Improved Circuit Synthesis with Amortized Bootstrapping for FHEW-like Schemes

Johannes Mono
Ruhr University Bochum
johannes.mono@rub.de

Kamil Kluczniak
CISPA Helmholtz Center for Information Security
kamil.kluczniak@cispa.de

Tim Güneysu
Ruhr University Bochum, DFKI GmbH
tim.gueneysu@rub.de

***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, mostly in the areas of performance and usability, to further advance the adoption of these technologies. This work provides multiple contributions to improve the current state-of-the-art in both areas.**

More specifically, we significantly simplify the bootstrapping idea by Carpov, Izabachène, and Mollimard [1] for Boolean-based FHE schemes such as FHEW or TFHE, making the concept usable in practice. Based on our simplifications, we provide an easy-to-use interface for amortized bootstrapping implementing our improvements in the open-source library FHE-Deck and provide new parameter sets for multi-bit encryptions with state-of-the-art security.

We build a toolset that compiles high-level code such as C++ to code that executes operations on encrypted data. For this toolset, we propose the first non-trivial FHE-specific optimizations in synthesizing privacy-preserving circuits from high-level code, namely look-up table (LUT) grouping and adder substitution. Using LUT grouping, we reduce the number of bootstrapping required by almost 35 % on average, while for adder substitution, we reduce the number of required bootstrapping by up to 80 % for certain use cases. Overall, the execution time is up to $3.8\times$ faster using our optimizations compared to previous state-of-the-art circuit synthesis.

1. Introduction

Encryption is a fundamental technology of today's society and is in use in different areas of public life, 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, such as reducing computation and hence economic costs, as well as improving usability to ease adoption.

Since Gentry's seminal work introducing the idea of bootstrapping in 2009 [2], multiple schemes have been proposed and improved over the years. Current state-of-the-art schemes are still exclusively based on bootstrapping, a process in which the error associated with the ciphertext is refreshed to allow for indefinite computation, and two strains have emerged.

Arithmetic-focused schemes such as BGV [3], BFV [4], [5] and CKKS [6] operate on many 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 many additions or multiplications, such as matrix multiplication.

In contrast, Boolean-based schemes such as FHEW [7] and TFHE [8] provide high flexibility encrypting either single bits or small bit groups and are thus a good fit for a wide variety of use cases in privacy-preserving computation. A target function is commonly represented as a circuit with Boolean gates where the input bits are encrypted. Every gate evaluation requires a bootstrapping which is relatively fast compared to arithmetic-focused schemes. However, bootstrapping is still the most expensive part of such circuits and thus, it is naturally important to reduce the number of gates requiring bootstrapping for evaluation when translating a use case to a circuit speeding up encrypted execution of the 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 usually results in better 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 [9], [10], [11].

Currently, two different approaches exist 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. Afterward, the individual instructions of the intermediate representation are mapped to small Boolean circuits and composed accordingly.

The second approach is based on existing hardware tooling, more specifically the synthesis process. First, high-level synthesis converts high-level code to a hardware description language (HDL) such as Verilog. Then, the HDL is passed to a synthesizer such as Yosys, and a Boolean circuit is generated as output. The resulting circuit is then transpiled

to FHE library code for both approaches.

The Google transpiler [11] is a state-of-the-art tool implementing both approaches. In the transpiler, the high-level synthesis framework XLS translates C++ code to an intermediate representation (IR) and performs common compiler optimizations. Afterward, the IR code is either mapped to small Boolean circuits for each instruction or translated to Verilog code for the synthesis-based approach.

For the latter, the Verilog code is then optimized and translated to a Boolean circuit using Yosys, an open-source synthesis tool. The result is a netlist, a textual description of a Boolean circuit, either based on single-bit gates with each ciphertext encrypting a single bit or multi-bit gates, so-called look-up tables (LUTs), where each ciphertext contains a small number of bits.

No matter the approach, the single-bit circuits can then be transpiled to FHE library code for different open-source libraries. For multi-bit circuits, however, the results can only be simulated using the transpiler's built-in simulator as current state-of-the-art libraries do not support encryption of multiple bits with secure parameters.

In this work, we make several significant contributions to improve the current state-of-the-art in FHE transpilation, primarily focusing on the synthesis-based approach:

- We significantly simplify the amortized bootstrapping idea proposed by Carpov, Izabachène, and Mollimard [1], improving the current state-of-the-art for bootstrapping in FHEW-like schemes. Our modifications make it feasible to use the amortization technique in practice.
- We propose new parameter sets for multi-bit encryptions with state-of-the-art security and implement an easy-to-use interface for amortized bootstrapping based on the open-source library FHE-Deck.
- 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.
- We also perform post-synthesis optimizations on the netlist based on amortized bootstrapping to evaluate multiple gates at once. Using our optimizations, execution times are up to $3.8\times$ faster compared to the previous state-of-the-art.

Outline. In Section 2, we start with some preliminaries on FHEW-like schemes and the synthesis process. Then, in Section 3, we provide our improvements and implementation for TFHE-based bootstrapping while our optimizations to the synthesis process are described and evaluated in Section 4. We discuss our results and their implications in Section 5 and provide a summary of our work in Section 6.

2. Preliminaries

In the following, we first introduce notation used in the remainder of this work and recall the definition of FHE. We continue by defining the Generalized Learning with Errors (GLWE) assumption as well as a cryptosystem based on this assumption. Afterward, we introduce concepts from hardware development, namely the synthesis process and circuit representations, concluding this section by explaining the general process of transpilation.

2.1. Notation

We denote as $\mathbb{Z}_q$ the group of integers modulo q. We denote as $\mathcal{R}$ the ring of polynomials $\mathbb{Z}_Q[X]/(X^N+1)$ where N is a power-of-two. We call the n-th root of unity in $\mathcal{R}$ an element $z \in \mathcal{R}$ such that $z^n = 1 \in \mathcal{R}$. 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. To differentiate ring elements from integers, we denote them with the `mathfrak` font like $\mathfrak{a} \in \mathcal{R}_Q$.

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 more generally $[n, m]_{i=n}^{m}$ the vector $[n, \ldots, m]^\top$. 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, respectively of the coefficients of the polynomial $\mathfrak{a}$. We denote any polynomial as $\mathsf{poly}(.)$. We denote as $\mathsf{negl}(\lambda)$ a negligible function in $\lambda \in \mathbb{N}$. That is, for any positive polynomial $\mathsf{poly}(.)$ 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. Fully Homomorphic Encryption

Below we recall the definition of fully homomorphic encryption (FHE) [2], [12].

Definition 1 (FHE)**.** *A FHE scheme consists of four algorithms* (Setup, Enc, Eval, Dec)*, each with the following syntax.*

$\mathsf{Setup}(\lambda)$: *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*.*

Correctness. We say that $\mathsf{FHE} = (\mathsf{Setup}, \mathsf{Enc}, \mathsf{Eval}, \mathsf{Dec})$ is correct if for all security parameters $\lambda \in \mathbb{N}$, the circuits

$\mathcal{C} : \mathcal{M}^n \mapsto \mathcal{M}$ 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

$$\mathsf{ct}_{\mathsf{out}} \leftarrow \mathsf{Eval}(\mathsf{ek}, [\mathsf{ct}_i]_{i=1}^n, \mathcal{C}).$$

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 $\mathcal{C}$.

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 has indistinguishable under chosen plaintext attack (IND-CPA) security if, given the advantage

$$\mathsf{Adv}^{\text{IND-CPA}}_{\mathcal{A},\mathsf{FHE}}(\lambda),$$

the following probability is at most $\mathsf{negl}(\lambda)$ for all PPT adversaries $\mathcal{A}$

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

where the oracle $\mathcal{O}$ on input of a message m outputs $\mathsf{ct} \leftarrow \mathsf{Enc}(\lambda, \mathsf{sk}, \mathsf{m})$.

Circuit Privacy. Let $\mathcal{C} : \mathcal{M}^n \mapsto \mathcal{M}$ be a polynomial size circuit. A FHE is said to be circuit private if there exists a PPT simulator Sim such that

$$\Delta(\mathsf{Sim}(\mathsf{ek}, m_{\mathsf{out}}), \mathsf{Eval}(\mathsf{ek}, c_1, \ldots, c_n, \mathcal{C})) \leq \mathsf{negl}(\lambda),$$

where $[m_i \leftarrow \mathsf{Dec}(\mathsf{sk}, c_i)]_{i=1}^n$, $m_{\mathsf{out}} \leftarrow \mathcal{C}(m_1, \ldots, m_n)$ and $(\mathsf{ek}, \mathsf{sk}) \leftarrow \mathsf{Setup}(\lambda)$.

2.3. Generalized Learning with Errors

Definition 2 (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 $\mathsf{RLWE}_\sigma(\mathfrak{s}, \mathsf{m}) = \mathsf{GLWE}_{\sigma,1,\deg,Q}$ 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 $\cdot$ when it is not necessary to refer to them within the scope of a function. We define the phase of $\vec{c} = \mathsf{GLWE}_{\sigma,n,\deg,Q}(\vec{s}, \mathsf{m})$, as $\mathsf{Phase}(\vec{c}, \vec{s}) = \langle \vec{c}, \vec{s} \rangle$. Additionally, we define the error of $\vec{c}$ as $\mathsf{Error}(\vec{c}, \mathsf{m}) = \mathsf{Phase}(\vec{c}) - \mathsf{m}$.

2.4. Computing on Encrypted Data

In this section we recall algorithms to compute on encrypted data and introduce additional notation that is neecessary for the main funcational bootstraping procedure in our work. In Table 1, we give a brief summary of the algorithms and, for detailed descriptions and experiments, refer the interested reader to previous work [7], [8], [13].

Below, we describe how to compose the algorithms from Table 1 to implement a system fitting Algorithm 1 and our model of computation.

$\mathsf{Setup}(\lambda)$. 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 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 $\mathfrak{s}$. Finally 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 $\mathsf{sk} = (\mathfrak{s}, \vec{s}_{\mathsf{ext}}, \vec{s})$.

$\mathsf{Enc}(\mathsf{sk}, \mathsf{m})$. 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 $m = \frac{Q}{p} \cdot m' \in \mathbb{Z}_Q$.

$\mathsf{Eval}(\mathsf{ek}, [\mathsf{ct}_i]_{i=1}^n, \mathcal{C})$. We can represent homomorphic computation as a circuit with gates of the form

$$f\left(b + \sum_{i=1}^k x_i \cdot a_i \in \mathbb{Z}_p\right) \in \mathbb{Z}_p,$$

where the a_i and b are scalars known by the evaluator and the x_i are the encrypted plaintexts. We compute the affine function using the additive homomorphism of the LWE samples and the function $f : \mathbb{Z}_p \mapsto \mathbb{Z}_p$ by applying the bootstrapping algorithm. In Section 3, we discuss in more detail how the bootstrapping algorithms compute functions.

TABLE 1. LIST OF PROCEDURES OUT OF WHICH WE BUILD AN FHEW/TFHE-STYLE BOOTSTRAPPING ALGORITHM.

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 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 $\mathsf{acc}_{\mathsf{out}} = \mathsf{RLWE}(\mathfrak{s},\vec{m}_{\mathsf{out}})$ with $\vec{m}_{\mathsf{out}} = \vec{m}_{\mathsf{acc}} \cdot X^{\mathsf{Phase}(\mathsf{ct})}$. **Description:** The blind rotation process [7], [8] 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 is a ring version of the GSW cryptosystem [14]. In this paper, we consider the concrete blind rotation from [8], 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:** Given a LWE sample $\mathsf{LWE}_{\cdot,n,Q}(\vec{s},\Delta_{Q,p} \cdot \mathsf{m})$ and a modulus $q < Q$, where $\mathsf{m} \in \mathbb{Z}_p$. **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 $\mathsf{LWE}_{\cdot,N,Q}(\vec{s}_{\mathsf{ext}},\mathsf{m})$, where $\mathsf{m} = \mathfrak{m}[1]$. That is, the LWE smaple encodes the constant coefficient of the polynomial $\mathfrak{m}$.
Bootstrap	**Input:** Takes as input a blind rotation key brKey, a key switching key ksKey, a LWE ciphertext $\vec{c} = \mathsf{LWE}_{\cdot,N,Q}(\vec{s},\cdot)$, a polynomial $\mathfrak{w} \in \mathcal{R}_Q$, and a vector $\vec{\mathfrak{v}} \in \mathcal{R}_Q^k$. **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 LWE ciphertext $\mathsf{LWE}_{\cdot,N,Q}(\vec{s},\mathfrak{w} \cdot X^{\vec{c},\vec{s}})$. For the amortized variant, the algorithm returns a vector of LWE ciphertexts $[\vec{c}_{\mathsf{out},i}]_{i=1}^k$, where $\vec{c}_{\mathsf{out},i} = \mathsf{LWE}_{\cdot,N,Q}(\vec{s},\mathfrak{w} \cdot X^{\vec{c},\vec{s}}\vec{\mathfrak{v}}[i])$. **Description:** The algorithm usually calls ModSwitch, KeySwitch, SampleExtract and BlindRotate as subprocedures. In Section 3, we describe the exact bootstrapping algorithms that we implement and use for your experiments.

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

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

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

2.5. Synthesis

Synthesis is a process in hardware development that transforms a circuit design, commonly described using a HDL such as Verilog, to a textual representation of a low-level circuit, also referred to as netlist (see also Subsection 2.6). During synthesis, the circuit is optimized, often heuristically, according to specific parameters such as area usage or power consumption.

Extending the idea of synthesis from HDL code to high-level code such as C++ code is called high-level synthesis. It enables designers to work at higher levels of abstraction and facilitates the re-use of existing high-level code for hardware circuits.

An example of a high-level synthesis tool is Google's XLS framework[1], translating C++ code to Verilog code, and an example of a low-level synthesis tool is Yosys[2], transforming Verilog code to netlists. Both tools are used in Google's FHE transpiler [11].

2.6. Circuit Representation

After synthesis, circuits are often represented using a textual representation, also referred to as a netlist. Usually, netlists are either based on storage elements (such as registers) and logic gates (for example, NAND-gates, XOR-gates, multiplexers, ...) or based on storage elements combined with LUTs where each LUT has an initialization string.

The initialization string is a bit string and defines the output for each input. For a `LUT2`, for example, the bit at

1. https://github.com/google/xls
2. https://github.com/yosyshq/yosys

index zero of the initialization string defines the output for the input `00`, the bit at index one for the input `01`, the bit at index two for `10` and the bit at index three for `11`.

In this work, we also use a full adder gate. A full adder receives two input bits x and y as well as a carry-in bit c_i. It outputs two bits, the sum s and the carry-out c_o:

$$s = x \oplus y \oplus c_i$$
$$c_o = (x \cdot y) + (c_i \cdot (x \oplus y))$$

A helpful perspective on a netlist is viewing it as directed acyclic graph, where the gates from the netlist are the vertices of the graph, and the edges are the wires connecting the gates. 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. Extending this idea to netlists ensures that, if we evaluate the gates in their topological order, all the inputs for a given gate have been evaluated previously.

2.7. 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. Usually, transpilation converts code at similar levels of abstraction without changing the code's logic or functionality.

In the context of FHE, transpilation refers to converting high-level code implementing functionality in the unencrypted domain to FHE library code. As an example, Google's FHE transpiler [11] 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.

The first step is thus often translating the high-level code to an IR where optimizations are performed and then further processed using 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 (see also Subsection 2.5).

3. Amortized Functional TFHE: Parameters and Efficient Implementation

In this section we specify how we compute functions on ciphertexts while keeping a constant noise level. Among the most efficient types of FHE schemes are schemes based on the FHEW/TFHE style bootstrapping algorithms [7], [8]. There are multiple variants [1], [8], [15], [16], [17], [18] and improvements [19], [20], [21], [22] of these algorithms.

Nevertheless, the at the core of these algorithms is the idea, introduced by Alperin-Sheriff and Peikert in [23], of homomorphically rotating a vector of elements in way that the homorphic rotation resenbles the decryption function on the input ciphertext. Specifically, suppose we have a 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 $\langle \vec{c}, \vec{s} \rangle$ of computing the phase can be realized within a cyclic algebraic group, more specifically, the group of rotations. The idea is to realize the rounding function 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 [23] over polynomial rings. In particular, the observation first made by Ducas and Micciancio [7] 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

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

At the core of FHEW-like algorithms is the BlindRotate procedure. In short, as part of the blind rotation procedure, we homomorphically compute $\mathfrak{w} \cdot X^{\langle \vec{c}, \vec{s} \rangle} = \mathfrak{w} \cdot X^{M+e}$. Since all computation take place over RLWE ciphertexts, we obtain at the end of the blind rotation procedure a RLWE ciphertext of $\mathfrak{w} \cdot X^{M+e}$. Finally, Ducas and Micciancio [7] 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 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 and switch the extracted LWE ciphertext to a LWE ciphertexts that is suitable for another bootstrapping step.

3.1. Amortized Functional Bootstraping

At a high level, the idea to amortize computation of different functions on the smaple input ciphertext is based on a previous work by Carpov, Izabachène, and Mollimard [1], but in this paper, we significantly simplify execution of the idea. Furthermore, we choose parameters that satisfy our constrains and integrate the algorithm in the the open-source library FHE-Deck [24]. More importantly, we provide high-level interfaces in FHE-DECK for all algorithms, making them easily accesible for FHE researchers and developers.

We give a version of the bootstrapping algorithm in Algorithm 1 providing 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. After this step, the LWE

ciphertext $\vec{c}$ has a smaller dimension $n \in \mathbb{N}$, and a secret key from a smaller distribution, for instance binary, ternary or Gaussian.

Algorithm 1: $\mathsf{Bootstrap}(\mathsf{brKey}, \mathsf{ksKey}, \vec{c}, \mathfrak{w}, \vec{\mathfrak{v}})$

Input:

The blind rotation key brKey;
The key switching key 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 $\vec{\mathfrak{v}} \in \mathcal{R}_p^k$.

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 $\left[\vec{c}_{\mathsf{out},i}\right]_{i=1}^{k}$;

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 $m_{\mathsf{acc}} \cdot \vec{v}[i]$. 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 bound of $\vec{\mathfrak{v}}$ is too large.

Carpov, Izabachène, and Mollimard [1] suggest a polynomial factorization algorithm that takes $\mathfrak{q}$ and returns factors $\mathfrak{w}_0$ and $\mathfrak{w}_1$ to tackle this problem. To plug this into our notation we would set $\mathfrak{w}$ to $\mathfrak{w}_0$ and and insert $\mathfrak{w}_1$ into the vector $\vec{\mathfrak{v}}$. This factorization algorithms works as follows. Essentially, they set $\mathfrak{w}_0 = \sum_{i=1}^{N} X^i$ and $\mathfrak{w}_1 = \sum_{i=1}^{N} t'_i X^i$. 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. In their implementation [1], the authors only tests the bootstrapping for random rotation polynomials, and there is no implementation of the linear system solver.

Nevertheless, for $N = 2^{14}$ [1], we can roughly calculate that the number of modular multiplications in the Gaussian elimination algorithm[3] 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 [1].

In our version, we use a simpler and less involved solution that requires only linear time to build the polynomials. In fact, in our implementation, the polynomials can be constructed online without any significant slowdown during computation. We observe that, when we are only interested in extracting bits, the polynomials in $\vec{\mathfrak{v}}$ can already be sparse and of small infinity norm.

Later, we can compute a simple binary composition before running the new bootstrap and computing the next LUT. Concretely, we set $\mathfrak{w} = \Delta_{Q,p} \cdot X$. 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 $f : \mathbb{Z}_p \mapsto \{0, 1\}$.

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

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

Crucially, observe that the sum takes each coefficient from the polynomails once, and at most $2N/3$ of the coefficents of

3. 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.

$\vec{\mathfrak{v}}[i]$ are non-zero. All non-zero coefficents of $\vec{\mathfrak{v}}[i]$ are either 1 or -1. Hence we have that

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

□

3.2. New Parameters

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 [25]. We also include a Python script to estimate the statistical security.

In Table 2, we specify three parameter sets. The `tfhe-11-ntt` parameter set is based on previous work by Kluczniak [26] 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 [25] and the modulus is below 51-bits to allow for faster multiplication of ring elements using the HEXL library [27].

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 as 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 [7], [28]. 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-ntt` will require the least amount of polynomials multiplications. In particular, recall that the number of polynomial multiplicaitons 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 Subsection 4.5.

```
FHEContext ctx;
ctx.generate_context(tfhe_11_NTT_amortized);

auto bit0 = [](long m) -> long {
    return m & 1;
};
auto bit1 = [](long m) -> long {
    return (m >> 1) & 1;
};
auto bit2 = [](long m) -> long {
    return (m >> 2) & 1;
};

std::vector<RotationPoly> lut;
lut.push_back(ctx.genrate_lut(bit0));
lut.push_back(ctx.genrate_lut(bit1));
lut.push_back(ctx.genrate_lut(bit2));

int msg = 6;
Ciphertext ct = ctx.encrypt_public(msg);
std::vector<Ciphertext> out = \
    ctx.eval_lut_amortized(&ct, lut);

std::cout << ctx.decrypt(&out[0])
          << ctx.decrypt(&out[1])
          << ctx.decrypt(&out[2])
          << std::endl;
```

Listing 1. Code example showing the easy-to-use interface for amortized bootstrapping.

3.3. Implementation Details

We implemented, tested and integrated the amortization algorithm into the FHE-Deck library [24]. A significant part of our work was to integrate different functional bootstraping algorithms, message encodings, amortization and circuit privacy support under a single interface.

At Listing 1, we show a simple example of evaluating a LUT using the new interface. Roughly speaking, we can specify C++ lambda expressions to evaluate during bootstrapping on encrypted data. FHE-Deck is based on the HEXL library [27] for fast polynomial multiplcation utilizing advanced vector instructions.

4. Circuit Synthesis

In the following, we provide a detailed description of our two main optimizations applied during synthesis and on the resulting netlist. We start by explaining our processing pipeline with Yosys [29], an open-source synthesis suite, and HAL [30], a netlist analysis tool. Then, we describe the idea of LUT grouping used for amortized bootstrapping and continue explaining adder substitutions for faster additions. We conclude this section with additional minor optimizations and evaluate and discuss our optimizations.

TABLE 2. NEW PARAMETER CHOICES.

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

4.1. (Post-)Synthesis Processing

Synthesis with Yosys can be done using pre-defined Yosys scripts, the HDL code is read in by Yosys and then transformed into a netlist executing the steps specified in the script. We base our scripts on the default synthesis script in Yosys, removing steps and optimizations that do not apply to circuits only consisting of combinatorial logic gates.

For Boolean circuits, we additionally use a so-called liberty file defining all gates supported in state-of-the-art FHE libraries; we source the file from the Google transpiler. The liberty file defines each gate to consume an area corresponding to the required bootstrappings per gate invocation. This includes inversion, which is free for FHEW-like schemes, and defined to consume no area.

For LUT synthesis, we include a step mapping the technology to a pre-defined gate library. HAL supports the same gate library; however, we slightly modify the default library to remove non-needed gates and include our custom full adder gate. The latter is instantiated if adder optimizations are enabled to map certain operations to these custom gates during synthesis (see also Subsection 4.3).

Using HAL's Python interface, the post-synthesis optimizations are performed using two different Python scripts, one for the Boolean circuits and one for the LUT-based circuits. Common functionality, for example, topological sorting or translating inputs and outputs to library code, is extracted to a third script.

Transpiling the Boolean-based netlists in HAL is relatively straightforward: We apply a topological sorting to the netlist and transpile all gates from the liberty file as required. As each ciphertext is either zero or one, we transpile inversion by subtracting the ciphertext from the constant value one.

Translating the LUT-based netlists is more involved: First, we apply a topological sorting as before. Afterward, we group all full adders (see Subsection 4.3) followed by the LUT grouping (see Subsection 4.2). We then apply another topological sort on the final groups as grouping the adders can "unsort" the graph (without introducing cycles, however).

The individual LUTs are translated as follows: first, we permute the inputs and the initialization string to a fixed order consistent in each LUT grouping. Then, the initialization string is extended to the size of the plaintext modulus covering the full range of possible input values, and the input ciphertext to a LUT is composed using constant multiplications by powers-of-two as well as additions.

Each LUT outputs a ciphertext with the output bit stored in the least significant bit of the ciphertext. Thus, composing inputs as described is a well-defined operation and does not wrap around the plaintext modulus (which would require one additional bootstrapping). Finally, the initialization string of each LUT is translated to library code and added to an amortized bootstrapping call for each grouping.

We depict the stages and tools during the entire transpilation process in Figure 1. Our transpilation code, including tests and examples, will be publicly available on GitHub.

4.2. Look-Up Table Grouping

As described in Subsection 3.1, the most expensive part of bootstrapping is computing the rotation polynomial while evaluating a specific function itself is cheap. Thus, as long as the input ciphertext is the same, we can generate multiple outputs with almost no overhead.

Mapping the amortized bootstrapping technique to LUTs is relatively straightforward: As long as the inputs to a LUT are equal and in the same order, we can perform an amortized bootstrapping evaluating multiple LUT at once and "save" on bootstrapping operations. This also applies to LUTs where we only evaluate a subset of inputs (for example, grouping a `LUT2` with a `LUT3`).

There is, however, a limitation: our input is a directed acyclic graph of Boolean gates sorted topologically. We still require the grouped graph to be acyclic for the output, as we cannot compute the circuit otherwise. Thus, we parse the LUTs sequentially and only group a LUT to a previous one if its input is a subset, including the case with equal inputs.

The reason is that grouping can introduce cycles to the graph if a previous LUT is a proper subset of the current one. Note that such a grouping can introduce a cycle into the netlist graph but does not have to. Nevertheless, detecting these cases is non-trivial, and thus we do not perform such groupings.

An example for LUT grouping is depicted in Figure 2. Here, the two `LUT3` in the bottom row contain the same inputs and hence can be grouped. Additionally, we can add the `LUT2` in the bottom row to the same group, as it is a subset of the inputs and appears later in the graph. The `LUT2` in the middle row has a unique input combination and is its own group. As for the top row, the inputs of the `LUT2` are a subset of the inputs from the bottom row. Adding it to the group at the bottom would introduce a cycle, however.

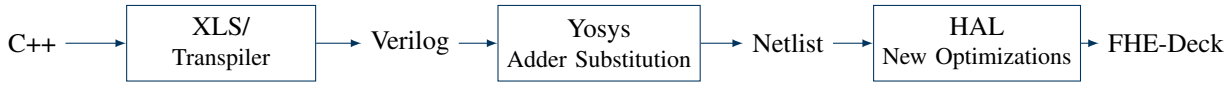

Figure 1. The full transpilation pipeline from C++ code implementing high-level functionality in the unencrypted domain to FHE library code, including our optimizations during and after synthesis. For some examples, we skip the high-level synthesis based on XLS and directly use available Verilog code.

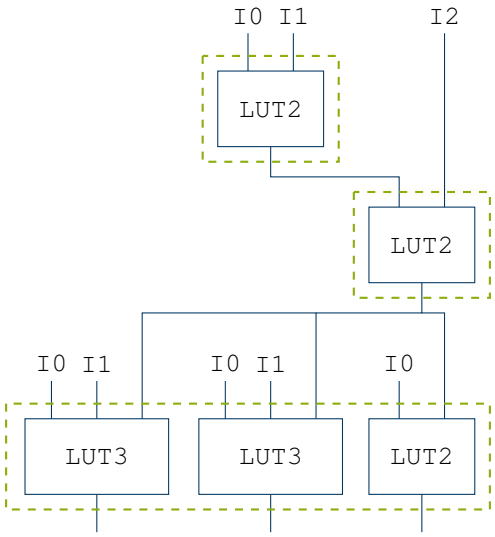

Figure 2. Grouping LUTs in an example circuit by matching the inputs while preserving topological order, each group is surrounded by a dashed line in green. The `LUT2` in the top row cannot be grouped with the bottom row as it would introduce a cycle.

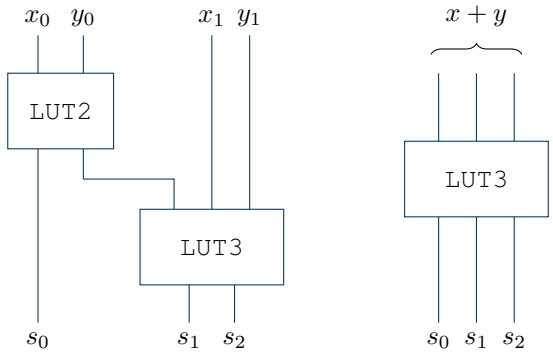

Figure 3. Comparing a two-bit addition $s = x + y$ based only on LUTs (left) and based on ciphertext addition with decomposition (right) for the plaintext modulus 2^3. The former requires two amortized bootstrappings while the latter only requires one. Each "LUT" can produce multiple outputs for different initialization strings, corresponding to a LUT grouping.

4.3. Adder Substitution

The general idea for our adder substitution optimization is based on the following observation: Using LUTs to realize additions is generally more costly than using the native addition capabilities of the scheme and decomposing the resulting sum to individual bits using a single amortized bootstrapping. Figure 3 depicts an example of a two-bit addition for the plaintext modulus 2^3.

The process of adder substitution is split into multiple steps. First, we hook into the synthesis process, replacing addition and subtraction units in the Yosys IR with custom adder gates. Note that we exclude constant additions as the optimizer is very good at combining them with surrounding logic into LUTs.

Second, during netlist post-processing, the adders are grouped into addition chains and split into groups so that we can add two ciphertexts without wrap-around. This is necessary as any wrap-around would require a second bootstrapping step for decomposition resulting in a more costly circuit overall.

Finally, the adders are transpiled to library code, and an amortized bootstrapping is used to decompose the ciphertext into individual bits. In the following, we will provide additional details for the first two steps describing our optimization in more detail, then the final step follows trivially.

Replacing Addition and Subtraction Units. We hook into the synthesis process with a custom mapping to replace addition and subtraction units in the Yosys IR. For readers familiar with hardware development, this process is generally used to replace certain operations with special gates such as multiplications with digital signal processing units.

As Yosys allows custom mappings for arbitrary instructions, we provide such mappings for additions and subtractions. For both operations, we first check if any of the inputs is a constant as replacing constant additions is inefficient and we thus abort processing for this specific instance if applicable. Otherwise, we continue with the replacement process.

Since additions and subtractions in Yosys are arbitrary-width operations, we sign-extend both operands to the same width and replace each bit addition with a full adder, connecting the adders with a carry chain. For addition, we set a constant value of zero to the carry-in of the first full adder. For subtraction, we set the carry-in to one and place inverters for the second operand; this corresponds to negation in the two's complement representation. As inversions are free for FHEW-like schemes, this does not add any costs to the resulting circuit.

Finally, we proceed with the rest of the synthesis process as usual and use ABC [31], a system for sequential logic synthesis and formal verification, to map the other parts to LUTs, including the ignored constant additions or subtractions.

Post-Processing Full Adder Gates. So far, we only placed single-bit full adders during the synthesis process. For multi-bit circuits, however, we can process multiple full adders

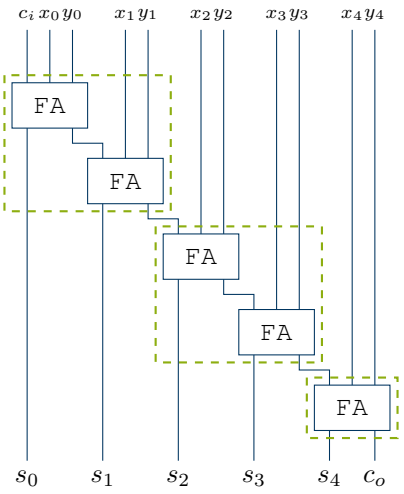

Figure 4. Grouping individual full adders in the post-synthesis netlist for a 5-bit addition and a plaintext modulus of 2^3, each grouping is marked by a dashed green box.

simultaneously and decompose the result to individual bits using amortized bootstrapping.

First, we detect the adders in the netlist by looking at the carry-ins. If it is either zero (for additions) or one (for subtractions), we set this full adder as the root of an adder chain. Then, we follow the carry bits to determine the entire chain.

Afterward, we group the adders where each group consists of one less adder than the bit size of the plaintext; for example, when using a plaintext modulus of 2^3 encrypting 3 bits, we group two adders to ensure no wrap-around with the carry happens. We can then compose the addition inputs using constant multiplications with powers-of-two and additions as described in Subsection 4.1, performing a ciphertext addition afterward. The sum is then input to a LUT, decomposing the sum into individual bits. Here, the highest bit corresponds to the carry-out of the multi-bit addition and is used afterward as carry-in to the next adder group in the chain. We depict this process in Figure 4 for a 5-bit addition and a plaintext modulus of 2^3.

4.4. Other Optimizations

In addition to our two major optimizations described in Subsection 4.2 and Subsection 4.3, we perform two additional minor optimizations: inverter conversion and type sorting. For inverter conversion, we use the fact that any `LUT1` corresponds to an inverter and hence does not require a bootstrapping (hardware tools usually use buffers for the identity function, not `LUT1` gates). As a sanity check, we make sure that each `LUT1` is actually an inverter.

For type sorting, we group the netlist graph into layers during topological sorting according to their depth and then sort each layer with respect to gate types. Most importantly, we sort 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.

4.5. Evaluation

For evaluation, we use our own examples as well as multiple extracted Verilog designs from the Google transpiler [11]. More specifically, the following examples are used:

- `add3`, `add4`, `add32`: Compute the sum of 3-bit, 4-bit or 32-bit integers, respectively.
- `calc`: Calculate the addition, subtraction or multiplication of two 16-bit integers.
- `const4`: Compute the sum of a 4-bit integer with a constant.
- `img-blur`: Apply blurring to a small image.
- `img-ricker`: Apply a ricker wavelet transformation to a small image.
- `img-sharp`: Apply sharpening to a small image.
- `relu`: Compute a rectified linear unit function.
- `sqrt`: Compute the square root of a 16-bit integer.
- `strrev`: Reverse an array of up to eight characters.
- `structs1d`: Compute the sum over a one-dimensional array of structs.
- `structs3d`: Compute the sum over a three-dimensional array of structs.
- `sum3d`: Compute the sum over a three-dimensional array of integers.

Additionally, we compare multiple different synthesis processes with or without optimizations:

- [11]: The Yosys script used with the Google transpiler for single-bit circuits without optimizations.
- `bool`: Our Yosys script for single-bit circuits without optimizations.
- `none2`, `none3`: Our Yosys script for multi-bit circuits using no optimizations during or after synthesis supporting up to 2-bit or 3-bit LUTs, respectively.
- `none2fa`, `none3fa`: Our Yosys script for multi-bit circuits using adder translations with amortized bootstrapping and no other post-synthesis optimizations, supporting up to 2-bit or 3-bit LUTs, respectively.
- `lut2`, `lut3`: Our Yosys script for multi-bit circuits with type sorting, LUT grouping, and inverter conversion after synthesis supporting up to 2-bit or 3-bit LUTs, respectively.
- `lut2fa`, `lut3fa`: Our Yosys script for multi-bit circuits with all optimizations enabled supporting up to 2-bit or 3-bit LUTs, respectively.

We summarize the number of bootstrappings that are required for each approach considering all examples in Table 3.

Evaluating LUT Grouping. To evaluate LUT grouping in isolation, we compare the results for `none2` and `none3` with `lut2` and `lut3`, respectively. For the chosen use

TABLE 3. RESULTS COMPARING MULTIPLE EXAMPLES WITH MULTIPLE SYNTHESIS PROCESSES AND OPTIMIZATION TECHNIQUES. EACH ROW CONTAINS THE NUMBER OF BOOTSTRAPPINGS FOR A GIVEN EXAMPLE WHILE THE COLUMNS CONTAIN THE SYNTHESIS PROCESSES.

	[11]	bool	none2	none2fa	lut2	lut2fa	none3	none3fa	lut3	lut3fa
add3	12	12	12	3	7	3	6	2	3	2
add4	17	17	17	4	10	4	8	2	4	2
add32	165	165	163	32	102	32	63	32	32	16
calc	948	896	884	857	655	653	491	479	351	358
const4	4	4	4	4	2	2	3	3	1	1
img-blur	318	318	316	74	193	74	146	37	90	37
img-ricker	342	341	334	90	212	89	145	48	95	47
img-sharp	183	183	194	76	125	65	94	43	57	35
relu	30	30	15	15	15	15	15	15	15	15
sqrt	195	200	224	334	183	307	102	179	85	162
strrev	668	690	805	805	744	744	364	364	341	341
structs1d	159	159	157	31	98	31	61	16	31	16
structs3d	1091	1091	1056	217	654	217	460	112	246	112
sum3d	954	954	926	203	553	203	396	105	199	105

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 %. Hence, we recommend always enabling LUT grouping to improve performance for synthesized FHE circuits.

Evaluating Adder Substitution. For adder substitution, we compare the results for `none2` and `none3` with `none2fa` and `none3fa`, respectively, to our optimization in isolation. 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, and there is no improvement to the number of bootstrappings. 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 2, 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 substitutions with many constant input bits into LUTs is relatively efficient, the default optimizations outperform our adder substitution. We suggest transpiling circuits with and without adder substitutions discussing possibilities for future work in Subsection 5.4 to avoid such scenarios.

Benchmarking. We run our 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.

```
[...]
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 2. 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-amort	0.29	81.7	81.8
tfhe-12-amort	0.58	217.9	163.6
tfhe-11-ntt	0.24	44.8	78.5

For the new parameter sets, we summarize our results in Table 4 regarding runtime and memory consumption, confirming our observations from Subsection 3.2.

We use the parameter set `tfhe-11-ntt` for `bool`, `lut2` as well as `lut2fa` while for `lut3` and `lut3fa`, we use the parameter set `tfhe-11-amort`. For `lut4` and `lut4fa`, we use the parameter set `tfhe-12-amort`.

Our benchmarking results for all examples are summarized in Table 5. Here, we can make multiple observations. As expected, the execution time highly correlates with the number of bootstrappings. In general, we receive the best

speed-ups for `lut3fa` (for `sqrt` with `lut3`). But, there are exceptions such as the `relu` example where `lut2` and `lut2fa` perform the best.

Using `lut4` and `lut4fa` is generally not worth due to the increased polynomial degree and thus the longer bootstrapping time; using `lut4` is actually worse than `bool`. Sometimes, the adder substitution optimization still is enough to provide a speed-up compared to `bool`, however, the speed-ups for the smaller LUT-based circuits provide better execution times in all our examples.

5. Discussion

We put our work in the context of the current scientific discourse discussing first 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 Boolean-based schemes from high-level languages.

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 of circuit privacy by default as well as interface support with correct and secure parameter sets for amortized bootstrapping. Both FHE-Deck and TFHE-rs support different algorithms for functional bootstrapping (also known as programmable bootstrapping).

In particular, FHE-Deck supports the full domain bootstrapping algorithm based on work by Liu, Micciancio, and Polyakov [21] while THFE-rs supports the algorithm by Chilotti et al. [20]. Moreover, both libraries support simple padding-based functional bootstrapping. The TFHE library [32] supports only binary gates. Open-FHE [33], which is derived from PALISADE [34], implements binary as well as full domain bootstrapping based on work by Liu, Micciancio, and Polyakov [21].

There are implementations for LUT evaluation [35]. 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 [15] and its improvement [36] 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 [1] 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 3.1, 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 generates these polynomials. Moreover, the bootstrapping parameters chosen in the implementation do not allow generating the rotation polynomials as suggested [1] 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 [1] 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 [1] 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 Google transpiler [11] as well as two earlier works named Cingulata, originally released under the name Armadillo [9], and the E3 framework [10].

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 [31], 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 Google 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

TABLE 5. EXECUTION TIME IN SECONDS INCLUDING THE CORRESPONDING (ROUNDED) SPEED-UPS COMPARED TO BOOL FOR ALL EXAMPLES AND A SELECTION OF SYNTHESIS PROCESSES WITH OPTIMIZATIONS.

	bool	lut2		lut2fa		lut3		lut3fa		lut4		lut4fa	
Example	t_b	t	t_b/t	t	t_b/t	t	t_b/t	t	t_b/t	t	t_b/t	t	t_b/t
add3	2.40	1.57	1.5	0.91	2.6	1.31	1.8	1.01	2.4	2.79	0.9	2.19	1.1
add4	3.33	2.18	1.5	1.23	2.7	1.75	1.9	1.14	2.9	3.88	0.9	3.28	1.0
add32	30.90	21.03	1.5	9.69	3.1	13.91	2.2	9.13	3.3	36.44	0.8	23.39	1.3
calc	146.50	109.10	1.3	107.60	1.4	104.30	1.4	106.00	1.4	149.60	1.0	149.10	1.0
const4	0.86	0.54	1.6	0.54	1.6	0.50	1.7	0.59	1.5	1.38	0.6	1.36	0.6
img-blur	56.00	35.88	1.6	17.00	3.3	30.82	1.8	15.70	3.6	61.71	0.9	35.05	1.6
img-ricker	59.88	39.11	1.5	19.38	3.1	31.98	1.9	18.67	3.2	64.19	0.9	40.28	1.5
img-sharp	32.39	22.97	1.4	13.26	2.4	19.34	1.7	12.90	2.5	36.60	0.9	24.82	1.3
relu	5.94	3.54	1.7	3.53	1.7	5.50	1.1	5.43	1.1	12.89	0.5	12.88	0.5
sqrt	33.23	30.55	1.1	52.51	0.6	25.64	1.3	50.77	0.7	41.63	0.8	80.00	0.4
strrev	115.10	124.60	0.9	123.70	0.9	103.10	1.1	103.60	1.1	171.50	0.7	171.30	0.7
structs1d	30.01	20.25	1.5	9.42	3.2	13.38	2.2	9.00	3.3	36.56	0.8	22.78	1.3
structs3d	193.20	123.40	1.6	53.68	3.6	88.20	2.2	50.62	3.8	202.80	1.0	110.20	1.8
sum3d	167.10	103.20	1.6	46.33	3.6	71.13	2.3	43.86	3.8	161.90	1.0	94.46	1.8

TABLE 6. FUNCTIONALITY COMPARISON OF DIFFERENT FHE LIBRARIES FOR FHEW-LIKE SCHEMES. BY **CP**, WE DENOTE WHETHER THE PARAMTER SETS AND INTERFACES SUPPORT CIRCUIT PRIVACY. FOR FUNCTIONAL BOOTSTRAPPING, WE DENOTE AS ● THE PLAIN FHEW/TFHE ALGORITHM TO EVALUATE BOOLEAN GATES. BY ◐, WE DENOTE A FULL DOMAIN FUNCTIONAL BOOTSTRAPPING ALGORITHM [19], [20], [21], [22]. BY ●, WE DENOTE SUPPORT FOR OUR IMPROVED FUNTIONAL BOOTSTRAPPING ALGORITHM. IN THIS COMPARISON, A HIGH-LEVEL INTERFACE FOR FUNCTIONAL BOOTSTRAPPING IS REQUIRED.

Lib	**Lang**	**CP**	Func. Bootstrap	Amortized
FHE-Deck	C++	✓	●	✓
Open-FHE	C++	×	◐	×
TFHE	C++	×	○	×
tfhe-rs/CONCRETE	Rust	×	●	×

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 4.5, 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 automated circuit generation.

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 usabilities 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 [1]. Additionally, we provide 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.

With respect to usability, we provide new and secure parameter sets for multi-bit encryptions which can be used by researchers and developers alike. Additionally, we implement a high-level interface for amortized bootstrapping based on the open-source library FHE-Deck which supports circuit privacy by default.

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 and by the Cryptography Research Center at Technology Innovation Institute (TII), Abu Dhabi. Kamil Kluczniak is supported by the German Ministry for Education and Research through funding for the project CISPA-Stanford Center for Cybersecurity (Funding number: 16KIS0927).

References

[1] S. Carpov, M. Izabachène, and V. Mollimard, "New techniques for multi-value input homomorphic evaluation and applications," in *Topics in Cryptology – CT-RSA 2019*, ser. Lecture Notes in Computer Science, M. Matsui, Ed., vol. 11405. San Francisco, CA, USA: Springer, Heidelberg, Germany, Mar. 4–8, 2019, pp. 106–126.

[2] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *41st Annual ACM Symposium on Theory of Computing*, M. Mitzenmacher, Ed. Bethesda, MD, USA: ACM Press, May 31 – Jun. 2, 2009, pp. 169–178.

[3] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without bootstrapping," *ACM Trans. Comput. Theory*, vol. 6, no. 3, pp. 13:1–13:36, 2014. [Online]. Available: https://doi.org/10.1145/2633600

[4] Z. Brakerski, "Fully homomorphic encryption without modulus switching from classical GapSVP," in *Advances in Cryptology – CRYPTO 2012*, ser. Lecture Notes in Computer Science, R. Safavi-Naini and R. Canetti, Eds., vol. 7417. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, Aug. 19–23, 2012, pp. 868–886.

[5] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," Cryptology ePrint Archive, Report 2012/144, 2012, https://eprint.iacr.org/2012/144.

[6] J. H. Cheon, A. Kim, M. Kim, and Y. S. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *Advances in Cryptology – ASIACRYPT 2017, Part I*, ser. Lecture Notes in Computer Science, T. Takagi and T. Peyrin, Eds., vol. 10624. Hong Kong, China: Springer, Heidelberg, Germany, Dec. 3–7, 2017, pp. 409–437.

[7] L. Ducas and D. Micciancio, "FHEW: Bootstrapping homomorphic encryption in less than a second," in *Advances in Cryptology – EUROCRYPT 2015, Part I*, ser. Lecture Notes in Computer Science, E. Oswald and M. Fischlin, Eds., vol. 9056. Sofia, Bulgaria: Springer, Heidelberg, Germany, Apr. 26–30, 2015, pp. 617–640.

[8] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds," in *Advances in Cryptology – ASIACRYPT 2016, Part I*, ser. Lecture Notes in Computer Science, J. H. Cheon and T. Takagi, Eds., vol. 10031. Hanoi, Vietnam: Springer, Heidelberg, Germany, Dec. 4–8, 2016, pp. 3–33.

[9] S. Carpov, P. Dubrulle, and R. Sirdey, "Armadillo: A compilation chain for privacy preserving applications," in *Proceedings of the 3rd International Workshop on Security in Cloud Computing*, ser. SCC '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 13–19. [Online]. Available: https://doi.org/10.1145/2732516.2732520

[10] E. Chielle, O. Mazonka, H. Gamil, N. G. Tsoutsos, and M. Maniatakos, "E3: A framework for compiling c++ programs with encrypted operands," Cryptology ePrint Archive, Paper 2018/1013, 2018, https://eprint.iacr.org/2018/1013. [Online]. Available: https://eprint.iacr.org/2018/1013

[11] S. Gorantala, R. Springer, S. Purser-Haskell, W. Lam, R. Wilson, A. Ali, E. P. Astor, I. Zukerman, S. Ruth, C. Dibak, P. Schoppmann, S. Kulankhina, A. Forget, D. Marn, C. Tew, R. Misoczki, B. Guillen, X. Ye, D. Kraft, D. Desfontaines, A. Krishnamurthy, M. Guevara, I. M. Perera, Y. Sushko, and B. Gipson, "A general purpose transpiler for fully homomorphic encryption," Cryptology ePrint Archive, Report 2021/811, 2021, https://eprint.iacr.org/2021/811.

[12] R. Rivest, L. Adleman, and M. Dertouzos, "On data banks and privacy homomorphism," 1978.

[13] D. Micciancio and Y. Polyakov, "Bootstrapping in fhew-like cryptosystems," in *Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, ser. WAHC '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 17–28. [Online]. Available: https://doi.org/10.1145/3474366.3486924

[14] C. Gentry, A. Sahai, and B. Waters, "Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based," in *Advances in Cryptology – CRYPTO 2013, Part I*, ser. Lecture Notes in Computer Science, R. Canetti and J. A. Garay, Eds., vol. 8042. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, Aug. 18–22, 2013, pp. 75–92.

[15] D. Micciancio and J. Sorrell, "Ring packing and amortized FHEW bootstrapping," in *ICALP 2018: 45th International Colloquium on Automata, Languages and Programming*, ser. LIPIcs, I. Chatzigiannakis, C. Kaklamanis, D. Marx, and D. Sannella, Eds., vol. 107. Prague, Czech Republic: Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Jul. 9–13, 2018, pp. 100:1–100:14.

[16] G. Bonnoron, L. Ducas, and M. Fillinger, "Large FHE gates from tensored homomorphic accumulator," in *AFRICACRYPT 18: 10th International Conference on Cryptology in Africa*, ser. Lecture Notes in Computer Science, A. Joux, A. Nitaj, and T. Rachidi, Eds., vol. 10831. Marrakesh, Morocco: Springer, Heidelberg, Germany, May 7–9, 2018, pp. 217–251.

[17] A. Guimarães, E. Borin, and D. F. Aranha, "Revisiting the functional bootstrap in tfhe," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2021, no. 2, p. 229–253, Feb. 2021. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/8793

[18] K. Kluczniak, "NTRU-v-um: Secure fully homomorphic encryption from NTRU with small modulus," in *ACM CCS 2022: 29th Conference on Computer and Communications Security*, H. Yin, A. Stavrou, C. Cremers, and E. Shi, Eds. Los Angeles, CA, USA: ACM Press, Nov. 7–11, 2022, pp. 1783–1797.

[19] Z. Yang, X. Xie, H. Shen, S. Chen, and J. Zhou, "TOTA: Fully homomorphic encryption with smaller parameters and stronger security," Cryptology ePrint Archive, Report 2021/1347, 2021, https://eprint.iacr.org/2021/1347.

[20] I. Chillotti, D. Ligier, J.-B. Orfila, and S. Tap, "Improved programmable bootstrapping with larger precision and efficient arithmetic circuits for TFHE," in *Advances in Cryptology – ASIACRYPT 2021, Part III*, ser. Lecture Notes in Computer Science, M. Tibouchi and H. Wang, Eds., vol. 13092. Singapore: Springer, Heidelberg, Germany, Dec. 6–10, 2021, pp. 670–699.

[21] Z. Liu, D. Micciancio, and Y. Polyakov, "Large-precision homomorphic sign evaluation using FHEW/TFHE bootstrapping," in *Advances in Cryptology – ASIACRYPT 2022, Part II*, ser. Lecture Notes in Computer Science, S. Agrawal and D. Lin, Eds., vol. 13792. Taipei, Taiwan: Springer, Heidelberg, Germany, Dec. 5–9, 2022, pp. 130–160.

[22] K. Kluczniak and L. Schild, "Fdfb: Full domain functional bootstrapping towards practical fully homomorphic encryption," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2023, no. 1, p. 501–537, Nov. 2022. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/9960

[23] J. Alperin-Sheriff and C. Peikert, "Faster bootstrapping with polynomial error," in *Advances in Cryptology – CRYPTO 2014, Part I*, ser. Lecture Notes in Computer Science, J. A. Garay and R. Gennaro, Eds., vol. 8616. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, Aug. 17–21, 2014, pp. 297–314.

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

[25] M. R. Albrecht, R. Player, and S. Scott, "On the concrete hardness of learning with errors," *Journal of Mathematical Cryptology*, vol. 9, no. 3, pp. 169–203, 2015. [Online]. Available: https://doi.org/10.1515/jmc-2015-0016

[26] K. 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.

[27] F. Boemer, S. Kim, G. Seifu, F. D. de Souza, V. Gopal *et al.*, "Intel HEXL (release 1.2)," https://github.com/intel/hexl, September 2021.

[28] Y. Lee, D. Micciancio, A. Kim, R. Choi, M. Deryabin, J. Eom, and D. Yoo, "Efficient FHEW bootstrapping with small evaluation keys, and applications to threshold homomorphic encryption," in *Advances in Cryptology – EUROCRYPT 2023, Part III*, ser. Lecture Notes in Computer Science, C. Hazay and M. Stam, Eds., vol. 14006. Lyon, France: Springer, Heidelberg, Germany, Apr. 23–27, 2023, pp. 227–256.

[29] C. Wolf, "Yosys open synthesis suite," https://yosyshq.net/yosys/.

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

[31] A. Mishchenko, "System for sequential logic synthesis and formal verification," https://github.com/berkeley-abc/abc.

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

[33] A. Al Badawi, J. Bates, F. Bergamaschi, D. B. Cousins, S. Erabelli, N. Genise, S. Halevi, H. Hunt, A. Kim, Y. Lee, Z. Liu, D. Micciancio, I. Quah, Y. Polyakov, S. R.V., K. Rohloff, J. Saylor, D. Suponitsky, M. Triplett, V. Vaikuntanathan, and V. Zucca, "Openfhe: Open-source fully homomorphic encryption library," in *Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, ser. WAHC'22. New York, NY, USA: Association for Computing Machinery, 2022, pp. 53–63. [Online]. Available: https://doi.org/10.1145/3560827.3563379

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

[35] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "Faster packed homomorphic operations and efficient circuit bootstrapping for TFHE," in *Advances in Cryptology – ASIACRYPT 2017, Part I*, ser. Lecture Notes in Computer Science, T. Takagi and T. Peyrin, Eds., vol. 10624. Hong Kong, China: Springer, Heidelberg, Germany, Dec. 3–7, 2017, pp. 377–408.

[36] A. Guimarães, H. V. L. Pereira, and B. van Leeuwen, "Amortized bootstrapping revisited: Simpler, asymptotically-faster, implemented," Cryptology ePrint Archive, Report 2023/014, 2023, https://eprint.iacr.org/2023/014.