Improved Circuit Synthesis with Amortized Bootstrapping for FHEW-like Schemes

Johannes Mono[1], Kamil Kluczniak[2] and Tim Güneysu[1,3]

[1] Ruhr University Bochum, first.last@rub.de
[2] secunet Security Networks, kamil.kluczniak@gmail.com
[3] DFKI GmbH

Abstract. In recent years, the research community has made great progress in improving techniques for privacy-preserving computation, such as fully homomorphic encryption (FHE). Despite the progress, there remain open challenges, mainly in performance and usability, to further advance the adoption of these technologies. This work provides multiple contributions that improve the current state-of-the-art in both areas. More specifically, we significantly simplify the bootstrapping by Carpov, Izabachène, and Mollimard [CIM19] for Boolean-based FHE schemes such as FHEW or TFHE, making the concept usable in practice. Based on our simplifications, we implement an easy-to-use interface for amortized bootstrapping in the open-source library FHE-Deck [fhe23], derive new parameter sets for multi-bit encryptions with state-of-the-art security, and build a toolset that translates high-level code to multi-bit operations on encrypted data using circuit synthesis. We propose the first non-trivial FHE-specific optimizations in synthesizing privacy-preserving circuits: look-up table (LUT) grouping and adder substitution. Using LUT grouping, we reduce the number of bootstrapping operations by almost 35 % on average, while for adder substitution, we reduce the number of required bootstrappings by up to 80 % for certain use cases. Overall, the execution time is up to 3.8× faster using our optimizations compared to previous state-of-the-art circuit synthesis.

Keywords: fully homomorphic encryption · FHEW · TFHE · circuit synthesis

1 Introduction

Encryption is a fundamental technology of today's society, securing data and communications around the globe. One exciting application is privacy-preserving computation, where techniques such as fully homomorphic encryption (FHE) encrypt and protect sensitive data during computation. However, there remain open challenges in adopting these technologies, for example, reducing computational costs or improving usability.

Gentry's seminal work [Gen09] introduces bootstrapping, in which the error associated with a homomorphic ciphertext is refreshed to allow for indefinite computation. Current state-of-the-art schemes are still based on this ingenious idea, and two strains have emerged. Word-based schemes such as BFV [Bra12, FV12], BGV [BGV14], and CKKS [CKKS17] operate on multiple large elements at a time and excel at highly parallelizable tasks. These schemes usually perform many operations followed by an expensive bootstrapping procedure and are often used for specific use cases requiring mainly additions or multiplications.

In contrast, Boolean-based schemes such as FHEW [DM15] and TFHE [CGGI16a] provide high flexibility encrypting single bits or small bit groups and are thus a good fit for a wide variety of use cases. A target function is commonly represented as a circuit composed of Boolean gates with encrypted input bits, and every gate evaluation requires a comparatively fast bootstrapping. However, bootstrapping is still the most expensive part of such circuits, and thus, it is crucial to reduce the number of gates when translating a use case to a circuit.

Translating use cases is either done manually for critical tasks (this can be compared to hand-written assembly) or using tools to automate translation from high-level code to Boolean circuits (similar to code compilation). Although the former can result in the best performance for a given use case, the process is rather tedious, and there has been some effort by the research community to provide automatic translations from high-level code to circuits [CDS15, CMG+18, GSPH+21], improving usability and easing adoption for non-experts interested in FHE.

Currently, two different approaches exist for automatically converting high-level code to Boolean circuits. The first approach is based on instruction mapping, where the high-level code is translated to an intermediate representation (IR). Afterward, the individual instructions of the IR are mapped to Boolean primitives and composed accordingly. The second approach is based on existing hardware tooling using so-called synthesizers. First, high-level synthesis converts high-level code to a hardware description language (HDL). Then, a synthesizer converts the HDL to a Boolean circuit. Finally, no matter the approach, the resulting circuit is translated to FHE library code.

In this work, we introduce several contributions to enhance the current state-of-the-art in FHE transpilation, with a primary focus on the synthesis-based approach:

- We significantly simplify the amortized bootstrapping idea proposed by Carpov, Izabachène, and Mollimard [CIM19], improving the current state-of-the-art for bootstrapping in FHEW-like schemes. Our modifications make it feasible to use this amortization technique in practice and enhance the capabilities of homomorphic gates (Subsection 3.2).

- We introduce the first non-trivial FHE-specific optimizations for single- and multi-bit circuits, modifying the circuit synthesis process to handle additions specifically optimized for FHE circuits. This results in circuits requiring up to 80 % less bootstrappings compared to non-optimized circuits (Subsection 3.7).

- We also perform post-synthesis optimizations on the netlist based on our improved amortized bootstrapping to evaluate multiple gates at once. With our optimizations, execution times are up to 3.8× faster compared to the previous state-of-the-art (Subsection 3.6).

- We propose new parameter sets for multi-bit FHEW-like encryptions with state-of-the-art security. We also implement an easy-to-use interface for amortized bootstrapping in the open-source library FHE-Deck [fhe23] (Subsection 3.4).

2 Preliminaries

In the following, we introduce the background to our work. We start with notations used throughout the papers, provide a formal definition of Boolean and homomorphic circuits, shortly describe FHE and its hardness assumption and conclude with a short section on transpilation in the context of FHE.

2.1 Notation

We denote as $\mathbb{Z}_Q$ the group of integers modulo Q and, for a power-of-two N, as $\mathcal{R}$ the ring of polynomials $\mathbb{Z}_Q[X]/(X^N+1)$. We call $z \in \mathcal{R}$ with $z^n = 1 \in \mathcal{R}$ the n-th root of unity. Note that $\mathcal{R}_Q$ has $2N$ roots of unity of the form X^a for $a \in \mathbb{Z}_{2N}$. Furthermore, the roots of unity in $\mathcal{R}_Q$ form an algebraic group of order $2N$ with respect to multiplication. For clarity, we denote ring elements as $\mathfrak{a} \in \mathcal{R}_Q$. We denote a n-dimensional column vector as

$[f(\cdot,i)]_{i=1}^{n}$, where $f(\cdot,i)$ defines the i-th coordinate. For brevity, we will also denote as $[n]$ the vector $[i]_{i=1}^{n}$, and as $[n,m]_{i=n}^{m}$ the vector $[n,\dots,m]^\top$. We address the i-th entry of a vector $\vec{v}$ by $\vec{v}[i]$.

By $x \leftarrow_R \mathcal{S}$, we denote sampling a random variable from the set $\mathcal{S}$. By default, we sample from the uniform distribution and explicitly state when referring to other distributions. For a random variable $a \in \mathbb{Z}$, we denote as $\mathsf{Var}(a)$ the variance of a, its expectation as $\mathsf{E}(x)$, and its standard deviation as $\mathsf{SD}(a)$. For $\mathfrak{a} \in \mathcal{R}_Q$, we define $\mathsf{Var}(\mathfrak{a})$ and $\mathsf{E}(\mathfrak{a})$ to be the variance and expectation of the coefficients of the polynomial $\mathfrak{a}$, respectively. We denote any polynomial as $\mathsf{poly}(\cdot)$. We denote as $\mathsf{negl}(\lambda)$ a negligible function in $\lambda \in \mathbb{N}$; that is, for any positive polynomial $\mathsf{poly}(\cdot)$ there exists $c \in \mathbb{N}$ such that for all $\lambda \geq c$ we have $\mathsf{negl}(\lambda) \leq \frac{1}{\mathsf{poly}(\lambda)}$.

2.2 Boolean and Homomorphic Circuits

We define a Boolean gate as an arbitrary Boolean function $\mathcal{G}_n : \mathbb{F}_2^n \to \mathbb{F}_2$ for a positive integer n. Any Boolean gate can be represented as LUT which stores 2^n outputs $O \in \mathbb{F}_2$, one for each input $I \in \mathbb{F}_2^n$; we often use the terms Boolean gate and LUT interchangably. An example for a Boolean gate $\mathcal{G}_1$ is an inverter gate

$$\mathtt{INV} : \mathbb{F}_2 \to \mathbb{F}_2,\;\; I \mapsto O = I + 1.$$

We can generalize a Boolean gate to multiple outputs as $\mathcal{G}_{n,m} : \mathbb{F}_2^n \to \mathbb{F}_2^m$ which can be represented with m LUTs, one for each output bit. Here, an example is a full adder with the inputs $x, y \in \mathbb{F}_2$ and a carry-in $c_i \in \mathbb{F}_2$ defined as

$$\mathtt{FA} : \mathbb{F}_2^3 \to \mathbb{F}_2^2,\;\; (x, y, c_i) \mapsto (s, c_o) = (x + y + c_i, x \cdot y + c_i \cdot (x + y))$$

for the sum s and the carry-out c_o. The number of input wires is also called fan-in of a gate, and the number of output wires fan-out.

A Boolean circuit $\mathcal{C}_2 : \mathbb{F}_2^s \to \mathbb{F}_2^t$ is a directed acyclic graph where the vertices are Boolean gates $\mathcal{G}_{n,m}$ and the edges connect gate outputs to the inputs of other gates with s unconnected global inputs and t unconnected global outputs. A well-known fact is that, for every directed acyclic graph, there exists a topological ordering of the graph. A topological ordering is an ordering such that for every edge, the start vertex of this edge appears before the end vertex in the sorted list of vertices. For a circuit $\mathcal{C}_2$, sorting the graph topologically and evaluating the sorted gates for a global input $I \in \mathbb{F}_2^s$ ensures that, for any given gate $\mathcal{G}_{n,m}$, all n inputs are known.

We can generate a Boolean circuit $\mathcal{C}_2$ from a circuit design in a HDL through a process called synthesis. The output, a textual representation of $\mathcal{C}_2$, is also referred to as netlist. Generally, a netlist can include additional elements such as registers for storage; however, in this work, a netlist only contains Boolean gates. During synthesis, the circuit is optimized, mostly heuristically, according to specific parameters, such as area usage or power consumption. Extending the idea of synthesis to high-level code is called high-level synthesis. Examples are the high-level synthesis tool XLS[1], translating C++ code to the HDL Verilog, and the low-level synthesis tool Yosys[2], generating a netlist from Verilog.

Usually, synthesis tools output a netlist either based on a gate library, a list of available gates for a given hardware platform, or output a LUT-based netlist with many LUTs for different functions $\mathcal{G}_n$ and some upper bound on fan-in[3]. For hardware with special high-performance gates, state-of-the-art synthesis tools are able to replace certain subcircuit

[1]https://github.com/google/xls

[2]https://github.com/yosyshq/yosys

[3]We note for completeness that synthesis tools also place gates $\mathcal{G}_{n,m}$ with $m > 1$, such as registers. Since we configure these tools to only output LUTs, which, by definition, only have a single output, this work assumes a netlist only containing a Boolean circuit $\mathcal{C}_2$ composed of gates $\mathcal{G}_n$.

patterns with optimized primitives; for example, multiplications are often realized in digital signal processing units.

We define a homomorphic circuit $\mathcal{C} : \mathcal{M}^s \to \mathcal{M}^t$ consisting of homomorphic gates of the form $f(b + \sum x_i \cdot a_i)$ for a function $f : \mathbb{Z}_p \to \mathbb{Z}_p$, known scalars $a_i, b \in \mathbb{Z}_p$, and encrypted plaintexts x_i. We compute the affine part $b + \sum x_i \cdot a_i$ using the homomorphic capabilities of the FHE scheme and the function f during bootstrapping, also known as functional bootstrapping (see Subsection 3.2). Note that we compute the affine part over the integers, but require the input to f in $\mathbb{Z}_p$; this has to be taken into account during homomorphic circuit design.

2.3 Fully Homomorphic Encryption

A FHE scheme consists of four algorithms (Setup, Enc, Eval, Dec), each with the following syntax [RAD78, Gen09].

- $\mathsf{Setup}(\lambda)$: This probabilistic polynomial time (PPT) algorithm takes as input a security parameter λ and outputs an evaluation key ek and a secret key sk.

- $\mathsf{Enc}(\mathsf{sk}, \mathsf{m})$: This PPT algorithm takes as input a secret key sk as well as a message m and returns a ciphertext ct.

- $\mathsf{Eval}(\mathsf{ek}, [\mathsf{ct}_i]_{i=1}^n, \mathcal{C})$: Given an evaluation key ek, ciphertexts $[\mathsf{ct}_i]_{i=1}^n$, and a circuit $\mathcal{C}$, this (non-)deterministic algorithm outputs a ciphertext ct.

- $\mathsf{Dec}(\mathsf{sk}, \mathsf{ct})$: Given a secret key sk and a ciphertext ct, this deterministic algorithm outputs a message m.

Informally, we say that an FHE scheme is correct, if the outcome of the evaluation of a circuit $\mathcal{C}$ on ciphertxts encrypting messages $\mathsf{m}_1, \ldots, \mathsf{m}_n$ decrypts to $\mathcal{C}(\mathsf{m}_1, \ldots, \mathsf{m}_n)$. Formally, we say that FHE is correct if for all security parameters $\lambda \in \mathbb{N}$, the circuits $\mathcal{C} : \mathcal{M}^n \to \mathcal{M}$ over the message space $\mathcal{M}$ of depth $\mathsf{poly}(\lambda)$, and all messages $[\mathsf{m}_i \in \mathcal{M}]_{i=1}^n$ we have

$$\Pr\left[\mathsf{Dec}(\mathsf{sk}, \mathsf{ct}_{\mathsf{out}}) = \mathcal{C}([\mathsf{m}_i]_{i=1}^n)\right] = 1 - \mathsf{negl}(\lambda),$$

where $\mathsf{sk} \leftarrow \mathsf{Setup}(\lambda)$, $\left[\mathsf{Dec}(\mathsf{sk}, \mathsf{ct}_i) = \mathsf{m}_i\right]_{i=1}^n$ and $\mathsf{ct}_{\mathsf{out}} \leftarrow \mathsf{Eval}(\mathsf{ek}, [\mathsf{ct}_i]_{i=1}^n, \mathcal{C})$. For efficiency, we require that Setup, Enc and Dec run in polynomial time in the security parameter, that is $\mathsf{poly}(\lambda)$, and Eval runs in $\mathsf{poly}(\lambda, |\mathcal{C}|)$. Finally, we say that a FHE scheme is compact if the size of the output of Eval is independent of the size of the circuit $\mathcal{C}$. More specifically, we require that $|\mathsf{Eval}(\mathsf{ek}, [\mathsf{ct}_i]_{i=1}^n, \mathcal{C})|$ is $\mathsf{poly}(\lambda, |\mathcal{M}|)$.

Indistinguishability Under Chosen Plaintext Attack. Let $\lambda \in \mathbb{N}$ be a security parameter and $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$ be a PPT adversary. We say that a FHE scheme is indistinguishable under chosen plaintext attack (IND-CPA)-secure if the probability

$$\Pr\left[\mathcal{A}_1(\mathsf{ct}_b, \mathsf{st}) = b \colon \begin{array}{c} \mathsf{sk} \leftarrow \mathsf{Setup}(\lambda), \\ (\mathsf{st}, \mathsf{m}_0, \mathsf{m}_1) \leftarrow \mathcal{A}_0^{\mathcal{O}(\mathsf{sk},.)}(\lambda), \\ b \leftarrow_R \{0,1\}, \\ \mathsf{ct}_b \leftarrow \mathsf{Enc}(\lambda, \mathsf{sk}, \mathsf{m}_b) \end{array}\right]$$

is at most $\mathsf{negl}(\lambda)$ for all PPT adversaries $\mathcal{A}$; the oracle $\mathcal{O}$ on input of a message m outputs $\mathsf{ct} \leftarrow \mathsf{Enc}(\lambda, \mathsf{sk}, \mathsf{m})$.

2.4 Generalized Learning with Errors

Definition 1 (GLWE)**.** Let $\mathcal{D}_{\mathsf{sk}}$ be a (not necessarily uniform) distribution over $\mathcal{R}_Q$, and $\sigma > 0$, $n \in \mathbb{N}$ and $N \in \mathbb{N}$ be a power-of-two, that are chosen according to a security parameter λ. For $\vec{\mathfrak{a}} \leftarrow_R \mathcal{R}_Q^n$, $\mathfrak{e} \leftarrow_R \mathcal{D}_{\mathcal{R},\sigma}$ and $\vec{\mathfrak{s}} \in \mathcal{D}_{\mathsf{sk}}^n$, we define a Generalized Learning with Errors (GLWE) sample of a message $\mathfrak{m} \in \mathcal{R}_Q$ with respect to $\vec{\mathfrak{s}}$, as

$$\mathsf{GLWE}_{\sigma,n,N,Q}(\vec{\mathfrak{s}}, \mathfrak{m}) = \begin{bmatrix} -\vec{\mathfrak{a}}^\top \cdot \vec{\mathfrak{s}} + \mathfrak{e} \\ \vec{\mathfrak{a}}^\top \end{bmatrix} + \begin{bmatrix} \mathfrak{m} \\ \vec{0} \end{bmatrix} \in \mathcal{R}_Q^{(n+1)}.$$

We say that the $\mathsf{GLWE}_{\sigma,n,N,Q}$-assumption holds if for any PPT adversary $\mathcal{A}$ we have

$$\left| \Pr\left[\mathcal{A}(\mathsf{GLWE}_{\sigma,n,N,Q}(\vec{\mathfrak{s}}, 0))\right] - \Pr\left[\mathcal{A}(\mathcal{U}_Q^{n+1})\right] \right| \leq \mathsf{negl}(\lambda)$$

where $\mathcal{U}_Q^{n+1}$ is the uniform distribution over $\mathcal{R}_Q^{n+1}$.

We denote a Learning with Errors (LWE) sample as $\mathsf{LWE}_{\sigma,n,Q}(\vec{s},\, \mathsf{m}) = \mathsf{GLWE}_{\sigma,n,1,Q}$, which is a special case of a GLWE sample where the ring is $\mathbb{Z}_Q[X]/(X+1)$. Similarly we denote a Learning with Errors over Rings (RLWE) sample as $\mathsf{RLWE}_\sigma(\mathfrak{s}, \mathsf{m}) = \mathsf{GLWE}_{\sigma,1,N,Q}$ which is the special case of an GLWE sample with $n = 1$. For simplicity, we omit to state the modulus and ring dimension for RLWE samples because we always use $\mathcal{R}_Q = \mathbb{Z}_Q[X]/(X^N + 1)$ where N is a power-of-two. For LWE samples, we will be switching between different moduli and different dimensions; hence we will indicate the current modulus in the notation. We sometimes leave the inputs unspecified and substitute them with a dot $(\cdot)$ when it is not necessary to refer to them within the scope of a function. We define the error of $\vec{c}$ as $\mathsf{Error}(\vec{c}, \mathsf{m}) = \langle \vec{c}, \vec{s} \rangle - \mathsf{m}$. Finally, we define the symbol $\Delta_{Q,t} = \lfloor \frac{Q}{t} \rceil$.

2.5 FHE Transpilation

Transpilation, also known as source-to-source compilation, is a process in which the source code written in one programming language is converted into the source code of another language. It converts code at similar levels of abstraction without changing the code's logic or functionality. In the context of FHE, transpilation mostly refers to converting high-level code implementing functionality in the unencrypted domain to FHE library code in the homomorphic realm. For example, the FHE transpiler [GSPH+21] converts a subset of C++ code to C++ or Rust, depending on the chosen output library.

Although FHE transpilation operates at a similar abstraction level with respect to the input and output programming language, the process itself closely resembles a compilation process as FHE libraries commonly only implement low-level operations on the encrypted data. Thus, FHE transpilation first converts high-level code to an IR where optimizations are performed. Then, the IR is further processed with instruction mapping or synthesis. For instruction mapping, each IR instruction is mapped to the low-level operations exposed by the FHE library, while for synthesis, hardware synthesis tools are used to convert the IR to a low-level circuit matching the low-level operations provided by the chosen library.

3 Contributions

In the following, we describe our theoretical contributions across the circuit synthesis process; some of these ideas also apply to FHE transpilation with instruction mapping which we note appropriately. We describe our synthesis toolchain top to bottom, afterward highlighting our contributions in the reverse starting with the modified FHE bootstrapping and ending with our new optimizations for the synthesis process.

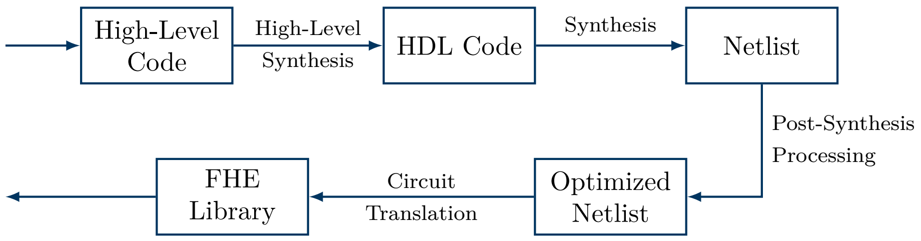

Figure 1: Our synthesis toolchain with different stages and their transitions.

3.1 Synthesis Toolchain

Our synthesis toolchain closely follows the structure and even reuses parts of the FHE transpiler [GSPH$^+$21], which currently provides the best publicly available solution to circuit synthesis. More specifically, high-level C++ code is converted to Verilog via the XLS framework[4]. Then, as in the transpiler, we synthesize Verilog to a Boolean circuit $\mathcal{C}_2$ via Yosys[5]. We configure synthesis in Yosys with pre-defined Yosys scripts specifing the exact steps Yosys should execute during synthesis. Instead of using the rather simple script of the transpiler, we create our own scripts based on the default Yosys script for LUT-based hardware. In contrast to the transpiler, there are two important differences:

- Using our new synthesis scripts, we can substitute certain subcircuits with custom gates. We use this to replace additions and subtractions with an optimized homomorphic gate structure using FHE-specific primitives (Subsection 3.7).
- We perform post-synthesis optimizations on the circuit $\mathcal{C}_2$ with HAL [Emb19], a netlist analysis tool, using its Python interface (Subsection 3.6, Subsection 3.8). Our Python scripts directly translates the optimized circuit to FHE-Deck code [fhe23].

In Figure 1, we depict the different stages of our toolchain including their transitions. All of our code, including tests and examples, is publicly available on GitHub [6].

3.2 Efficient Amortized Functional Bootstrapping

In the following, we describe our amortized bootstrapping algorithm, our instantiation of the FHE scheme and how we enhance homomorphic gate evaluation from the function f to a function $f_m : \mathbb{Z}_p \mapsto \mathbb{Z}_p^m$ at the cost of computing just one f. Since the amortized bootstrapping algorithm is based on FHEW-style bootstrapping algorithms [DM15, CGGI16a], we first give an informal overview of these algorithms in a black-box fashion limiting ourselves to their input-output relation, which is enough to understand the bootstrapping algorithm. Second, we elaborate on the practical problems with previous work on amortized bootstrapping and how we solve these problems.

Among the most efficient types of FHE schemes are schemes based on the FHEW-style bootstrapping algorithms [DM15]. There are multiple variants [CGGI16a, MS18, CIM19, BDF18, GBA21, Klu22b] and improvements [YXS$^+$21, CLOT21, LMP22, KS22] of these algorithms. The goal is to homomorphically compute the decryption function on a LWE ciphertext and compute an arbitrary function on the encrypted plaintext message along the way.

Suppose we have an LWE ciphertext $\vec{c}$ with $\langle \vec{c}, \vec{s} \rangle = M + e$, where $\vec{s}$ is the secret key, M the message, and e the error. The operation $\langle \vec{c}, \vec{s} \rangle$ can be realized within a cyclic algebraic group, more specifically, the group of rotations. The idea is to realize the rounding

[4]https://github.com/google/xls
[5]https://github.com/yosyshq/yosys
[6]https://github.com/Chair-for-Security-Engineering/fhewsyn

function $\lfloor\cdot\rceil$ by setting the elements of the vector such that messages are encoded in intervals of appropriate size to handle the noise term e. Bootstraping algorithms for FHEW-like schemes use the design pattern established by Alperin-Sheriff and Peikert [AP14] over polynomial rings. In particular, the observation first made by Ducas and Micciancio [DM15] is that in the ring $\mathbb{Z}[X]/(X^N+1)$, the product of any ring element with a root of unity (negacyclicly) rotates the coefficients of that ring element. In other words, given a polynomial $\mathfrak{w} = \sum_{i=0}^{N-1} w_i \cdot X^i$, we have

$$\mathfrak{w} \cdot X^y = \sum_{i=0}^{N-y-1} w_i \cdot X^{i+y} - \sum_{i=N-y}^{N-1} w_i \cdot X^{i+y-N}.$$

As part of the blind rotation procedure, we compute $\mathfrak{w} \cdot X^{\langle \vec{c}, \vec{s} \rangle} = \mathfrak{w} \cdot X^{M+e}$ homomorphically. Since computation takes place over RLWE ciphertexts, we obtain at the end of the blind rotation procedure an RLWE ciphertext of $\mathfrak{w} \cdot X^{M+e}$. Finally, Ducas and Micciancio [DM15] observe that given such RLWE ciphertext, one can extract a LWE ciphertext that encrypts the constant coefficent of the message; more specifically, the element $\mathfrak{w} \cdot X^{M+e}[1]$. The step is done via the $\mathsf{SampleExtract}$ procedure (see Table 1). Then, the final step is to choose the polynomial $\mathfrak{w}$ such that $\mathfrak{w} \cdot X^{M+e}[1]$ encodes the desired value, according to a given function $f : \mathbb{Z}_p \to \mathbb{Z}_p$, and switch the extracted LWE ciphertext to a LWE ciphertexts that is suitable for another bootstrapping step.

It is easy to see that with a bootstrapping algorithm which computes a function $f : \mathbb{Z}_p \mapsto \mathbb{Z}_p$, we can modify it to compute a function $f_m : \mathbb{Z}_p \mapsto \mathbb{Z}_p^m$ by grouping m distinct f, one for every output of f_m; this would require m bootstrapping invocations. The goal of amortized bootstrapping is to compute all m functions at the cost of only one bootstrapping. The general idea to amortize computation of different functions on the same input ciphertext is based on a previous work by Carpov, Izabachène, and Mollimard [CIM19], but in this paper, we significantly simplify execution of the idea. We give a version of the bootstrapping algorithm Algorithm 1 and provide a bound on the bootstrapping noise in Theorem 1.

The algorithm takes as input the blind rotation key brKey and the key switching key ksKey, a LWE ciphertext $\vec{c} \in \mathbb{Z}_Q^{N+1}$, and polynomials $\mathfrak{w}$ and $\vec{\mathfrak{v}}$. First, the algorithm switches the LWE key; this results in a LWE ciphertext $\vec{c}$ with a smaller dimension $n \in \mathbb{N}$ and a secret key from a smaller distribution, for instance binary, ternary, or Gaussian. Afterward, the algorithm switches the modulus from Q to $2N$ (recall that the roots of unity in the ring $\mathcal{R}_Q$ form an algebraic group of order $2N$). Then, we run $\mathsf{BlindRotate}$ which homomorphically computes $m_{\mathsf{acc}} \leftarrow \mathfrak{w} \cdot X^{\langle \vec{c}, \vec{s} \rangle}$. Finally, we execute a for-loop that multiplies the ciphertext with a polynomial from the vector $\vec{\mathfrak{v}}$. Consequently, we obtain and return a vector of ciphertexts $\left[\vec{c}_{\mathsf{out},i}\right]_{i=1}^{m}$. The i-th ciphertext in the returned vector encrypts the message $m_{\mathsf{acc}} \cdot \vec{\mathfrak{v}}[i]$. The problem when using this construction is that the multiplications by the elements of the vector $\vec{\mathfrak{v}}$ may blow up the error, ultimately destroying the ciphertext. This can happen if the norm bounds of elements in $\vec{\mathfrak{v}}$ are too large.

Previous Work

Carpov, Izabachène, and Mollimard [CIM19] suggest a polynomial factorization algorithm that takes a polynomial $\mathfrak{q}$ and returns factors $\mathfrak{w}_0$ and $\mathfrak{w}_1$ to tackle this problem. Aligned with our notation, we would set $\mathfrak{w}$ to $\mathfrak{w}_0$ and and insert $\mathfrak{w}_1$ into the vector $\vec{\mathfrak{v}}$. Essentially, their factorization algorithms works as follows: First, they set $\mathfrak{w}_0 = \sum_{i=1}^{N} X^i$ and $\mathfrak{w}_1 = \sum_{i=1}^{N} t'_i X^i$. Then, to compute the t'_i coefficients, they build and solve a large system of N linear equations. In practice, the polynomial degree is usually a power-of-two $N \geq 2^{11}$ and Gaussian elimination runs in cubic time in the number of variables. Hence we may expect that soving such system may take considerable time in practice.

```
Algorithm 1: $\mathsf{Bootstrap}(\mathsf{brKey}, \mathsf{ksKey}, \vec{c}, \mathfrak{w}, \vec{\mathfrak{v}})$
  Input:
    The blind rotation key $\mathsf{brKey}$;
    The key switching key $\mathsf{ksKey}$;
    A LWE ciphertext $\vec{c} \in \mathbb{Z}_q^{n+1}$;
    Polynomial $\mathfrak{w} \in \mathbb{Z}_Q^N$; and
    A vector of polynomials $\vec{\mathfrak{v}} \in \mathcal{R}_p^k$.
  Output: We consider two different versions of the bootstrapping algorithm: the
    classic variant and the amortized variant. For the classic variant, the vector $\vec{\mathfrak{v}}$ is
    empty ($k = 0$), and the algorithm returns a ciphertext $\mathsf{LWE}_{\cdot,N,Q}(\vec{s}, \mathfrak{w} \cdot X^{\langle \vec{c}, \vec{s} \rangle})$.
    For the amortized variant, the algorithm returns a vector of LWE ciphertexts
    $[\vec{c}_{\mathsf{out},i}]_{i=1}^k$, where $\vec{c}_{\mathsf{out},i} = \mathsf{LWE}_{\cdot,N,Q}(\vec{s}, \mathfrak{w} \cdot X^{\langle \vec{c}, \vec{s} \rangle} \vec{\mathfrak{v}}[i])$.
  begin
    Run $\vec{c}_{\mathsf{ksKey}} \leftarrow \mathsf{KeySwitch}(\vec{c}, \mathsf{ksKey}) \in \mathbb{Z}_Q^{n+1}$ ;
    Run $\vec{c}_{\mathsf{in}} \leftarrow \mathsf{ModSwitch}(\vec{c}_{\mathsf{ksKey}}, 2N) \in \mathbb{Z}_{2N}^{n+1}$ ;
    $\vec{\mathfrak{c}}_{\mathsf{acc}} \leftarrow \mathsf{BlindRotate}(\mathsf{brKey}, \mathfrak{w}, \vec{c}_{\mathsf{in}})$ ;
    for $i = 1 \dots k$ do
      Compute $\vec{\mathfrak{c}}_{\mathsf{acc},i} \leftarrow \vec{\mathfrak{c}}_{\mathsf{acc}} \cdot \vec{\mathfrak{v}}[i]$ ;
      Compute $\vec{c}_{\mathsf{out},i} \leftarrow \mathsf{SampleExtract}(\vec{\mathfrak{c}}_{\mathsf{acc},i})$ ;
    Return $[\vec{c}_{\mathsf{out},i}]_{i=1}^k$ ;
```

In their implementation [CIM19], the authors only test the bootstrapping for random rotation polynomials, and there is no implementation of the linear system solver. Nevertheless, for $N = 2^{14}$ [CIM19], we can roughly calculate that the number of modular multiplications in the Gaussian elimination algorithm[7] will be over 2^{40}. Furthermore, we need to assume that the system is solvable, and that elements of the matrix that we build for Gaussian elimination are invertible modulo Q which, with high probability, will not be the case if Q is a power-of-two as in many TFHE implementations inlcuding the bootstrapping implementation from [CIM19].

Our Contribution

In our work, we use a simpler and less involved solution that requires only linear time to build the polynomials. In fact, the polynomials in our approach can be constructed online without any significant slowdown during computation. Furthermore, for our solution to work, we do not need to assume that a system of linear equations is solvable modulo Q. As a consequence, we are not restricted to special moduli in contrast to [CIM19]. We observe that when we are only interested in extracting bits the polynomials in $\vec{\mathfrak{v}}$ are already sparse and of small infinity norm. Later, we compute a simple binary composition before running the new bootstrapping algorithm and computing the next LUT function. Concretely, we set $\mathfrak{w} = \Delta_{Q,p}$. Then, the polynomials in $\vec{\mathfrak{v}}$ are of the form

$$f(0) - \sum_{i=1}^{N} f(\lfloor i/2N \rceil) \cdot X^{N-i},$$

where $f : \mathbb{Z}_p \mapsto \{0, 1\}$.

[7]Recall that Gaussian elimination requires $N(N+1)/2$ divisions, $(2N^3 + 3N^2 - 5N)/6$ multiplications, and $(2N^3 + 3N^2 - 5N)/6$ subtractions modulo Q.

In the worst case, all coefficients of the polynomials in the vector $\vec{\mathfrak{v}}$ could be 1 and -1. Such polynomials, however, would not compute any interesting function, essentially the outcome of every bootstrapping would be the constant function computing 1. In practice, we have that at least one block must be equal to zero. Hence, the infinity norm of the polynomials in $\vec{\mathfrak{v}}$ can be bounded by $2N/3$.

Theorem 1 (Bootstrapping Correctness). *Let $\vec{\mathfrak{c}}_{\mathsf{acc}} \in \mathcal{R}_Q^2$ be an RLWE ciphertext returned by* $\mathsf{BlindRotate}$ *in an execution of Algorithm 1. Let $\mathfrak{e}_{\mathsf{acc}} = \mathsf{Error}(\vec{\mathfrak{c}}_{\mathsf{acc}}, \mathfrak{m}_{\mathsf{acc}})$ where $\mathfrak{m}_{\mathsf{acc}} = \Delta_{Q,p} \cdot X^M$, and $M = \langle \vec{c}_{\mathsf{in}}, \vec{s} \rangle \mod 2N$. Then, for all $i \in [k]$, we have*

$$\mathsf{SD}\big(\mathsf{Error}(\vec{\mathfrak{c}}_{\mathsf{acc},i}, \mathfrak{m}_{\mathsf{acc}} \cdot \vec{v}[i])\big) \leq \sqrt{\frac{2N}{3} \cdot \mathsf{Var}(\mathfrak{e}_{\mathsf{acc}})}$$

Proof. Recall that we assume that at most $2N/3$ of all coefficients in the polynomials in the $\vec{\mathfrak{v}}$ vector are non-zero and that $\mathfrak{e}_{\mathsf{acc}} \in \mathcal{R}_Q$. When multiplying the RLWE ciphertext $\vec{\mathfrak{c}}_{\mathsf{acc}}$ by $\vec{\mathfrak{v}}[i]$, we multiply the resulting error polynomial which is then equal to $\mathfrak{e}_{\mathsf{acc},i} = \mathfrak{e}_{\mathsf{acc}} \cdot \vec{\mathfrak{v}}[i]$. The d-th coefficient of $\mathfrak{e}_{\mathsf{acc},i}$ can be written as

$$\mathfrak{e}_{\mathsf{acc},i}[d] = \sum_{j=1}^{d} \mathfrak{e}_{\mathsf{acc}}[j] \cdot \vec{\mathfrak{v}}[i][d-i+1] + \sum_{j=d+1}^{N} \mathfrak{e}_{\mathsf{acc}}[j] \cdot \vec{\mathfrak{v}}[i][N+d-i+1].$$

Crucially, observe that the sum takes each coefficient from the polynomials only once and that at most $2N/3$ of the coefficents of $\vec{\mathfrak{v}}[i]$ are non-zero. All non-zero coefficents of $\vec{\mathfrak{v}}[i]$ are either 1 or -1. Hence, we have that

$$\mathsf{SD}(\mathfrak{e}_{\mathsf{acc},i}) \leq \sqrt{\sum_{i=1}^{2N/3} \mathsf{Var}(\mathfrak{e}_{\mathsf{acc}})} \leq \sqrt{\frac{2N}{3} \cdot \mathsf{Var}(\mathfrak{e}_{\mathsf{acc}})}.$$

□

3.3 The FHE Scheme

We now combine the algorithms from Table 1, containing subprocedures from previous work [DM15], and our new amortized bootstrapping from Algorithm 1 to a FHE system fitting the formal scheme definition in Subsection 2.3 and our model of computation. We provide high-level interfaces in FHE-Deck [fhe23] for all algorithms, making them easily accesible for researchers and developers.

Setup(λ). The setup algorithm consists of three main parts.

1. We choose the modulus Q, a power-of-two dimension N of the ring $\mathcal{R}_Q$, and a LWE dimension $n \in \mathbb{N}$ according to the security parameter λ. Then, we choose $\mathfrak{s} \in \mathcal{R}_Q$ for the RLWE key and set $\vec{s}_{\mathsf{ext}}$ to be the coefficient vector of $\mathfrak{s}$. We choose $\vec{s} \in \{0,1\}^n$ for the LWE key.
2. We run $\mathsf{ksKey} \leftarrow \mathsf{KSSetup}(\vec{s}, \vec{s}_{\mathsf{ext}}, \ell_{\mathsf{ksKey}}, \sigma_{\mathsf{ksKey}})$.
3. We run $\mathsf{brKey} \leftarrow \mathsf{BRSetup}(\vec{s}, \mathfrak{s}, \ell_{\mathsf{brKey}}, \sigma_{\mathsf{brKey}})$.

Finally, we set the evaluation key $\mathsf{ek} = (\mathsf{brKey}, \mathsf{ksKey})$ and the secret key $\mathsf{sk} = (\mathfrak{s}, \vec{s}_{\mathsf{ext}}, \vec{s})$.

Enc(sk, m). To encrypt a message $m' \in \mathbb{Z}_p$, we compute

$$\vec{c} \leftarrow \mathsf{LWE}_{\sigma,N,Q}(\vec{s}_{\mathsf{ext}}, m) \in \mathbb{Z}_Q^{N+1},$$

where $m = \frac{Q}{p} \cdot m' \in \mathbb{Z}_Q$.

Table 1: List of subprocedures commonly used in FHEW-like schemes [DM15].

	Key Switching
KSSetup	**Input:** Takes as input two LWE secret keys $\vec{s} \in \{0,1\}^n$, $\vec{s}_{\mathsf{ext}} \in \mathbb{Z}_Q^N$, a performance parameter $\ell_{\mathsf{ksKey}} \in \mathbb{N}$, and a standard deviation $\sigma_{\mathsf{ksKey}} \in \mathbb{R}$. **Output:** Generates a key switching key ksKey which consists of $N \cdot \ell_{\mathsf{ksKey}}$ $\mathsf{LWE}_{\sigma_{\mathsf{ksKey}},n,Q}(\vec{s},\cdot)$ ciphertexts.
KeySwitch	**Input:** Takes as input a key switching key ksKey and a $\mathsf{LWE}_{\cdot,N,Q}(\vec{s}_{\mathsf{ext}},\mathsf{m})$ sample of a message $\mathsf{m} \in \mathbb{Z}_Q$. **Output:** Returns a $\mathsf{LWE}_{\cdot,n,Q}(\vec{s},\mathsf{m})$ sample under the key $\vec{s}$ encoding the same message m. **Description:** The key switching process consists of $N \cdot \ell_{\mathsf{ksKey}}$ scalar multiplications in $\mathbb{Z}_Q$. The parameter ℓ_{ksKey} largely determines the time and space efficiency; that is, the smaller ℓ_{ksKey}, the faster the computation and the smaller the space complexity of the key material, but the bigger the noise induced by the key switching operation.
	Blind Rotation
BRSetup	**Input:** Takes as input the LWE key $\vec{s} \in \{0,1\}^n$, a RLWE key $\mathfrak{s} \in \mathcal{R}_Q$, a performance parameter $\ell_{\mathsf{brKey}} \in \mathbb{N}$, and a standard deviation σ_{brKey}. **Output:** Generates a blind rotation key brKey that consists of $2n\ell_{\mathsf{brKey}}$ $\mathsf{RLWE}_{\sigma_{\mathsf{brKey}}}(\mathfrak{s},\cdot)$ ciphertexts.
BlindRotate	**Input:** Takes as input a blind rotation key brKey, a LWE sample ct under modulus $2N$, and an accumulator $\mathsf{acc} = \mathsf{RLWE}(\mathfrak{s},\vec{m}_{\mathsf{acc}})$. **Output:** BlindRotate returns a sample $\mathsf{acc}_{\mathrm{out}} = \mathsf{RLWE}(\mathfrak{s},\vec{m}_{\mathsf{out}})$ with $\vec{m}_{\mathsf{out}} = \vec{m}_{\mathsf{acc}} \cdot X^{\langle \mathsf{ct},\vec{s}\rangle}$. The blind rotation process [DM15, CGGI16a] consists of $2n \cdot (\ell_{\mathsf{ksKey}} + 1)$ polynomial multiplications of elements in $\mathcal{R}_Q$. In particular, at the heart of a blind rotation algorithm, there is a ring version of the GSW cryptosystem [GSW13]. In this paper, we consider the concrete blind rotation from [CGGI16a], hence the number of polynomial multiplications. Similarly to key switching, the smaller the parameter ℓ_{brKey}, the faster the blind rotation algorithms and the smaller the blind rotation key (at the cost of larger noise).
	Other
ModSwitch	**Input:** Takes as input a LWE sample $\mathsf{LWE}_{\cdot,n,Q}(\vec{s},\Delta_{Q,p}\cdot\mathsf{m})$ and a modulus $q < Q$ with $\mathsf{m} \in \mathbb{Z}_p$. **Output:** Returns a LWE sample $\mathsf{LWE}_{\cdot,n,q}(\vec{s},\Delta_{q,p}\cdot\mathsf{m})$ under modulus q.
SampleExtract	**Input:** Takes as input a RLWE encryption $\mathsf{RLWE}_{\sigma_{\mathsf{brKey}}}(\mathfrak{s},\mathfrak{m})$ of a message $\mathfrak{m} \in \mathcal{R}_Q$. **Output:** Returns a LWE sample $\mathsf{LWE}_{\cdot,N,Q}(\vec{s}_{\mathsf{ext}},\mathsf{m})$ with $\mathsf{m} = \mathfrak{m}[1]$; that is, the LWE sample encodes the constant coefficient of the polynomial $\mathfrak{m}$.

Table 2: New parameter choices.

		BR Key				KS Key		
Set	Amort.	Q	N	ℓ_{BR}	SD	n	ℓ_{KS}	SD
`tfhe-11-bin`	$\times$	2^{48}	2^{11}	2	3.2	912	6	2^{26}
`tfhe-11-amort`	✓	2^{51}	2^{11}	3	3.2	950	6	2^{18}
`tfhe-12-amort`	✓	2^{50}	2^{12}	6	3.2	950	6	2^{18}

Eval(ek, $[\mathsf{ct}_i]_{i=1}^n, \mathcal{C}$). For the homomorphic circuit $\mathcal{C}$, we compute each individual gate

$$f_m\left(\sum_{i=1}^{k} x_i \cdot 2^{i-1} \in \mathbb{Z}_p\right) \in \mathbb{Z}_p^m,$$

where where $p = 2^k$ and k is the maximum fan-in of the gates in the circuit $\mathcal{C}$. While the output domain is $\mathbb{Z}_p^m$, we compute m bits in $\{0,1\}$ at an amortized cost of running just one bootstrapping from Algorithm 1.

Dec(sk, ct). To decrypt a LWE sample $\vec{c} \in \mathbb{Z}_Q^{N+1}$, we compute $\langle \vec{c}, \vec{s}_{\mathsf{ext}} \rangle = \frac{Q}{p} \cdot m' + e \in \mathbb{Z}_Q$, rescale and round the result obtaining

$$\left\lceil \frac{p}{Q}\left(\frac{Q}{p} \cdot m' + e\right) \right\rfloor = m'$$

if $|e| \leq \frac{Q}{2p}$.

3.4 New Parameters for Amortized Bootstrapping

We choose our parameter sets to target 128-bit security for the LWE and RLWE samples. The parameters are listed in Table 2. We estimate the security using the latest commit of the Lattice Estimator [APS15]. We also include a Python script to estimate the statistical security. In Table 2, we specify three parameter sets. The `tfhe-11-bin` parameter set is based on previous work by Kluczniak [Klu22a] and chosen for binary ciphertexts. The parameter sets `tfhe-11-amort` and `tfhe-12-amort` are new parameter sets to support amortized bootstraping for 3-bit and 4-bit LUTs, respectively.

We choose our parameters accoring to the following strategy. For the bootstrapping key, we choose two rings, one with dimension deg $= 2^{11}$ and one with dimensiton deg $= 2^{12}$. The idea is to choose the highest modulus such that the RLWE problem remains 128-bit secure according to the Lattice Estimator [APS15] and the modulus is below 51-bits to allow for faster multiplication of ring elements using the HEXL library [BKS$^+$21]. The larger ring gives us a larger group of the roots of unity, and we thus correctly process larger messages. For the LWE parameters, we set $n = 950$ for both rings and a binary secret key, since there are asymptotic reductions from binary LWE to LWE with uniform keys. Moreover, we stress that we choose the secret key vector uniformly from the binary distributions. In particular, we do not use sparse secret keys and we do not fix the hamming weight, but there are algorithms to handle other key distributions [DM15, LMK$^+$23]. However, these bootstrapping algorithms are usually slightly slower or require larger bootstrapping keys. Then, we choose the decomposition bases to minize the number of polynomial multiplications ℓ while at the same time preserving correctness with a probability of at most 2^{-80} for a faulty bootstraping.

Based on Table 2, we can conclude that `tfhe-11-bin` will require the least amount of polynomials multiplications. In particular, recall that the number of polynomial multiplications is given by $n \cdot (2 \cdot (\ell_{\mathsf{brKey}} + 1))$. For `tfhe-11-bin`, we need 5472 polynomial multiplicaiotns while for `tfhe-11-amort` and `tfhe-12-amort`, we need 7600 and 13300 polynomial multiplications, respectively. Furthermore, note that the ring dimension in `tfhe-12-amort` is doubled compared to the other parameter sets. Hence, a polynomial multiplication in this ring will be slower. We provide benchmarking results confirming these observations in Section 4.

3.5 Circuit Mapping

We now move from the bottom layer of our toolchain, the FHE scheme and its capabilities, to the post-synthesis layer above, that is to the netlist storing the synthesis output $\mathcal{C}_2$ and how the Boolean circuit is translated to a homomorphic circuit. Due to our efficient and secure amortized bootstrapping, we upgrade homomorphic gate capabilities and are able to compute m outputs within a single bootstrapping via f_m. This enables a more generic approach to circuit mapping, which we describe in the following.

Recall that we assume a $\mathcal{C}_2$ which only consists of gates $\mathcal{G}_n$. To convert a Boolean circuit $\mathcal{C}_2$ to a homomorphic circuit $\mathcal{C}$, we compose the gate inputs via the affine part of the FHE scheme and translate each gate $\mathcal{G}_n$ to a function f. Let $k = \max n$ be the maximum fan-in for all gates $\mathcal{G}_n$ in $\mathcal{C}_2$. We set $p = 2^k$ and compose the input to f as $x = \sum_{i=0}^{n-1} 2^i \cdot x_i$ homomorphically for encrypted values $x_i \in \mathbb{F}_2$. Theoretically, we could encrypt any value $x_i \in \mathbb{Z}_p$, but by encrypting single bits, we have $x \in \mathbb{Z}_p$ and thus x serving as valid input to f by definition. Finally, we map the gate functionality of $\mathcal{G}_n$ to a function f and compute the homomorphic equivalent as $f(x)$.

For example, consider the gate $\mathtt{AND} : \mathbb{F}_2^2 \to \mathbb{F}_2, (I_0, I_1) \mapsto I_0 \cdot I_1$. We define $f_{\mathtt{AND}}$ mapping the encrypted plaintexts $\{0, 1, 2, 3\}$ to $\{0, 0, 0, 1\}$, respectively, and $f_{\mathtt{AND}}(I_0 + 2I_1)$ computes an `AND` gate homomorphically. Note that, while the output domain of f is $\mathbb{Z}_p$, we will output only bits in $\{0, 1\}$. For a circuit $\mathcal{C}_2$ with gates $\mathcal{G}_{n,m}$, we trivially extend our mapping to a function f_m with m distinct functions f, one for every output bit.

3.6 Optimization: LUT Grouping

In the following, we introduce our first optimization performed during post-synthesis processing. Synthesis outputs $\mathcal{C}_2$ only consisting of single LUT gates $\mathcal{G}_{n,1}$, and our first optimization is constructing gates $\mathcal{G}_{n,m}$ to reduce the number of gates. As described in Subsection 3.5, we then trivially map these to homomorphic gates computing f_m via amortized bootstrapping, thus reducing the number of bootstrappings compared to before. The straightforward case is two gates $\mathcal{G}_n$ with exactly the same inputs in the same order which we can group as $\mathcal{G}_{n,2}$. There are, however, other opportunities enabling us to group different LUTs into a single LUT with multiple outputs. More specifically, we can group $\mathcal{G}_{n'}$ to $\mathcal{G}_n$ as long as (1) the inputs of $\mathcal{G}_{n'}$ are a subset of the inputs of $\mathcal{G}_n$ (or the other way around), and (2) grouping $\mathcal{G}_{n'}$ to $\mathcal{G}_n$ does not introduce cycles to $\mathcal{C}_2$.

Condition (1) follows relatively straightforward from our amortized bootstrapping technique: We compute the blind rotation for an encrypted input $x \in \mathbb{Z}_p$ to the function f_m which, for a gate $\mathcal{G}_{n,m}$ and encrypted values x_i, is composed as $x = \sum_{i=0}^{n-1} 2^i \cdot x_i$. We do have to be careful when grouping the individual f to a single f_m, however, as the x_i for all gates have to be in the same position i within a grouping. During post-synthesis optimization, we thus first extend each gate in the grouping to have all x_i as input, adding new inputs as required without modifying the gate output. Second, we order the inputs according to their unique IDs in HAL always composing the same x; hence, we compute $\mathcal{G}_{n,m}$ correctly.

Condition (2) ensures that $\mathcal{C}_2$ still matches our definition from Subsection 2.2 preventing cycles in $\mathcal{C}_2$ as we otherwise could not evaluate it anymore. Consider the following scenario: Two inputs to some gates $\mathcal{G}_2$ and $\mathcal{G}_3$ are the same while the third input to $\mathcal{G}_3$ is the output of

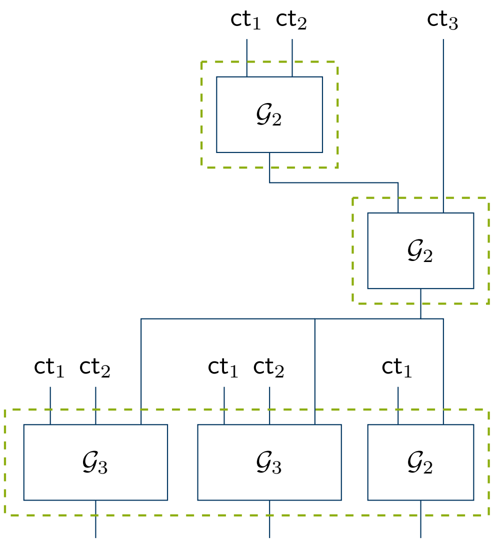

Figure 2: Grouping gates in an example circuit by matching the inputs while preserving topological order, each group is surrounded by a dashed line in green. The gate $\mathcal{G}_2$ in the top row cannot be grouped with the bottom row as it would introduce a cycle.

$\mathcal{G}_2$; here, grouping would introduce a cycle. This example generalizes to more complicated settings with additional gates in between and detecting such cases specifically is rather difficult; it would require checking all inputs to gates which are directly or indirectly connected to the output of a grouping candidate. We therefore choose a simpler, although non-optimal method, and only group $\mathcal{G}_{n'}$ to $\mathcal{G}_n$ if $\mathcal{G}_n$ appears before $\mathcal{G}_{n'}$ in the topological order and the inputs of $\mathcal{G}_{n'}$ are a subset of $\mathcal{G}_n$ (but not the other way around).

An example for LUT grouping where both conditions are at play is depicted in Figure 2. Here, the two gates $\mathcal{G}_3$ in the bottom row have the same inputs and hence can be grouped. Additionally, we can add $\mathcal{G}_2$ in the bottom row to the same group, as its inputs are a subset (we assume it appears later in the sorted graph). $\mathcal{G}_2$ in the middle row has a unique input combination and is its own group. As for the top row, the inputs of $\mathcal{G}_2$ are a subset of the inputs from the bottom row. But, adding it to the group at the bottom would introduce a cycle, and it thus remains its own group.

3.7 Optimization: Adder Substitution

Adder substitution, our second optimization, is based on the following observation: Using LUTs to realize additions is generally more costly than using the native addition capabilities of the FHE scheme (the same idea can be applied to transpilation based on instruction mapping). For example, with $p = 2^3$, a two-bit addition with LUT grouping requires two amortized bootstrappings while adder substitution reduces it to one bootstrapping as depicted in Figure 3.

At its core, adder substitution takes place during synthesis moving us up another layer in the toolchain, but for simplicity, we split its implementation into two steps, one performed during synthesis in Yosys and one during post-synthesis processing in HAL: First, we hook into the synthesis process mapping additions and subtractions from the Yosys IR to custom adder gates. In the second step, we group these adders depending on the plaintext modulus p. In the following, we explore both steps in more detail.

During synthesis, we instruct Yosys to use single-bit full adder gates

$$\texttt{FA} : \mathbb{F}_2^3 \to \mathbb{F}_2^2,\ \ (x, y, c_i) \mapsto (s, c_o) = (x + y + c_i, x \cdot y + c_i \cdot (x + y))$$

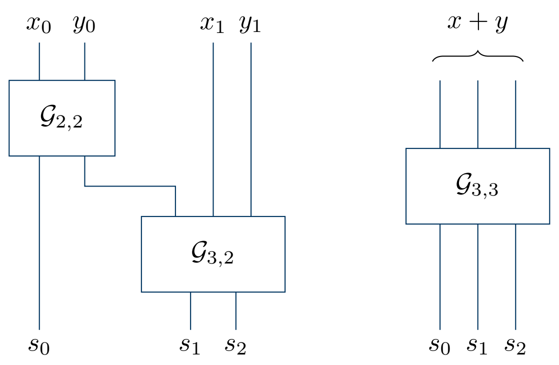

Figure 3: Comparing a two-bit addition $s = x + y$ with a three-bit output based on LUTs for computation (left) and based on ciphertext addition with decomposition (right) with $p = 2^3$. The former requires two amortized bootstrappings while the latter requires one.

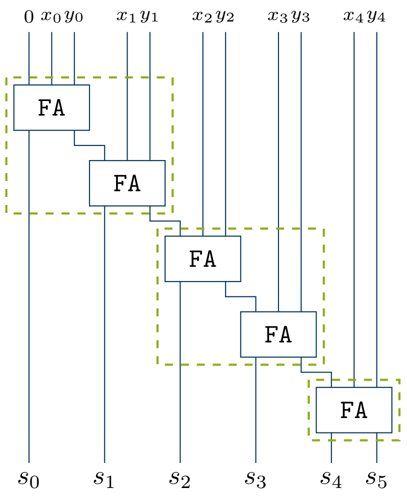
0 x_0 y_0 x_1 y_1 x_2 y_2 x_3 y_3 x_4 y_4
FA
FA
FA
FA
FA
s_0 s_1 s_2 s_3 s_4 s_5

Figure 4: Grouping FA gates for a 5-bit addition $s = x + y$ with $p = 2^3$ and $c_i = 0$, each grouping is marked by a dashed green box.

for addition. Since an addition $x + y$ in the Yosys IR has arbitrary width, we start with a single-bit addition by adding the least significant bits x_0 and y_0 with $c_i = 0$, building up to an arbitrary-width addition by connecting the FA gates via the carry bits as required. For a subtraction $x - y$, we place inversion gates

$$\mathtt{INV} : \mathbb{F}_2 \to \mathbb{F}_2, \quad I \mapsto O = I + 1$$

for y and set $c_i = 1$ for the least significant bit, this corresponds to a negation in the two's complement representation. Note that, for a ciphertext ct, computing inversion as $1 - \mathsf{ct}$ does not require a bootstrapping and is considered free (see also Subsection 3.8).

During post-synthesis processing, we identify adder chains and group them into groups of $\log p - 1$ gates. Using $\log p - 1$ gates ensures that addition does not wrap around modulo p, which we require for correctness. The FA for the least significant bit serves as root of a chain and is easy to identify with a constant $c_i \in \mathbb{F}_2$. Figure 4 contains an example for a 5-bit addition with $p = 2^3$.

As an additional optimization, we exclude constant addition or subtraction from adder substitution. The reason is rather straightforward: Since the individual bits of one operand are known at synthesis time, they can be folded with surrounding logic very efficiently and using our custom `FA` gate would result in worse performance. We evaluate the effect of constant bits on adder substitution in more detail in Section 4.

3.8 Additional Optimizations

Beyond the improvements achieved by LUT grouping and adder substituions, we employ three further minor optimizations:

- We use the well-known fact that inversion of a ciphertext `ct` encrypting a value in $\mathbb{F}_2$ corresponds to the affine function $1 - \texttt{ct}$, also known as *free inversion* optimization; hence, no bootstrapping is required (this is also applicable to transpilation based on instruction mapping).

- We employ *LUT conversion*, during which we convert `LUT1` gates to affine functions on a ciphertext without performing a bootstrapping. The mapping is trivial as a `LUT1` either computes the identity function or an inversion.

- *Type sorting* is applied during the topological sorting of the circuit graph. We split the gates according to their depth and sort each layer with respect to gate types. Most importantly, we sort the LUTs according to the number of inputs in descending order. This ensures that we do not miss out on LUT groupings within a layer, as these cannot introduce cycles to the graph.

4 Evaluation

In the following, we evaluate our contributions with a multitude of examples; most designs stem from the FHE transpiler [GSPH+21], the remaining ones are our own addition. Overall, we use the following examples: Compute the sum of 3-bit, 4-bit or 32-bit integers, respectively (`add3`, `add4`, `add32`); calculate the addition, subtraction or multiplication of two 16-bit integers (`calc`); compute the sum of a 4-bit integer with a constant (`const4`); apply blurring to a small image (`img-blur`); apply a ricker wavelet transformation to a small image (`img-ricker`); apply sharpening to a small image (`img-sharp`); compute a rectified linear unit function (`relu`); compute the square root of a 16-bit integer (`sqrt`); reverse an array of up to eight characters (`strrev`); compute the sum over a one-dimensional array of structs (`structs1d`); compute the sum over a three-dimensional array of structs (`structs3d`); compute the sum over a three-dimensional array of integers (`sum3d`).

For these examples, we target several architectures:

- `prev`: The Yosys script from the FHE transpiler without optimizations [GSPH+21].

- `noX`: Our Yosys script with a maximum LUT-size of X bit without optimizations.

- `noXfa`: `noX` with adder substitution.

- `optX`: `noX` with all optimizations except adder substitution.

- `optXfa`: `optX` with adder substitution.

In Table 3, we list the number of bootstrappings for all examples with a wide variety of architectures.

Table 3: Results for all examples with different synthesis processes and optimization techniques; each row contains the number of bootstrappings.

	`prev`	`no2`	`no2fa`	`opt2`	`opt2fa`	`no3`	`no3fa`	`opt3`	`opt3fa`
`add3`	12	12	3	7	3	6	2	3	2
`add4`	17	17	4	10	4	8	2	4	2
`add32`	165	163	32	102	32	63	32	32	16
`calc`	948	884	857	655	653	491	479	351	358
`const4`	4	4	4	2	2	3	3	1	1
`img-blur`	318	316	74	193	74	146	37	90	37
`img-ricker`	342	334	90	212	89	145	48	95	47
`img-sharp`	183	194	76	125	65	94	43	57	35
`relu`	30	15	15	15	15	15	15	15	15
`sqrt`	195	224	334	183	307	102	179	85	162
`strrev`	668	805	805	744	744	364	364	341	341
`structs1d`	159	157	31	98	31	61	16	31	16
`structs3d`	1091	1056	217	654	217	460	112	246	112
`sum3d`	954	926	203	553	203	396	105	199	105

4.1 Evaluating LUT Grouping

To evaluate LUT grouping, we compare the results for architectures `no2` and `no3` with architectures `opt2` and `opt3`, respectively. For the chosen use cases, we reduce the number of bootstrappings by up to 66 % and, for most use cases, by at least 30 %. On average, using LUT grouping reduces the number of bootstrappings by almost 35 %. Due to its effectiveness with only a slight overhead in the parameters, we recommend to always enable LUT grouping to improve performance for FHE circuit synthesis.

4.2 Evaluating Adder Substitution

For adder substitution, we compare the results for `no2` and `no3` with `no2fa` and `no3fa`, respectively. Here, results are more mixed than before and we can make multiple interesting observations. As expected, for use cases without additions such as `strrev`, the number of bootstrappings stays the same. For use cases with lots of additions, however, improvements are much more drastic compared to before, and the number of bootstrappings is reduced by up to 80 % and, on average, we reduce the number of bootstrappings by almost 44 %.

Nevertheless, in the use case `sqrt`, we actually perform worse with our optimization. In Listing 1, we extract the culprits for this result. The subtractions performed depend on many constant bits for the second input. However, we cannot detect this during synthesis in Yosys as the built-in constant folding is not aggressive enough to mark the appropriate subset of input bits as constant. Since folding additions and subtractions with many constant input bits into LUTs is relatively efficient, the default LUT optimizations outperform adder substitution.

Overall, as efficient arithmetic is one of the main selling points of word-based schemes compared to Boolean-based schemes, we believe that the above optimization can be an important step to gain efficient arithmetic while keeping flexibility for FHE computations. For now, we suggest transpiling circuits with and without adder substitutions and choosing the better performing option (this can happen automatically); we discuss possibilities for future work in Subsection 5.4 to avoid such scenarios.

```
[...]
assign sub_1935 = sel_1912 - \
    ({1'h0, sel_1910, 13'h0000} | 16'h1000);
[...]
assign sub_1962 = sel_1939 - \
    ({1'h0, sel_1937, 11'h000} | 16'h0400);
[...]
assign sub_1989 = sel_1966 - \
    ({1'h0, sel_1964, 9'h000} | 16'h0100);
[...]
assign sub_2016 = sel_1993 - \
    ({1'h0, sel_1991, 7'h00} | 16'h0040);
[...]
assign sub_2043 = sel_2020 - \
    ({1'h0, sel_2018, 5'h00} | 16'h0010);
[...]
assign sub_2070 = sel_2047 - \
    ({1'h0, sel_2045, 3'h0} | 16'h0004);
[...]
```

Listing 1: Subtractions in the `sqrt` example containing constant bits.

Table 4: Performance of the new parameter sets.

Set	Boot. [s]	brKey [MB]	ksKey [MB]
`tfhe-11-bin`	0.24	44.8	78.5
`tfhe-11-amort`	0.29	81.7	81.8
`tfhe-12-amort`	0.58	217.9	163.6

4.3 Benchmarking

We run benchmarks on Ubuntu 20.04.4 with an Intel Core i7-11850H central processing unit (CPU) at 2.50 GHz featuring 8 cores. Our system has 16 GiB of available memory. For the new parameter sets, we summarize our results in Table 4 regarding runtime and memory consumption, confirming our observations from Subsection 3.4. For `prev`, we use the parameter sets `tfhe-11-bin` while we use `tfhe-11-amort` for `opt2fa`, `opt3`, and `opt3fa`. For `opt4`, we use `tfhe-12-amort`.

Our benchmarking results for all examples are summarized in Table 5. As expected, the execution time highly correlates with the number of bootstrappings. In most cases, we receive the best speed-ups for 3-bit LUTs. But, there are exceptions such as the `relu` example where 2-bit LUTs perform the best. Using 4-bit LUTs is generally not worth due to the increased polynomial degree and thus the longer bootstrapping time.

5 Discussion

In the following, we put our work in the context of current research, first discussing related work on FHEW-like implementations followed by related work on FHE circuit synthesis. Afterward, we discuss limitations of our optimizations and explore multiple opportunities for future work to further optimize the tool-based generation of circuits for FHE.

5.1 Related Work on FHEW-like Implementations

In Table 6, we roughly compare different libraries implementing FHEW-like schemes. A distinguishing feature of FHE-Deck is the support for correct and secure parameter sets

Table 5: Execution time in seconds including the corresponding rounded speed-ups compared to `prev` for all examples and a selection of synthesis processes with optimizations on our benchmarking setup, an Ubuntu 20.04.4 with an Intel Core i7-11850H CPU.

	`prev`	`opt2fa`		`opt3`		`opt3fa`		`opt4`	
	t	t'	t/t'	t'	t/t'	t'	t/t'	t'	t/t'
`add3`	2.40	0.91	2.6	1.31	1.8	1.01	2.4	2.79	0.9
`add4`	3.33	1.23	2.7	1.75	1.9	1.14	2.9	3.88	0.9
`add32`	30.90	9.69	3.1	13.91	2.2	9.13	3.3	36.44	0.8
`calc`	146.50	107.60	1.4	104.30	1.4	106.00	1.4	149.60	1.0
`const4`	0.86	0.54	1.6	0.50	1.7	0.59	1.5	1.38	0.6
`img-blur`	56.00	17.00	3.3	30.82	1.8	15.70	3.6	61.71	0.9
`img-ricker`	59.88	19.38	3.1	31.98	1.9	18.67	3.2	64.19	0.9
`img-sharp`	32.39	13.26	2.4	19.34	1.7	12.90	2.5	36.60	0.9
`relu`	5.94	3.53	1.7	5.50	1.1	5.43	1.1	12.89	0.5
`sqrt`	33.23	52.51	0.6	25.64	1.3	50.77	0.7	41.63	0.8
`strrev`	115.10	123.70	0.9	103.10	1.1	103.60	1.1	171.50	0.7
`structs1d`	30.01	9.42	3.2	13.38	2.2	9.00	3.3	36.56	0.8
`structs3d`	193.20	53.68	3.6	88.20	2.2	50.62	3.8	202.80	1.0
`sum3d`	167.10	46.33	3.6	71.13	2.3	43.86	3.8	161.90	1.0

for amortized bootstrapping. Both FHE-Deck and TFHE-rs support different algorithms for functional bootstrapping (also known as programmable bootstrapping), but without amortization. In particular, FHE-Deck supports the full domain bootstrapping algorithm based on work by Liu, Micciancio, and Polyakov [LMP22] while THFE-rs supports the algorithm by Chilotti et al. [CLOT21]. Moreover, both libraries support simple padding-based functional bootstrapping. The TFHE library [CGGI16b] supports only binary gates. Open-FHE [ABBB$^+$22], which is derived from PALISADE [PAL21], implements binary as well as full domain bootstrapping based on work by Liu, Micciancio, and Polyakov [LMP22].

There are implementations for LUT evaluation [CGGI17]. However, the techniques are vastly different, as the authors evaluate a LUT on so-called RGSW ciphertexts requiring numerous bootstrapping invocations (the number depends on the chosen parameters) for each output bit of the LUT. In contrast, we focus on computing LUTs using a single bootstrapping invocation. Finally, the amortized bootstrapping technique by Micciancio and Sorrel [MS18] and its improvement [GPvL23] compute functional bootstrapping over many input ciphertexts at a cheaper cost than bootstrapping ciphertexts separately. In particular, the functional difference is that we amortize computation for many output functions on the same input ciphertexts while they compute the same functions on multiple ciphertexts. Combining both amortization techniques in a practical way is an interesting open problem for future work.

Amortized Bootstrapping in TFHE

The TFHE library implements amortized bootstrapping [CIM19] in a separate branch. However, the method is not integrated into a usable interface, and parameters are hardcoded in the low-level code mainly for benchmarking purposes. In particular, the performance tests do not switch the key back to the LWE form, thereby disallowing to use the implementation in applications. Furthermore, as we addressed in Subsection 3.2, the implementation only tests the performance of bootstrapping itself for randomly chosen polynomials $\mathfrak{w}$ and $\vec{\mathfrak{v}}$. Unfortunately, there is no implementation of the procedures that generate these

Table 6: Functionality Comparison of Different FHE Libraries for FHEW-like schemes. For functional bootstrapping, we denote as ○ the plain FHEW/TFHE algorithm to evaluate boolean gates. By ◐, we denote a full domain functional bootstrapping algorithm [YXS+21, CLOT21,LMP22,KS22]. By ●, we denote support for our improved funtional bootstrapping algorithm. In this comparison, a high-level interface for functional bootstrapping is required.

Library	Language	Bootstrapping	
		Functional	Amortized
FHE-Deck	C++	●	✓
Open-FHE	C++	◐	×
TFHE	C++	○	×
tfhe-rs/CONCRETE	Rust	●	×

polynomials. Moreover, the bootstrapping parameters chosen in the implementation do not allow generating the rotation polynomials as suggested [CIM19] because the ciphertext moduli are a powers-of-two. Hence the system of linear equations will not be solvable.

To fix the problem, we may choose prime power moduli. However, even with prime power moduli, the method for the suggested parameters requires over 2^{40} multiplications and modulus reductions. While 2^{40} isn't considered cryptographically hard, it is a considerable time in practice. Additionally, we note that previous parameters [CIM19] use a much larger ring of dimension 2^{14}, which may be justified by the larger target precision. However, the choice of the LWE dimension seems to be controversial with respect to security. In particular, the dimension n is only 803 with a sparse binary secret key of hamming weight 63. Finally, the online bootstrapping algorithm used in [CIM19] is the same as Algorithm 1. Hence differences in running time may be attributed to differences in implementation, benchmarking setups, or parameter choices.

5.2 Related Work on Circuit Synthesis

There are a couple of previous works on circuit synthesis for FHEW-like schemes, the previously mentioned FHE transpiler [GSPH+21] as well as two earlier works named Cingulata, originally released under the name Armadillo [CDS15], and the E3 framework [CMG+18]. In Cingulata, the authors divide their toolchain in three parts: the front-end translating C++ code to a Boolean circuit, the middle-end optimizing the circuit and the back-end transpiling the circuit to a FHE library. The mapping from C++ to a Boolean circuit in the front-end defers optimizations to the middle-end based on ABC [Mis], which we also use as part of our toolchain via Yosys. No other optimizations are performed. The E3 framework also uses hardware tooling for transpilation, however, no details regarding optimizations are available in their publication.

The FHE transpiler currently improves upon all previously known work and thus serves as a good foundation to evaluate new research ideas. Common compiler optimizations such as constant folding or dead code elimination are performed in the XLS-based high-level synthesis layer. However, to the best of our knowledge, the only FHE-specific optimization currently performed is rather trivial treating inversion as free for Boolean-based circuits. For LUT-based circuits, there is currently no post-processing implementing this optimization.

5.3 Limitations

There are some limitations for our proposed optimizations. First, as highlighted in Subsection 4.2, using adder substitution can result in worse performance when the inputs contain many constant bits which can be non-detectable using our current approach. Therefore, a user has to manually check for the better circuit. One solution and useful contribution in future work would be improving constant bit detection and constant folding in the Yosys IR. Second, although using three-bit ciphertexts tends to provide the largest speed-ups, sometimes other bit sizes can be more beneficial such as using two-bit ciphertexts for the `relu` example. Exploring the root causes and detecting such cases, especially if also done for subcircuits, would further improve circuit synthesis for FHE.

5.4 Future Work

An in our opinion important observation is that the current state-of-the-art in bootstrapping for FHEW-like schemes can still be improved upon. We also believe that there is still room for improvement regarding performance for TFHE implementations as well as regarding usability for currently available libraries, including, but not limited to, FHE-Deck. As for circuit synthesis, our work is a first step in FHE-specific optimizations and we believe that there is a multitude of other possibilities making automatically generated circuits more efficient. For example, amortized bootstrapping greatly benefits from LUTs with the same inputs which current hardware tooling is not optimizing for.

Another important contribution would be providing high-level implementations for a representative set of use cases serving as foundation to better evaluate optimizations (similar to compiler benchmarking where, for specific use cases, the performance of the compiler is evaluated considering compilation time and output quality). This is necessary as optimizations are often heuristic in nature and hard to evaluate generically. Overall, we are looking forward to future work in the area of circuit synthesis for Boolean-based FHE schemes.

6 Conclusion

In this work, we improve performance and usability of FHEW-like schemes by extending the current state-of-the-art in bootstrapping as well as circuit synthesis. To improve performance, we significantly simplify the bootstrapping idea proposed by Carpov, Izabachène, and Mollimard [CIM19] and provide the first efficient implementation for amortized bootstrapping. With respect to usability, we provide new and secure parameter sets for multi-bit encryptions, which can be used by researchers and developers alike, and implement a high-level interface for amortized bootstrapping in the open-source library FHE-Deck [fhe23].

We provide a generalized model for mapping Boolean circuits to homomorphic circuits and introduce the first non-trivial FHE-specific optimizations for generating circuits from high-level code: LUT grouping and adder substitution. Using LUT grouping, generated circuits require almost 35 % less bootstrappings on average and adder substitution reduces the number of required bootstrappings by up to 80 %. Overall, our performance improvements result in up to 3.8× faster execution times compared to previous synthesized circuits with state-of-the-art methods.

6.1 Acknowledgements

Johannes Mono and Tim Güneysu are supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972.

References

[ABBB$^+$22] Ahmad Al Badawi, Jack Bates, Flavio Bergamaschi, David Bruce Cousins, Saroja Erabelli, Nicholas Genise, Shai Halevi, Hamish Hunt, Andrey Kim, Yongwoo Lee, Zeyu Liu, Daniele Micciancio, Ian Quah, Yuriy Polyakov, Saraswathy R.V., Kurt Rohloff, Jonathan Saylor, Dmitriy Suponitsky, Matthew Triplett, Vinod Vaikuntanathan, and Vincent Zucca. Openfhe: Open-source fully homomorphic encryption library. In *Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, WAHC'22, pages 53–63, New York, NY, USA, 2022. Association for Computing Machinery.

[AP14] Jacob Alperin-Sheriff and Chris Peikert. Faster bootstrapping with polynomial error. In Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology – CRYPTO 2014, Part I*, volume 8616 of *Lecture Notes in Computer Science*, pages 297–314, Santa Barbara, CA, USA, August 17–21, 2014. Springer, Heidelberg, Germany.

[APS15] Martin R. Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. *Journal of Mathematical Cryptology*, 9(3):169–203, 2015.

[BDF18] Guillaume Bonnoron, Léo Ducas, and Max Fillinger. Large FHE gates from tensored homomorphic accumulator. In Antoine Joux, Abderrahmane Nitaj, and Tajjeeddine Rachidi, editors, *AFRICACRYPT 18: 10th International Conference on Cryptology in Africa*, volume 10831 of *Lecture Notes in Computer Science*, pages 217–251, Marrakesh, Morocco, May 7–9, 2018. Springer, Heidelberg, Germany.

[BGV14] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Trans. Comput. Theory*, 6(3):13:1–13:36, 2014.

[BKS$^+$21] Fabian Boemer, Sejun Kim, Gelila Seifu, Fillipe DM de Souza, Vinodh Gopal, et al. Intel HEXL (release 1.2). https://github.com/intel/hexl, September 2021.

[Bra12] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical GapSVP. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 868–886, Santa Barbara, CA, USA, August 19–23, 2012. Springer, Heidelberg, Germany.

[CDS15] Sergiu Carpov, Paul Dubrulle, and Renaud Sirdey. Armadillo: A compilation chain for privacy preserving applications. In *Proceedings of the 3rd International Workshop on Security in Cloud Computing*, SCC '15, page 13–19, New York, NY, USA, 2015. Association for Computing Machinery.

[CGGI16a] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *Advances in Cryptology – ASIACRYPT 2016, Part I*, volume 10031 of *Lecture Notes in Computer Science*, pages 3–33, Hanoi, Vietnam, December 4–8, 2016. Springer, Heidelberg, Germany.

[CGGI16b] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: Fast fully homomorphic encryption library, August 2016. https://tfhe.github.io/tfhe/.

[CGGI17] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Faster packed homomorphic operations and efficient circuit bootstrapping for TFHE. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology – ASIACRYPT 2017, Part I*, volume 10624 of *Lecture Notes in Computer Science*, pages 377–408, Hong Kong, China, December 3–7, 2017. Springer, Heidelberg, Germany.

[CIM19] Sergiu Carpov, Malika Izabachène, and Victor Mollimard. New techniques for multi-value input homomorphic evaluation and applications. In Mitsuru Matsui, editor, *Topics in Cryptology – CT-RSA 2019*, volume 11405 of *Lecture Notes in Computer Science*, pages 106–126, San Francisco, CA, USA, March 4–8, 2019. Springer, Heidelberg, Germany.

[CKKS17] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yong Soo Song. Homomorphic encryption for arithmetic of approximate numbers. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology – ASIACRYPT 2017, Part I*, volume 10624 of *Lecture Notes in Computer Science*, pages 409–437, Hong Kong, China, December 3–7, 2017. Springer, Heidelberg, Germany.

[CLOT21] Ilaria Chillotti, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. Improved programmable bootstrapping with larger precision and efficient arithmetic circuits for TFHE. In Mehdi Tibouchi and Huaxiong Wang, editors, *Advances in Cryptology – ASIACRYPT 2021, Part III*, volume 13092 of *Lecture Notes in Computer Science*, pages 670–699, Singapore, December 6–10, 2021. Springer, Heidelberg, Germany.

[CMG$^+$18] Eduardo Chielle, Oleg Mazonka, Homer Gamil, Nektarios Georgios Tsoutsos, and Michail Maniatakos. E3: A framework for compiling c++ programs with encrypted operands. Cryptology ePrint Archive, Paper 2018/1013, 2018. https://eprint.iacr.org/2018/1013.

[DM15] Léo Ducas and Daniele Micciancio. FHEW: Bootstrapping homomorphic encryption in less than a second. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 617–640, Sofia, Bulgaria, April 26–30, 2015. Springer, Heidelberg, Germany.

[Emb19] Embedded Security Group. HAL - The Hardware Analyzer. https://github.com/emsec/hal, 2019.

[fhe23] Fhe-deck. https://github.com/FHE-Deck, September 2023.

[FV12] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144, 2012. https://eprint.iacr.org/2012/144.

[GBA21] Antonio Guimarães, Edson Borin, and Diego F. Aranha. Revisiting the functional bootstrap in tfhe. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(2):229–253, Feb. 2021.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In Michael Mitzenmacher, editor, *41st Annual ACM Symposium on Theory of Computing*, pages 169–178, Bethesda, MD, USA, May 31 – June 2, 2009. ACM Press.

[GPvL23] Antonio Guimarães, Hilder V. L. Pereira, and Barry van Leeuwen. Amortized bootstrapping revisited: Simpler, asymptotically-faster, implemented. Cryptology ePrint Archive, Report 2023/014, 2023. https://eprint.iacr.org/2023/014.

[GSPH+21] Shruthi Gorantala, Rob Springer, Sean Purser-Haskell, William Lam, Royce Wilson, Asra Ali, Eric P. Astor, Itai Zukerman, Sam Ruth, Christoph Dibak, Phillipp Schoppmann, Sasha Kulankhina, Alain Forget, David Marn, Cameron Tew, Rafael Misoczki, Bernat Guillen, Xinyu Ye, Dennis Kraft, Damien Desfontaines, Aishe Krishnamurthy, Miguel Guevara, Irippuge Milinda Perera, Yurii Sushko, and Bryant Gipson. A general purpose transpiler for fully homomorphic encryption. Cryptology ePrint Archive, Report 2021/811, 2021. https://eprint.iacr.org/2021/811.

[GSW13] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology – CRYPTO 2013, Part I*, volume 8042 of *Lecture Notes in Computer Science*, pages 75–92, Santa Barbara, CA, USA, August 18–22, 2013. Springer, Heidelberg, Germany.

[Klu22a] Kamil Kluczniak. NTRU-ν-um: Secure fully homomorphic encryption from NTRU with small modulus. Cryptology ePrint Archive, Report 2022/089, 2022. https://eprint.iacr.org/2022/089.

[Klu22b] Kamil Kluczniak. NTRU-v-um: Secure fully homomorphic encryption from NTRU with small modulus. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022: 29th Conference on Computer and Communications Security*, pages 1783–1797, Los Angeles, CA, USA, November 7–11, 2022. ACM Press.

[KS22] Kamil Kluczniak and Leonard Schild. Fdfb: Full domain functional bootstrapping towards practical fully homomorphic encryption. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2023(1):501–537, Nov. 2022.

[LMK+23] Yongwoo Lee, Daniele Micciancio, Andrey Kim, Rakyong Choi, Maxim Deryabin, Jieun Eom, and Donghoon Yoo. Efficient FHEW bootstrapping with small evaluation keys, and applications to threshold homomorphic encryption. In Carmit Hazay and Martijn Stam, editors, *Advances in Cryptology – EUROCRYPT 2023, Part III*, volume 14006 of *Lecture Notes in Computer Science*, pages 227–256, Lyon, France, April 23–27, 2023. Springer, Heidelberg, Germany.

[LMP22] Zeyu Liu, Daniele Micciancio, and Yuriy Polyakov. Large-precision homomorphic sign evaluation using FHEW/TFHE bootstrapping. In Shweta Agrawal and Dongdai Lin, editors, *Advances in Cryptology – ASIACRYPT 2022, Part II*, volume 13792 of *Lecture Notes in Computer Science*, pages 130–160, Taipei, Taiwan, December 5–9, 2022. Springer, Heidelberg, Germany.

[Mis] Alan Mishchenko. System for sequential logic synthesis and formal verification. https://github.com/berkeley-abc/abc.

[MS18] Daniele Micciancio and Jessica Sorrell. Ring packing and amortized FHEW bootstrapping. In Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx, and Donald Sannella, editors, *ICALP 2018: 45th International Colloquium on Automata, Languages and Programming*, volume 107 of *LIPIcs*, pages

100:1–100:14, Prague, Czech Republic, July 9–13, 2018. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

[PAL21] PALISADE Lattice Cryptography Library (release 1.11.5). https://palisade-crypto.org/, September 2021.

[RAD78] Ronald Rivest, Len Adleman, and Michael Dertouzos. On data banks and privacy homomorphism, 1978.

[YXS+21] Zhaomin Yang, Xiang Xie, Huajie Shen, Shiying Chen, and Jun Zhou. TOTA: Fully homomorphic encryption with smaller parameters and stronger security. Cryptology ePrint Archive, Report 2021/1347, 2021. https://eprint.iacr.org/2021/1347.