blocks which are adopted or improved by succeeding works such as IRON and Bumblebee. Along with Cheetah, RHOMBUS and GALA [71–73] devised efficient methods for linear computation for Hybrid protocols. However, their works do not target reducing the errors for NAFs thus cannot improve NN accuracy. SLOTHE could exploit those works to improve NN inference latency while maintaining its accuracy benefits. Crypten [38] and SIGMA [35] involves a trusted third party to reduce the overhead, but their threat model is accordingly different with ours. MUSE [74] and SIMC [75] propose extensions to support security against a malicious party. The choice for NAF approximations are orthogonal to their extentions, SLOTHE could also be used to generate approximations for such protocols that tolerate malicious adversaries. However, we expect severe optimizations required as the building blocks used for malicious security are computationally heavy and they only demonstrate simple ReLU implementation, while supporting complex NAFs (such as those in transformers) could induce significant overhead to compute with protocols for malicious security (*e.g.*, Zero-Knowledge proof over HE).

8 Conclusion

We presented SLOTHE, a technique that automatically generates NAF approximations satisfying specific needs of the developers. Unlike prior works that adopt eager approximation, SLOTHE focuses on the fact that NAFs are codes that are composed of various sub-routines, which we can in turn consider as the target for approximation. We observe that delaying approximations by recursively analyzing the code involved in the function calls has the potential of reducing the error than EA. To address potential computational overheads from the increased number of paths in the code, we devised FHE-aware code optimizations to eliminate and merge paths that are redundant when computed over FHE. SLOTHE is able to produce NAF approximations that exhibits upto 42,138× lower error compared to state-of-the-art EA works. Applying the results for transformer inferences achieved up to both 1.18× speedup and 6% higher accuracy than a state-of-the-art work. SLOTHE sometimes even achieves zero loss in accuracy with 6% of additional latency.

9 Ethics Considerations

We honor and support the code of conduct for USENIX Security 2025. This section discusses several potential sources of ethics violations and reasons why our work does not correspond to them. We may not be able to address all existing ethic aspects, but have made our best effort to cover a broad range, as elaborated below.

About Harms to Humans and Plagiarism. Our work does not involve any human participants for surveys or experiments. For the psychological well-being of our research team, we ensured that no harmful data, such as hate speech or socially biased content, was involved in our work. As far as we know, our work is original until the submission date.

About Likelihood of Harm to Potential Stakeholders. The stakeholders of Privacy-Preserving NN inference services would be the user and the service provider, and SLOTHE do not cause harm to them. Rather, SLOTHE brings benefits to them. FHE-based NN inferences basically keep the privacy of the two parties from being disclosed to each other. The user's input data is hidden with FHE and the service provider's model parameters are hidden since computation is processed on the service provider's server. SLOTHE is a work that enhances the performance and accuracy of the service, thus rather is likely to increase the user's satisfaction as well as the service provider's income (if the number of users increase). In a systematic view, the only external input to the system running SLOTHE is the user's requirements (See Fig.1). These requirements are submitted in a form visible to SLOTHE. If the user-provided values are not in the format

they are supposed to be, SLOTHE can detect and terminate the launch. From the user's perspective, his private input data is encrypted on his local side, and in kept encrypted during the whole NN inference. The user is in fact the only one who can decrypt the result which also takes place locally. The secret key is always on the client local environment, never disclosed to other parties. On the server's perspective, all computation happens in his own environment, thus his model parameters are not disclosed to other parties. System attacks on the two environment could extract the key, but such attacks are orthogonal to our work and can happen not only in our scenario but for all sorts of computer related ones. On minor notes, our work does not exploit vulnerabilities, nor does it promote specific commercial products. Based on these grounds, we claim that there are no concrete stakeholders or potential victims affected by our work.

About Malicious Software. SLOTHE does not use any harmful software nor allow unautorized execution that can potentially damage the system of the developers. We highlight that at the bottom line, we use FHE, a well-known privacy-preserving technique for protecting the private data of both parties in MLaaS, where all computations are simply undertaken in the encrypted state. The process do not introduce any systematic vulnerabilities, just as previous FHE based NN inference works do [10–12,27,40]. Our work does not include any experiments about attacks on systems. SLOTHE is composed with only a set of methods with the goal of analyzing an NAF code and modify it into a FHE compatible state. It does not hold any potential ethics violations. Our evaluations do not include performing attacks or analyzing vulnerabilities, thus, no potential vulnerability or threat invoked on the deployed system by the results of our work.

Unethical Experiments. As discussed above, our work does not impose any risks to the system and we conducted all experiments in a controlled, closed environment. All programs we use are extensions of open-sources, including the math libraries, FHE tools, NN frameworks, NN architectures and the corresponding pre-trained codes [47–49], and datasets [19,58]. Note that we used pre-trained models in their original form as provided online. To the best of our knowledge, this is the most objective way to demonstrate how FHE-based NN inference offer good accuracy (less loss) since training a model for our own leaves the possibility of biased training using a model trained to fit our specific needs. We also cross-check that the public models are as they are in their original literature (we strictly follow the instructions in their repositories for fine-tuning and running the transformers) and works that already have used them, to validate (to the best of our efforts) the authenticity and righteousness. Our work do not rely on additional or re-training adapted to FHE, thus we do not need to modify the NN itself except for the replacement of NAFs into approximations. This also only happens during inference, and it is indeed not in our favor in terms of performance and accuracy.

10 Open Science

We acknowledge and fully support the new open science policy which we will follow respectfully. As a matter of fact, our work also benefitted from artifacts of previous works, and we are willing to spread the positive effect. We are certain that there are no licensing restrictions or any other kind of potential problems in the process of sharing our artifacts. Specifically, our evaluation section also used open-source for the underlying FHE library and open-sources of previous works, such as Liberate, NEXUS.

We plan to share artifacts to let future researchers fully understand and run the experiments[1][2]. The core artifact is the code that generates the approximation in IRB form starting from the NAF C code. We will provide with execution script along with running examples. For practicality and thoroughness of our artifact, we will provide shell codes that installs the software dependencies. From a given NAF C code, we will provide codes that first translate them into IRB, then use CF-Optimizer to show the optimized IRB, then run FBA to generate the SLOTHE results as IRBs. We will also provide a short guideline how to install and run the example codes.

Acknowledgments

This work was partly supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (RS-2023-00277326), the Institute of Information & Communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT) (No.RS-2023-00277060, No. 2021-0-00528, No.RS-2024-00438729, IITP-2023-RS-2023-00256081), the BK21 FOUR program of the Education and Research Program for Future ICT Pioneers, Seoul National University in 2025, and the Inter-University Semiconductor Research Center (ISRC).

References

[1] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. A full rns variant of approximate homomorphic encryption. In *Selected Areas in Cryptography–SAC 2018: 25th International Conference, Calgary, AB, Canada, August 15–17, 2018, Revised Selected Papers 25*, pages 347–368. Springer, 2019.

[2] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Paper 2012/144, 2012. https://eprint.iacr.org/2012/144.

[1] https://doi.org/10.5281/zenodo.15600623
[2] https://github.com/SNUSOR-PECT/SLOTHE

[3] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)*, 6(3):1–36, 2014.

[4] Rebecca HK Emanuel, Paul D Docherty, Helen Lunt, and Knut Möller. The effect of activation functions on accuracy, convergence speed, and misclassification confidence in cnn text classification: a comprehensive exploration. *The Journal of Supercomputing*, 80(1):292–312, 2024.

[5] Jae Hyung Ju et al. Neujeans: Private neural network inference with joint optimization of convolution and fhe bootstrapping. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 4361–4375, 2024.

[6] Ricardo Pachón and Lloyd N Trefethen. Barycentric-remez algorithms for best polynomial approximation in the chebfun system. *BIT Numerical Mathematics*, 49:721–741, 2009.

[7] Adrien Marie Legendre. *Nouvelles méthodes pour la détermination des orbites des comètes: avec un supplément contenant divers perfectionnemens de ces méthodes et leur application aux deux comètes de 1805*. Courcier, 1806.

[8] Robert E Goldschmidt. *Applications of division by convergence*. PhD thesis, Massachusetts Institute of Technology, 1964.

[9] Jaehee Jang et al. Privacy-preserving deep sequential model with matrix homomorphic encryption. In *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, pages 377–391, 2022.

[10] Jiawen Zhang et al. Secure transformer inference made non-interactive. *Cryptology ePrint Archive*, 2024.

[11] Eunsang Lee et al. Low-complexity deep convolutional neural networks on fully homomorphic encryption using multiplexed parallel convolutions. In *International Conference on Machine Learning*, pages 12403–12422. PMLR, 2022.

[12] Ran Gilad-Bachrach et al. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *International conference on machine learning*, pages 201–210. PMLR, 2016.

[13] Alon Brutzkus et al. Low latency privacy preserving inference. In *International Conference on Machine Learning*, pages 812–821. PMLR, 2019.

[14] Seonyoung Cheon et al. Dacapo: Automatic bootstrapping management for efficient fully homomorphic encryption. pages 4697–4714, 2023.

[15] Qi Pang, Jinhao Zhu, Helen Möllering, Wenting Zheng, and Thomas Schneider. Bolt: Privacy-preserving, accurate and efficient inference for transformers. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 4753–4771. IEEE, 2024.

[16] Wen-jie Lu et al. Bumblebee: Secure two-party inference framework for large transformers. *Cryptology ePrint Archive*, 2023.

[17] Kevin Nam et al. Lohen: Layer-wise optimizations for neural network inferences over encrypted data with high performance or accuracy. In *34th USENIX Security Symposium (USENIX Security 25)*, 2025.

[18] Yinhan Liu. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 364, 2019.

[19] Alex Wang. Glue: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461*, 2018.

[20] Jean-Philippe Bossuat et al. Security guidelines for implementing homomorphic encryption. *Cryptology ePrint Archive*, 2024.

[21] CrytptoLab. Official HEaaN Library, 2022. https://heaan.it/.

[22] Microsoft seal (release 3.7). https://github.com/Microsoft/SEAL, 2021.

[23] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. Bootstrapping for approximate homomorphic encryption. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 360–384. Springer, 2018.

[24] Eunsang Lee et al. Minimax approximation of sign function by composite polynomial for homomorphic comparison. *IEEE Transactions on Dependable and Secure Computing*, 2021.

[25] Yongwoo Lee et al. Elasm:error-latency-aware scale management for fully homomorphic encryption. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 4697–4714, 2023.

[26] Youngjin Bae et al. Meta-bts: Bootstrapping precision beyond the limit. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 223–234, 2022.

[27] Song Bian et al. Heir: A unified representation for cross-scheme compilation of fully homomorphic computation. *Cryptology ePrint Archive*, 2023.

[28] I. Chillotti et al. Programmable bootstrapping enables efficient homomorphic inference of deep neural networks. In *Cyber Security Cryptography and Machine Learning: 5th International Symposium*. Springer, 2021.

[29] Jung Hee Cheon, Wootae Kim, and Jai Hyun Park. Efficient homomorphic evaluation on large intervals. *IEEE Transactions on Information Forensics and Security*, 17:2553–2568, 2022.

[30] Yongwoo Lee et al. Hecate: Performance-aware scale optimization for homomorphic encryption compiler. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 193–204. IEEE, 2022.

[31] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. {GAZELLE}: A low latency framework for secure neural network inference. In *27th USENIX security symposium (USENIX security 18)*, pages 1651–1669, 2018.

[32] Deevashwer Rathee et al. Sirnn: A math library for secure rnn inference. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1003–1020. IEEE, 2021.

[33] Meng Hao et al. Iron: Private inference on transformers. *Advances in neural information processing systems*, 35:15718–15731, 2022.

[34] Xiaoyu Fan et al. Nfgen: Automatic non-linear function evaluation code generator for general-purpose mpc platforms. In *2022 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2022.

[35] Kanav Gupta et al. Sigma: Secure gpt inference with function secret sharing. *Cryptology ePrint Archive*, 2023.

[36] Qian Lou and Lei Jiang. She: A fast and accurate deep neural network for encrypted data. *Advances in Neural Information Processing Systems*, 32, 2019.

[37] Kevin Nam, Hyunyoung Oh, Hyungon Moon, and Yunheung Paek. Accelerating n-bit operations over tfhe on commodity cpu-fpga. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, pages 1–9, 2022.

[38] Brian Knott, Shobha Venkataraman, Awni Hannun, Shubho Sengupta, Mark Ibrahim, and Laurens van der Maaten. Crypten: Secure multi-party computation meets machine learning. *Advances in Neural Information Processing Systems*, 34:4961–4973, 2021.

[39] Junghyun Lee et al. Precise approximation of convolutional neural networks for homomorphically encrypted data. *IEEE Access*, 11:62062–62076, 2023.

[40] Raghav Malik, Kabir Sheth, and Milind Kulkarni. Coyote: A compiler for vectorizing encrypted arithmetic circuits. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 118–133, 2023.

[41] Alexander Viand, Patrick Jattke, Miro Haller, and Anwar Hithnawi. Heco: Automatic code optimizations for efficient fully homomorphic encryption. *arXiv preprint arXiv:2202.01649*, 2022.

[42] Wen-jie Lu et al. Pegasus: bridging polynomial and non-polynomial evaluations in homomorphic encryption. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1057–1073. IEEE, 2021.

[43] Zama. TFHE-rs: A Pure Rust Implementation of the TFHE Scheme for Boolean and Integer Arithmetics Over Encrypted Data, 2022. https://github.com/zama-ai/tfhe-rs.

[44] K. Matsuoka et al. Virtual secure platform: A five-stage pipeline processor over tfhe. In *USENIX Security Symposium*, 2021.

[45] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Tfhe: fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33(1):34–91, 2020.

[46] Joon-Woo Lee et al. High-precision bootstrapping of rns-ckks homomorphic encryption using optimal minimax polynomial approximation and inverse sine function. In *EUROCRYPT 2021, October 17–21, 2021*, pages 618–647. Springer, 2021.

[47] Jacob Devlin. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[48] Google Research. Albert: A lite bert for self-supervised learning of language representations, 2019. https://github.com/google-research/ALBERT.

[49] Google Research. Concrete ML: a privacy-preserving machine learning library using fully homomorphic encryption for data scientists, 2019. https://github.com/facebookresearch/fairseq.

[50] Zhenzhong Lan. Albert: A lite bert for self-supervised learning of language representations. *arXiv preprint arXiv:1909.11942*, 2019.

[51] Free Software Foundation. Gnu c math library, 2024. https://www.gnu.org/software/libc/.

[52] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[53] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[54] Lolremez. https://github.com/samhocevar/lolremez.

[55] Junghyun Lee et al. Precise approximation of convolutional neural networks for homomorphically encrypted data. *IEEE Access*, 2023.

[56] Adam Paszke et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[57] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[58] Guokun Lai et al. Race: Large-scale reading comprehension dataset from examinations. *arXiv preprint arXiv:1704.04683*, 2017.

[59] Jaiyoung Park et al. Aespa: Accuracy preserving low-degree polynomial activation for fast private inference. *arXiv preprint arXiv:2201.06699*, 2022.

[60] Wei Ao and Vishnu Naresh Boddeti. Autofhe: Automated adaption of cnns for efficient evaluation over fhe. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 2173–2190, 2024.

[61] Donghwan Kim et al. Hyphen: A hybrid packing method and optimizations for homomorphic encryption-based neural networks. *arXiv preprint arXiv:2302.02407*, 2023.

[62] DESILO. Liberate.FHE. https://github.com/Desilo/liberate-fhe.

[63] Rakesh Kumar et al. Enhanced channel-wise homomorphic encryption for image inference based on pairwise activation functions. In *2023 International Conference on Intelligent Systems, Advanced Computing and Communication (ISACC)*, pages 1–7. IEEE, 2023.

[64] Austin Ebel et al. Orion: A fully homomorphic encryption framework for deep learning. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 734–749, 2025.

[65] Kaustubh Shivdikar et al. Gme: Gpu-based microarchitectural extensions to accelerate homomorphic encryption. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 670–684, 2023.

[66] Christina Boura et al. Chimera: Combining ring-lwe-based fully homomorphic encryption schemes. *Journal of Mathematical Cryptology*, 14(1):316–338, 2020.

[67] Ahmad Al Badawi et al. Towards the alexnet moment for homomorphic encryption: Hcnn, the first homomorphic cnn on encrypted data with gpus. *IEEE Transactions on Emerging Topics in Computing*, 9(3):1330–1343, 2020.

[68] Roshan Dathathri et al. Chet: An optimizing compiler for fully-homomorphic neural-network inferencing. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 142–156, 2019.

[69] Song Bian et al. Chloe: Loop transformation over fully homomorphic encryption via multi-level vectorization and control-path reduction. *Cryptology ePrint Archive*, 2024.

[70] Karthik Garimella et al. Characterizing and optimizing end-to-end systems for private inference. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 89–104, 2023.

[71] Zhicong Huang et al. Cheetah: Lean and fast secure {Two-Party} deep neural network inference. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 809–826, 2022.

[72] Jiaxing He et al. Rhombus: Fast homomorphic matrix-vector multiplication for secure two-party inference. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 2490–2504, 2024.

[73] Qiao Zhang et al. Gala: Greedy computation for linear algebra in privacy-preserved neural networks. *arXiv preprint arXiv:2105.01827*, 2021.

[74] Ryan Lehmkuhl et al. Muse: Secure inference resilient to malicious clients. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2201–2218, 2021.

[75] Nishanth Chandran et al. {SIMC}:{ML} inference secure against malicious clients at {Semi-Honest} cost. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 1361–1378, 2022.

Tbl. 13: Comparison of GeLU approximations.

		Amortized Latency (us)	Speedup
SLOTHE (T-50)		1.52	-
Poly EA [10]		8.76	5.8×
LUT [27]	q=55	27790	18283×
	q=32	13283	8739×
	q=16	4100	2697×
Bool. Circuit [44]		1783293	1173219×

Tbl. 10: The security requirements for $\lambda = 128$.

N	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}
$\log(QP)$	214	430	868	1747	3523

Tbl. 11: Moduli chain used in our paper

Prime	Value	Prime	Value
Base	288230376147386369		288230376132182017
	4398044938241		288230376131788801
S2C	4398043496449		288230376129691649
	4398042972161	*ModRed*	288230376128643073
	1125899903827969		288230376126545921
	1125899902124033		288230376122613761
	1125899887312897		288230376121434113
	1125899886395393		288230376121040897
Mult	1125899885740033	*C2S*	288230376118026241
	1125899884167169		288230376115535873
	1125899884036097		1152921504606584833
	1125899883642881		1152921504598720513
	1125899883380737	*Temp*	1152921504597016577
ModRed	288230376138735617		1152921504595968001
	288230376135196673		1152921504592822273

Tbl. 12: Error bound of operations in RNS-CKKS.

Operation	bound
Encode	$1/\Delta$
Encryption	$6\sigma\sqrt{N} + \frac{32}{3}\sqrt{6\sigma N}$
Rescale	$\sqrt{3N} + \frac{8\sqrt{2}N}{3}$
Key switching (CMult, Rotation)	$\frac{8\sqrt{3}\cdot \mathtt{dnum}\cdot\sigma N}{3P} + \sqrt{3N} + \frac{8\sqrt{2}N}{3}$

A Security of RNS-CKKS and Our Setting

RNS-CKKS is constructed based on a lattice-based problem, specifically the Ring Learning With Error (RLWE) problem. The estimated concrete security of the RLWE problem is dependent on pre-defined parameters, such as N, $\log(QP)$, and standard deviation σ. The maximum allowable values of QP to meet the $\lambda = 128$ security level recommended in [**?**] for a standard deviation of $\sigma = 3.19$ and a given N are provided in Tbl. 10. This indicates that satisfying $\lambda = 128$ security for a specific N requires careful selection of the individual moduli in the moduli chain of QP, ensuring that $\log(QP)$ does not exceed the values listed in the table. These security requirements were calculated using the Lattice Estimator, a widely accepted tool for estimating the security of Learning with Errors (LWE) instances against known attacks on (R)LWE problems.

The outline of the FHE parameter setting used in our paper is proposed in Tbl. 1. Since our parameter $\log(QP) = 1630$ is less than the security requirements for $N = 2^{16}$, our parameter setting satisfies $\lambda = 128$. In detail, Tbl. 11 describes the exact full values of the moduli composed of QP. The primes $S2C, ModRed, C2S$ are consumed during the bootstrapping procedure, denoting slot-to-coeff, modular reduction, and coeff-to-slot respectively, and $Temp$ are temporal primes used during key switching routines. After the BTS, the ciphertext modulus becomes $Q = \Sigma_{i=0}^{12} q_i$.

B Inherent Errors of RNS-CKKS

It is known that several FHE operations introduce computational errors. The additional errors induced by RNS-CKKS operations are summarized in Tbl. 12, based on the error analysis from previous work, where P represents the auxiliary modulus, and `dnum` denotes the number of auxiliary moduli used for key switching. Other operations not listed in the table do not introduce *additional* errors and are therefore omitted. For example, the error resulting from the multiplication between two ciphertexts (CMult) is calculated by the product of the inherent errors of the two ciphertexts, along with the key switching error. Thus, the key switching error is the only additional error introduced by this operation. Notably, rescaling operations induce a relatively larger error compared to other operations, making it being considered as the actual error bound.

C LUT and Boolean Circuit Evaluation

Tbl. 13 shows the extended version of the latencies for LUT and boolean circuit evaluation compared to SLOTHE. As discussed in §2.3, boolean circuit evaluation induces unrealistic overhead, being over $10^6\times$ slower than SLOTHE).

D Resulting Polynomials

Tbl. 14 shows exemplary polynomials generated by SLOTHE for FHE. The polynomials include the high-degree sub-polynomials that are used for divisions and comparisons (note that those sub-polynomials are not visually presented in previous works although they also require them [10]).

Tbl. 14: SLOTHE result polynomials (Tanh, GeLU, Swish, Mish). Results include the computation of sub-polynomials for divisions and comparisons.

NAF	UDC	Result polynomial
Tanh	T-40	$f(x) = -4.2808 \cdot 10^{-6} x^{11} + 2.1731 \cdot 10^{-4} x^9 - 4.3071 \cdot 10^{-3} x^7 + 4.2847 \cdot 10^{-2} x^5 - 2.3741 \cdot 10^{-1} x^3 + 9.6647 \cdot 10^{-1} x$
	T-300	$f(x) = f_2(f_1(x))$ where $\begin{cases} f_1(x) = 8.376 \cdot 10^{-17} x^{16} + 1.3481 \cdot 10^{-15} x^{15} + 5.8105 \cdot 10^{-17} x^{14} - 9.6073 \cdot 10^{-16} x^{13} + 1.9515 \cdot 10^{-12} x^{12} \\ +2.3576 \cdot 10^{-11} x^{11} + 1.6018 \cdot 10^{-10} x^{10} + 1.5947 \cdot 10^{-9} x^9 + 1.7111 \cdot 10^{-8} x^8 + 1.3706 \cdot 10^{-7} x^7 + 9.18 \cdot 10^{-7} x^6 \\ +5.5059 \cdot 10^{-6} x^5 + 2.7833 \cdot 10^{-5} x^4 + 1.1134 \cdot 10^{-4} x^3 + 3.3319 \cdot 10^{-4} x^2 + 6.6635 \cdot 10^{-4} x^1 + 1.3334 \cdot 10^{-3} \\ f_2(x) = 1 - \frac{2-x}{750} \cdot \prod_{i=1}^{13} \left(1 + (1-x)^{2^i}\right) \end{cases}$
	T-900	$f(x) = f_2(f_1(x))$ where $\begin{cases} f_1(x) = \frac{1}{1500}(9.5436 \cdot 10^{-11} x^7 + 5.3457 \cdot 10^{-9} x^6 + 2.5430 \cdot 10^{-7} x^5 + 1.0172 \cdot 10^{-5} x^4 + 3.2552 \cdot 10^{-4} x^3 \\ +7.8124 \cdot 10^{-3} x^2 + 1.25 \cdot 10^{-1} x + 1)^{16} + \frac{1}{1500} \\ f_2(x) = 1 - \frac{2-x}{750} \cdot \prod_{i=1}^{13} \left(1 + (1-x)^{2^i}\right) \end{cases}$
GeLU	T-40	$f(x) = 7.6262 \cdot 10^{-14} x^{18} - 2.3522 \cdot 10^{-11} x^{16} + 3.0770 \cdot 10^{-9} x^{14} - 2.2266 \cdot 10^{-7} x^{12} + 9.7566 \cdot 10^{-6} x^{10} - 2.6703 \cdot 10^{-4} x^8 + 4.5696 \cdot 10^{-3} x^6$ $-4.8581 \cdot 10^{-2} x^4 + 3.7451 \cdot 10^{-1} x^2 + 0.5x + 5.6080 \cdot 10^{-3}$
	T-50	$f(x) = 4.9915 \cdot 10^{-16} x^{26} - 1.0699 \cdot 10^{-13} x^{24} + 1.0222 \cdot 10^{-11} x^{22} - 5.7474 \cdot 10^{-10} x^{20} + 2.1150 \cdot 10^{-8} x^{18} - 5.3645 \cdot 10^{-7} x^{16} + 9.6301 \cdot 10^{-6} x^{14}$ $-1.2394 \cdot 10^{-4} x^{12} + 1.1491 \cdot 10^{-3} x^{10} - 7.6852 \cdot 10^{-3} x^8 + 3.7127 \cdot 10^{-2} x^6 - 1.3096 \cdot 10^{-1} x^4 + 3.9849 \cdot 10^{-1} x^2 + 0.5x$
	T-700	$f(x) = \frac{x}{2} \times (2 - f_3(x)) \times f_4(-0.5626 + R(f_1(x))) \times f_2(S(f_1(x)))))$ where $\begin{cases} f_1(x) = \frac{1}{32}(2 - \frac{x^2}{32}) \cdot \prod_{i=1}^{12} \left(1 + (1 - \frac{x^2}{32})^{2^i}\right) \\ f_2(x) = (2-x) \cdot \prod_{i=1}^{16} \left(1 + (1-x)^{2^i}\right) \\ f_3(x) = -4.6382 \cdot 10^{-9} x^{15} + 3.2090 \cdot 10^{-7} x^{14} - 1.0172 \cdot 10^{-5} x^{13} + 1.9581 \cdot 10^{-4} x^{12} - 2.5582 \cdot 10^{-3} x^{11} \\ +2.4002 \cdot 10^{-2} x^{10} - 1.6698 \cdot 10^{-1} x^9 + 8.7644 \cdot 10^{-1} x^8 - 3.4977 x^7 + 10.6105 x^6 - 24.2861 x^5 + 41.3428 x^4 \\ -51.2730 x^3 + 45.157 x^2 - 27.0525 x + 9.126 \\ f_4(x) = 7.5565 \cdot 10^{-5} x^7 + 1.0584 \cdot 10^{-3} x^6 + 7.8272 \cdot 10^{-3} x^5 + 4.1221 \cdot 10^{-2} x^4 + 1.6645 \cdot 10^{-1} x^3 \\ +4.9995 \cdot 10^{-1} x^2 + x + 1 \\ R(x) = -7.4877 \cdot 10^{-5} x^7 - 0.0006 x^6 - 0.0014 x^5 - 0.0012 x^4 - 0.0005 x^3 - 8.0556 \cdot 10^{-5} x^2 \\ -5.2937 \cdot 10^{-6} x - 7.5263 \cdot 10^{-8} \\ S(x) = -4.6100 \cdot 10^{-7} x^8 + 5.013 \cdot 10^{-5} x^7 + 0.0008 x^6 + 0.0033 x^5 + 0.0049 x^4 + 0.0033 x^3 \\ +0.0011 x^2 + 0.0001 x + 7.6294 \cdot 10^{-6} \end{cases}$
Swish	T-40	$f(x) = 1.3694 \cdot 10^{-14} x^{18} - 4.3137 \cdot 10^{-12} x^{16} + 5.8067 \cdot 10^{-10} x^{14} - 4.3763 \cdot 10^{-8} x^{12} + 2.0387 \cdot 10^{-6} x^{10} - 6.1562 \cdot 10^{-5} x^8 + 1.2482 \cdot 10^{-3} x^6$ $-1.8094 \cdot 10^{-2} x^4 + 2.4624 \cdot 10^{-1} x^2 + 0.5x + 8.7448 \cdot 10^{-4}$
	T-50	$f(x) = -2.1803 \cdot 10^{-16} x^{24} + 3.6750 \cdot 10^{-14} x^{22} - 2.7852 \cdot 10^{-12} x^{20} + 1.2597 \cdot 10^{-10} x^{18} - 3.8169 \cdot 10^{-9} x^{16} + 8.2890 \cdot 10^{-8} x^{14} - 1.3608 \cdot 10^{-6} x^{12}$ $+1.7864 \cdot 10^{-5} x^{10} - 2.0043 \cdot 10^{-4} x^8 + 2.0642 \cdot 10^{-3} x^6 - 2.0815 \cdot 10^{-2} x^4 + 2.4999 \cdot 10^{-1} x^2 + 0.5x + 4.7323 \cdot 10^{-7}$
	T-600	$f(x) = x \cdot f_2(f_1(x))$ where $\begin{cases} f_1(x) = -3.1542 \cdot 10^{-17} x^{17} + 5.3733 \cdot 10^{-16} x^{16} - 5.4397 \cdot 10^{-15} x^{15} + 8.1492 \cdot 10^{-14} x^{14} - 1.2705 \cdot 10^{-12} x^{13} \\ +1.6521 \cdot 10^{-11} x^{12} - 1.9544 \cdot 10^{-10} x^{11} + 2.1498 \cdot 10^{-9} x^{10} - 2.1532 \cdot 10^{-8} x^9 + 1.9379 \cdot 10^{-7} x^8 \\ -1.5501 \cdot 10^{-6} x^7 + 1.0851 \cdot 10^{-5} x^6 - 6.5104 \cdot 10^{-5} x^5 + 3.2552 \cdot 10^{-4} x^4 - 1.3021 \cdot 10^{-3} x^3 \\ +3.9063 \cdot 10^{-3} x^2 - 0.0078 x + 0.0156 \\ f_2(x) = \frac{2-x}{27} \cdot \prod_{i=1}^{10} \left(1 + (1-x)^{2^i}\right) \end{cases}$
Mish	T-40	$f(x) = -2.4036 \cdot 10^{-8} x^{11} + 7.9048 \cdot 10^{-7} x^{10} + 2.2340 \cdot 10^{-6} x^9 - 5.9687 \cdot 10^{-5} x^8 - 8.4269 \cdot 10^{-5} x^7 + 1.7599 \cdot 10^{-3} x^6 + 1.6678 \cdot 10^{-3} x^5 - 2.6830 \cdot 10^{-2} x^4$ $-1.8459 \cdot 10^{-2} x^3 + 2.9397 \cdot 10^{-1} x^2 + 6.0117 \cdot 10^{-1} x + 6.0906 \cdot 10^{-3}$
	T-50	$f(x) = 1.7504 \cdot 10^{-11} x^{15} + 3.2955 \cdot 10^{-9} x^{14} - 1.1733 \cdot 10^{-9} x^{13} - 3.3125 \cdot 10^{-7} x^{12} + 1.3009 \cdot 10^{-10} x^{11} + 1.3792 \cdot 10^{-5} x^{10} + 2.1764 \cdot 10^{-6} x^9 - 3.1113 \cdot 10^{-4} x^8$ $-8.7474 \cdot 10^{-5} x^7 + 4.2364 \cdot 10^{-3} x^6 + 1.6944 \cdot 10^{-3} x^5 - 3.8304 \cdot 10^{-2} x^4 - 1.8468 \cdot 10^{-2} x^3 + 3.1345 \cdot 10^{-1} x^2 + 6.0099 \cdot 10^{-1} x + 9.4745 \cdot 10^{-4}$
	T-200	$f(x) = x \cdot f_2(f_1(x))$ where $\begin{cases} f_1(x) = 1.9147 \cdot 10^{-13} x^{18} - 2.5610 \cdot 10^{-11} x^{16} + 1.5119 \cdot 10^{-9} x^{14} - 5.2657 \cdot 10^{-8} x^{12} + 1.2382 \cdot 10^{-6} x^{10} \\ -2.1941 \cdot 10^{-5} x^8 + 3.3372 \cdot 10^{-4} x^6 - 5.1863 \cdot 10^{-3} x^4 + 1.2499 \cdot 10^{-1} x^2 + 0.5x + 6.9315 \cdot 10^{-1} \\ f_2(x) = 6.8255 \cdot 10^{-6} x^{11} - 1.9724 \cdot 10^{-4} x^{10} + 2.4565 \cdot 10^{-3} x^9 - 1.7119 \cdot 10^{-2} x^8 + 7.1956 \cdot 10^{-2} x^7 \\ -1.7908 \cdot 10^{-1} x^6 + 2.1682 \cdot 10^{-1} x^5 + 4.1711 \cdot 10^{-2} x^4 - 3.9615 \cdot 10^{-1} x^3 + 2.4237 \cdot 10^{-2} x^2 \\ +9.9689 \cdot 10^{-1} x + 6.3127 \cdot 10^{-5} \end{cases}$