

FHEDA: Efficient Circuit Synthesis with Reduced Bootstrapping for Torus FHE

Animesh Singh¹, Smita Das¹, Anirban Chakraborty¹, Rajat Sadhukhan¹,
Ayantika Chatterjee¹, and Debdeep Mukhopadhyay¹

¹IIT Kharagpur

June 18, 2024

Abstract

Fully Homomorphic Encryption (FHE) schemes are widely used cryptographic primitives for performing arbitrary computations on encrypted data. However, FHE incorporates a computationally intensive mechanism called *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 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 two consecutive Boolean gate evaluations over fresh encryptions require only one bootstrapping instead of two, based on appropriate parameter choices. 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) which 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 the 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 the pay-per-use model. While in the first case, a user makes use of the computational power of the cloud by outsourcing both model and data (in encrypted form) to the server¹, 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, which means it allows computations on a collection of ciphertexts for a set of plaintext m_1, \dots, m_ℓ and returns an encryption of a function/circuit evaluated on the input plaintexts $f(m_1, \dots, m_\ell)$, without the knowledge of secret key. The security of FHE schemes relies on hard problems such as Learning With Error (LWE) or its ring variant (RLWE). While the idea of Homomorphic Encryption (HE) has been around in the cryptography community since the 1970s, the significant impetus to the field of FHE came after the breakthrough of Gentry’s work [5] in 2009, which proposed a Lattice-based scheme allowing 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 second-generation FHE schemes, that could perform arbitrary homomorphic computations. The bootstrapping operation and the memory overhead for the bootstrapping key (encryption of the secret key) have unfortunately restricted the wide adoption of this promising primitive. 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.

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 operations, which operates after the homomorphic evaluation of every binary gate. 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

¹We use the terms cloud and server interchangeably.

on the cloud. In order to take advantage of the computational capabilities of FHE libraries and integrate them 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.
- We incorporate three novel optimizations in our framework to reduce the number of gate and bootstrapping operations. First, we show that using multiple fan-in gates 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 intersections. We could exhibit a percentage reduction in circuit depth and homomorphic evaluation time up to 50%. Finally, we describe an end-to-end implementation for the oblivious inference of a CNN on encrypted data using our FHEDA flow.

Comparison with existing works. To the best of our knowledge it’s the first work on developing EDA frameworks for generating efficient FHE circuits by reducing the total number of bootstrapping operations. Whereas, recent works [9, 10] have proposed synthesis flows related to FHE circuits considering bootstrapping time optimizations. A recently proposed FHE-specific circuit synthesis tool [9] performs amortized bootstrapping for FHEW-like FHE schemes. However, these flows are only applicable to the Field Programmable Gate Array (FPGA) domain as they are only capable of performing Look-Up-Table (LUT) optimizations. In contrast, our proposed flow is more generic and can work on gate-level circuits integrating three distinct optimizations (as elaborated later in the paper). As bootstrapping is the most resource-intensive operation, we present, both empirical and theoretical evidence demonstrating that two consecutive gates in TFHE can be computed without the need for bootstrapping leading to the efficient use of 3-input Boolean gates. However, some other works [11, 12, 13] propose circuit synthesis flows with the objective of optimizing Garbled and MPC circuits concerning latency, gate count, and area footprint. It may be emphasized that the optimization techniques adopted in these MPC and Garbled circuit-related approaches are not applicable to the FHE domain. For instance, Songhori et al.[11] propose Garbled circuits optimization with the objective of using more NAND gates to generate the latency-optimized circuit as it is preferred by classical flow over AND-XOR gates due to their lower area footprint and latency costs. In contrast, our proposed flow uses more AND-XOR gates during optimization to synthesize FHE-friendly circuits. Similarly, CryptoNAS [13] offers a ReLU network optimization strategy by reducing ReLU layers for latency optimization related to MPC problems. In contrast, our approach adopts

a bottom-up optimization method for FHE circuits, building FHE-friendly components from fundamental levels ensuring FHE-related constraints are met at the gate level in the design cycle process.

2 Related Works

Adapting the hardware at a refined level facilitates 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 [14], HELib[15], TFHE, PALISADE² accelerate the performance of different FHE schemes. Several research illustrated that GPU-based implementations [16, 17, 18, 19] make use of inherent parallelism in FHE. Intel proposed Intel Homomorphic Encryption Acceleration Library (HEXL) [20] 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 [21, 22, 23, 24, 25, 26] 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 [27]. The computations on encrypted AND gates on the TFHE scheme take 13ms [3] on a CPU. However, these improvements are also slow and to mitigate the speed limitations, FPT [28], a Fixed-Point FPGA accelerator is proposed for TFHE which is compute-bound with $937\times$ faster than CPU implementations and $2.5\times$ faster than the prior FHE accelerator, MATCHA [29] by Jiang et al. and cuFHE. *SynCirc* [30], 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 [31, 32]. Cheetah [32] introduced algorithmic and hardware optimizations for HE DNN and used MPC instead of bootstrapping to reduce the errors during the HE operation. F1 [22] is the first programmable FHE accelerator that has achieved ASIC-level performance and introduced effective design by accelerating *primitive* FHE schemes. BTS [33], a bootstrappable FHE accelerator achieved a speedup of $2237\times$ in HE multiplication throughput in contrast to the state-of-the-art CPU implementations. However, the existing literature does not offer any automated framework for synthesizing FHE-amenable circuits.

3 Preliminaries and Background

3.1 Notations and Mathematical Background

We use \mathbb{T} to denote the Torus (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

²<https://gitlab.com/palisade/palisade-release>

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.³

3.2 Fully Homomorphic Encryption

Here we briefly discuss the definition of FHE and its functionality. For example, in an additive homomorphic encryption scheme, $\text{FHE.Dec}(\text{FHE.Enc}(m_0+m_1)) = \text{FHE.Dec}(\text{FHE.Enc}(m_0)) + \text{FHE.Dec}(\text{FHE.Enc}(m_1))$. 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 called bootstrapping that reduce the noise growth without altering the result.

3.3 Torus FHE (TFHE)

Here we provide a formal definition of TFHE scheme [3], which is the underlying encryption scheme in the present work.

Definition 1 (Torus FHE) *TFHE scheme is a tuple of five PPT (Probabilistic Polynomial Time) algorithms (TFHE.Setup, TFHE.KeyGen, TFHE.Enc, TFHE.Dec, TFHE.Eval).*

- $\text{params} \leftarrow \text{TFHE.Setup}(1^\lambda)$: Given a security parameter λ , this function returns a public parameter params .
- $(\text{sk}, \text{pk}) \leftarrow \text{TFHE.KeyGen}(\text{params})$: It takes the public parameter params as input and generates a secret key sk and the corresponding public key pk .
- $\text{ct} \leftarrow \text{TFHE.Enc}(m, \text{pk})$: It takes a message $m \in \{0,1\}$ and the public key $\text{pk} = \langle \mathbf{a}, \text{sk} \rangle + e$ as input. Samples a random vector $\mathbf{a} \in \mathbb{Z}_q^n$ and a noise value $e \leftarrow \mathcal{G}_\alpha$ and computes $b = \sum_{\mathcal{S}} \text{pk} + \Delta(m)$, where, $\sum_{\mathcal{S}}$ denotes random subset sum over the set \mathcal{S} , $\Delta(m)$ be the encoding of the plaintext m and \mathcal{G}_α be a Gaussian distribution with standard deviation α . It finally returns a ciphertexts $\text{ct} = (b, \mathbf{a})$.
- $m \leftarrow \text{TFHE.Dec}(\text{ct}^*, \text{sk})$: Given an evaluated ciphertext $\text{ct}^* = (b^*, \mathbf{a}^*) \in \mathbb{T}^{n+1}$ and the secret key sk , returns a message bit $m^* \in \{0,1\}$ which minimizes $|b^* - \langle \mathbf{a}^*, \text{sk} \rangle - \Delta(m^*)|$.
- $\text{ct}^* \leftarrow \text{TFHE.Eval}(\{\text{ct}_i\}_{i \in \ell}, \text{pk}, \mathcal{C})$: It takes a set of ciphertexts $\{\text{ct}_1, \dots, \text{ct}_\ell\}$, the public key pk and a circuit $\mathcal{C} : \{0,1\}^\ell \rightarrow \{0,1\}^{\ell'}$. Returns an evaluated ciphertext ct^* which is an encryption of $m^* = \mathcal{C}(m_1, \dots, m_\ell)$ under the secret key sk , where m_l is the underlying plaintext of ct_l for all $l \in \ell$.

³A function $f : \mathbb{N} \rightarrow \mathbb{N}$ is said to be negligible in λ if for every positive polynomial p , $f(\lambda) < 1/p(\lambda)$ when λ is sufficiently large.

3.4 Security Definition of TFHE Scheme

We now discuss the formal security definition of our underlying encryption scheme TFHE [3].

Definition 2 (IND-Security) We define $G_{\text{TFHE}, C_h, \mathcal{A}_d}(1^\lambda)$ a game between a PPT challenger C_h and a PPT adversary \mathcal{A}_d .

$G_{\text{TFHE}, C_h, \mathcal{A}_d}(1^\lambda)$:

Initialization Phase.

1 C_h samples public key pk and secret key sk and the public key pk is sent to \mathcal{A}_d .

Query Phase. In the query phase \mathcal{A}_d may generate polynomially many queries. Each of the queries consists of the following two steps:

2 \mathcal{A}_d sends ℓ plaintexts $\{m_0, \dots, m_{\ell-1}\} \in \mathcal{M}$ to C_h . C_h encrypts each m_i to generate ciphertexts $\text{ct}_i = \text{TFHE.Enc}(m_i, \text{sk})$. \mathcal{A}_d is handed over $\{\text{ct}_i\}_{i \in [\ell]}$.

3 \mathcal{A}_d sends a circuit $C : \{0, 1\}^\ell \rightarrow \{0, 1\}^*$. C_h computes the evaluated ciphertext as

$$\overline{\text{ct}} = \text{TFHE.Eval}(\{\text{ct}_i\}_{i \in \{0, 1\}}, C, \text{pk}).$$

\mathcal{A}_d is handed over $\overline{\text{ct}}$.

Challenge Phase.

4 In the challenge phase, \mathcal{A}_d sends $m'_0, m'_1 \in \mathcal{M}$ and $m'_i \notin \{m_0, \dots, m_{\ell-1}\}, \forall i \in \{0, 1\}$ of its choice. C_h then samples $r \leftarrow \{0, 1\}$ randomly and returns $\text{ct}'_r = \text{TFHE.Enc}(m'_r, \text{sk})$ to \mathcal{A}_d .

Output Phase.

5 Adversary \mathcal{A}_d eventually outputs a bit $r' \in \{0, 1\}$.

6 if $r' = r$, the game outputs 1, otherwise it outputs 0.

We say that the TFHE scheme is IND-secure if, for any security parameter $\lambda \in \mathbb{N}$, for any PPT adversary \mathcal{A}_d , letting $\gamma_\beta = \Pr[G_{\text{TFHE}, C_h, \mathcal{A}_d}(1^\lambda) = \beta]$, for $\beta \in \{0, 1\}$ (where the probability is over the random coins used by TFHE.KeyGen , TFHE.Enc , TFHE.Eval , and the adversary \mathcal{A}_d), we have $|\gamma_0 - \gamma_1| \leq \text{negl}(\lambda)$.

3.5 HDL Synthesis using Bristol Format

In this section, we present an essential step in the design flow of digital hardware circuits or, 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 Solution⁴. 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.

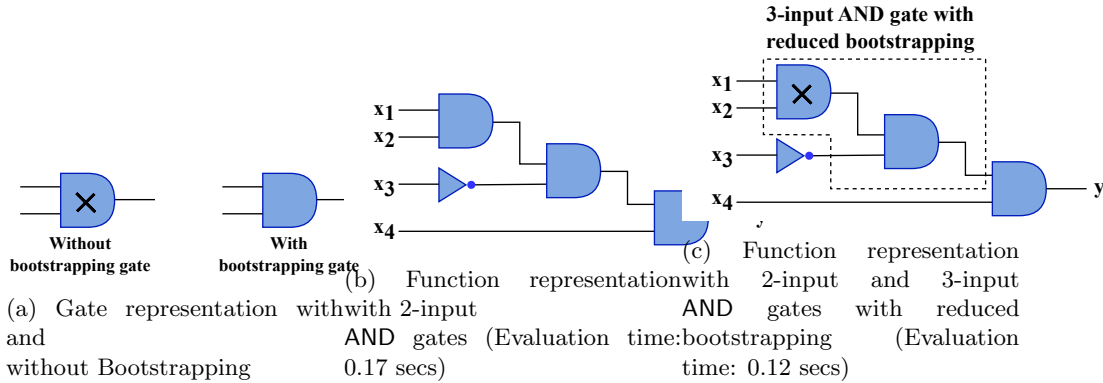


Figure 1: Notation of Bootstrapping and Comparison between 2-input and 3-input AND gates with reduced bootstrapping with a toy example

Now, we provide a brief description of the structure of Bristol formats⁵. Consider the Bristol format of 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’ refer 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’ denote 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 optimal Boolean representations of circuits, which is used for homomorphic evaluation using the TFHE scheme.

Listing 1: Bristol Format of Half Adder

```

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

```

⁴https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/synthesis/genus-synthesis-solution.html

⁵<https://homes.esat.kuleuven.be/~nsmart/MPC/old-circuits.html>

4 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 have 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. Compared to other FHE schemes like BGV/BFV, TFHE provides much faster bootstrapping for binary gate evaluation⁶. As mentioned earlier, our present work is to generate an optimal Boolean representation of circuits by introducing a few optimization techniques, which are currently focused on the TFHE scheme.

Due to the notion of the gate-bootstrapping technique in the TFHE scheme, one can evaluate a binary gate followed by a bootstrapping operation. However, bootstrapping is essentially homomorphic decryption, which means only one binary gate evaluation can be performed without bootstrapping (refer to Section 5.1 for more details). This means that evaluating one binary gate without bootstrapping will result in correct decryption, but the output will be a noisy ciphertext, which when fed into another binary gate may result in incorrect decryption. This happens due to the noise growth after the second binary gate, that might exceed the threshold of correct decryption. However, in this work, we propose 3-input gates instead of 2-input (or, binary) gates, where a 3-input gate requires only one bootstrapping. While in the original TFHE scheme, to evaluate a 3-input Boolean gate, two binary gates are required resulting in two bootstrapping. In Section 5.5 we discuss why it is sufficient to have only 3-input gates rather than n -input gates with $n \geq 4$. This helps us to construct optimal circuit descriptions for the TFHE scheme. In Figure. 1a, we denote the Boolean gate with “ \times ” symbol as “without bootstrapping”, whereas the other one as “with bootstrapping”. We now mention the optimization techniques for the homomorphic circuit evaluation, which is as follows:

- In Figure. 1c we can reduce one bootstrapping from the first gate given a chain of two gates connected in series (shown within dotted lines), resulting in a construction of 3-input gate. Thus a total of two bootstrappings are required for the entire circuit evaluation. We use this method of “reduced bootstrapping” to replace such a chain of Boolean gates. Consequently, if a Boolean circuit has n inputs, then with the *reduced bootstrapping* technique we can effectively reduce the total number of bootstrapping operations almost by $\lceil \frac{n}{2} \rceil$, as illustrated in the subsequent section.
- The NOT gates in TFHE require no bootstrapping, this motivates us to reduce other bootstrappable gates such as XORs, ANDs by 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

⁶Note 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 large number of ciphertexts.

replace the above algebraic operation with $a \wedge (\bar{b})$ and $b \wedge (\bar{a})$ respectively. Thus we reduce one bootstrappable 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”.

We use the above-mentioned techniques to further optimize the 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 explore some intriguing test cases that will play a pivotal role in optimizing circuit synthesis for our framework.

$$\begin{aligned} \text{Lib_Set1} &= \{\text{XOR, AND, NOT}\}, \\ \text{Lib_Set2} &= \{\text{NAND, NOR, OR}\} \cup \text{Lib_Set1}, \end{aligned} \tag{1}$$

4.1 Notable Observations on Evaluating 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 two binary gates can be evaluated without bootstrapping (ref. Figure. 1c) with suitable parameter choices (discussed in Section 5.3). This allows us to construct efficient 3-input Boolean gates requiring only one bootstrapping. 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 6). 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 a 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 reduced bootstrapping. Here, with “reduced” we refer that our 3-input gates require only one bootstrapping unlike in the original TFHE scheme. 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 our 3-input gates support the per gate bootstrapping notion like in the 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. 5), we explore this observation in greater detail and provide theoretical insights to substantiate the underlying reasons behind the above 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 *Inverter based optimization* 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 circuit 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. 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: $(\bar{x}_0 \wedge \bar{x}_1) \oplus (x_2 \bar{\wedge} x_3)$. 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 FHE domain.

5 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

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

Expt.	$\tau(\text{sec.}), \delta$		% Gain (τ, δ)	Inference
	Initial	Final		
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

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.

5.1 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

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

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 $\|e_1\|_\infty, \|e_2\|_\infty < \frac{1}{16}$.

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

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

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

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

Now, let us analyze what will happen if we perform decryption of \mathbf{ct}^* without performing the bootstrapping over \mathbf{ct}^* 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}^* , first compute a phase $\Phi = b^* - \mathbf{a}^* \cdot \mathbf{sk} \pmod{1}$, now if $\Phi > 0$ then return 1, otherwise return 0.

Consider the following analysis of the decryption operation of \mathbf{ct}^* 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 $0 \leq e_1, e_2 < \frac{1}{16}$, or $-\frac{1}{16} < e_1, e_2 \leq 0$,

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

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

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

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., $\Delta(m_1) = \Delta(m_2) = \frac{1}{8}$,

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

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,

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

5.2 Construction of 3-input Gates

The above analysis motivates us to construct 3-input AND and XOR gates. In TFHE homomorphic XOR is defined as $\text{HomXOR}(\mathbf{ct}_1, \mathbf{ct}_2) = \left(\frac{1}{4}, \mathbf{0} \right) + (\mathbf{ct}_1 + \mathbf{ct}_2)$. Now to perform homomorphic XOR over three inputs \mathbf{ct}_1 , \mathbf{ct}_2 and \mathbf{ct}_3 , we perform the following,

$$\begin{aligned}
\mathbf{ct}_{tmp} &= \left(\frac{1}{4}, \mathbf{0} \right) + (\mathbf{ct}_1 + \mathbf{ct}_2), \\
\mathbf{ct}^* &= \left(\frac{1}{4}, \mathbf{0} \right) + (\mathbf{ct}_{tmp} + \mathbf{ct}_3).
\end{aligned}$$

Here, \mathbf{ct}_{tmp} can be computed without bootstrapping as described earlier, but \mathbf{ct}^* should result in a decryption correctness. Note here that, to achieve correctness of a 3-input XOR gate taking \mathbf{ct}_1 , \mathbf{ct}_2 and \mathbf{ct}_3 as input, the noise in the resultant ciphertext \mathbf{ct}^* should lie within the threshold of decryption correctness even without bootstrapping; because the bootstrapping operation is essentially a homomorphic decryption operation which only helps to retain correctness in further computations by reducing the noise within a required threshold. For that, we propose a 3-input XOR gate construction that results in correct decryption even without bootstrapping. However, our 3-input gate is followed by a bootstrapping to retain the correctness in further homomorphic computation. We define our 3-input homomorphic XOR as,

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

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

and, we define our 3-input homomorphic AND gate as,

$$\text{HomAND}(\mathbf{ct}_1, \mathbf{ct}_2, \mathbf{ct}_3) = \left(-\frac{1}{8}, \mathbf{0}\right) + \frac{1}{2} \cdot (\mathbf{ct}_1 + \mathbf{ct}_2 + \mathbf{ct}_3).$$

In the following section, we prove the correctness of our proposed 3-input gates. As bootstrapping is the primary bottleneck of evaluating a Boolean gate, both the 2-input and 3-input gates take approximately similar evaluation time (ref. Table 2).

5.3 Proof of Correctness of 3-input Gates

In this section, we provide a detailed mathematical explanation for the correctness of our proposed 3-input Boolean gates. Recall from Section 5.2 that our 3-input XOR is defined as,

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

Before, diving into the correctness, we recall that our correctness of 3-input gates relies on the fact that the noise \hat{e} in the ciphertext after two Boolean operations remain within the threshold of correct decryption, i.e., $\hat{e} < \frac{1}{16}$, as mentioned in the TFHE scheme. There are two extreme limits of Gaussian noise distribution are mentioned in TFHE; one distribution having standard deviation $\alpha_{min} = 2^{-15}$ corresponds to the minimum value to achieve the required security (128-bit) and the other standard deviation $\alpha_{max} = 2^{-6}$ provides the maximum value to retain correctness. From the property of Gaussian distribution, we have $|\hat{e}| < 4 \cdot \alpha_{max} = \frac{1}{16}$. Observe that, in a Boolean gate the noises in the input ciphertexts are added, and after bootstrapping they are reduced within a required threshold to retain correctness. A bootstrapping operation is a homomorphic decryption; thus the output of a 3-input homomorphic Boolean gate should result in correct decryption even without bootstrapping (ref. Section 5.2), for that the noise in the resultant ciphertext should lie within the threshold. Therefore, we sample the encryption noises from α_{min} , maintaining 128-bit security. Hence, the initial noise e in the input ciphertexts lies within $|e| < 4 \cdot \alpha_{min} = 2^{-13}$. As a result, the final noise \hat{e} after a 3-input homomorphic gate lies within 2^{-4} ; and after bootstrapping \hat{e} is removed and a fresh noise sampled from $\mathcal{G}_{\alpha_{min}}$, a Gaussian distribution with standard deviation α_{min} is added.

Let us now delve into the correctness of a 3-input gate without bootstrapping. Here, we choose 3-input HomXOR as an example. Extending from Section 5.1, the phase computed during decryption of after a 3-input HomXOR can be written as,

$$\Phi = \frac{1}{2} + 2 \cdot (\Delta(m_1) + \Delta(m_2) + \Delta(m_3)) + 2 \cdot (e_1 + e_2 + e_3)$$

Consider the analysis of the decryption operation of with $m_1, m_2, m_3 = 0$, i.e., $\Delta(m_1) = \Delta(m_2) = \Delta(m_3) = -\frac{1}{8}$.

$$\begin{aligned}\Phi &= \frac{1}{2} + 2 \cdot \left(-\frac{1}{8} - \frac{1}{8} - \frac{1}{8} \right) + 2 \cdot (e_1 + e_2 + e_3), \\ &= -\frac{1}{4} + 2 \cdot (e_1 + e_2 + e_3), \\ &= -\frac{1}{4} + \delta \quad [|\delta| < 2^{-10}, \text{ as } |e_1|, |e_2|, |e_3| < 2^{-13}].\end{aligned}$$

In this cases, $\Phi \approx -\frac{1}{4}$, or, $\Phi \pmod{1} = -\frac{1}{4} < 0$. Hence, the decryption of 3-input XOR upon $m_1 = m_2 = m_3 = 0$ is 0, which is a correct decryption. Now, consider when the plaintexts are $m_1 = m_2 = m_3 = 1$, or, $\Delta(m_1) = \Delta(m_2) = \Delta(m_3) = \frac{1}{8}$,

$$\begin{aligned}\Phi &= \frac{1}{2} + 2 \cdot \left(\frac{1}{8} + \frac{1}{8} + \frac{1}{8} \right) + 2 \cdot (e_1 + e_2 + e_3), \\ &= \frac{5}{4} + 2 \cdot (e_1 + e_2 + e_3), \\ &= \frac{5}{4} + \delta \quad [|\delta| < 2^{-10}, \text{ as } |e_1|, |e_2|, |e_3| < 2^{-13}].\end{aligned}$$

In this cases, $\Phi \approx \frac{5}{4}$, or, $\Phi \pmod{1} \approx \frac{5}{4} - 1 = \frac{1}{4} > 0$ hence, the decryption of 3-input XOR upon $m_1 = m_2 = m_3 = 1$ is 1, which is a correct decryption. Now, consider when two plaintext bits are 0, without loss of generality consider $m_1 = m_2 = 0$, and $m_3 = 1$, or, $\Delta(m_1) = \Delta(m_2) = -\frac{1}{8}$, and $\Delta(m_3) = \frac{1}{8}$,

$$\begin{aligned}\Phi &= \frac{1}{2} + 2 \cdot \left(-\frac{1}{8} - \frac{1}{8} + \frac{1}{8} \right) + 2 \cdot (e_1 + e_2 + e_3), \\ &= \frac{1}{4} + 2 \cdot (e_1 + e_2 + e_3), \\ &= \frac{1}{4} + \delta \quad [|\delta| < 2^{-10}, \text{ as } |e_1|, |e_2|, |e_3| < 2^{-13}].\end{aligned}$$

In this cases, $\Phi \approx \frac{1}{4}$, or, $\Phi \pmod{1} = \frac{1}{4} > 0$, hence the decryption of 3-input XOR upon $m_1 = m_2 = 0$, and $m_3 = 1$ is 1, which is a correct decryption. Now, consider when two plaintext bits are 1, without loss of generality consider $m_1 = m_2 = 1$, and $m_3 = 0$, or, $\Delta(m_1) = \Delta(m_2) = \frac{1}{8}$, and $\Delta(m_3) = -\frac{1}{8}$,

$$\begin{aligned}\Phi &= \frac{1}{2} + 2 \cdot \left(\frac{1}{8} + \frac{1}{8} - \frac{1}{8} \right) + 2 \cdot (e_1 + e_2 + e_3), \\ &= \frac{3}{4} + 2 \cdot (e_1 + e_2 + e_3), \\ &= \frac{3}{4} + \delta \quad [|\delta| < 2^{-10}, \text{ as } |e_1|, |e_2|, |e_3| < 2^{-13}].\end{aligned}$$

In this cases, $\Phi \approx \frac{3}{4}$, or, $\Phi \pmod{1} \approx \frac{3}{4} - 1 = -\frac{1}{4} < 0$ hence, the decryption of 3-input XOR upon $m_1 = m_2 = 1$, and $m_3 = 0$ is 0, which is a correct decryption.

Similarly, for 3-input AND gate, i.e., $\text{HomAND}(\mathbf{ct}_1, \mathbf{ct}_2, \mathbf{ct}_3) = (-\frac{1}{8}, \mathbf{0}) + \frac{1}{2} \cdot (\mathbf{ct}_1 + \mathbf{ct}_2 + \mathbf{ct}_3)$ we can prove the correctness as follows, from Section 5.1, the phase computed during decryption of after a 3-input HomAND can be written as,

$$\Phi = -\frac{1}{8} + \frac{1}{2} \cdot (\Delta(m_1) + \Delta(m_2) + \Delta(m_3)) + \frac{1}{2} \cdot (e_1 + e_2 + e_3)$$

Consider the analysis of the decryption operation of with $m_1, m_2, m_3 = 0$, i.e., $\Delta(m_1) = \Delta(m_2) = \Delta(m_3) = -\frac{1}{8}$.

$$\begin{aligned} \Phi &= -\frac{1}{8} + \frac{1}{2} \cdot \left(-\frac{1}{8} - \frac{1}{8} - \frac{1}{8}\right) + \frac{1}{2} \cdot (e_1 + e_2 + e_3), \\ &= -\frac{5}{16} + \frac{1}{2} \cdot (e_1 + e_2 + e_3), \\ &= -\frac{5}{16} + \delta \quad [|\delta| < 2^{-12}, \text{ as } |e_1|, |e_2|, |e_3| < 2^{-13}]. \end{aligned}$$

In this cases, $\Phi \approx -\frac{5}{16}$, or, $\Phi \pmod{1} = -\frac{5}{16} < 0$, hence, the decryption of 3-input AND upon $m_1 = m_2 = m_3 = 0$ is 0, which is a correct decryption. Now, consider when the plaintexts are $m_1 = m_2 = m_3 = 1$, or, $\Delta(m_1) = \Delta(m_2) = \Delta(m_3) = \frac{1}{8}$,

$$\begin{aligned} \Phi &= -\frac{1}{8} + \frac{1}{2} \cdot \left(\frac{1}{8} + \frac{1}{8} + \frac{1}{8}\right) + \frac{1}{2} \cdot (e_1 + e_2 + e_3), \\ &= \frac{1}{16} + 2 \cdot (e_1 + e_2 + e_3), \\ &= \frac{1}{16} + \delta \quad [|\delta| < 2^{-12}, \text{ as } |e_1|, |e_2|, |e_3| < 2^{-13}]. \end{aligned}$$

In this cases, $\Phi \approx \frac{1}{16}$, or, $\Phi \pmod{1} \approx \frac{1}{16} > 0$ hence, the decryption of 3-input AND upon $m_1 = m_2 = m_3 = 1$ is 1, which is a correct decryption. Now, consider when two plaintext bits are 0, without loss of generality consider $m_1 = m_2 = 0$, and $m_3 = 1$, or, $\Delta(m_1) = \Delta(m_2) = -\frac{1}{8}$, and $\Delta(m_3) = \frac{1}{8}$,

$$\begin{aligned} \Phi &= -\frac{1}{8} + \frac{1}{2} \cdot \left(-\frac{1}{8} - \frac{1}{8} + \frac{1}{8}\right) + \frac{1}{2} \cdot (e_1 + e_2 + e_3), \\ &= -\frac{3}{16} + \frac{1}{2} \cdot (e_1 + e_2 + e_3), \\ &= -\frac{3}{16} + \delta \quad [|\delta| < 2^{-12}, \text{ as } |e_1|, |e_2|, |e_3| < 2^{-13}]. \end{aligned}$$

In this cases, $\Phi \approx -\frac{3}{16}$, or, $\Phi \pmod{1} = -\frac{3}{16} < 0$, hence the decryption of 3-input AND upon $m_1 = m_2 = 0$, and $m_3 = 1$ is 0, which is a correct decryption. Now, consider when two plaintext bits are 1, without loss of generality consider $m_1 = m_2 = 1$, and $m_3 = 0$, or, $\Delta(m_1) = \Delta(m_2) = \frac{1}{8}$, and $\Delta(m_3) = -\frac{1}{8}$,

$$\begin{aligned}
\Phi &= -\frac{1}{8} + \frac{1}{2} \cdot \left(\frac{1}{8} + \frac{1}{8} - \frac{1}{8} \right) + \frac{1}{2} \cdot (e_1 + e_2 + e_3), \\
&= -\frac{1}{16} + \frac{1}{2} \cdot (e_1 + e_2 + e_3), \\
&= -\frac{1}{16} + \delta \quad [|\delta| < 2^{-12}, \text{ as } |e_1|, |e_2|, |e_3| < 2^{-13}].
\end{aligned}$$

In this cases, $\Phi \approx -\frac{1}{16}$, or, $\Phi \pmod{1} \approx -\frac{1}{16} < 0$ hence, the decryption of 3-input AND upon $m_1 = m_2 = 1$, and $m_3 = 0$ is 0, which is a correct decryption.

5.4 Security Analysis of 3-input Gates

In this section, we discuss the impact on the security notion of the underlying TFHE scheme for using 3-input homomorphic Boolean gates. Here, we define our security definition as,

Definition 3 (Our Security Definition) *Our modified TFHE scheme (TFHE_m) using 3-input gates is IND-secure for any security parameter $\lambda \in \mathbb{N}$, for any PPT adversary \mathcal{A}_d , i.e., $|\Pr[\mathbf{G}_{\text{TFHE}_m, \mathcal{C}_h, \mathcal{A}_d}(1^\lambda) = 0] - \Pr[\mathbf{G}_{\text{TFHE}_m, \mathcal{C}_h, \mathcal{A}_d}(1^\lambda) = 1]| \leq \text{negl}(\lambda)$.*

We prove our security notion by showing indistinguishability between two hybrid models (**Hybrid**₀ and **Hybrid**₁) of $\mathbf{G}_{\text{TFHE}, \mathcal{C}_h, \mathcal{A}_d}(1^\lambda)$ (from Section 3.4). By Definition 2, **Hybrid**₀ follows the notion of IND-security, whereas the **Hybrid**₁ depicts the security of our modified TFHE scheme, that incorporates 3-input homomorphic gates. These two hybrids only differ in the stage 3 of *Query Phase* of the security game $\mathbf{G}_{\text{TFHE}, \mathcal{C}_h, \mathcal{A}_d}(1^\lambda)$, as follows

Hybrid₀. This game is between challenger \mathcal{C}_h and adversary \mathcal{A}_d .

Query Phase.

- 3 \mathcal{A}_d queries a circuit $C : \{0, 1\}^3 \rightarrow \{0, 1\}$, having 3-input and 1-output structure. Now, C is evaluated using two consecutive 2-input gates, requiring a total of two bootstrapping. \mathcal{C}_h computes the evaluated ciphertext as $\overline{\mathbf{ct}}_0 = \text{TFHE.Eval}(\{\mathbf{ct}_i\}_{i \in \{0, 1\}}, C, \mathbf{pk})$. \mathcal{A}_d is handed over $\overline{\mathbf{ct}}$.

Hybrid₁. This game is between challenger \mathcal{C}_h and adversary \mathcal{A}_d .

Query Phase.

- 3 \mathcal{A}_d queries a circuit $C : \{0, 1\}^3 \rightarrow \{0, 1\}$, that has 3-input and 1-output structure. Now, the circuit C is evaluated using a 3-input gate, that requires only one bootstrapping in total. \mathcal{C}_h computes the evaluated ciphertext as $\overline{\mathbf{ct}}_1 = \text{TFHE.Eval}(\{\mathbf{ct}_i\}_{i \in \{0, 1\}}, C, \mathbf{pk})$. \mathcal{A}_d is handed over $\overline{\mathbf{ct}}$.

Indistinguishability Argument. In **Hybrid**₀ two consecutive 2-input gates are used, and the evaluated ciphertext $\overline{\mathbf{ct}}_0$ contains a fresh noise e^0 due to bootstrapping. While in **Hybrid**₁ a 3-input gate is used, for which the evaluated ciphertext $\overline{\mathbf{ct}}_1$ contains a different but fresh noise e^1 due to the final bootstrapping. Both e^0 and e^1 are sampled from the

same Gaussian distribution \mathcal{G}_α and are independent of each other. Thus in the view of \mathcal{A}_d both the evaluated ciphertexts $\overline{\mathbf{ct}_0}, \overline{\mathbf{ct}_1}$ are indistinguishable due to the LWE hardness assumption. Therefore, **Hybrid**₀ and **Hybrid**₁ are indistinguishable and **Hybrid**₁ achieves the notion of IND security.

5.5 3-input vs. n -input Gate ($n \geq 4$)

In this section, we elaborate on why an n -input gate for $n \geq 4$ is not possible to construct using TFHE. For example, let us construct a 4-input XOR following a similar approach mentioned in Section 5.2. We can now define a 4-input XOR as, $\text{HomXOR}(\mathbf{ct}_1, \mathbf{ct}_2, \mathbf{ct}_3, \mathbf{ct}_4) = (1, \mathbf{0}) + 2 \cdot (\mathbf{ct}_1 + \mathbf{ct}_2 + \mathbf{ct}_3 + \mathbf{ct}_4) \pmod{1} = 2 \cdot (\mathbf{ct}_1 + \mathbf{ct}_2 + \mathbf{ct}_3 + \mathbf{ct}_4)$. Now, the decryption of a 4-input XOR can be written as,

$$\Phi = 2 \cdot (\Delta(m_1) + \Delta(m_2) + \Delta(m_3) + \Delta(m_4)) + 2 \cdot (e_1 + e_2 + e_3 + e_4)$$

Consider, $2 \cdot (e_1 + e_2 + e_3 + e_4) = \delta$, where δ being small given $e_i \leftarrow \mathcal{G}_{\alpha_{min}}, \forall i \in [1, 4]$. Now, for $m_1, m_2, m_3, m_4 = 0$, or, $\Delta(m_1) = \Delta(m_2) = \Delta(m_3) = \Delta(m_4) = -\frac{1}{8}$, where m_i be the underlying plaintext of $\mathbf{ct}_i, \forall i \in [1, 4]$, we have

$$\Phi = 2 \cdot \left(-\frac{1}{8} - \frac{1}{8} - \frac{1}{8} - \frac{1}{8} \right) + \delta = -1 + \delta.$$

Now, $\Phi = -1 + \delta \pmod{1} = \delta$. Therefore, the decryption of 4-input XOR depends upon the sign of δ , which can be either positive or negative; thus the decryption is invalid.

Similarly, for other values of m_i 's and 4-input AND gate, we can show that the decryption will fail. This implies in TFHE scheme it is hard to construct a 4-input gate (similarly for n -input gates $n \geq 4$) with only one bootstrapping. Therefore, to evaluate a 4-input gate, we require a combination of a 3-input and a 2-input gate as shown in Figure. 2. Consequently, for any n -input gate ($n \geq 4$) an optimal combination of 3-input and 2-input gates can be constructed with at most $\lceil \frac{n}{2} \rceil$ bootstrapping operations in total.

6 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. 3. 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. 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 7.2) to generate multiple gate-level netlists of the circuit. ②

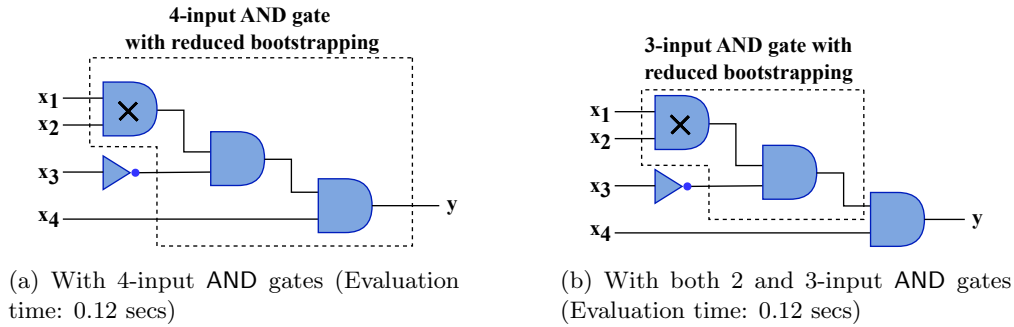


Figure 2: Comparison between 4-input and 3-input AND gates with reduced bootstrapping.

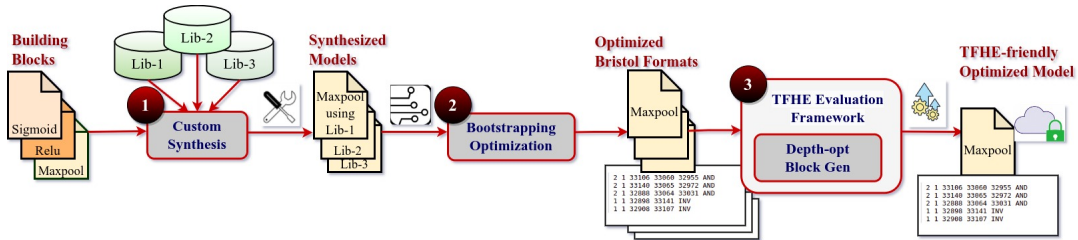


Figure 3: Proposed EDA flow to Generate Optimized TFHE-friendly Circuit

converts the generated gate-level netlists to the corresponding Bristol format and optimizes these with the objective of minimizing the number of bootstrapping, thereby significantly improving the overall performance and efficiency of the circuit evaluation. In the final stage ③, we evaluate the optimized Bristol formats 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 sections, we delve into these stages in greater detail.

6.1 Logic Synthesis Challenges 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 5 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 of optimizing 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 4.1. 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.

6.2 FHEDA Flow: Stages

Figure. 3 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

6.2.1 Stage: ①- Custom Synthesis

Our third experimental demonstration in Section 4.1 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 circuit depth and bootstrapping requirements. Furthermore, in Section 5, 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 ($set_1, set_2, \dots, set_N$) (described elaborately in Section 7.2). 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.

6.2.2 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 4.1) without altering the circuit functionality in the Bristol representation, 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. 4. Consider N sets ($set_1, set_2, \dots, set_N$) containing different combinations of logic gates, e.g., set_i may contain 2-input and 3-input XOR, AND, and NOT along with

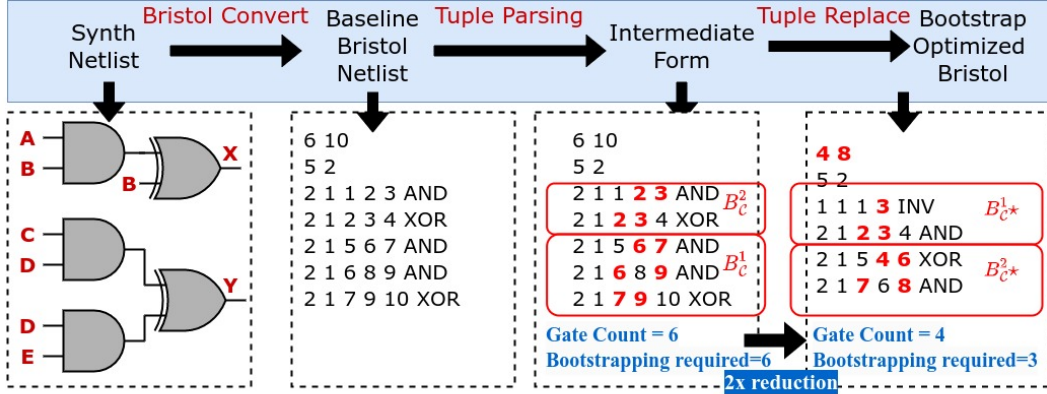


Figure 4: Optimization of Bootstrapping Components

NAND, NOR and OR gates. Assume, $C : \{0, 1\}^* \rightarrow \{0, 1\}^*$ be a Boolean circuit and let B_C be its Bristol representation. The functionality in B_C is then converted into multiple Boolean formats using the Boolean gates from each of the sets in $\{set_1, \dots, set_N\}$ using our $\text{Convert}(set_i, B_C)$ algorithm that takes a set set_i and the baseline Bristol Format B_C as input and returns a different circuit description B_C^i of C only using the Boolean gates from set set_i . Running $\text{Convert}(\cdot)$ algorithm N times we get the set (B_C^1, \dots, B_C^N) representing N circuit descriptions of the same circuit.

After the above conversion we again parse all the N circuit representations, i.e., (B_C^1, \dots, B_C^N) using our $\text{Parse}(\cdot)$ algorithm. This algorithm takes a set struct and (B_C^1, \dots, B_C^N) as input and outputs another set $(B_C^{1*}, \dots, B_C^{N*})$. The set struct consists of a set of tuples described as follows: $\text{struct} = \left\{ \left((a \cdot b) \oplus a, a \cdot (\bar{b}) \right), \left((a \cdot b) \oplus b, b \cdot (\bar{a}) \right), \left((a \cdot b) \oplus (b \cdot c), b \cdot (a \oplus c) \right), \left((a \oplus b) \oplus (a \oplus c), (b \oplus c) \right) \right\}$, here a, b and c are Boolean variables, \oplus, \wedge and $\bar{\cdot}$ denotes XOR, AND and NOT gate operation respectively. Now, our $\text{Parse}(\text{struct}, B_C^i)$ algorithm parses each $B_C^{i \in [N]}$ 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 $\text{Parse}(\cdot)$ algorithm gives another set containing N circuit descriptions with compact representations, i.e., $(B_C^{1*}, \dots, B_C^{N*})$. 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. 4, where the gate count of the sample circuit is reduced from 6 to 4, resulting in only 3 bootstrappable gates due to our *Inverter based optimization*, that 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.

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	$a \cdot (b)$	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_C , struct

Ensure: $(B_C^{1*}, \dots, B_C^{N*})$

- 1: Initialize empty sets $B = \{\}$ and $B^* = \{\}$
 - 2: **for** $i = 1$ **to** N
 - 3: $B_C^i \leftarrow \text{Convert}(set_i, B_C)$
 - 4: $B \leftarrow B \cup B_C^i$
 - 5: **for** $i = 1$ **to** N
 - 6: $B_C^{i*} \leftarrow \text{Parse}(\text{struct}, B[i])$
 - 7: $B^* \leftarrow B^* \cup B_C^{i*}$
 - 8: **return** B^*
-

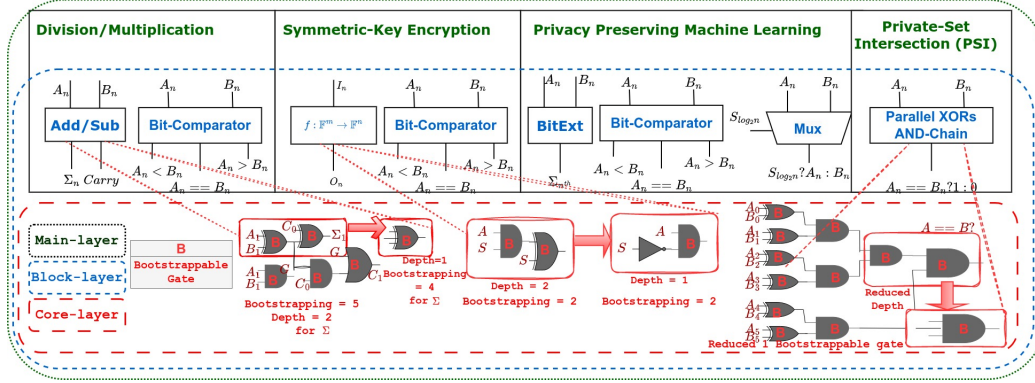


Figure 5: Three-layered Execution of Proposed EDA Flow

6.2.3 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, \dots, m_ℓ) be the inputs of a circuit $C^* : \{0, 1\}^\ell \rightarrow \{0, 1\}^{\ell'}$. Now, before evaluating the circuit C^* we parse its Bristol Format through the procedure mentioned in Algorithm 1 to retrieve the optimal Boolean representations of C^* . Let us assume, $\{B_{C^*}^1, \dots, B_{C^*}^N\}$ are the N optimal Boolean representation of C^* . Now, to evaluate C^* , we first encrypt the inputs (m_1, \dots, m_ℓ) using public key pk and return the ciphertexts (μ_1, \dots, μ_ℓ) , such that $\mu_i =$

Algorithm 2: Evaluation of Optimal Circuit

Require: $\{B_{C^*}^1, \dots, B_{C^*}^N\}$, pk , $\{m_1, \dots, m_\ell\}$
Ensure: $\text{eval}^*, B_{C^*}^*$

- 1: Initialize empty sets $\mu = \{\}$ and $\text{eval} = \{\}$
 - 2: **for** $i = 1$ **to** ℓ
 - 3: $\mu_i \leftarrow \text{TFHE.Enc}(m_i, \text{pk})$
 - 4: $\mu \leftarrow \mu \cup \mu_i$
 - 5: **for** $i = 1$ **to** N
 - 6: $\text{eval}_i \leftarrow \text{TFHE.Eval}(\mu, \text{pk}, B^*[i])$
 - 7: $\text{eval} \leftarrow \text{eval} \cup \text{eval}_i$
 - 8: $\text{eval}^* = \min(\text{eval}_1, \dots, \text{eval}_N)$
 - 9: Set $B_{C^*}^*$ as the optimal circuit with the minimum evaluation time
 - 10: **return** $\text{eval}^*, B_{C^*}^*$
-

$\text{TFHE.Enc}(m_i, \text{pk})$, $\forall i \in [\ell]$. Each $B_{C^*}^j$, $\forall j \in [N]$, we invoke $\text{TFHE.Eval}((\mu_1, \dots, \mu_\ell), \text{pk}, B_{C^*}^j)$, $\forall j \in [N]$ and compute the homomorphic evaluation time eval_j for circuit representation $B_{C^*}^j$. Finally, we retrieve the minimum of $\{\text{eval}_1, \dots, \text{eval}_N\}$ as eval^* and also returns the optimal Boolean representation for which the minimum evaluation time is achieved.

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

7 Three Layer Execution of FHEDA

Our proposed flow FHEDA follows a three-tiered execution strategy as shown in Figure. 5. 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.

7.1 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 FHEDA flow (Figure. 3) we have used Cadence Genus (version:17.24-s038.1) for synthesis. The standard cell library used for synthesis is the 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 stage ③ 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 followed by the bootstrapping operation implemented in the TFHE library⁷ 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 the different three tiers of our proposed FHEDA execution flow.

7.2 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 4.1, 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 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 Lib – Set2 that consist of both 2 and 3-input XOR and AND gates with unary NOT gates. Using 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 5). This results in significant time savings and improved depth efficiency during the homomorphic evaluation process. Lib – Set3 incorporates all fundamental 2-input universal gates, including NAND and NOR, alongside XOR, AND, OR, and NOT gates. By combining Lib – Set2 with Lib – Set3, we establish 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,

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

where GATE_p is a Boolean gate GATE with p inputs.

⁷<https://github.com/nucypher/TFHE.jl.git>

7.3 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 display 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

```
1 module multiplexer_2x1
2 (input A, B, S, output Y);
3   INV n1 (nS, S);
4   NAND2 a1 (Y1, nS, A);
5   NAND2 a2 (Y2, S, B);
6   NAND2 o1 (Y, Y1, Y2);
7 endmodule
```

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

```
1 module multiplexer_2x1
2 (input A, B, S, output Y);
3   INV n1 (nS, S);
4   AND2 a1 (Y1, nS, A);
5   INV aa1 (nY1, Y1);
6   AND2 a2 (Y2, S, B);
```

```

7     INV aa2 (nY2, Y2);
8     AND2 o1 (nY, nY1, nY2);
9     INV o2 (Y, nY);
10  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.

7.4 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. 6 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) ($k' = 8$). These 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

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
	FC ₂	$84 \rightarrow 10$	0.93	0.77	0.88	0.71	23.65
5	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

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 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 an 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 [34] and encrypted databases [35]. In this work, we considered AES [36], PRESENT [37], and LowMC [35] ciphers as our benchmark netlists. The AES implementation from Archer et al. [38] was employed and processed through our FHEDA flow. By utilizing Lib – Set2, we managed to achieve an improvement of up to 19% in the evaluation time for AES. In the case of LowMC, Lib – Set3 resulted in the maximum depth improvement, while Lib – Set4 led to better evaluation time improvement. Finally, for PRESENT, 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 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.

8 Homomorphic Inference of Convolution Neural Network

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 [39] followed by a non-linear activation function [40] and pooling layers [41]. 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 [42], 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 [43] with a window size of (2, 2) and stride of 2, and the Tanh [44] activation function except for the last layer which uses a Softmax activation [45]. 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 [46] and ReLU activation [47] in our present CNN architecture. In Table 5 we present different layers of our CNN model with input/output specifications. Below is our CNN architecture in short: (CONV + ReLU + MaxPool) → (CONV + ReLU + MaxPool) → (CONV + ReLU) → (FC₁ + ReLU) → (FC₂ + 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, MaxPool denotes a max-pooling layer with window size (2, 2) and stride 2, FC₁ and FC₂ 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 public key **pk** 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 using process-level threads for parallel computation and homomorphic evaluation results in Table 5 using our four different library sets.

9 Conclusion

The growing focus on data privacy has sparked significant interest in 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 FHEDA, an EDA framework for generating efficient TFHE-friendly Boolean circuits. 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 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 neural network inference. 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] J. Mono, K. Kluczniak, and T. Güneysu, “Improved circuit synthesis with amortized bootstrapping for fhe-like schemes,” Cryptology ePrint Archive, Paper 2023/1223, 2023, <https://eprint.iacr.org/2023/1223>. [Online]. Available: <https://eprint.iacr.org/2023/1223>
- [10] S. Carpov, M. Izabachène, and V. Mollimard, “New techniques for multi-value input homomorphic evaluation and applications,” in *Topics in Cryptology – CT-RSA 2019*, M. Matsui, Ed. Cham: Springer International Publishing, 2019, pp. 106–126.
- [11] E. M. Songhori, S. U. Hussain, A.-R. Sadeghi, T. Schneider, and F. Koushanfar, “Tinygarble: Highly compressed and scalable sequential garbled circuits,” in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 411–428.
- [12] D. Demmler, G. Dessouky, F. Koushanfar, A.-R. Sadeghi, T. Schneider, and S. Zeitouni, “Automated synthesis of optimized circuits for secure computation,” in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 1504–1517.
- [13] Z. Ghodsi, A. K. Veldanda, B. Reagen, and S. Garg, “Cryptonas: Private inference on a relu budget,” *Advances in Neural Information Processing Systems*, vol. 33, pp. 16 961–16 971, 2020.
- [14] “Microsoft SEAL (release 4.1),” <https://github.com/Microsoft/SEAL>, Jan. 2023, microsoft Research, Redmond, WA.
- [15] S. Halevi and V. Shoup, “Helib-an implementation of homomorphic encryption,” *Cryptology ePrint Archive, Report 2014/039*, 2014.
- [16] 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.
- [17] 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.
- [18] 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.
- [19] 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.

- [20] 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.
- [21] 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.
- [22] 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.
- [23] 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.
- [24] 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.
- [25] 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.
- [26] 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.
- [27] J. Zhang, X. Cheng, L. Yang, J. Hu, X. Liu, and K. Chen, “Sok: Fully homomorphic encryption accelerators,” *arXiv preprint arXiv:2212.01713*, 2022.
- [28] 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.
- [29] 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.
- [30] 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.
- [31] 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.

- [32] 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.
- [33] 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.
- [34] 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>
- [35] 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.
- [36] 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.
- [37] 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.
- [38] D. Archer, V. A. Abril, S. Lu, P. Maene, N. Mertens, D. Sijacic, and N. Smart, “Bristol fashion mpc circuits,” 2021.
- [39] 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.
- [40] 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.
- [41] M. Sun, Z. Song, X. Jiang, J. Pan, and Y. Pang, “Learning pooling for convolutional neural network,” *Neurocomputing*, vol. 224, pp. 96–104, 2017.
- [42] 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.
- [43] 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.
- [44] 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.

- [45] 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.
- [46] N. Murray and F. Perronnin, “Generalized max pooling,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2014, pp. 2473–2480.
- [47] 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.

.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 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. 6 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 [42] 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 CNN⁸, 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. 6, 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

⁸As 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.

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 $c = 6$).

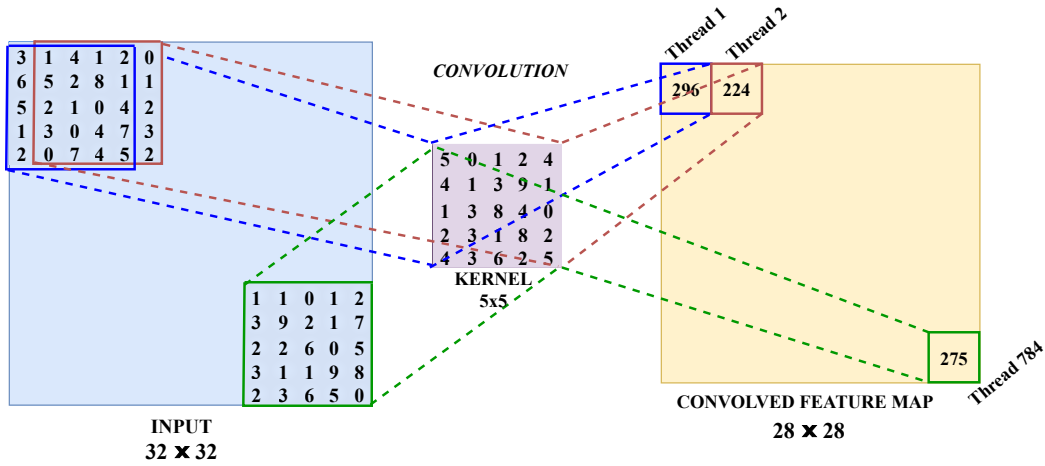


Figure 6: Convolutional Layer

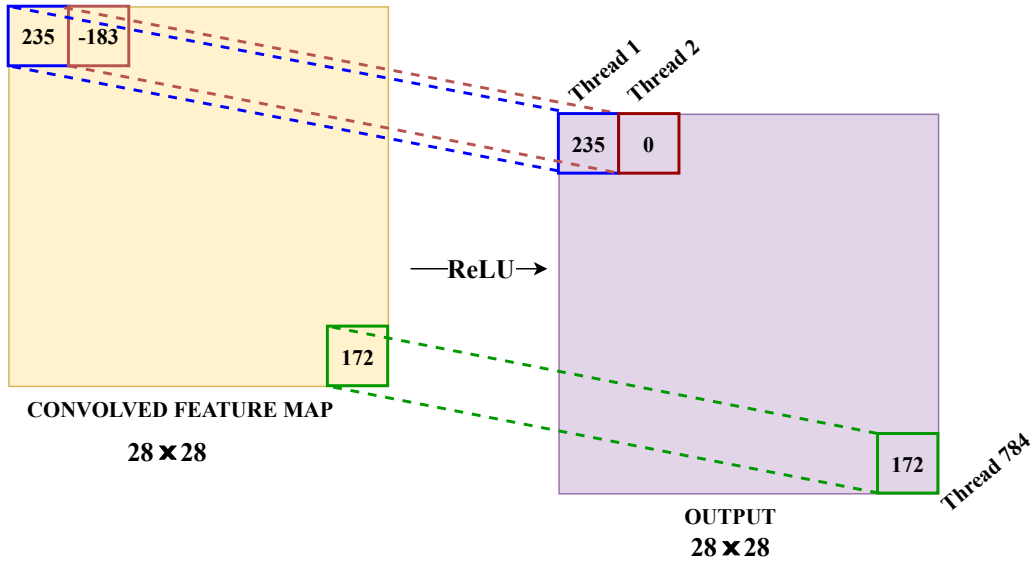


Figure 7: Rectified Linear Unit

.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 $\text{ReLU}(x) = \max(0, x)$, 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. 7.

.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 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. 8. 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.

.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 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. 9). 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.

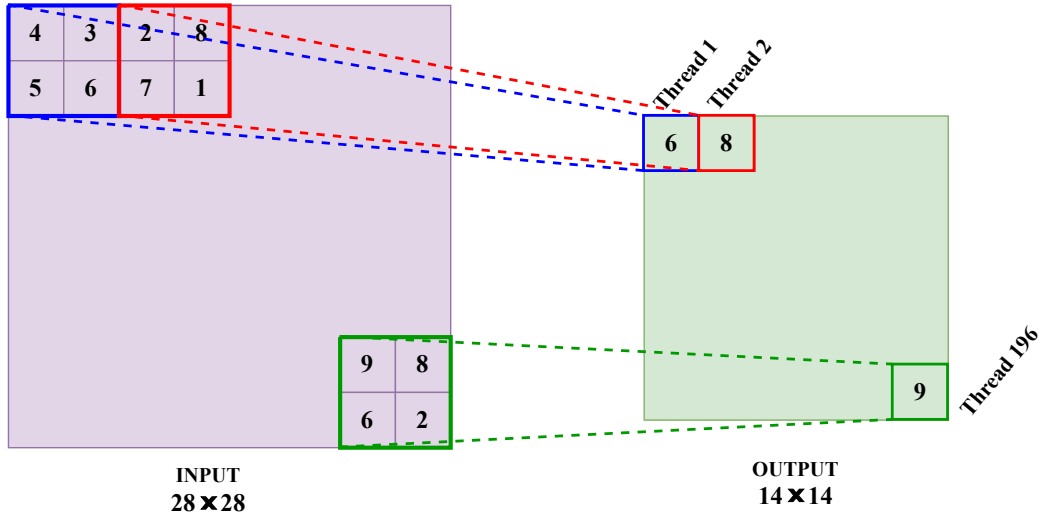


Figure 8: Maxpool Layer

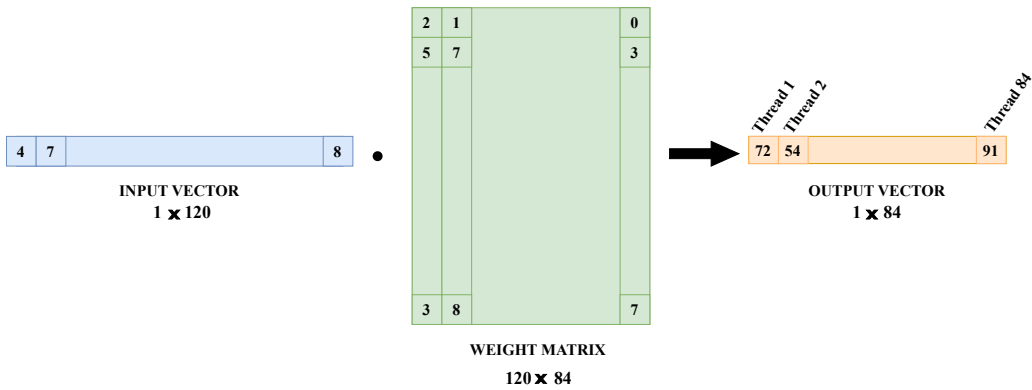


Figure 9: Fully Connected Layer

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 is 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.

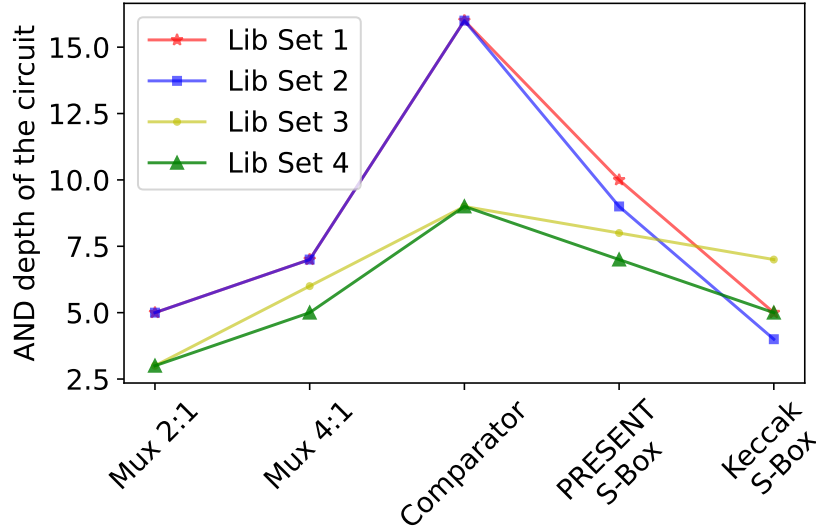


Figure 10: AND depth of block layer

.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 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 [14], HELib[15], TFHE, PALISADE⁹ accelerate the performance of different FHE schemes. Several research illustrated that GPU based implementations [16, 17, 18, 19] make use of inherent parallelism in FHE. Intel proposed Intel Homomorphic Encryption Acceleration Library (HEXL) [20] 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 [21, 22, 23, 24, 25, 26] 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 [27]. 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 [28], a Fixed-Point FPGA accelerator is proposed for TFHE which is compute-bound with $937\times$ faster than CPU implementations and $2.5\times$ faster than the prior FHE accelerator, MATCHA [29] by Jiang *et al.* and cuFHE. *SynCirc* [30], 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 [31, 32] Cheetah [32] introduced algorithmic and hardware optimizations for HE DNN and uses MPC instead of bootstrapping for reducing the errors

⁹<https://gitlab.com/palisade/palisade-release>

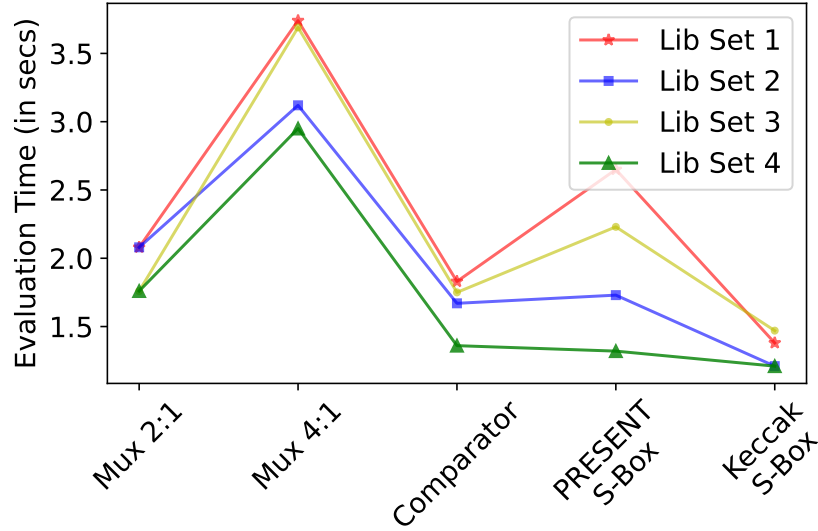


Figure 11: Evaluation time of block layer

during the HE operation. F1 [22] is the first programmable FHE accelerator that has achieved ASIC-level performance and introduced effective design by accelerating *primitive* FHE scheme. BTS [33], a bootstrappable FHE accelerator achieved a speedup of $2237\times$ in HE multiplication throughput in contrast to the state-of-the-art CPU implementations. 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. 10 we present a comparison of multiple block layer circuits with respect to AND-depth vs. gate count of the block layer circuits. Similarly, Figure. 11 shows the variations of homomorphic evaluation time with respect to the number of gates present in the different block layer circuits.