FHEDA: Efficient Circuit Synthesis with Reduced Bootstrapping for Torus FHE

Animesh Singh[1], Smita Das[1], Anirban Chakraborty[1], Rajat Sadhukhan[1], Ayantika Chatterjee[1], and Debdeep Mukhopadhyay[1]

[1]IIT Kharagpur

September 3, 2023

Abstract

Fully Homomorphic Encryption (FHE) is a widely used cryptography primitive for performing arbitrary computations on encrypted data. However, FHE incorporates a computationally intensive mechanism known as "bootstrapping", that resets the noise in the ciphertext to a lower level allowing the computation on circuits of arbitrary depth. This process can take significant time, ranging from several minutes to hours. To address the above issue, in this work, we propose an Electronic Design Automation (EDA) framework FHEDA that generates efficient Boolean representations of circuits compatible with the Torus-FHE (ASIACRYPT 2020) scheme. To the best of our knowledge, this is the first work in the EDA domain of FHE. We integrate logic synthesis tricks and gate optimization techniques into our FHEDA framework for reducing the total number of bootstrapping operations in a Boolean circuit, which leads to a significant (up to 50%) reduction in homomorphic computation time. Our FHEDA is built upon the observation that in Torus-FHE at most one Boolean gate over fresh encryptions does not require bootstrapping. By integrating this observation with logic replacement techniques into FHEDA, we could reduce the total number of bootstrapping operations along with the circuit depth. This eventually reduces the homomorphic evaluation time of Boolean circuits. In order to verify the efficacy of our approach, we assess the performance of the proposed EDA flow on a diverse set of representative benchmarks including privacy-preserving machine learning and different symmetric key block ciphers.

1 Introduction

In the era of increasing digitization and data-driven technologies, ensuring the security and privacy of sensitive information has become a paramount concern. One of the privacy-preserving technologies that made rapid inroads in recent years is Fully Homomorphic Encryption (FHE) that allows arbitrary computations on encrypted data. This enables users to offload their private and sensitive data on "honest-but-curious" cloud servers enabling secure computations on sensitive data while preserving confidentiality. FHE holds immense promise for various applications, such as cloud computing, machine learning, and data outsourcing, where privacy concerns have hindered the adoption of conventional data processing approaches. In particular, Machine Learning (ML) based application has seen rapid advances in recent years, driven by availability of data and sophisticated algorithms. Most ML models that are developed to solve real-world problems involve complex structures and require high computational power to work on large datasets. Moreover, in multiple real-world scenarios, ML models are developed and trained by one party (or entity) and made available to the public as ML-as-a-service (MLaaS) in pay-per-use model. While in the first case, an user makes use of the computational power of the cloud by outsourcing both model and data (in encrypted form) to the server[1], in the second case, it aims to avail services provided by the cloud keeping its own data private.

Fully Homomorphic Encryption [1, 2, 3, 4] has enabled the power of computing on encrypted data by allowing servers to compute directly on ciphertext. In other words, it allows computations on a collection of ciphertexts for some plaintext $m_1, \dots m_\ell$ into a ciphertext which is an encryption of the output of a function/circuit evaluated on the input plaintexts, i.e., $f(m_1, \dots m_\ell)$, without the knowledge of secret key. The security of modern FHE schemes relies on hard problems such as Learning With Error (LWE) or its ring variant (RLWE). The intractability of these schemes depend upon the idea of *noise* (alternatively called *error*), a small value that is added to the ciphertexts during encryption, which grows when homomorphic operations are performed on these ciphertexts. Due to the accumulation of noise, a refreshing operation, called *bootstrapping* is performed in order to ensure the noise contained in the homomorphically computed ciphertext does not exceed the required threshold.

While the idea of Homomorphic Encryption (HE) has been around in the cryptography community since 1970s, the significant impetus to the field of FHE came after the breakthrough of Gentry's work [5] in 2009, where he proposed Lattice-based scheme that allow any computations, albeit computationally intensive but can perform arbitrary depth circuit using the technique called "bootstrapping". Owing to this seminal work and the introduction of bootstrapping, multiple practical FHE schemes were proposed such as BF/V [6], BGV [2], CKKS [4], etc. These are considered as second-generation FHE schemes, that could perform arbitrary homomorphic computations. The bootstrapping operation and the memory overhead for the bootstrapping key (encryption of the original secret key) effectively restricted the wide adoption. Finally, the third generation schemes, namely FHEW [7] and TFHE [3], allow arbitrary computations with unrestricted depth of circuits. These schemes have revisited the concept of bootstrapping, making it relatively cheaper and faster. But after every gate a bootstrapping operation is performed. Therefore, the number of gates as well as the depth of the circuit contributes to the overall latency and throughput of FHE-based applications. Some additional works related to making FHE efficient in accelerated frameworks is presented in Appendix .2.

1.1 Motivation and Contribution

While FHE resolves the problem of computing homomorphic operations on encrypted data, it comes with a computationally expensive operation, called bootstrapping. Amongst other widely used FHE schemes like BGV/BFV, TFHE provides one of the most efficient gate wise bootstrapping operation, which operates after homomorphic evaluation of every binary gate. Because, TFHE operates on binary message space and provides all basic logic gates

[1]We will use the terms cloud and server interchangeably in this paper.

evaluation in the encrypted domain, it becomes a suitable choice for most of the FHE applications. However, bootstrapping is still the costliest computation, taking up almost 99% of the homomorphic gate computation in the TFHE scheme. Moreover, to implement a full-fledged application one needs to create FHE-amenable circuits using TFHE gates. Although manually designing such circuits is trivial for small applications, it becomes tedious for larger circuits to perform complex tasks, such as performing ML-based operations. While attempts have been made to provide synthesis frameworks for other privacy-preserving schemes like Multi-Party Computation (MPC) [8], the literature currently lacks an automated framework for synthesizing FHE-amenable circuits that can be utilized to perform homomorphic operations on the cloud. In order to take advantage of the computational capabilities of FHE libraries and integrate it with real-life applications, it is imperative to have an end-to-end framework for synthesizing FHE-amenable circuits. Additionally, such a framework must optimize the homomorphic gates used in the circuitry in order to minimize the huge overhead of bootstrapping. In this work, we propose the first hardware synthesis framework for the automated generation of FHE-friendly circuits for the TFHE scheme, wherein high-level function descriptions are automatically compiled to efficient and optimized circuit representation using logic synthesis and commercially available CAD tools. In particular, we make the following contributions:

- We propose FHEDA, an automated framework for synthesizing depth-optimized Boolean circuits with reduced bootstrapping and time-efficient implementation for TFHE scheme. For this, we augment industry-grade logic synthesis tools, to automatically and efficiently compile a function written in a Hardware Description Language into efficient depth-optimized representation.
- We incorporate three novel optimisations in our framework to reduce number of gate and bootstraping operations. First, we show that using multiple fan-in gate with reduced bootstrapping instead of 2-input can optimize the total number of bootstrapping required as well as reduce the depth of the circuit. Secondly, we perform NOT gate-based optimization which reduces the overall computation time for the circuit. Thirdly, we define custom standard cell library sets to automatically synthesize the most efficient circuit.
- We evaluated our proposed FHEDA flow on a set of representative benchmarks consisting of privacy-preserving neural networks, popular symmetric key ciphers, and private set intersection. We could exhibit a percentage reduction in circuit and homomorphic evaluation time, within the range of 30% to 40% and 13% to 50%, respectively. Finally, we describe an end-to-end implementation for the oblivious inference of a convolutional neural network (CNN) on encrypted data using our FHEDA flow.

1.2 Organization

The paper is organized as follows: Section 2 provides an overview of the necessary background, notations, and preliminary concepts essential for understanding the paper. In Section 3, various optimization techniques for representing Boolean circuits in an FHE-friendly manner are elaborated upon. Section 4 presents empirical and theoretical evidence that demonstrates the efficiency of FHE-friendly circuit evaluation by employing multi-input gates. The proposed FHEDA flow is described in Section 5, outlining the steps involved in the flow.

Experimental details of evaluating the tool on representative benchmarks and results are presented in Section 6. Section 7 presents an end-to-end evaluation a Convolutional neural network using FHEDA. Finally, the work is concluded in Section 8.

2 Preliminaries and Background

2.1 Notations and Mathematical Background

We use $\mathbb{T}$ to denote the Torus (i.e., the set of all real numbers modulo 1). We write $x \leftarrow \chi$ to represent that an element x is sampled uniformly at random from a set/distribution $\mathcal{X}$. For $a, b \in \mathbb{Z}$ such that $a, b \geq 0$, we denote by $[a]$ and $[a, b]$ the set of integers lying between 1 and a (both inclusive), and the set of integers lying between a and b (both inclusive) respectively. We denote $\langle \mathbf{x}, \mathbf{y} \rangle$ to represent a vector dot product between the vectors $\mathbf{x}$ and $\mathbf{y}$. We refer to $\lambda \in \mathbb{N}$ as the security parameter, and denote by $poly(\lambda)$ and $negl(\lambda)$ any generic (unspecified) polynomial function and negligible function in λ, respectively.[2]

2.2 Fully Homomorphic Encryption

Homomorphic Encryption enables arbitrary computation to be performed on encrypted data such that the results obtained using homomorphic computation post decryption is same as the equivalent operation in plaintext domain. For example, in an additive homomorphic encryption scheme, $\mathsf{Dec}(\mathsf{Enc}(x + y)) = \mathsf{Dec}(\mathsf{Enc}(x)) + \mathsf{Dec}(\mathsf{Enc}(y))$. Fully Homomorphic Encryption (FHE) extends this property to support both addition and multiplication over encrypted data. Modern FHE schemes rely on hardness assumptions like Learning with Errors (LWE) or Ring Learning with Errors (RLWE) to introduce carefully calibrated noise during encryption. However, this noise grows during computation, potentially leading to incorrect decryption once it surpasses a certain threshold. To address this issue, FHE schemes employ ciphertext maintenance operations that reduce the noise growth without altering the result.

2.3 Torus FHE (TFHE)

In this work, we focus on one of the well-known and widely used FHE scheme named Torus-FHE [3] (TFHE). The overall working principle of the scheme can be broadly classified into three stages - *encryption* of plaintext messages by the client using a secret key, *homomorphic gate evaluation and bootstrapping* by the server on the encrypted messages and finally *decryption* of the computed ciphertext by the client. We briefly explain each step, provide a mathematical formulation of the operations and discuss on Boolean representations of the FHE circuits.

[2]Note that a function $f : \mathbb{N} \to \mathbb{N}$ is said to be negligible in λ if for every positive polynomial p, $f(\lambda) < 1/p(\lambda)$ when λ is sufficiently large.

2.3.1 Encryption Stage

In the symmetric key setting, the encryption process starts with sampling a noise value $e \leftarrow \mathcal{G}_\alpha$ from a Gaussian distribution $\mathcal{G}$ with standard deviation α and adding it to the encoded message x to obtain an intermediate value of $b' = x + e$. It then samples a random vector $\mathbf{a} \in \mathbb{Z}_q^n$ and performs a vector dot product with the secret key $\mathbf{sk} \in \mathbb{B}^n$. Here, n be the secret key dimension, q be the modulus and $\mathbb{B} \in \{0, 1\}$. The result of this dot product is then added to the intermediate value b' to obtain its final value as $b = \langle \mathbf{a}, \mathbf{sk} \rangle + x + e$. The final ciphertext comes out to be $(\mathbf{a}, b)$[3]. Below we formally describe the encryption algorithm in TFHE.

$\mathsf{TFHE.Enc}(m, \mathbf{sk})$: This function works similarly to standard LWE encryption. It takes a message $m \in \{0, 1\}$ and a TFHE secret key $\mathbf{sk}$ as input. Samples a random vector $\mathbf{a} \in \mathbb{Z}_q^n$ and a noise value $e \leftarrow \mathcal{G}_\alpha$ and computes $b = \langle \mathbf{a}, \mathbf{sk} \rangle + \Delta(m) + e$, where $\Delta(m)$ be the encoding of the plaintext m. It finally returns a ciphertexts $\mathbf{ct} = (\mathbf{a}, b)$.

2.3.2 Evaluation and Bootstrapping

Assume, the server receives two ciphertexts $\mathbf{ct}_1 = (\mathbf{a}_1, b_1)$ and $\mathbf{ct}_2 = (\mathbf{a}_2, b_2)$ on which it performs homomorphic Boolean operations. The server first defines gate constant as a tuple $(\mathbf{0}, b_{gc}^T)$ where b_{gc}^T is uniquely defined for each of the 10 available homomorphic gates in the TFHE library. For each individual gate evaluation, the resultant ciphertext $\mathbf{ct} = (\mathbf{a}, b)$ is computed as $\mathbf{a} = \mathbf{0} \pm c \cdot (\mathbf{a_1} \pm \mathbf{a_2})$ and $b = b_{gc}^T \pm c \cdot (b_1 \pm b_2)$ under modulus $q = 2^{32}$, where the ordering of $+$ or $-$ and the constant c depend on the type of homomorphic gate being evaluated. Once the gate evaluation is completed, a *bootstrapping* operation takes place to refresh the accumulated noise in the resultant ciphertext. During bootstrapping, the bootstrapping key $\mathbf{bk}$ (a part of the evaluation key $\mathbf{ek}$), which is essentially encryption of the secret key $\mathbf{sk}$ under different key $\mathbf{sk'}$, modifies the evaluated ciphertext with lower noise and encrypted under different key $\mathbf{sk'}$. Therefore, a key-switching procedure is followed in order to bring the evaluated ciphertext back to a encryption under $\mathbf{sk}$ while keeping the noise unaltered[4].

$\mathsf{TFHE.Eval}(\{\mathbf{ct}_l\}_{l \in \ell}, \mathcal{C}, \mathbf{ek})$: It takes a set of TFHE ciphertexts $\{\mathbf{ct}_1, \ldots, \mathbf{ct}_\ell\}$, the evaluation key $\mathbf{ek}$ and a circuit $\mathcal{C} : \{0, 1\}^\ell \to \{0, 1\}^{\ell'}$. Returns an evaluated TFHE ciphertext $\overline{\mathbf{ct}}$ which is an encryption of $\mathcal{C}(m_1, \ldots, m_\ell)$ under the secret key $\mathbf{sk}$, where each m_l is the underlying plaintext of $\mathbf{ct}_l$ for all $l \in \ell$.

2.3.3 Decryption Stage

Once the client receives the evaluated ciphertext $\overline{\mathbf{ct}} = (\mathbf{a}, b)$, a result of some homomorphic computation, it then performs the decryption process by computing a phase $\phi = b - \langle \mathbf{a}, \mathbf{sk} \rangle$. As a result of this computation, the client receives a noisy version $x + e$ of the underlying encoded plaintext message $x = \Delta(m)$, where m being the evaluated plaintext. The sign of this phase is checked to determine the output, i.e., return 1 if $\phi > 0$ and 0 if $\phi \leq 0$.

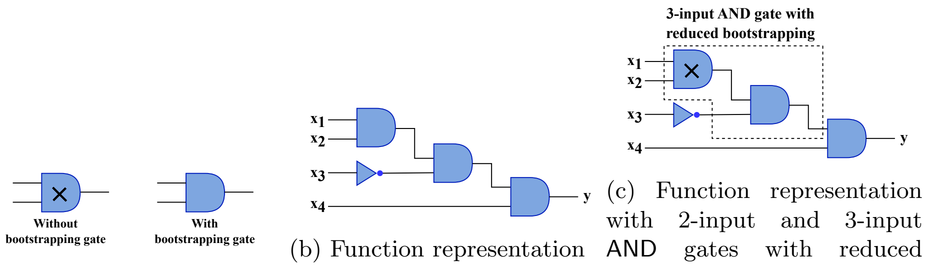
[3]Although TFHE scheme supports both private and public key encryption schemes, we consider private key for this work as the available TFHE library [9] can be directly used for private key without much modification. However, we note that our framework attempts to optimise the homomorphic circuit synthesis for TFHE and thus is not dependent on the nature of the scheme (private or public).

[4]For details of the *bootstrapping* procedure, refer to the original work of TFHE [3]

TFHE.Dec($\overline{\mathbf{ct}}, \mathbf{sk}$): Given an evaluated TFHE ciphertext $\overline{\mathbf{ct}} = (\mathbf{a}, b) \in \mathbb{T}^{n+1}$ and the secret key $\mathbf{sk}$, returns a message bit $m \in \{0, 1\}$ which minimizes $|b - \langle \mathbf{a}, \mathbf{sk} \rangle - \frac{1}{4}m|$.

2.4 HDL Synthesis using Bristol Format

In this section, we present an essential step in the design flow of digital hardware circuits i.e., Hardware Description Language (HDL) synthesis that involves the transformation of a high-level description of hardware functionality into gate-level representations amenable for hardware implementation. Using an HDL function description as input, a logic synthesis tool analyses it and generates an appropriate result for various hardware platforms such as FPGAs and ASICs. In our framework, we perform the logic synthesis using the industry-leading software, Cadence Genus Synthesis Solution[5]. The design flow begins with the Register Transfer Level (RTL) design behavioral description in Verilog HDL followed by the logic synthesis of the circuit in Cadence Design Framework to generate gate-level *synthesized* netlists, such as Bristol Formats.


Without bootstrapping gate
With bootstrapping gate
x1
x2
x3
x4
y
3-input AND gate with reduced bootstrapping
x1
x2
x3
x4
y

(a) Gate representation with and without Bootstrapping

(b) Function representation with 2-input AND gates (Evaluation time: 0.17 secs)

(c) Function representation with 2-input and 3-input AND gates with reduced bootstrapping (Evaluation time: 0.12 secs)

Figure 1: Notation of Bootstrapping and Comparison between 2-input and 3-input AND gates with reduced bootstrapping with a toy example

Now, we provide a brief description of the structure of Bristol Formats. A Bristol Format file starts with a line that defines the number of Boolean gates and the number of wires in the Boolean circuit. The second line symbolizes the first two numbers as the number of input wires to the function with its corresponding number of output wires. From the third line onwards, each line represents a binary gate and its input/output specifications. The first two values represent the number of input and output respectively. The next two values are input wire numbers, which are fed as the inputs of the binary gate; after evaluation, the result is written in the next wire number, which is the output wire. For example, consider the Half-Adder circuit presented in Listing 1. In line-1, '2' and '4' denote the number of gates and the wires respectively. In line-2, the first two values i.e., '2' and '2' refers to the size of the two inputs, and the third value '2' is the size of the output. The last entry in the line-3 represents XOR operation taking the third and the fourth values i.e., '1' and '2' as input wires, and the result after homomorphic XOR operation will be written in the output wire number '3'. The first two values '2' and '1' denotes the number of inputs and the number of outputs. The Bristol Formats are known to be TFHE-friendly as it is represented using only binary gates. In the following sections, we present a detailed discussion on constructing

[5]https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/synthesis/genus-synthesis-solution.html

optimal Boolean representations of circuits, which will be used for homomorphic evaluation using the TFHE scheme.

Listing 1: Bristol Format of Half Adder

```
2 4
2 2 2
2 1 1 2 3 XOR
2 1 1 2 4 AND
```

3 TFHE Friendly Circuit Representations

Despite recent advancements in homomorphic encryption schemes and optimization techniques, the performance overhead remains a significant challenge in practical implementations of FHE systems. This is due to the large overhead of the *bootstrapping* time during homomorphic computation. Ample FHE schemes has been proposed in the last decade; amongst them BGV [2], BFV [1], TFHE [3], CKKS [4] are widely used FHE schemes. Here, BGV and BFV can directly operate on integer plaintexts, whereas CKKS allows homomorphic operations on real numbers (or, complex numbers). But, TFHE is generally used for binary messages $\in \{0, 1\}$, and can evaluate any binary logic gate. TFHE supports bootstrapping after evaluation of every binary gate (except NOT gate), thus one can evaluate any arbitrary depth Boolean circuits using TFHE. Compared to other FHE schemes like BGV/BFV, TFHE provides much faster bootstrapping for binary gate evaluation[6]. As mentioned earlier, our present work is to generate an optimal Boolean representation of circuits by introducing a few optimization techniques (discussed later in this section). These optimization techniques are only for the TFHE scheme and one can achieve similar performance efficiency irrespective of the implementation language of the TFHE scheme. For our base implementation, we used the Julia language for TFHE[7] library due to the simplicity and plainness of the Julia. In our present setup the bootstrapping in TFHE-Julia takes approximately 86 ms, whereas in the C++ based TFHE library[8] implementation, this bootstrapping time is around 17 ms, for which one can observe a time difference in the homomorphic evaluation of Boolean circuits. For example, AES-128 takes around 2224.68 secs in TFHE-Julia, where using the C++-based implementation we get 352.39 secs for the same AES-128 circuit.

Now, we delve into several gate-level design techniques that incorporate variations in both circuit constructions and the number of inputs to each gate. By employing these optimizations, we generate Boolean circuits and assess their performance on the TFHE platform. As we mentioned earlier that at most one Boolean gate can be evaluated without bootstrapping operation, we present a detailed analysis on this in Section 4.1.

Due to the notion of the gate-bootstrapping technique in the TFHE scheme, one can evaluate a binary gate followed by a bootstrapping operation. But our observation is that a binary gate taking fresh ciphertexts as inputs can operate without performing the bootstrapping after homomorphic evaluation. Here, with fresh ciphertext, we denote either fresh encryption of plaintext or output of a bootstrapping gate evaluation. This means after evaluating one binary gate without bootstrapping will result in a correct decryption, but the output

[6]Note that, for arithmetic circuits, BGV/BFV can perform better than TFHE, because in TFHE evaluating arithmetic circuits requires first decomposing the input plaintexts into bits and encrypting each bit separately, resulting in huge number of ciphertetxs

[7]https://github.com/nucypher/TFHE.jl

[8]https://github.com/tfhe/tfhe.git

i.e., the noisy ciphertext when fed into another binary gate, a bootstrapping operation is compulsory for correct decryption of the output. This observation helps us to incorporate several optimization techniques that lead us to construct optimal circuit descriptions for the TFHE scheme. In Figure. 1a, we denote the Boolean gate with " $\times$ " symbol as "without bootstrapping gate", whereas the other one as "with bootstrapping gate". Now we mention the tricks to optimize the homomorphic circuit evaluation, which is as follows:

- Observe that, in Figure. 1c we can omit one bootstrapping from the first gate given a chain of two gates connected in series (shown within dotted lines), resulting in a total of two bootstrapping for the entire circuit. We use this trick of "reduced bootstrapping" to replace such a chain of Boolean gates. This refers to if a Boolean circuit has n inputs, then with the reduced bootstrapping techniques we can reduce the total number of bootstrapping operations almost by $\frac{n}{2}$, as illustrated in the subsequent section. This is basically done by using bootstrap-disabled (without bootstrapping) binary gates alternatively after every bootstrap-enabled (with bootstrapping) gates, keeping the last binary gate of the Boolean circuit as bootstrap-enabled to return a fresh resultant ciphertext. The dotted enclosure in Figure. 1c can be referred as 3-input ("multi-input" in general) gates with reduced bootstrapping. Note here that in a Boolean circuit with reduced bootstrapping technique, we always prefer to keep the bootstrapping operation in the last binary gate such that every Boolean circuit evaluation should return a fresh ciphertext as output.

- Because the NOT gates do not require bootstrapping, we will try to reduce other bootstrap-enabled gates such as XORs, ANDs while involving more number of NOT gates without altering the functionality of the circuits. For example, consider the following algebraic operations: $(a \wedge b) \oplus a$, $(a \wedge b) \oplus b$, where a and b are Boolean variables with $\oplus$, $\wedge$ and $\bar{\cdot}$ denoting XOR, AND and NOT gate operations respectively. Here, we can replace the above algebraic operation with $a \wedge (\bar{b})$ and $b \wedge (\bar{a})$ respectively. Observe that we reduce one bootstrap-enabled gate (XOR) and introduce one NOT gate, resulting in sub-optimal algebraic formulation with lesser homomorphic evaluation time. We denote this technique as "Inverter based optimization" or "Inv-Opt" in short.

We use the above-mentioned techniques to further optimize the most efficient Boolean representation generated using our circuit synthesis tool. Throughout our evaluation of Boolean circuits, we have made significant and crucial observations concerning two key factors: circuit depth and evaluation time. In the following section, we will explore some intriguing test cases that will play a pivotal role in optimizing circuit synthesis for our framework.

$$\begin{aligned} \mathsf{Lib\_Set1} &= \{\mathsf{XOR}, \mathsf{AND}, \mathsf{NOT}\}, \\ \mathsf{Lib\_Set2} &= \{\mathsf{NAND}, \mathsf{NOR}, \mathsf{OR}\} \bigcup \mathsf{Lib\_Set1}, \end{aligned} \tag{1}$$

3.1 Notable Observations on Evaluating Small Boolean Circuits in TFHE

As the TFHE scheme operates in binary message space, it provides an efficient evaluation of Boolean circuits of unrestricted depth for computing arbitrary functions, due to the per gate bootstrapping technique. As we already mentioned, in TFHE at most one binary gate can be evaluated without bootstrapping (refer to Figure. 1a). We present a detailed discussion on this in Section 4.1. We now note some unique observations related to a Boolean circuit evaluation in TFHE scheme that pave the way for optimized circuit synthesis for our framework (discussed in Section. 5). We resort to following three simple experiments by evaluating toy Boolean functions to showcase our observations.

Experiment 1. Let us consider a Boolean function, $y = x_1 \wedge x_2 \wedge \bar{x_3} \wedge x_4$. We synthesized and represented the Boolean function using Bristol Format for the circuits with different gate constructions as shown in Figure. 1b and Figure. 1c. We then evaluated both the circuit's constructions using the TFHE scheme and observed the following: an evaluation time of 0.17 secs is achieved using the construction as in Figure. 1b (using all bootstrap-enabled gates). While using the gate construction of Figure. 1c (using reduced number of bootstrapping), we achieve an evaluation time of 0.12 secs. Thus we gained an efficiency of around 30% on this simple Boolean function using the construction as shown in Figure. 1c. Now, consider the dotted line in Figure. 1c, where two binary gates are connected in series. In this case, we can avoid the bootstrapping in the first gate which leads to the construction of a 3-input Boolean gate with only one bootstrapping. Extending the notion of per-gate bootstrapping from TFHE scheme, we can introduce these 3-input gates with reduced bootstrapping as a part of basic Boolean gates to construct our modified TFHE scheme, because a 3-input supports the notion of per gate bootstrapping of TFHE scheme. With this introduction of 3-input gates with reduced bootstrapping technique, we can achieve the depth improvement of 33.3%, considering a 3-input gate has depth 1 due to single bootstrapping requirement. Therefore, we make the following inference.

Employing Boolean gates with reduced bootstrapping has the potential to enhance the efficiency of executing Boolean functions in our modified TFHE scheme.

In the succeeding section (cf. Section. 4), we will explore this observation in greater detail and provide theoretical insights to substantiate the underlying reasons behind this observation.

Experiment 2. Here we consider a Boolean function, $(x_1 \wedge x_2) \oplus x_1$, where $\oplus$ represents XOR operation. We conducted synthesis and evaluation of this function in the TFHE setup, which took approximately 0.174 secs with a reported circuit depth of 2. Next, we applied our Inv-Opt where we use NOT gate to transform the function into the form $x_1 \wedge \bar{x_2}$, resulting in a reduced depth of 1 and an evaluation time of 0.086 secs. As a result, we observed an approximate 50% improvement in both evaluation time and depth. Therefore, we make the following inference.

Inverter gates do not require bootstrapping, leading to more efficient homomorphic computation times when using Inverter-based optimization.

Experiment 3. We now elaborate on using standard cell library (shown in Equation 1), which includes basic AND, NAND, NOR, XOR and NOT gates to construct Boolean circuits.

Table 1: Summary of Results from Toy Experiments. (τ-Evalustion Time, δ-Depth of the Circuit)

Expt.	τ(sec.), δ		% Gain	Inference
	Initial	Final	(τ, δ)	
1	0.17, 3	0.12, 2	31.3, 33.3	Reduced bootstrapping gates
2	0.174, 2	0.086, 1	50.5, 50	Inverter-based optimization
3	0.2593, NA	0.2589, NA	0.0015, NA	Using NAND-OR gates during synthesis

In Lib_Set1 we keep only the Boolean gates used in Bristol formats and in Lib_Set2 we keep other basic Boolean gates along with XOR, AND and NOT, therefore, our optimal Boolean representation is not specific to Bristol formats, it can consist of Boolean gates from Lib_Set2. For example, consider a Boolean circuit: $(\overline{\bar{x_0} \wedge \bar{x_1}}) \oplus (x_2 \bar{\wedge} x_3)$. The homomorphic evaluation of the Bristol representation of the above circuit takes around $259.32ms$, but using Boolean gates from the standard library cell we achieve the following optimal representation: $(x_0 \vee x_1) \oplus (x_2 \bar{\wedge} x_3)$, where $\vee$ and $\bar{\wedge}$ represents OR and NAND gates respectively. Using the later Boolean representation, we achieve an improvement with $258.99ms$ of homomorphic evaluation time. We now make our third inference. This improvement will have a noticeable effect on large circuits such as AES block ciphers or neural networks.

Using Boolean gates from standard cell library in the TFHE scheme resulting in more efficient homomorphic evaluation time.

A summary of all our experimental observations is provided in Table 1. Overall, we show that this exploration of gate-level design techniques and their evaluation on the TFHE scheme contributes valuable insights towards advancing the field of homomorphic encryption and enhancing its practicality in various domains requiring privacy-preserving computations. We utilize these techniques to introduce the FHEDA flow, which will be further explored in greater detail in the subsequent sections. Before delving into that, our aim in the next section is to offer a comprehensive analytical and theoretical demonstration, highlighting how the integration of Boolean gates with reduced bootstrapping can enhance the efficiency of homomorphic evaluation of Boolean circuits in the fully homomorphic encryption (FHE) domain.

4 Boolean Gates with Reduced Bootstrapping

In the Bristol Formats, the circuits are mainly represented with 2-input XOR and AND gates with unary NOT gates. Interestingly, these Bristol representations can be modified to incorporate multi-input Boolean gates to improve the efficiency and throughput of the Boolean circuit evaluations. In this section, we show, both theoretically and analytically, that using 3-input gates with reduced bootstrapping in place of 2-input gates provides a significant advantage in terms of homomorphic computational time. In particular, we show that by using 3-input gates, we reduce the total number of bootstrapping operations required in a circuit and simultaneously lead to a lower depth of the circuits.

In general, the majority of standard cell libraries consist of both 2-input and multi-input gates, often supporting up to five inputs. In terms of hardware area/latency, in most cases using a multi-input gate is generally smaller and more efficient in terms of latency than the combined areas of multiple 2-input gates. Consequently, if a logic synthesizer is provided with a behavioral description of a system in a hardware description language and tasked with generating an area-optimized/latency-optimized circuit, it typically designs a circuit that utilizes a combination of gates having multiple inputs. In this section, we will dive deeper into this approach and demonstrate through both empirical and theoretical analysis that the utilization of a combination of 2-input and 3-input reduced bootstrapping gates leads to the creation of highly optimized TFHE-friendly circuits. Specifically, these circuits exhibit improved performance in terms of homomorphic evaluation.

4.1 Evaluating Single 2-input Gate Without Bootstrapping

In the previous section, we observed that in Figure. 1c, bootstrapping is not required for the first stage 2-input AND gate. Here, we present a theoretical explanation for why it is possible to evaluate a 2-input Boolean gate without the need for performing the bootstrapping operation within the Torus-FHE scheme. Let us assume, $\mathbf{ct}_1 = (b_1, \mathbf{a}_1)$ and $\mathbf{ct}_2 = (b_2, \mathbf{a}_2)$ are two TFHE ciphertexts encrypting $m_1 \in \{0,1\}$ and $m_2 \in \{0,1\}$ respectively. According to the TFHE library, the plaintexts m_1 and m_2 are scaled in the range $[-\frac{1}{8}, \frac{1}{8}]$. Assume, Δ be the scaling function that maps $\Delta(0) = -\frac{1}{8}$ and $\Delta(1) = \frac{1}{8}$. Hence, we have

$$b_1 = \langle \mathbf{a}_1, \mathbf{sk} \rangle + \Delta(m_1) + e_1,$$
$$b_2 = \langle \mathbf{a}_2, \mathbf{sk} \rangle + \Delta(m_2) + e_2,$$

where, $\mathbf{sk} \in \{0,1\}^n$ be the secret key and $e_1, e_2 \in \mathcal{G}_\alpha$ are the encryption noise sampled from a Gaussian distribution with standard deviation α. According to the TFHE library, to perform correct decryption the encryption noises should satisfy $\|e_1\|_\infty, \|e_1\|_\infty < \frac{1}{16}$.

Now, we demonstrate the homomorphic AND opertion between $\mathbf{ct}_1$ and $\mathbf{ct}_2$,

$$\mathsf{HomAND}(\mathbf{ct}_1, \mathbf{ct}_2) = \left(-\frac{1}{8}, \mathbf{0}\right) + (\mathbf{ct}_1 + \mathbf{ct}_2).$$

Let, $\mathbf{ct}^\star = (b^\star, \mathbf{a}^\star)$ be the output of the homomorphic AND operation $\mathsf{HomAND}(\mathbf{ct}_1, \mathbf{ct}_2)$. Hence, we have

$$\mathbf{a}^\star = (\mathbf{a}_1 + \mathbf{a}_2),$$
$$b^\star = -\frac{1}{8} + \mathbf{a}^\star \cdot \mathbf{sk} + (\Delta(m_1) + \Delta(m_2)) + (e_1 + e_2)$$

Now, let us analyze what will happen if we perform decryption of $\mathbf{ct}^\star$ without performing the bootstrapping over $\mathbf{ct}^\star$ for each of the cases of choosing the messages (m_1, m_2) and noises (e_1, e_2). In order to decrypt $\mathbf{ct}^\star$, first compute a phase $\Phi = b^\star - \mathbf{a}^\star \cdot \mathbf{sk} \pmod 1$, now if $\Phi > 0$ then return 1, otherwise return 0.

Consider the following analysis of the decryption operation of $\mathbf{ct}^\star$ with $m_1 = 0, m_2 = 0$, i.e., $\Delta(m_1) = -\frac{1}{8}, \Delta(m_2) = -\frac{1}{8}$ and the encryption noises with the same sign, i.e., either $0 \le e_1, e_2 < \frac{1}{16}$, or $-\frac{1}{16} < e_1, e_2 \le 0$,

$$\begin{aligned}
\Phi &= -\frac{1}{8} + (\Delta(m_1) + \Delta(m_2)) + (e_1 + e_2), \\
&= -\frac{1}{8} + \left(-\frac{1}{8} - \frac{1}{8}\right) + (e_1 + e_2), \\
&= -\frac{3}{8} + (e_1 + e_2), \\
&= -\frac{3}{8} + \delta \quad \left[0 \le \delta < \frac{1}{8}, \text{ as } 0 \le e_1, e_2 < \frac{1}{16}\right], \\
&= \delta' \quad \left[-\frac{3}{8} \le \delta' < -\frac{1}{4}, \text{ for both } m_1 = 0, m_2 = 0\right].
\end{aligned}$$

Now, with another extreme scenario when both the encryption noises are negative,

$$\begin{aligned}
&= -\frac{3}{8} + (e_1 + e_2), \\
&= -\frac{3}{8} + \delta \quad \left[0 \ge \delta > -\frac{1}{8}, \text{ as } -\frac{1}{16} < e_1, e_2 \le 0\right], \\
&= \delta' \quad \left[-\frac{1}{2} < \delta' \le -\frac{3}{8}, \text{ for both } m_1 = 0, m_2 = 0\right]
\end{aligned}$$

In both of these cases, $-\frac{1}{2} < \Phi < -\frac{1}{4}$, i.e., $\Phi \pmod 1 = \Phi < 0$, hence the decryption of AND upon $m_1 = 0$ and $m_2 = 0$ is 0, which is a correct decryption. Now, consider when both the plaintexts are $m_1 = m_2 = 1$, i.e., $\Delta(m_1) = \Delta(m_2) = \frac{1}{8}$,

$$\begin{aligned}
\Phi &= -\frac{1}{8} + (\Delta(m_1) + \Delta(m_2)) + (e_1 + e_2), \\
&= -\frac{1}{8} + \left(\frac{1}{8} + \frac{1}{8}\right) + (e_1 + e_2), \\
&= \frac{1}{8} + (e_1 + e_2), \\
&= \frac{1}{8} + \delta \quad \left[0 \le \delta < \frac{1}{8}, \text{ as } 0 \le e_1, e_2 < \frac{1}{16}\right], \\
&= \delta' \quad \left[\frac{1}{8} \le \delta' < \frac{1}{4}, \text{ for both } m_1 = 1, m_2 = 1\right].
\end{aligned}$$

Now, with another extreme when both the encryption noises are negative,

$$\begin{aligned}
&= \frac{1}{8} + (e_1 + e_2), \\
&= \frac{1}{8} + \delta \quad \left[0 \geq \delta > -\frac{1}{8}, \text{ as } -\frac{1}{16} < e_1, e_2 \leq 0\right], \\
&= \delta' \quad \left[0 \leq \delta' < \frac{1}{8}, \text{ for both } m_1 = 1, m_2 = 1\right].
\end{aligned}$$

In both of these cases, $0 \leq \Phi < \frac{1}{4}$, i.e., $\Phi \pmod 1 = \Phi > 0$, hence the decryption of AND upon $m_1 = 1$, $m_2 = 1$ is 1, which is a correct decryption. We now consider the case when both the plaintexts are of opposite sign, without loss of generality consider $m_1 = 0, m_2 = 1$, and thus $\Delta(m_1) = -\frac{1}{8}$ and $\Delta(m_2) = \frac{1}{8}$,

$$\begin{aligned}
\Phi &= -\frac{1}{8} + (\Delta(m_1) + \Delta(m_2)) + (e_1 + e_2), \\
&= -\frac{1}{8} + \left(-\frac{1}{8} + \frac{1}{8}\right) + (e_1 + e_2), \\
&= -\frac{1}{8} + (e_1 + e_2), \\
&= -\frac{1}{8} + \delta \quad \left[0 \leq \delta < \frac{1}{8}, \text{ as } 0 \leq e_1, e_2 < \frac{1}{16}\right], \\
&= \delta' \quad \left[-\frac{1}{8} < \delta' < 0, \text{ for } m_1 = 0, m_2 = 1\right].
\end{aligned}$$

Again consider another extreme case when both the encryption noises are negative,

$$\begin{aligned}
&= -\frac{1}{8} + (e_1 + e_2), \\
&= -\frac{1}{8} + \delta \quad \left[0 \geq \delta > -\frac{1}{8}, \text{ as } -\frac{1}{16} < e_1, e_2 \leq 0\right], \\
&= \delta' \quad \left[-\frac{1}{4} < \delta' < -\frac{1}{8}, \text{ for } m_1 = 0, m_2 = 1\right].
\end{aligned}$$

In both of these cases, $-\frac{1}{4} < \Phi < 0$, i.e., $\Phi \pmod 1 = \Phi < 0$ hence the decryption of AND upon $m_1 = 0$ and $m_2 = 1$ is 0, which is a correct decryption. So far, we have demonstrated that the extreme choices of encryption noises i.e., when both of them are of the same sign, we can achieve correct decryption without the bootstrapping and it is trivial to show similar cases when noises are of the opposite sign; that is the value of δ' in each of the above cases will remain in the same range. A similar approach as mentioned above can be achieved for homomorphic XOR operation on $\mathbf{ct}_1$ and $\mathbf{ct}_2$, which is,

$$\mathsf{HomXOR}(\mathbf{ct}_1, \mathbf{ct}_2) = \left(\frac{1}{4}, \mathbf{0}\right) + 2 \cdot (\mathbf{ct}_1 + \mathbf{ct}_2).$$

Table 2: Homomorphic Evaluation time in ms of 2-input and 3-input XOR gate

Fan-In	HomXOR time	**Bootstrapping Time**	**Total Eval. Time**
2	0.083	86.25	86.33
3	0.166	86.25	86.41

4.2 Constructing 3-input Gates

Now, with the above analysis, we show how to construct 3-input AND and XOR gates. As we already know, homomorphic AND is $\mathsf{HomAND}(\mathbf{ct}_1, \mathbf{ct}_2) = \left(-\frac{1}{8}, \mathbf{0}\right) + (\mathbf{ct}_1 + \mathbf{ct}_2)$; now perform homomorphic AND over $\mathbf{ct}_1$, $\mathbf{ct}_2$ and $\mathbf{ct}_3$ or $\mathsf{HomAND}(\mathbf{ct}_1, \mathbf{ct}_2, \mathbf{ct}_3)$ as follows,

$$\mathbf{ct}_{tmp} = \left(-\frac{1}{8}, \mathbf{0}\right) + (\mathbf{ct}_1 + \mathbf{ct}_2),$$
$$\mathbf{ct}^\star = \left(-\frac{1}{8}, \mathbf{0}\right) + (\mathbf{ct}_{tmp} + \mathbf{ct}_3).$$

Here, $\mathbf{ct}_{tmp}$ can be computed without bootstrapping as described above, but $\mathbf{ct}^\star$ should undergo a bootstrapping procedure for decryption correctness. Hence, to evaluate a 3-input AND gate we only need one bootstrapping instead of two. Compactly we can also write,

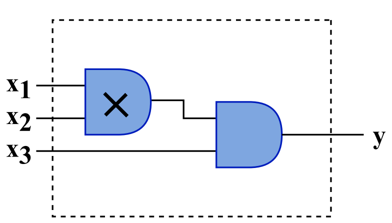
$$\mathsf{HomAND}(\mathbf{ct}_1, \mathbf{ct}_2, \mathbf{ct}_3) = \left(-\frac{1}{4}, \mathbf{0}\right) + (\mathbf{ct}_1 + \mathbf{ct}_2 + \mathbf{ct}_3).$$

Similarly, we can achieve a compact version of 3-input homomorphic XOR gate as follows,

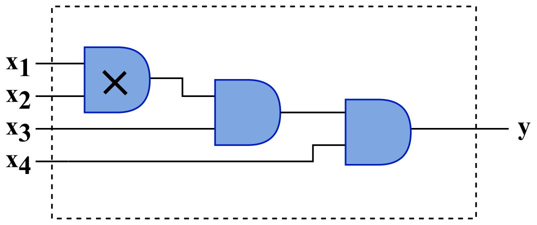
$$\mathsf{HomXOR}(\mathbf{ct}_1, \mathbf{ct}_2, \mathbf{ct}_3) = \left(\frac{1}{2}, \mathbf{0}\right) + 2 \cdot (\mathbf{ct}_1 + \mathbf{ct}_2 + \mathbf{ct}_3).$$

Note that, in TFHE scheme we can avoid the bootstrapping operation only for one 2-input binary gate, which allows us to construct 3-input Boolean gates with reduced bootstrapping using only one bootstrapping operation. As bootstrapping is the primary bottleneck of evaluating a Boolean gate, both the 2-input and 3-input versions will take approximately similar evaluation time (Table 2 presents the evaluation time for 2 and 3-input homomorphic XOR gate).

From the above observation, we conclude that at most one binary gate can be evaluated without bootstrapping, hence, it is sufficient to allow bootstrapping after the evaluation of two binary gates connected in series. This implies we can also evaluate a 3-input binary gate followed by a single bootstrapping operation. Therefore, using this observation we can use 3-input reduced bootstrapping gates (along with 2-input gate) in the Bristol representations of the circuits, leading to a significant improvement in the efficiency of the circuits in terms of homomorphic evaluation time as shown in Figure. 1b and Figure. 1c.

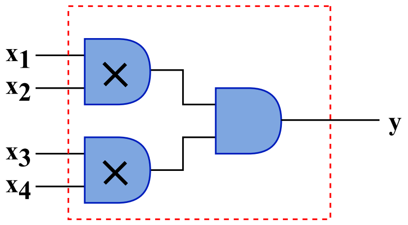


(a) 3-input AND gate with reduced bootstrapping

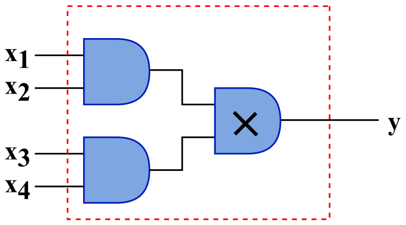


(b) 4-input AND gate with reduced bootstrapping

Figure 2: Multi-input AND gates



(a) Without bootstrapping



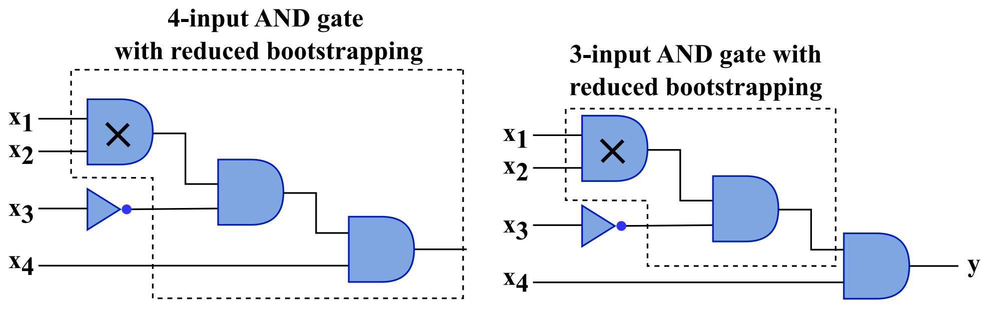
(b) With bootstrapping

Figure 3: Seemingly possible 4-input AND gate

4.3 3-input vs. n-input Gate ($n \geq 4$)

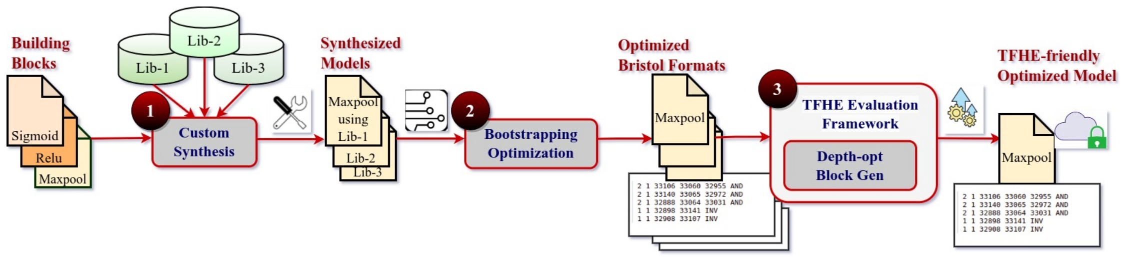
Here, we will elaborate on why it is sufficient to use 3-input Boolean gates over n-input Boolean gates with reduced bootstrapping for $n \geq 4$. This technique is applicable for any Boolean gate but we have presented the elaboration for only Boolean AND gate. As shown in Figure. 2b, in a 4-input Boolean gate we can avoid bootstrapping after evaluating the first two inputs x_1 and x_2, but for the rest of the inputs x_3 and x_4, we definitely require the bootstrapping operation because the gate with x_3 as input also takes a noisy ciphertext from the previous gate and the for the last gate we prefer to have a bootstrap-enabled gate to return a fresh ciphertext as output (discussed earlier). This results in a total of two bootstrapping operations for a 4-input gate, which actually violates the notion of per-gate bootstrapping in the TFHE scheme. Hence, in our modified TFHE scheme we only introduce 3-input Boolean gates with only one bootstrapping operation per gate.

Also, it is not hard to observe that one 4-input Boolean gate with reduced bootstrapping can be represented with a combination of one 2-input and one 3-input gate with reduced bootstrapping, in which we again require only two bootstrapping operations; hence a 4-input gate provides no more advantage over a combination of a 2-input and 3-input gates. Now, one might think of a representation of 4-input Boolean gate as mentioned in Figure. 3, i.e., in the first level gates, the inputs x_1 and x_2 can be evaluated with a 2-input bootstrap-disabled gate and the inputs x_3 and x_4 can also be evaluated in a similar manner, i.e., by a bootstrap-disabled 2-input gate. But observe that the second level gate has both the inputs as noisy ciphertexts which will result in a decryption failure even after bootstrapping (refer Figure. 3a). Further, revisiting our toy example $y = x_1 \cdot x_2 \cdot \bar{x_3} \cdot x_4$, we will try to represent



(a) With 4-input AND gates (Evaluation time: 0.12 secs) (b) With both 2 and 3-input AND gates (Evaluation time: 0.12 secs)

Figure 4: Comparison between 4-input and 3-input AND gates with reduced bootstrapping with a toy example



```
2 1 33106 33060 32955 AND
2 1 33140 33065 32972 AND
2 1 32888 33064 33031 AND
1 1 32898 33141 INV
1 1 32908 33107 INV
```

```
2 1 33106 33060 32955 AND
2 1 33140 33065 32972 AND
2 1 32888 33064 33031 AND
1 1 32898 33141 INV
1 1 32908 33107 INV
```

Figure 5: Proposed EDA flow to Generate Optimized TFHE-friendly Circuit

this circuit using 4-input and 3-input gates with reduced bootstrapping (along with 2-input gates) as shown in Figure. 4. It is easy to observe from Figure. 4 that in both cases two bootstrapping operations are required, resulting in a similar evaluation time; therefore using 4-input gates provides no extra advantage. Now, for the general case, i.e., a n-input Boolean gate can also be represented using an optimal combination of 2-input and 3-input Boolean gates with at-most $(\frac{n}{2})$ bootstrapping operations, which is trivially needed in a n-input Boolean gate with reduced bootstrapping structure.

5 FHEDA: Our Proposed EDA Flow

In this section, we elaborate on our automated FHE circuit synthesis framework, FHEDA. We start with the challenges of classical logic synthesis flow on FHE circuits, followed by a detailed description of our proposed flow. The high-level architecture of our framework is shown in Figure. 5. The purpose of this framework is to generate depth-optimized and bootstrap-efficient circuit modules for deployment into the secure computation using TFHE framework. In particular, the framework aims to automatically and efficiently synthesize a Boolean circuit written in HDL into an optimized representation, geared towards minimizing both bootstrapping time and circuit depth. Our proposed EDA flow FHEDA essentially comprised three major stages: ① stage deals with the custom synthesis of the given behavioral module constrained under various standard cell library sets (discussed in Section 6.2) to generate multiple gate-level netlists of the circuit. ② converts the generated gate-level netlists to the corresponding Bristol Format and optimizes these Bristol Formats with the objective of minimizing the number of most sophisticated and compute-intensive bootstrapping, thereby significantly improving the overall performance and efficiency of the circuit. In the final stage (③) of FHEDA, we evaluate the optimized Bristol Formats within the TFHE platform where we assess the circuits in terms of both circuit-depth and bootstrapping time. The goal of this step is to identify and select the most optimized circuit description, considering both of these essential factors. In the following subsections, we will delve into these stages in greater detail.

5.1 Challenges of Logic Synthesis for FHE Circuits

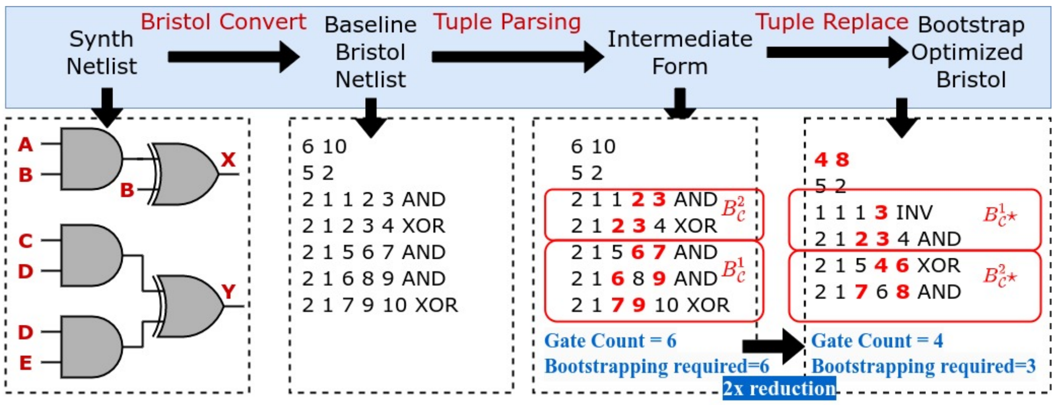
The generation of TFHE friendly circuits for secure computation using hardware synthesis presents two primary obstacles. Initially, the classical hardware synthesis flows (both commercial and open-source) are designed to optimize hardware platforms which come with distinct technology constraints concerning PPA (power, performance, and area) optimization not applicable to FHE Boolean circuits. In Section 4 we have seen that using a 4-input gate with reduced bootstrapping does not add any advantage to the evaluation time of the FHE circuit. But classical synthesis flow is typically biased towards using 4-input gates with the objective to optimize area and latency. The second challenge lies in the considerable variation in gate costs between classical synthesis flow and FHE circuits. In classical logic synthesis tools, Boolean NAND gates are preferred over AND-XOR gates due to their lower placement and area footprint costs. However, in FHE circuits, AND-XOR gates can be used to generate FHE-friendly circuits as demonstrated through Experiment 3 results in Section 3.1. As a result, it becomes imperative to modify and fine-tune classical logic synthesis tools to align with our objectives in security applications, especially when creating depth-optimized and bootstrapping time-optimized FHE-friendly Boolean circuits. In the following subsection, we will explore the different stages of our proposed FHEDA flow to generate optimized TFHE-friendly circuits in detail.

5.2 FHEDA Flow: Stages

Figure. 5 illustrates the high-level architecture of our proposed FHEDA framework. The primary objective of this framework is to produce FHE-friendly circuit modules that are optimized for depth and bootstrap efficiency, making them suitable for deployment within the secure computation in TFHE framework. The FHEDA flow comprises three key stages and in the following subsections, we will provide a detailed exploration of these stages

5.2.1 Stage:①- Custom Synthesis

Our third experimental demonstration in Section 3.1 has clearly shown that leveraging customized library sets provides substantial benefits in the FHE domain, resulting in noteworthy enhancements in computation efficiency. By tailoring the library sets to the specific library cells, we can achieve faster computation times and reduced resource consumption concerning



```
6 10
5 2
2 1 1 2 3 AND
2 1 2 3 4 XOR
2 1 5 6 7 AND
2 1 6 8 9 AND
2 1 7 9 10 XOR
```

```
6 10
5 2
2 1 1 2 3 AND
2 1 2 3 4 XOR
2 1 5 6 7 AND
2 1 6 8 9 AND
2 1 7 9 10 XOR
Gate Count = 6
Bootstrapping required=6
```

```
4 8
5 2
1 1 1 3 INV
2 1 2 3 4 AND
2 1 5 4 6 XOR
2 1 7 6 8 AND
Gate Count = 4
Bootstrapping required=3
```

Figure 6: Optimization of Bootstrapping Components

circuit depth and bootstrapping requirements. In Section 4, we have both empirically and theoretically demonstrated that using multi-input gates with reduced bootstrapping limited to a maximum of three inputs can significantly enhance the efficiency of FHE circuits. Taking into account the mentioned facts and figures, we primarily create standard cell library sets, denoted as N sets $(set_1, set_2, \ldots, set_N)$ (described elaborately in Section 6.2). Each set consists of various combinations of 2-input, 3-input, and/or both logic gates. These library sets are utilized for synthesizing a given FHE circuit effectively. Thus, with N number of sets, this stage produces N synthesized netlists for a given FHE behavioral model. These generated netlists are then converted into their respective Bristol Format (we call this our baseline circuit representation) and forwarded to stage ② of the flow. In this stage, each netlist is optimized to reduce bootstrapping components, as detailed in the following section.

5.2.2 Stage:②- Optimization of Bootstrapping Component

As we already know, in Bristol Formats the Boolean circuits are mainly represented using basic Boolean gates XOR, AND, and NOT. Among these gates, only NOT gate over encrypted input is computationally much lighter (almost $10^3\times$ faster) because NOT gate does not require bootstrapping, unlike other Boolean gates. Therefore, it is advisable to increase the number of NOT gates while reducing other Boolean gates by specific algebraic reductions (mentioned in Experiment 2, Section 3.1) without altering the circuit functionality in the Bristol format, or we can find ways to simplify Boolean clauses into their compact forms. By doing so, we can substantially reduce the number of bootstrappable gates, resulting in more efficient and faster computations. In this section, we provide a detailed discussion of this step as shown in Figure. 6. Consider N sets $(set_1, set_2, \ldots, set_N)$ containing different combinations of logic gates, e.g., set_i may contain 2-input and 3-input XOR, AND, and NOT along with NAND, NOR and OR gates. Assume, $\mathcal{C} : \{0,1\}^\star \to \{0,1\}^\star$ be a Boolean circuit and let $B_\mathcal{C}$ be its Bristol representation. The functionality in $B_\mathcal{C}$ is then converted into multiple Boolean formats using the Boolean gates from each of the sets in $\{set_1, \ldots, set_N\}$ using our Convert$(set_i, B_\mathcal{C})$ algorithm that takes a set set_i and the baseline Bristol Format

Table 3: Homomorphic Evaluation time of the tuples of the set struct

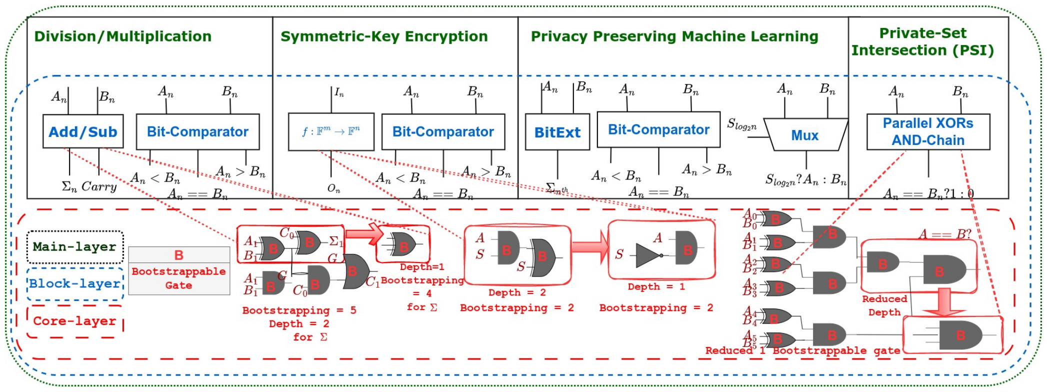
Expression Before Replacement	**Eval. time (in ms.)**	**Expression After Replacement**	**Eval. time (in ms.)**
$(a \cdot b) \oplus a$	172.80	$a \cdot (\bar{b})$	86.48
$(a \cdot b) \oplus b$	172.80	$b \cdot (\bar{a})$	86.51
$(a \cdot b) \oplus (b \cdot c)$	259.20	$b \cdot (a \oplus c)$	172.80
$(a \oplus b) \oplus (a \oplus c)$	259.20	$(b \oplus c)$	86.43

Algorithm 1: Optimal Circuit Formation

```
Require: $\{set_1, \dots, set_N\}$, $B_{\mathcal{C}}$, $\mathsf{struct}$
Ensure: $(B_{\mathcal{C}}^{1\star}, \dots, B_{\mathcal{C}}^{N\star})$
1: Initialize empty sets $\mathsf{B} = \{\}$ and $\mathsf{B}^\star = \{\}$
2: for $i = 1$ to $N$
3:     $B_{\mathcal{C}}^{i} \leftarrow \mathsf{Convert}(set_i, B_{\mathcal{C}})$
4:     $\mathsf{B} \leftarrow \mathsf{B} \cup B_{\mathcal{C}}^{i}$
5: for $i = 1$ to $N$
6:     $B_{\mathcal{C}}^{i\star} \leftarrow \mathsf{Parse}(\mathsf{struct}, \mathsf{B}[i])$
7:     $\mathsf{B}^\star \leftarrow \mathsf{B}^\star \cup B_{\mathcal{C}}^{i\star}$
8: return $\mathsf{B}^\star$
```

$B_{\mathcal{C}}$ as input and returns a different circuit description $B_{\mathcal{C}}^{i}$ of $\mathcal{C}$ only using the Boolean gates from set set_i. Running $\mathsf{Convert}(\cdot)$ algorithm N times we get the set $\left(B_{\mathcal{C}}^{1}, \dots, B_{\mathcal{C}}^{N}\right)$ representing N circuit descriptions of the same circuit.

After the above conversion we again parse all the N circuit representations, i.e., $\left(B_{\mathcal{C}}^{1}, \dots, B_{\mathcal{C}}^{N}\right)$ using our $\mathsf{Parse}(\cdot)$ algorithm. This algorithm takes a set struct and $\left(B_{\mathcal{C}}^{1}, \dots, B_{\mathcal{C}}^{N}\right)$ as input and outputs another set $\left(B_{\mathcal{C}}^{1\star}, \dots, B_{\mathcal{C}}^{N\star}\right)$. The set struct consists of a set of tuples described as follows: $\mathsf{struct} = \Big\{\Big((a \cdot b) \oplus a, a \cdot (\bar{b})\Big), \Big((a \cdot b) \oplus b, b \cdot (\bar{a})\Big), \Big((a \cdot b) \oplus (b \cdot c), b \cdot (a \oplus c)\Big), \Big((a \oplus b) \oplus (a \oplus c), (b \oplus c)\Big)\Big\}$, here a, b and c are Boolean variables, $\oplus$, $\wedge$ and $\bar{\cdot}$ denotes XOR, AND and NOT gate operation respectively. Now, our $\mathsf{Parse}(\mathsf{struct}, B_{\mathcal{C}}^{i})$ algorithm parse each $B_{\mathcal{C}}^{i_{i \in [N]}}$ representation to find the Boolean clauses present in the first component of the tuples in struct and replaces with the second component of the corresponding tuples. Hence, running $\mathsf{Parse}(\cdot)$ algorithm gives another set containing N circuit descriptions with compact representations, i.e., $(B_{\mathcal{C}}^{1\star}, \dots, B_{\mathcal{C}}^{N\star})$. This proposed technique of converting the original Bristol Format into an optimal circuit representation is briefly described in Algorithm 1. We demonstrate the improvement of our proposed struct replacement both mathematically and experimentally in Table 3. Furthermore, we visually depict the struct replacement in Figure. 6, where the gate count of the sample circuit is reduced from 6 to 4, resulting in only 3 bootstrappable gates due to our $\mathsf{Inv\text{-}Opt}$. This optimization yields a $2\times$ improvement in efficient computation. Moreover, in the upcoming section, we will explain the circuit depth optimization process by assessing the generated designs at this stage in the TFHE platform.



Division/Multiplication
Symmetric-Key Encryption
Privacy Preserving Machine Learning
Private-Set Intersection (PSI)
Add/Sub
Bit-Comparator
BitExt
Mux
Parallel XORs AND-Chain
Main-layer
Block-layer
Core-layer
Bootstrappable Gate
Bootstrapping = 5 Depth = 2 for Σ
Depth=1 Bootstrapping = 4 for Σ
Depth = 2 Bootstrapping = 2
Depth = 1 Bootstrapping = 2
Reduced Depth
Reduced 1 Bootstrappable gate

Figure 7: Three-layered Execution of Proposed EDA Flow

5.2.3 Stage-③: Depth-optimized Building Block Generation

In this section, we will describe the homomorphic evaluation of Boolean circuits using TFHE scheme that is generated in stage② of our proposed flow. As we already described in the previous section we can build an optimal Boolean representation of any circuit, and this optimal representation is now evaluated homomorphically over the encrypted inputs. In Algorithm 2 we briefly presented the homomorphic evaluation procedure. Let's assume, $(m_1, \ldots, m_\ell)$ be the inputs of a circuit $\mathcal{C}^\star : \{0,1\}^\ell \rightarrow \{0,1\}^{\ell'}$. Now, before evaluating the circuit $\mathcal{C}^\star$ we parse its Bristol Format through the procedure mentioned in Algorithm 1 to retrieve the optimal Boolean representations of $\mathcal{C}^\star$. Let us assume, $\{B^1_{\mathcal{C}^\star}, \ldots, B^N_{\mathcal{C}^\star}\}$ are the N optimal Boolean representation of $\mathcal{C}^\star$. Now, to evaluate $\mathcal{C}^\star$, we first encrypt the inputs $(m_1, \ldots, m_\ell)$ using secret key $\mathbf{sk}$ and return the ciphertexts $(\mu_1, \ldots, \mu_\ell)$, such that $\mu_i = \mathsf{TFHE.Enc}(m_i, \mathbf{sk})$, $\forall i \in [\ell]$. Each $B^j_{\mathcal{C}^\star}$, $\forall j \in [N]$, we invoke $\mathsf{TFHE.Eval}((\mu_1, \ldots, \mu_\ell), B^j_{\mathcal{C}^\star}, \mathbf{ek})$, $\forall j \in [N]$ and compute the homomorphic evaluation time eval_j for circuit representation $B^j_{\mathcal{C}^\star}$. Finally, we retrieve the minimum of $\{\mathsf{eval}_1, \ldots, \mathsf{eval}_N\}$ as $\mathsf{eval}^\star$ and also returns the optimal Boolean representation for which the minimum evaluation time is achieved.

6 Three Layer Execution of FHEDA

Our proposed flow FHEDA follows a three-tiered execution strategy as shown in Figure. 7. The three layers are comprised of the core layer, block layer, and main layer, constituting FHEDA. The main layer which represents the topmost tier, is eventually responsible for implementing a privacy-preserving version of various functionalities using FHE-friendly circuits. In the subsequent subsections, we will explore each of these layers in greater detail supported with experimental evaluation details and results on a set of representative benchmark circuits.

```
Algorithm 2: Evaluation of Optimal Circuit
Require: $\{B^1_{\mathcal{C}^\star}, \dots, B^N_{\mathcal{C}^\star}\}$, $\mathsf{pk}$, $\{m_1, \dots, m_\ell\}$
Ensure: $\mathsf{eval}^\star, B^\star_{\mathcal{C}}$
1: Initialize empty sets $\mu = \{\}$ and $\mathsf{eval} = \{\}$
2: for $i = 1$ to $\ell$
3:     $\mu_i \leftarrow \mathsf{TLWE.Enc}(m_i, \mathsf{pk})$
4:     $\mu \leftarrow \mu \cup \mu_i$
5: for $i = 1$ to $N$
6:     $\mathsf{eval}_i \leftarrow \mathsf{TLWE.Eval}(\mathsf{B}^\star[i], \mu, \mathsf{pk})$
7:     $\mathsf{eval} \leftarrow \mathsf{eval} \cup \mathsf{eval}_i$
8: $\mathsf{eval}^\star = \mathbf{min}(\mathsf{eval}_1, \dots, \mathsf{eval}_N)$
9: Set $B^\star_{\mathcal{C}}$ as the optimal circuit with the minimum evaluation time
10: return $\mathsf{eval}^\star, B^\star_{\mathcal{C}}$
```

Table 4: Evaluation Time Reduction and Depth Gains of FHE-circuits at Block and Main Level when Compared to Baseline Library. ($**$-Baseline Library, $*$-(I,O)-Input bit-width, Output bit-width))

Circuit	bitwidth (I,O)*	Lib-Set 1** Eval Time (in secs.)	Circuit Depth	Lib-Set 2 Eval Time (in secs.)	Circuit Depth	Lib-Set 3 Eval Time (in secs.)	Circuit Depth	Lib-Set 4 Eval Time (in secs.)	Circuit Depth	% Depth Gain	% Eval Time Reduction
Block Layer											
Carry Select Adder	16,16	11.52	39	10.2	39	11.66	20	9.96	20	48.72	13.54
Mux 2:1	8,8	2.08	5	2.08	5	1.76	3	1.76	3	40.00	15.38
Mux 4:1	8,8	3.74	7	3.12	7	3.69	6	2.95	5	28.57	21.12
Comparator	4,3	1.83	16	1.67	16	1.75	9	1.36	9	43.75	25.68
PRESENT S-Box	4,4	2.65	10	1.73	9	2.23	8	1.32	7	30	50.18
AES S-Box	8,8	58.57	39	40.84	24	49.48	40	38.55	25	38.46	34.18
Keccak S-Box	5,5	1.38	5	1.21	4	1.47	7	1.21	5	20.00	12.32
Main Layer											
Division	8,8	47.11	238	37.71	237	43.62	147	35.02	140	41.17	25.66
Multiplication	8,16	28.64	60	22.85	60	26.31	38	20.95	38	36.67	26.85
Private Set Intersection	40,10	2.16	2	1.43	2	2.16	2	1.43	2	0.00	33.80
PRESENT Encryption	64,64	1,314.4	310	858.08	279	1,106.08	248	654.72	217	30.00	50.19
AES Encryption	128,128	16,030.53	507	11,071.94	317	17,676.81	487	11,146.18	327	37.47	30.93
AES Non-Expanded	128,128	2758.37	NA	2224.60	NA	2758.24	NA	2232.40	NA	NA	19.35
LowMC Encryption	128,128	660.71	84	434.85	56	660.41	84	434.45	56	34.24	34.24
Maxpool	8,8	3.38	28	2.7	23	3.32	18	2.66	13	53.57	21.30
ArgMax	32,32	22.31	308	22.30	306	21.91	179	21.91	177	42.53	1.82
ReLU	32,32	2.16	2	2.16	2	2.16	2	2.16	2	0.00	0.00

6.1 Experimental Setup

Here, we describe the experimental setup and tools integrated to FHEDA. We evaluated our benchmarks on a high-end x86-based computing platform. At stage ① of FHEDAflow (Figure. 5) we have used Cadence Genus (version:17.24$-s$038_1) for synthesis. The standard cell library used for synthesis is TSL18FS120 cell library from Tower Semiconductor Ltd. at the 180nm technology node. In stage ② of the flow, we employ our custom Python-based utility to optimize the Bristol formats. At stage③ of the flow our baseline TFHE scheme is used to implement the homomorphic evaluation function for different Boolean circuits. Our evaluation function takes a Boolean representation of a circuit and sequentially evaluates the binary gates according to the circuit representation. We use TFHE built-in functions for different binary gate evaluations and followed by the bootstrapping operation implemented in the TFHE library itself. We implement our evaluation algorithm on a high-end workstation with an Intel(R) Xeon(R) CPU E5-2690 v4 CPU (2.60GHz clock-frequency), 28 physical cores, and 128GB RAM. We now discuss different the three tiers of our proposed FHEDA execution flow.

6.2 Core Layer

This layer serves as a foundation layer for the block layer which incorporates depth and reduced-bootstrapping versions of FHE-friendly circuits. As observed in Section 3.1, by introducing diverse gate types (beyond the classical Bristol Format's 2-input AND and XOR gates) and incorporating multi-input gates, we can achieve substantial enhancements in evaluation time and depth for Boolean circuits. Therefore, this layer deals with constructions of standard cell library sets that will be used to generate and evaluate FHE-friendly circuit blocks at the next layer by FHEDA flow. Primarily, we construct $\mathsf{Lib-Set1}$, which includes only 2-input XOR and AND gates along with NOT gates, aligning precisely with the Bristol Format and serves as our baseline library set. To incorporate multi-input gates we construct $\mathsf{Lib-Set2}$ that consist of both 2 and 3-input XOR and AND gates with unary NOT gates. Using $\mathsf{Lib-Set2}$ for Boolean representation leads to a notable reduction in the number of binary gates in circuits. This improvement is due to the fact that each 3-input gate requires approximately the same evaluation time (bootstrapping reduction) as compared to 2-input gates (as discussed in Section 4). This results in significant time savings and improved depth efficiency during the homomorphic evaluation process. $\mathsf{Lib-Set3}$ incorporates all fundamental 2-input universal gates, including NAND and NOR, alongside XOR, AND, OR, and NOT gates. By combining $\mathsf{Lib-Set2}$ with $\mathsf{Lib-Set3}$, we establish $\mathsf{Lib-Set4}$, harnessing the benefits of both sets to achieve an optimal reduction in homomorphic computation time. We present these library sets below for readers' convenience,

$$\begin{aligned}
\mathsf{Lib-Set1} &= \{\mathsf{XOR}_2, \mathsf{AND}_2, \mathsf{NOT}\},\\
\mathsf{Lib-Set2} &= \{\mathsf{XOR}_3, \mathsf{AND}_3\} \bigcup \mathsf{Lib-Set1},\\
\mathsf{Lib-Set3} &= \{\mathsf{NAND}_2, \mathsf{NOR}_2, \mathsf{OR}_2\} \bigcup \mathsf{Lib-Set1},\\
\mathsf{Lib-Set4} &= \{\mathsf{XOR}_3, \mathsf{AND}_3\} \bigcup \mathsf{Lib-Set3}.
\end{aligned}$$

where GATE_p is a Boolean gate GATE with p inputs.

6.3 Block Layer

By skillfully designing personalized library sets, as demonstrated in the previous section, the synthesis tool incorporated into our FHEDA flow employs these custom sets to synthesize and generate FHE-friendly netlists of fundamental building blocks in this layer. This approach allows for the creation of numerous critical functionalities, such as carry select adder, multiplexer, comparator circuit, and S-Boxes with dimensions 4×4 and 5×5 (utilized in AES, PRESENT, and Keccak ciphers). The computation time and circuit depth achieved by employing different library sets for these block-level circuits are presented in Table 4. One can observe that the percentage reduction in circuit depth and FHE-evaluation time of these fundamental blocks under various library sets is in the range from $30-40\%$ and $13-50\%$, respectively. An interesting observation to note here is that even though the overall percentage reduction values are significant, they are not uniform. For instance, let's consider the PRESENT S-Box, which exhibits a time reduction of approximately 50.18%, while the AES S-Box shows only a 34.18% reduction. Similarly, for Mux2:1 the percentage gain in circuit-depth is 40%, while evaluation time reduction is just 15%. The reasons for these differences can be explained as follows:

Listings 2 and 3 displays a synthesized netlist of a Mux2:1 schematic as synthesized by Cadence. The circuit depicted in Listing 2 is generated using Lib − Set2, while the one in Listing 3 is generated using Lib − Set4. Even though Lib − Set2 and Lib − Set4 comprises 3-input gates, however, the synthesis tool could not perform 3-input gate replacement in Mux2:1, resulting in similar evaluation time. One can also notice a higher reduction in evaluation time in the case of PRESENT S-Box than AES S-Box. The PRESENT S-Box is represented using gates from Lib − Set1, encompassing both XOR and AND gates, along with NOT gates. When converting the Boolean representations from Lib − Set1 to Lib − Set2 (comprising 2 and 3-input XOR and AND gates), some of the 2-input gates in both XOR and AND are merged into 3-input gates, resulting in a significant reduction of gate counts by approximately 25% for AND and 14% for XOR. On the other hand, the AES S-Box only employs AND and NOT gates and lacks XOR gates. As a consequence, the gate counts are reduced by approximately 22% solely for the AND gates when converting the Boolean representations from Lib − Set1 to Lib − Set2. Therefore, the PRESENT S-Box achieves a higher cumulative reduction compared to the AES S-Box. In Appendix .3 we provide a brief comparison between different block layer circuits with respect to the homomorphic evaluation time and circuit depth using our four library sets.

Listing 2: Synthesis of Mux2:1 using Lib − Set2

```
module multiplexer_2x1
(input A, B, S, output Y);
   INV n1 (nS, S);
   NAND2 a1 (Y1, nS, A);
   NAND2 a2 (Y2, S, B);
   NAND2 o1 (Y, Y1, Y2);
endmodule
```

Listing 3: Synthesis of Mux2:1 using Lib − Set4

```
module multiplexer_2x1
(input A, B, S, output Y);
   INV n1 (nS, S);
   AND2 a1 (Y1, nS, A);
   INV aa1 (nY1, Y1);
   AND2 a2 (Y2, S, B);
   INV aa2 (nY2, Y2);
   AND2 o1 (nY, nY1, nY2);
   INV  o2 (Y, nY);
endmodule
```

Furthermore, to capitalize on the advantages achieved in these circuits, these fundamental components are assembled and combined to create advanced functionality blocks in the FHE domain. This takes place at the main level of the FHEDA flow, as elaborated in the following section.

6.4 Main Layer

In order to make the most of the benefits obtained from the functional circuit blocks obtained as described in the last section, these basic elements are brought together and integrated to form sophisticated FHE functional units comprising of Privacy Preserving Neural Networks, Private Set Intersection (PSI), etc. This process occurs at the main layer of our proposed FHEDA flow, as detailed in this section. We generated the following FHE-units through our FHEDA flow:

Privacy Preserving Neural Networks: In this work, we use a Convolutional Neural Network (CNN), in which the convolutional layer performs a convolution operation with the encrypted inputs and weight matrix (known as kernel/filter). A convolution operation consists of element-wise "multiplication" between a part of the input and the weights and followed by an "addition" operation (for more details of the convolution operation we refer to Figure. 8 in Appendix .1). This "multiplication" is implemented using a k-bit multiplier ($k = 8$ in our case) and the encrypted "addition" is implemented using a k'-bit Full Adder (FA) ($k' = 8$). This fundamental circuits are presented at the block layer (Table 4) with depth and evaluation time reduction of 48% and 13% respectively. The next significant component is the activation function that determines the neuron's output. In this work, we consider the rectified linear unit (ReLU) denoted as $ReLU(y) = max(0, y)$, with y as the input. The output of this function is characterized by the maximum value between encrypted 0 and encrypted y and realized using the Mux2:1. Note that, no gain in circuit-depth or evaluation time is observed, because Mux2:1 operation cannot be represented using 3-input Boolean gates as shown in Listing 2 and 3. Next, in the pooling layer, the MaxPool circuit is implemented using an encrypted multiplexer that chooses the maximum of the two inputs using a comparator circuit. We have obtained efficient Mux and comparator at the block level of FHEDA flow and that is utilized to generate a MaxPool circuit with 53% and 21% reduction in circuit depth and evaluation time.

Symmetric Key Ciphers: Using symmetric-key ciphers a party can encrypt the inputs using the cipher's key and at the server's end a homomorphic decryption operation is performed to convert the encryption into a FHE ciphertext. This helps to reduce the bandwidth of communication between the party and the computing server (supported by Hybrid homomorphic encryption). This emerges in applications like private set intersection [10] and encrypted databases [11]. In this work, we considered AES [12], PRESENT [13], and LowMC [11] ciphers as our benchmark netlists. The AES implementation from Archer et al. [14] was employed and processed through our FHEDA flow. By utilizing $\mathsf{Lib-Set2}$, we managed to achieve an improvement of up to 19% in the evaluation time for AES. In the case of LowMC, $\mathsf{Lib-Set3}$ resulted in the maximum depth improvement, while $\mathsf{Lib-Set4}$ led to better evaluation time improvement. Finally, for PRESENT, $\mathsf{Lib-Set4}$ produced the best circuit in terms of both depth and evaluation time enhancement.

Private Set Intersection (PSI): PSI allows two parties to securely compute a function based on the common elements in their respective private input sets. Each set is represented as a binary vector, and the intersection of these sets is determined by performing a bit-wise AND operation on the sets provided by both parties. Since this circuit mainly involves an AND gate chain, we observed an improvement in evaluation time when using $\mathsf{Lib-Set4}$. However, there is no improvement in circuit depth as synthesis won't replace the pure AND chain with other gate types, as doing so might introduce an increase in depth. In the next section, we present a comprehensive implementation of a Convolutional Neural Network (CNN) using FHEDA flow.

7 Homomorphic Inference of Convolution Neural Network

In this section, we describe an end-to-end implementation for the homomorphic evaluation of a convolutional neural network (CNN) on encrypted inputs. A CNN basically consists of a series of convolution operations [15] followed by a non-linear activation function [16] and pooling layers [17]. The output of the convolution layers is finally fed to a series of "fully connected" layers (FC), which are also followed by an activation function. A detailed description of the architecture for different layers of a CNN is presented in Appendix .1. In our present work, we have used Lenet-5 [18], a simple convolution neural network with only 5-network layers (3 convolution and 2 fully connected layers). The Lenet-5 architecture uses the convolution operation with a filter/kernel size of $(5, 5)$ and stride of 1, Average pooling layer [19] with a window size of $(2, 2)$ and stride of 2, and the Tanh [20] activation function except for the last layer which uses a Softmax activation [21]. But, in this work, we used a simple modified version of the Lenet-5 network, where all other parameters remain the same except for activation and the pooling layers. We chose Maxpool layer [22] and ReLU activation [23] in our present CNN architecture. In Table 5 (Appendix .1) we present different layers of our CNN model with input/output specification. Below is our CNN architecture in short: $(\mathsf{CONV} + \mathsf{ReLU} + \mathsf{MaxPool}) \rightarrow (\mathsf{CONV} + \mathsf{ReLU} + \mathsf{MaxPool}) \rightarrow (\mathsf{CONV} + \mathsf{ReLU}) \rightarrow (\mathsf{FC}_1 + \mathsf{ReLU}) \rightarrow (\mathsf{FC}_2 + \mathsf{ReLU})$. Here, CONV denotes a convolution layer with filter/kernel size of $(5, 5)$ and stride 1, ReLU denotes the Rectified linear unit activation layer, $\mathsf{MaxPool}$ denotes a max-pooling layer with window size $(2, 2)$ and stride 2, FC_1 and FC_2 denote fully connected layers with weight matrices of dimensions $(120, 84)$ and $(84, 10)$, respectively. In our current setup, we used the MNIST image (grayscale) dataset with dimensions of $(32, 32)$ for every image sample. In the homomorphic inference of CNN, we encrypt both the inputs and the network weights using TFHE secret key **sk**. We then outsource these encrypted values and the TFHE evaluation key **ek** to a high-end computing platform for performing the homomorphic evaluation. Due to the page limitations, we refer to Appendix .1 for a detailed discussion of our CNN implementation, evaluation approach that uses process-level threads for parallel computation and homomorphic evaluation results in Table 5 using our four different library sets. Note that, the total evaluation time of our CNN architecture for each library set is mentioned in the last row of Table 5, where we achieve around 8.67 hrs. using $\mathsf{Lib} - \mathsf{Set1}$ and get the optimal homomorphic evaluation time of 6.66 hrs. (approx.) for $\mathsf{Lib} - \mathsf{Set4}$. Therefore, we accomplish approximately a 23.14% reduction in the total homomorphic evaluation time for our CNN model.

8 Conclusion

The growing focus on data privacy has sparked significant interest in Fully Homomorphic Encryption (FHE) from both industry and academia. FHE enhances data security by introducing noise, but this can lead to increased computation time and costs. To address these issues we developed an EDA framework FHEDA for generating efficient FHE-friendly Boolean circuits compatible with Torus-FHE platform. Our proposed framework reduces the number of bootstrapping operations in Boolean circuits, thereby resulting in up to 50% faster homomorphic computation when compared to baseline netlists (comprising of 2-input AND, XOR gates). To validate our approach, we tested the proposed EDA flow on various benchmarks, including privacy-preserving machine learning blocks, symmetric key block ciphers like AES and LowMC, and oblivious inference on neural networks like CNN. In the future direction, we would like to extend the work further to develop an efficient FHE accelerator.

References

[1] Z. Brakerski, "Fully homomorphic encryption without modulus switching from classical gapsvp," in *Annual Cryptology Conference*. Springer, 2012, pp. 868–886.

[2] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without bootstrapping," *ACM Transactions on Computation Theory (TOCT)*, vol. 6, no. 3, pp. 1–36, 2014.

[3] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "Tfhe: fast fully homomorphic encryption over the torus," *Journal of Cryptology*, vol. 33, no. 1, pp. 34–91, 2020.

[4] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *Advances in Cryptology–ASIACRYPT 2017: 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I 23*. Springer, 2017, pp. 409–437.

[5] C. Gentry, *A fully homomorphic encryption scheme*. Stanford university, 2009.

[6] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," *Cryptology ePrint Archive*, 2012.

[7] L. Ducas and D. Micciancio, "Fhew: bootstrapping homomorphic encryption in less than a second," in *Annual international conference on the theory and applications of cryptographic techniques*. Springer, 2015, pp. 617–640.

[8] P. Mukherjee and D. Wichs, "Two round multiparty computation via multi-key fhe," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2016, pp. 735–763.

[9] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "TFHE: Fast fully homomorphic encryption library," August 2016, https://tfhe.github.io/tfhe/.

[10] D. Kales, C. Rechberger, T. Schneider, M. Senker, and C. Weinert, "Mobile private contact discovery at scale," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1447–1464. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/kales

[11] M. R. Albrecht, C. Rechberger, T. Schneider, T. Tiessen, and M. Zohner, "Ciphers for mpc and fhe," in *Advances in Cryptology – EUROCRYPT 2015*, E. Oswald and M. Fischlin, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 430–454.

[12] V. Rijmen and J. Daemen, "Advanced encryption standard," *Proceedings of federal information processing standards publications, national institute of standards and technology*, vol. 19, p. 22, 2001.

[13] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin, and C. Vikkelsoe, “Present: An ultra-lightweight block cipher,” in *Cryptographic Hardware and Embedded Systems - CHES 2007*, P. Paillier and I. Verbauwhede, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 450–466.

[14] D. Archer, V. A. Abril, S. Lu, P. Maene, N. Mertens, D. Sijacic, and N. Smart, “Bristol fashion mpc circuits,” 2021.

[15] J. Gu, Z. Wang, J. Kuen, L. Ma, A. Shahroudy, B. Shuai, T. Liu, X. Wang, G. Wang, J. Cai *et al.*, “Recent advances in convolutional neural networks,” *Pattern recognition*, vol. 77, pp. 354–377, 2018.

[16] W. Hao, W. Yizhou, L. Yaqin, and S. Zhili, “The role of activation function in cnn,” in *2020 2nd International Conference on Information Technology and Computer Application (ITCA)*. IEEE, 2020, pp. 429–432.

[17] M. Sun, Z. Song, X. Jiang, J. Pan, and Y. Pang, “Learning pooling for convolutional neural network,” *Neurocomputing*, vol. 224, pp. 96–104, 2017.

[18] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[19] J. Kakarla, B. V. Isunuri, K. S. Doppalapudi, and K. S. R. Bylapudi, “Three-class classification of brain magnetic resonance images using average-pooling convolutional neural network,” *International Journal of Imaging Systems and Technology*, vol. 31, no. 3, pp. 1731–1740, 2021.

[20] L. Alzubaidi, J. Zhang, A. J. Humaidi, A. Al-Dujaili, Y. Duan, O. Al-Shamma, J. Santamaría, M. A. Fadhel, M. Al-Amidie, and L. Farhan, “Review of deep learning: Concepts, cnn architectures, challenges, applications, future directions,” *Journal of big Data*, vol. 8, pp. 1–74, 2021.

[21] I. Kouretas and V. Paliouras, “Simplified hardware implementation of the softmax activation function,” in *2019 8th international conference on modern circuits and systems technologies (MOCAST)*. IEEE, 2019, pp. 1–4.

[22] N. Murray and F. Perronnin, “Generalized max pooling,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2014, pp. 2473–2480.

[23] G. Lin and W. Shen, “Research on convolutional neural network based on improved relu piecewise activation function,” *Procedia computer science*, vol. 131, pp. 977–984, 2018.

[24] “Microsoft SEAL (release 4.1),” https://github.com/Microsoft/SEAL, Jan. 2023, microsoft Research, Redmond, WA.

[25] S. Halevi and V. Shoup, “Helib-an implementation of homomorphic encryption,” *Cryptology ePrint Archive, Report 2014/039*, 2014.

[26] W. Dai and B. Sunar, “cuhe: A homomorphic encryption accelerator library,” in *Cryptography and Information Security in the Balkans: Second International Conference, BalkanCryptSec 2015, Koper, Slovenia, September 3-4, 2015, Revised Selected Papers 2*. Springer, 2016, pp. 169–186.

[27] A. Al Badawi, B. Veeravalli, C. F. Mun, and K. M. M. Aung, "High-performance fv somewhat homomorphic encryption on gpus: An implementation using cuda," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 70–95, 2018.

[28] W. Jung, S. Kim, J. H. Ahn, J. H. Cheon, and Y. Lee, "Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with gpus," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 114–148, 2021.

[29] Ö. Özerk, C. Elgezen, A. C. Mert, E. Öztürk, and E. Savaş, "Efficient number theoretic transform implementation on gpu for homomorphic encryption," *The Journal of Supercomputing*, vol. 78, no. 2, pp. 2840–2872, 2022.

[30] F. Boemer, S. Kim, G. Seifu, F. DM de Souza, and V. Gopal, "Intel hexl: Accelerating homomorphic encryption with intel avx512-ifma52," in *Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, 2021, pp. 57–62.

[31] D. B. Cousins, J. Golusky, K. Rohloff, and D. Sumorok, "An fpga co-processor implementation of homomorphic encryption," in *2014 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2014, pp. 1–6.

[32] N. Samardzic, A. Feldmann, A. Krastev, S. Devadas, R. Dreslinski, C. Peikert, and D. Sanchez, "F1: A fast and programmable accelerator for fully homomorphic encryption," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 238–252.

[33] D. B. Cousins, K. Rohloff, and D. Sumorok, "Designing an fpga-accelerated homomorphic encryption co-processor," *IEEE Transactions on Emerging Topics in Computing*, vol. 5, no. 2, pp. 193–206, 2016.

[34] S. S. Roy, F. Turan, K. Jarvinen, F. Vercauteren, and I. Verbauwhede, "Fpga-based high-performance parallel architecture for homomorphic computing on encrypted data," in *2019 IEEE International symposium on high performance computer architecture (HPCA)*. IEEE, 2019, pp. 387–398.

[35] M. S. Riazi, K. Laine, B. Pelton, and W. Dai, "Heax: An architecture for computing on encrypted data," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 1295–1309.

[36] F. Turan, S. S. Roy, and I. Verbauwhede, "Heaws: An accelerator for homomorphic encryption on the amazon aws fpga," *IEEE Transactions on Computers*, vol. 69, no. 8, pp. 1185–1196, 2020.

[37] J. Zhang, X. Cheng, L. Yang, J. Hu, X. Liu, and K. Chen, "Sok: Fully homomorphic encryption accelerators," *arXiv preprint arXiv:2212.01713*, 2022.

[38] M. Van Beirendonck, J.-P. D'Anvers, and I. Verbauwhede, "Fpt: a fixed-point accelerator for torus fully homomorphic encryption," *arXiv preprint arXiv:2211.13696*, 2022.

[39] L. Jiang, Q. Lou, and N. Joshi, "Matcha: A fast and energy-efficient accelerator for fully homomorphic encryption over the torus," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, 2022, pp. 235–240.

[40] A. Patra, T. Schneider, A. Suresh, and H. Yalame, "Syncirc: Efficient synthesis of depth-optimized circuits for secure computation," in *2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2021, pp. 147–157.

[41] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan, "{GAZELLE}: A low latency framework for secure neural network inference," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 1651–1669.

[42] B. Reagen, W.-S. Choi, Y. Ko, V. T. Lee, H.-H. S. Lee, G.-Y. Wei, and D. Brooks, "Cheetah: Optimizing and accelerating homomorphic encryption for private inference," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 26–39.

[43] S. Kim, J. Kim, M. J. Kim, W. Jung, J. Kim, M. Rhu, and J. H. Ahn, "Bts: An accelerator for bootstrappable fully homomorphic encryption," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 711–725.

.1 Architecture and Implementation of CNN

Convolutional Neural Network (CNN) is a type of Neural Network which has redefined various fields like image classification, face recognition, object detection, etc. A CNN consists of multiple convolutional layers, activation layers, and pooling layers followed by a series of fully connected (FC) layers.

.1.1 Convolutional Layer

Architecture. Convolutional layers are the core building blocks of a CNN model. A Convolutional layer takes a 2-D image (or, feature map) as input and performs the Convolutional operation using a kernel/filter of pre-defined size. The kernel basically performs shifts based

Table 5: Lenet-5 Architecture Evaluation Time Reduction of FHE-circuits when Compared to Baseline Library. (**-Baseline Library))

Layer	Layer Type	Specification	Execution Time Lib-Set 1** (in hrs.)	Execution Time Lib-Set 2 (in hrs.)	Execution Time Lib-Set 3 (in hrs.)	Execution Time Lib-Set 4 (in hrs.)	Efficiency (%)
1	CONV	$32 \times 32 \rightarrow 28 \times 28$	1.65	1.36	1.56	1.27	23.03
	ReLU	$28 \times 28 \rightarrow 28 \times 28$	0.0036	0.0036	0.0036	0.0036	0.00
	MaxPool	$28 \times 28 \rightarrow 14 \times 14$	0.0169	0.0135	0.0166	0.0133	21.30
2	CONV	$14 \times 14 \rightarrow 10 \times 10$	4.40	3.63	4.16	3.39	22.95
	ReLU	$10 \times 10 \rightarrow 10 \times 10$	0.0096	0.0096	0.0096	0.0096	0.00
	MaxPool	$10 \times 10 \rightarrow 5 \times 5$	0.045	0.037	0.044	0.035	22.22
3	CONV	$5 \times 5 \rightarrow 1 \times 1$	0.275	0.227	0.26	0.212	22.90
	ReLU	$1 \times 1 \rightarrow 1 \times 1$	0.0006	0.0006	0.0006	0.0006	0.00
4	FC_1	$120 \rightarrow 84$	1.34	1.10	1.26	1.02	23.88
	ReLU	$84 \rightarrow 84$	0.0006	0.0006	0.0006	0.0006	0.00
5	FC_2	$84 \rightarrow 10$	0.93	0.77	0.88	0.71	23.65
	ReLU	$10 \rightarrow 10$	0.0006	0.0006	0.0006	0.0006	0.00
	Total	$32 \times 32 \rightarrow 10$	8.672	7.152	8.196	6.665	23.14

on the value of the stride across the input/feature map. Each sifting operation of the kernel results in an element-wise multiplication of the values in the kernel with the elements in the input image and accumulates these results using addition operation (Figure. 8 shows a pictorial representation of the Convolutional operation). Assume, an input of dimension (N, N) is fed into a Convolutional layer with kernel/filter size (f, f) and stride s. It outputs a feature map (known as "convolved feature map") of dimension $\left(\frac{N-f+1}{s}, \frac{N-f+1}{s}\right)$. As already mentioned, we used a simple modified Lenet-5 [18] CNN architecture as described in Table 5. Our CNN model uses a Convolutional kernel of size 5×5 and stride 1. The first Convolutional layer takes a $(32, 32)$ image as input and outputs a $(28, 28)$ convolved feature map, the other Convolutional layers perform a similar operation using the feature maps from the previous layers as input. Now, we briefly discuss our implementation approach of a Convolutional layer.

Implementation. In this work, we present a CPU-based implementation of CNN[9], in which we adopt parallelism using process-level threads. There are several approaches to perform parallel implementation using threads, but in our current work, we initialize the number of threads equal to the number of elements in the output feature map. Let us observe this scenario using Figure. 8, in this figure a $(32, 32)$ image is convolved using a kernel of size $(5, 5)$, consequently the output feature map will have the dimension of $(28, 28)$. For each element in the output feature map, we initialize a thread that will perform the Convolutional operation, i.e., $(5 \times 5) = 25$ multiplications followed by 24 addition operations. Hence, a total of $(28 \times 28) = 784$ threads will be initialized and thus we are able to perform 784 Convolutional operations in parallel. Now, if a Convolutional operation has c output channels, we perform a serialized computation for these c channels (e.g., the first Convolutional layer in our CNN model has $c = 6$).

.1.2 Rectified Linear Unit

Architecture. Every Convolutional layer in our CNN architecture is followed by a Rectified Linear Unit (ReLU) activation layer, which maps every element in the convolved feature map to a new feature map of the same dimension. A ReLU function is defined as $\mathsf{ReLU}(x) = max(0, x)$, where $max(a, b)$ is denoted as finding the maximum between a, b.

Implementation. The ReLU activation function is also implemented using thread-level parallelism. In this approach, the ReLU activation function on each element from the convolved feature map is performed by a thread. Hence, applying ReLU activation on a feature map of dimension $(28, 28)$ requires a total of $(28 \times 28) = 784$ threads, as described in Figure. 9.

.1.3 Maxpool Layer

Architecture. After performing the ReLU activation function, our CNN model performs a max-pooling operation with a window size of $(2, 2)$ and stride of 2. Assume, a $(28, 28)$ image is fed into a Maxpool layer, it will output a feature map of dimension $(14, 14)$. The primary work of a Maxpool layer is to extract the relevant and important features from a

[9]As we are evaluating a CNN on encrypted inputs and weights, it is not trivial to perform the Convolutional operation on TFHE ciphertexts on a GPU-based platform.

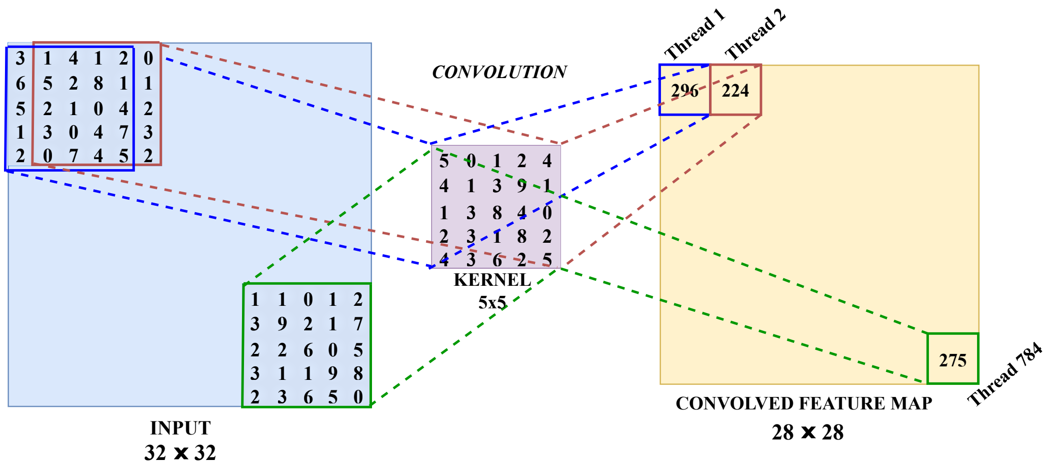


Figure 8: Convolutional Layer

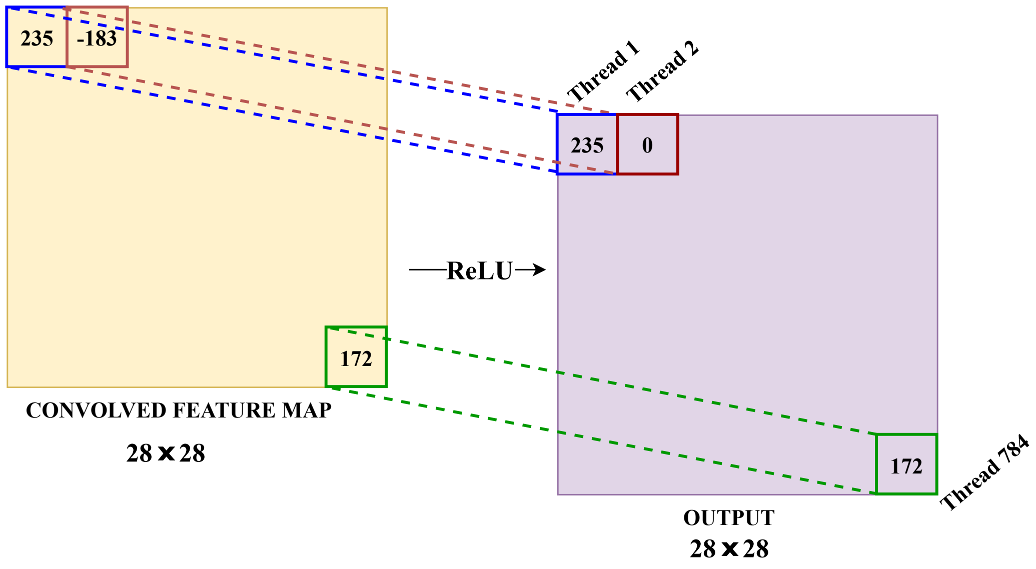


Figure 9: Rectified Linear Unit

feature map, thus reducing the feature map size which helps in faster and precise learning of the CNN.

Implementation. The Maxpool layer in a CNN model is parallelized similarly to a Convolutional layer as shown in Figure. 10. For each element in the output feature map of a Maxpool, we initialize a process-level thread, which means for an output feature map of dimension $(14, 14)$ we initialize a total of $(14 \times 14) = 196$ threads. Each thread is responsible for computing the maximum of the elements in the input feature map that lies within the boundaries of the Maxpool window (which is $(2, 2)$ in our case). Similarly, for c number of channels in the input feature map, we perform c sequential max-pooling operations.

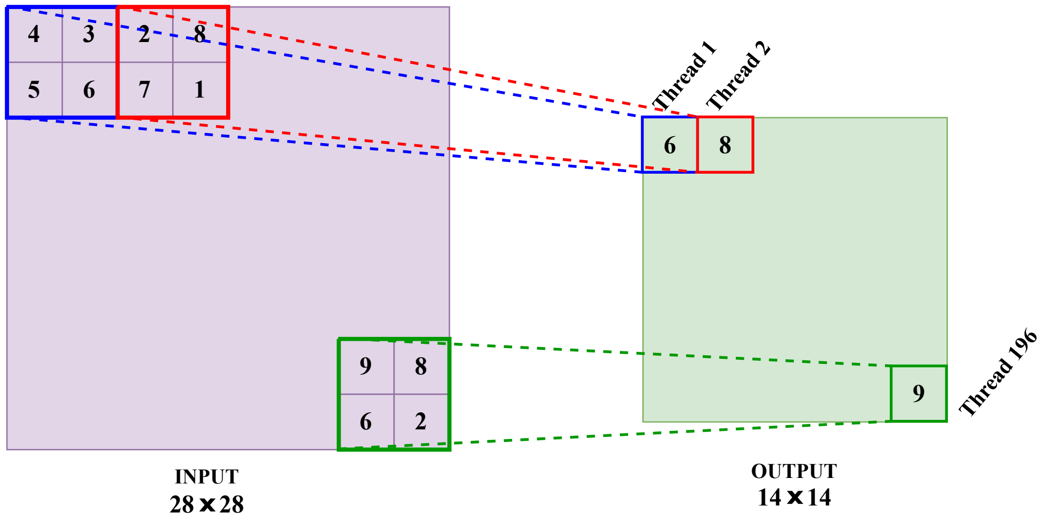


Figure 10: Maxpool Layer

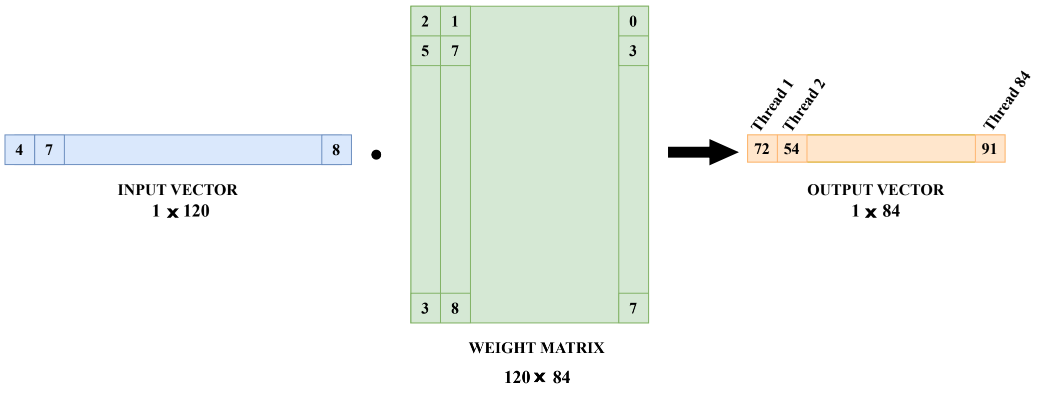


Figure 11: Fully Connected Layer

.1.4 Fully Connected Layer

Architecture. A fully connected (FC) layer is an essential part of a CNN architecture and generally presents in the last layers of a CNN model. An FC layer consists of a 2-D matrix and a vector of random elements, known as weights and biases respectively. These weights and biases are optimized during the training of CNN (along with the elements in the Convolutional kernel). In our CNN model, we have two FC layers; the first FC layer takes an input vector of dimension $(1, 120)$ and multiplies it with a weight matrix of dimension $(120, 84)$ and outputs a vector of dimension $(1, 84)$. The output feature vector of an FC

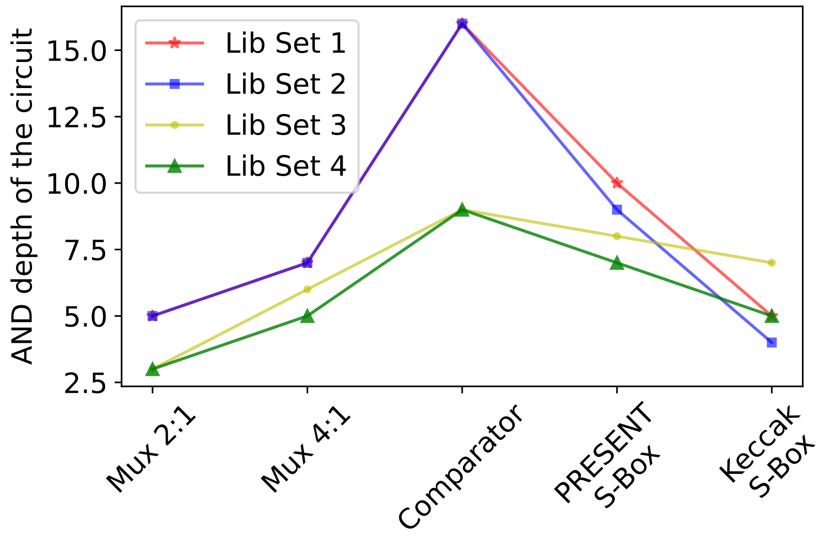


Figure 12: AND depth of block layer

layer is then followed by a ReLU activation function (except for the last layer that uses a Softmax activation). The final FC layer takes a vector of dimension $(1, 84)$ and outputs a vector of dimension $(1, 10)$, i.e., it uses a weight matrix of dimension $(84, 10)$, as specified in Table 5.

Implementation. For each element in the output vector, we initialized a thread and a total of 84 threads perform in parallel (see Figure. 11). Each thread is responsible for computing a vector dot product between the input vector and each column vector of the weight matrix. Similarly, for the final FC layer, we have 10 threads perform in parallel to produce the final output (after passing through the Softmax activation) of our CNN architecture.

Note. Here, we describe a different approach of performing parallel computation using process-level threads. Consider the scenario when after convolution/pooling operation the output feature map size if smaller in compared to the kernel/window size, i.e., $\left(\frac{N-f+1}{s} < f\right)$ or, $\left(\frac{N}{w} < w\right)$, for the input dimension (N, N), convolution filter size (f, f), pooling window size (w, w) and stride s. Then initializing threads according to the output map size like mentioned above will not be efficient. Rather, in the situation of this kind, we initialize a total of $(5 \times 5) = 25$ threads for performing convolution operation using kernel size $(5, 5)$. That means, 25 element-wise multiplications are getting performed in parallel; and followed by 24 additions, which can be done using a logarithmic depth adder circuit. A similar case happens for our MaxPool operation, i.e., we initialize $(2 \times 2) = 4$ threads to be executed in parallel. Thus, we can optimize of thread-based parallel implementation of CNN.

.2 Some Additional Related Works

EDA tools play a pivotal role in developing efficient and secure hardware implementation in the context of FHE and MPC, that form an integral part of the privacy preserving

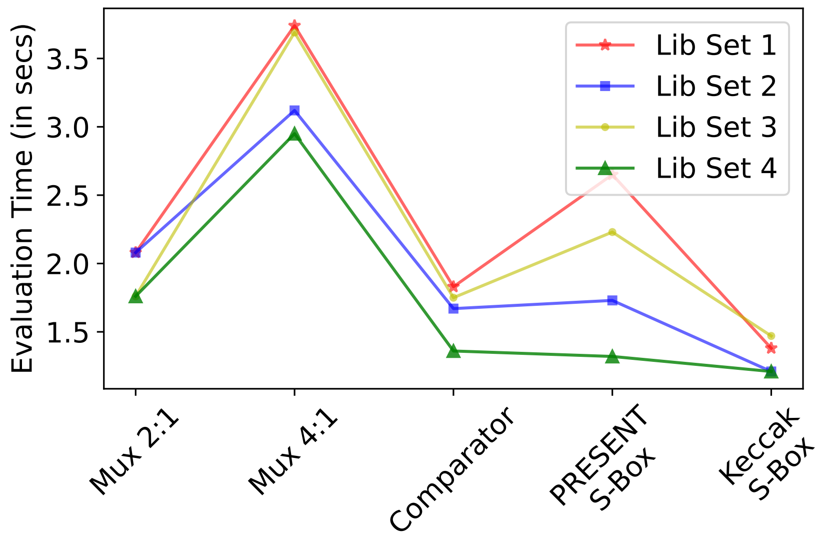


Figure 13: Evaluation time of block layer

computations. Adapting the hardware at a refined level facilitate detailed parallel processing and efficient resource utilization thereby yielding enhanced performance and energy efficiency. Multiple optimizations and acceleration strategies are being explored to handle the computational and memory requirements of FHE. In the realm of FHE computations on CPU, many software libraries such as SEAL [24], HELib[25], TFHE, PALISADE[10] accelerate the performance of different FHE schemes. Several research illustrated that GPU based implementations [26, 27, 28, 29] make use of inherent parallelism in FHE. Intel proposed Intel Homomorphic Encryption Acceleration Library (HEXL) [30] for fast number theoretic transform (NTT) operations. Several NTTs are inefficient on CPUs and GPUs, however, can be accelerated using specific functional units for which prior literature studies [31, 32, 33, 34, 35, 36] focus on Application Specific Integrated Circuit (ASIC) and Field Programmable Gate Array (FPGA) based accelerators. Existing literature suggests GPU-enabled TFHE libraries such as cuFHE, NuFHE [37]. The computations on encrypted AND gates on TFHE scheme take 13ms [3] on a CPU. However, these improvements are also slow and to mitigate the speed limitations, FPT [38], a Fixed-Point FPGA accelerator is proposed for TFHE which is compute-bound with 937× faster than CPU implementations and 2.5× faster than the prior FHE accelerator, MATCHA [39] by Jiang *et al.* and cuFHE. *SynCirc* [40], an efficient hardware synthesis framework is designed to generate multiplicative depth optimizations for secure MPC applications. Past studies have proposed ASIC accelerators that combine homomorphic encryption with MPC [41, 42] Cheetah [42] introduced algorithmic and hardware optimizations for HE DNN and uses MPC instead of bootstrapping for reducing the errors during the HE operation. F1 [32] is the first programmable FHE accelerator that has achieved ASIC-level performance and introduced effective design by accelerating *primitive* FHE scheme. BTS [43], a bootstrappable FHE accelerator achieved a speedup of 2237× in HE multiplication throughput in contrast to the state-of-the-art CPU implementations.

[10]https://gitlab.com/palisade/palisade-release

However, the existing literature does not offer any automated framework for synthesizing FHE-amenable circuits capable of performing homomorphic operations on cloud platforms.

.3 Comparison of Different Block layer Circuits

In this section, we present a brief discussion on the circuit depth and evaluation time reduction for a set of circuits mentioned in the block layer in Table 4. In Figure. 12 we present a comparison of multiple block layer circuits with respect to AND-depth vs. gate count of the block layer circuits. Similarly, Figure. 13 shows the variations of homomorphic evaluation time with respect to the number of gates present in the different block layer circuits.