A Haskell to FHE Transpiler With Circuit Parallelisation

Anne Müller
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
Graduate School of Computer Science, Saarland University
Germany
anne.mueller@cispa.de

Mohd Kashif
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
Saarland University
Germany
md.kashif8858@gmail.com

Nico Döttling*
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
doettling@cispa.de

Abstract

Fully Homomorphic Encryption (FHE) enables the evaluation of programs directly on encrypted data. However, because only basic operations can be performed on ciphertexts, programs must be expressed as boolean or arithmetic circuits. This low-level representation makes implementing applications for FHE significantly more cumbersome than writing code in a high-level language.

To reduce this burden, several transpilers have been developed that translate high-level code into circuit representations. In this work, we extend the range of high-level languages that can target FHE by introducing a transpiler for Haskell, which converts Haskell programs into Boolean circuits suitable for homomorphic evaluation.

Our second contribution is the automatic parallelization of these generated circuits. We implement an evaluator that executes gates in parallel by parallelizing each layer of the circuit. We demonstrate the effectiveness of our approach on two key applications: Private Information Retrieval (PIR) and the AES encryption standard. Prior work has parallelized AES encryption manually. We demonstrate that the automated method outperforms some but not all manual parallelizations of AES evaluations under FHE. We achieve an evaluation time of 28 seconds for a parallel execution with 16 threads and an evaluation time of 8 seconds for a parallel execution with 100 threads.

CCS Concepts

• **Security and privacy** → Public key encryption.

Keywords

Fully Homomorphic Encryption; circuit synthesis; circuit parallelization; AES

*Funded by the European Union (ERC, LACONIC, 101041207). Views and opinions expressed are those of the author(s) only and do not necessarily reflect those of the European Union or the ERC. Neither the European Union nor the granting authority can be held responsible.

WAHC '25, Taipei, Taiwan

ACM ISBN 979-8-4007-1900-4/25/10
https://doi.org/10.1145/3733811.3767311

1 Introduction

Fully Homomorphic Encryption (FHE) is a foundational tool for secure cloud computing and privacy-preserving machine learning. Since Gentry's groundbreaking work in 2009 [21], the field has developed rapidly. However, the broad adoption of FHE still faces several challenges. Most notably, the computation on encrypted data is very inefficient compared to the computation on plain data. Moreover, only basic operations are possible on ciphertexts. Therefore, the program has to be specified as a Boolean or arithmetic circuit.

To mitigate the aforementioned issues many improvements in the area of homomorphic encryption have been introduced. New schemes have been developed that achieve better efficiency. Additionally, many compilers have been developed that transform code from a high-level language such as C++ or Python into a low-level circuit representation [35].

However, many of these compilers support only a restricted subset of their respective source languages. In contrast, our approach starts with programs written in Clash, a functional hardware description language that supports most features of Haskell. Naturally, FHE-specific constraints still apply, such as no data-dependent recursion. After the compilation step we evaluate the circuit using the TFHE library [15].

The issue of low efficiency can be addressed by parallelizing FHE programs. But, such parallelism cannot typically be implemented in the source language: to the best of our knowledge, no current compiler supports automatic translation of libraries containing parallel execution instructions. As a result, developers have manually identified parallelizable code regions, compiled them separately, and reassembled them for parallel circuit evaluation—a cumbersome, error-prone process that must be repeated for each new application.

Contrary to this approach, we investigate the performance of simple layer-based parallelization. A circuit naturally consists of layers such that each gate in a layer can be evaluated independently of the gates in the same layer.

We test our compiler on an AES implementation. AES is a common benchmark for FHE implementations and a popular target for manual parallelization efforts. Trama et al. [33] report an evaluation of AES using 16 threads in 36 seconds on an AMD-server or 28 seconds on an i7-server. Our evaluation similarly achieves an AES evaluation in 28 seconds using 16 threads. The choice of 16 threads for parallelizing AES is no coincidence. The state throughout the AES execution consists of 16 bytes and in each round some of the operations can be performed simultaneously on each byte of the

state. Therefore, for a manual parallelization of AES the choice of 16 threads is natural. Our parallelization algorithm is oblivious to any structure other than the number of gates at each depth level of the circuit. Therefore, the parallel execution can be extended to any number of threads; as such we achieve an evaluation time of 8 seconds using 100 threads.

In a recent paper [4] the 'Hyppogryph' implementation of AES achieves a single-threaded execution in 32 seconds and an execution with 32 threads in 1.1 seconds. The implementation is highly specialized and combines multiple methods for different sections of the AES cipher. Therefore, the manual parallelization in this approach seems necessary.

Contributions

- We present the first compiler that translates Haskell (via Clash) into circuits suitable for fully homomorphic encryption, making FHE accessible to a broader audience, particularly those in the functional programming community.
- As our second contribution, we show that the natural circuit parallelization technique achieves good results on important benchmarks such as AES and should not be neglected. We further evaluate our compiler on a second benchmark where we implement private information retrieval (PIR).

The code, including all benchmarks, is publicly available[1].

2 Background

2.1 Fully Homomorphic Encryption

A fully homomorphic encryption scheme consists of the following polynomial time algorithms:

- $\mathsf{KeyGen}(1^\lambda) \to (\mathsf{pk}, \mathsf{sk})$ KeyGen is a probabilitstic algorithm that on input the security parameter λ produces a public key, secret key pair $(\mathsf{pk}, \mathsf{sk})$.
- $\mathsf{Enc}(\mathsf{pk}, \mathsf{m}) \to \mathsf{ct}$ Enc is a probabilisitc algorithm that on input the public key pk and a message m produces a ciphertext ct.
- $\mathsf{Eval}(\mathsf{pk}, C, \mathsf{ct}) \to \mathsf{ct}'$ Eval is a probabilistic algorithm that on input the public key pk and a circuit C and a ciphertext ct produces a ciphertext ct'.
- $\mathsf{Dec}(\mathsf{sk}, \mathsf{ct}) \to \mathsf{m}/\perp$ Dec is a deterministic algorithm that given a secret key sk and a ciphertext ct outputs a message m or $\perp$.

The FHE scheme also has to fulfill security and efficiency properties. Informally, we require that given a ciphertext, one cannot learn anything about the encrypted message. We also require that the size of the ciphertext that is produced by Eval does not depend on the size of the circuit C. Of course, we also require correctness, which means that given an encryption of a message m, after applying $\mathsf{Eval}(\mathsf{pk}, C, \mathsf{ct}) \to \mathsf{ct}'$ the resulting ciphertext ct' contains an encryption of $C(\mathsf{m})$. Note that the circuit C is not hidden.

The circuit C that can be evaluated by the Eval algorithm depends on the particular FHE scheme. For example, GSW [23], FHEW [19] and TFHE [14] allow the execution of boolean circuits while BGV [9] and BFV [8, 20] allow the evaluation of arithmetic circuits with modular arithmetic. The CKKS [11] scheme allows to evaluate arithmetic circuits on real numbers. In this work we make use of the TFHE library [15] which works on boolean circuits.

A levelled homomorphic encryption scheme is a homomorphic encryption scheme that can only evaluate an a priori bounded number of gates. The number of gates has to be known at key-generation time. This is in contrast to a fully homomorphic encryption scheme which can evaluate an unbounded number of gates on a ciphertext. This is achieved through an expensive bootstrapping operation. With each homomorphic operation, noise accumulates in the ciphertext, which would eventually lead to decryption errors. To avoid this the error has to be reduced periodically by applying the bootstrapping algorithm which is a time-consuming operation. In the FHEW scheme the cost of the bostrapping operation was amortized by combining the boostrapping with a NAND gate. This operation was further improved in its efficiency in the TFHE scheme. A ciphertext of these schemes can also encrypt multiple bits. Then, the functional boostrapping technique allows to evaluate a small boolean function with a few input and output bits, typically 3-8, within a boostrapping operation.

2.2 Circuit Representation of FHE Programs

A circuit can be represented as a directed acyclic graph $G = (V, E)$ where V is the set of vertices in the graph and E is the set of edges. An edge indicates a wire in the circuit. A vertex v is annotated with a gate, for example in the case of boolean circuits the gate can be `AND`, `OR`, `NAND`, `XOR`. We call v_1 a *parent* of v_2 if there is an edge from v_1 to v_2. We call v_2 a *child* of v_1 if there is an edge going from v_1 to v_2.

The depth of a node is defined as the maximum length path one can take from an 'input' node, i.e. a node that has no incoming edges to the node. The depth of the DAG is defined as the maximum depth of all nodes.

2.3 Related Work

2.3.1 Automatic Parallelisation. Automatic parallelization is, in general, a very difficult task since most programs rely on data-dependent branching and loops. Then, possible opportunities for parallelisation have to be identified dynamically at runtime [32]. In FHE programs however no such data-dependent operations are allowed, making it a prime target for automated parallelisation. The question of optimally parallelising FHE computations can be defined as the task of splitting a circuit into equally sized, independent subcircuits. In general the task of dividing a circuit into k subcircuits of equal size such that the number of edges between subgraphs is minimised is NP-hard [1]. We employ a not necessarily optimal but natural parallelisation technique for circuits by parallelising each layer of the circuit individually.

2.3.2 FHE Compilers. With the progress in FHE schemes also comes progress in the FHE compiler development literature such that the new and improved features of FHE schemes can be leveraged. Among the first FHE compilers were Cingulata [10], Alchemy [12], Marble [36], E3 [13] and Ramparts [2] which take high-level code and transform it into arithmetic or boolean circuits to be executed on BGV or GSW ciphertexts. An important optimization technique that has received much attention is packing or SIMD

[1] https://github.com/Anne-Me/Haskell2FHE

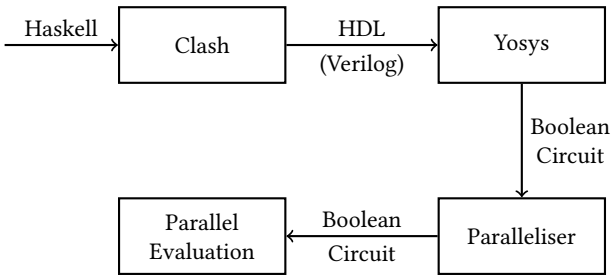

Figure 1: Our synthesis toolchain

batching which allows to pack multiple inputs into one ciphertext. Some compilers that implement SIMD batching require the user to provide the input in vectorized format [2, 17]. To reduce the effort and expertise required by the user progress was made to automatically translate SIMD-friendly programs by the compilers HECO, Fhelipe, HEIR and Porcupine [16, 24, 27, 34]. EVA [17], Ant-Ace [28] and nGraph-HE [5] are compiler frameworks that are developed with a focus on machine-learning applications such as encrypted interference. They evaluate the finalised circuits under CKKS since its inherent support for real numbers makes it practical for machine-learning tasks.

The functional bootstrapping approach requires the creation of circuits consisting of LUTs (look-up tables) instead of boolean or arithmetic gates. LUTs have long been a fundamental concept in hardware-accelerated computation, such as in FPGAs (Field-Programmable Gate Arrays). Therefore, many compilers can easily be extended to compile to LUT tables instead of boolean circuits by leveraging hardware synthesis tools such as Yosys. The HEIR [24] compiler and the compiler by Mono et al. [30] support LUT tables via Yosys compilation.

In the domain of arithmetic circuit optimisation Oraqle [37] performs depth-aware restructuring of the circuit by trying to balance the multiplicative depth which is responsible for noise growth with the multiplication count which increases the runtime.

3 The Transpiler

The toolchain of our transpiler consists of four main stages which are depicted in fig. 1. First, the code written in the high-level language Haskell is compiled into the hardware description language (HDL) Verilog using the Clash[2] compiler. Then, the verilog code undergoes optimisations and is translated into a boolean circuit using Yosys[3]. This boolean circuit enters the parallelization stage of our compiler where the circuit is prepared for the parallel execution. This stage is described in more detail in 3.1. Finally, the circuit is evaluated in a multi-threaded manner by the Evaluation procedure which uses the TFHE library.

The toolchain is similar to other compilers such as HEIR [24] which also leverages hardware language synthesis and optimization tools. HEIR focuses on creating the program in a Multi-Level Intermediate Representation (MIRL) which is then optimized and compiled into Boolean circuits. We opted to use the compiler Clash since it supports native Haskell code. A programmer who is familiar with Haskell can immediately start implementing and does not need familiarize themselves with a new language. Of course, some limitations that are inherent to the representation of programs in circuits apply. Clash does not support data-dependent recursion or recursive data types, therefore only a subset of the Haskell language is supported.

Yosys performs many optimisations such as constant-folding, dead-code elimination and depth reduction. In particular, depth reduction is useful for the following step in our workflow since it increases the number of gates that can be executed in parallel.

3.1 Parallelization

To prepare the circuit for parallel execution each gate is assigned a depth. We note that the term depth might be used differently in different works. In the context of FHE circuits depth often refers to the multiplicative depth of the circuit, i.e. the maximum number of sequential multiplications. This is usually a relevant measure due to the noise growth which is stronger in multiplication operations than in addition operations. In our application we need to identify the layer in which the gate sits so that we know at which point it can be executed. The depth of a node is the maximum of all its parents' depths + 1. This ensures that a gate is only evaluated after its parents have been evaluated. To assign a depth to the gate we first compute a topological ordering of all gates. In a topological sorting a node v_1 appears before a node v_2 if there is an edge from v_1 to v_2. The topological sorting can be computed in time O(|V|+|E|) using depth-first search. Start at an input node and continue through the graph according to the depth-first search. Upon reaching a terminal node, i.e. a node with no outgoing edges or a node where all child nodes are already collected, append the node to the list of sorted gates and continue. After all nodes are collected, assign depth 0 to all input nodes. Then traverse the list of topologically sorted nodes and for every node v_1 if $v_1.depth = i$ then for all children v_2 of v_1 set $v_2.depth = \max\{i + 1, v_2.depth\}$.

During evaluation we are given a number of threads k and for each layer all gates are split equally between the available threads.

4 Applications

In this section, we evaluate our approach on two important FHE applications: AES encryption and Private Information Retrieval (PIR). We evaluat the applications on a AMD EPYC 9374F 32-Core Processor with 128 available vCPUs running a Ubuntu 24.04.2 LTS server. We instantiated the TFHE library with a security level of 128 bits.

4.1 AES

AES is a symmetric encryption algorithm which was standardized by NIST in 2001. It operates on 128-bit blocks of plaintext and produces 128-bit ciphertexts. The algorithm supports key sizes of 128, 192, or 256 bits. In this benchmark, we focus exclusively on **AES-128**, which uses a 128-bit key. The algorithm consists of two main phases: 1. Key Expansion and 2. Encryption. In the key expansion step the 128-bit key is expanded into 11 round keys each of size 128 bits. The encryption phase of AES begins by XORing the plaintext to the first round key. For the following steps it is useful to

[2] https://clash-lang.org/
[3] https://github.com/yosyshq/yosys

imagine the 128 bit state to be split into 16 byte-sized words which are arranged in a 4-by-4 matrix. Then the algorithm proceeds in 9 rounds where each round consists of the following steps

(1) SubBytes: A non-linear substitution where each byte is replaced according to a fixed S-box.
(2) ShiftRows: Rotate the last three rows of the state by a fixed number of bytes.
(3) MixColumns: A linear mixing operation combining bytes within one column.
(4) AddRoundKey: XOR the current state with the corresponding round key.

One final round is added where only SubBytes, ShiftRows and AddRoundKey are applied.

While traditional AES implementations often use a lookup table for the S-box, this approach is suboptimal in FHE settings. Instead we implement the S-box using a Boyar-Peralta circuit [7]. This circuit uses only 128 gates to implement the S-box.

Due to the large ciphertext size in a FHE scheme it is preferable to send the inputs of a computation encrypted under the AES block cipher. The AES key is encrypted under the FHE scheme such that the inputs can be homomorphically decrypted on the server side. Although more FHE-friendly ciphers have been proposed for practical use, AES remains an important benchmark.

Following prior work [31, 33], we benchmark only the encryption phase of AES, as the key expansion can be performed in advance by the client. Our transpiler produces a circuit with 27987 boolean gates. In table 1 we report the execution times of previous works and our work.

The work of Mella and Susella [29] implements AES and other cryptographic algorithms such as SHA-256, Salsa20 and KECCAK. Their implementation targets the BGV scheme. The implementation of [22] also targets the BGV variant and they report execution times of AES under FHE in 22 minutes. Additionally, they achieve an execution of AES in 4 minutes in a levelled homomorphic encryptino scheme, therefore, the ciphertext cannot be used in further computations after the AES execution. In the domain of levelled homomorphic encryption [39] and [38] achieve strong improvements which we summarize in table 2. The work of Trama et al. [33] and Bon et al. [6] target LUT based TFHE implementations. Bon et al. [6] implement a general framework to decompose a boolean circuit into boolean functions that can be implemented as a LUT table. They apply a p-encoding approach to encode a bit with redundancy. The gadgets for boolean functions can be applied which reduces the number of functional bootstrappings necessary for the evaluation of the function. To find the gadgets for a boolean function and to find partial boolean functions in a larger circuit they develop heuristic algorithms. They also discuss parallelization opportunities for their scheme but they do not report execution times. They mention the well-known 16-thread parallelization technique and also suggest the layer-based parallelization technique to be applied locally within one S-Box execution.

Trama et al. [33] focus on transforming the steps of each AES round into 8-by-8-bit or 4-by-4-bit LUT tables to match the structure of the AES byte structure. Then they manually parallelize the SubBytes, MixColumns and AddRoundKey functions in each round of the AES encryption step. We demonstrate that we can automatically parallelize the circuit based on the layer-parallelization approach and achieve similar execution times. Both the approach [33] and our automatic parlallelization achieve an execution time of 28 seconds for 16 gates. Furthermore the layer-paralellization approach yields good results for any number of available threads while the manual parallelization approach does not allow a variable number of threads. We have tested the AES execution up to 100 threads and achieve an execution in 8.447 seconds.

In a recent work [4] achieve a strong improvement by combining the LUT-based techniques of Trama et al. [33] and the p-encoding technique of Bon et al. [6] in an implementation they name Hippogryph. This approach is highly specialized to the AES circuit and requires encoding switching between the roundkeys and the subBytes steps of each round in the AES implementation. Their results are included in table 1.

	Threads	**Time**
Gentry et al. [22]	1	18min
Mella et al.[29]	1	22 min
Stracovsky et al. [31]	16	4.2min
Bon et al. [6]	1	135s
Trama et al. [33]		
`AMD-server`	1	342s (5.7min)
`AMD-server`	16	36.39s
`i7-server`	16	28.73s
Hyppogryph [4]	1	32s
Hyppogryph [4]	32	1.1s
Our work		
`AMD EPYC`	1	311.301s (5.18min)
`AMD EPYC`	16	28.359s
`AMD EPYC`	20	23.596s
`AMD EPYC`	50	11.046s
`AMD EPYC`	100	8.447s

Table 1: Our execution times of AES-128 and those of previous works.

	Threads	**Time**
Gentry et. al. [22]	1	4min
Fregata [39]	1	86s
Fregata [39]	16	9s
Thunderbird [38]	1	46s

Table 2: Execution times for AES encryption in levelled homomorphic encryption schemes.

How much can AES be parallelized? Due to the narrow circuit structure of AES a much bigger speedup due to parallelization only cannot be expected. The AES circuit produced by our scheme has a depth of 204. In all but 1 layer there are less than 200 gates and

more than 50% of the layers have fewer than 150 gates. Since we can have at most as many parallel threads as we have gates in a layer, the maximum speed-up could be achieved for 200 threads. Of course, creating threads and waiting for threads to join can create additional overhead. Therefore, more than 100 gates will likely not yield a much bigger improvement and different compilation techniques have to be applied.

4.2 Private Information Retrieval

Imagine a hospital has a database with patient records that a research team wants to access. The hospital needs to keep its records on the server encrypted to comply with regulations and the research team does not want to reveal which illness they are researching. This scenario can be solved using Private Information Retrieval (PIR). PIR is the task of obliviously selecting a database entry based on the client's request without learning anything about the client's data. In the real world, Microsoft Edge's Password Monitor [25] uses FHE to check whether a user's saved passwords appear in known breach datasets without revealing the passwords to the server. This application is a prime candidate for parallelisation as one could imagine the database being split into individual sections that are processed in parallel.

In the **PIR-100** benchmark we evaluate the performance of our compiler for the PIR task on a database of size 100 where each entry is a 32-bit integer. In the **PIR-500** benchmark we evaluate the performance of our compiler for the PIR task on a database of size 500 where each entry is a 32-bit integer. In table 3 and table 4 respectively we report the evaluation of the benchmark with different levels of parallelisation.

	Threads	Time
AMD EPYC	1	74s
AMD EPYC	20	4.913s
AMD EPYC	50	2.26s
AMD EPYC	100	1.69s

Table 3: The PIR-100 benchmark.

	Threads	Time
AMD EPYC	1	375.625s
AMD EPYC	20	24.073s
AMD EPYC	50	10.360s
AMD EPYC	100	7.561s

Table 4: The PIR-500 benchmark.

5 Discussion

We have implemented a compiler from Haskell to a simple boolean gate-based TFHE evaluation. As a next step the library will be extended to connect to more advanced FHE implementations such as tfhe-rs [40] or FHE-Deck [26] which implement functional bootstrapping which allows the evaluation of LUT tables. We expect that the depth-based parallelization algorithm can achieve similar speedups for LUT-based circuits.

Other approaches to FHE parallelization include parallelization of individual FHE operations targeting both CPU and GPU optimizations. This approach is implemented by the libraries OpenFHE [3] and Desilo [18]. It is an interesting question for future work to compare different parallelization techniques.

Acknowledgments

We would like to thank Felix Klein from QBayLogic for helpful discussions regarding Clash.

References

[1] Konstantin Andreev and Harald Räcke. 2004. Balanced graph partitioning. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*. 120–124.

[2] David W Archer, José Manuel Calderón Trilla, Jason Dagit, Alex Malozemoff, Yuriy Polyakov, Kurt Rohloff, and Gerard Ryan. 2019. Ramparts: A programmer-friendly system for building homomorphic encryption applications. In *Proceedings of the 7th acm workshop on encrypted computing & applied homomorphic cryptography*. 57–68.

[3] Ahmad Al Badawi, Andreea Alexandru, Jack Bates, Flavio Bergamaschi, David Bruce Cousins, Saroja Erabelli, Nicholas Genise, Shai Halevi, Hamish Hunt, Andrey Kim, Yongwoo Lee, Zeyu Liu, Daniele Micciancio, Carlo Pascoe, Yuriy Polyakov, Ian Quah, Saraswathy R.V., Kurt Rohloff, Jonathan Saylor, Dmitriy Suponitsky, Matthew Triplett, Vinod Vaikuntanathan, and Vincent Zucca. 2022. OpenFHE: Open-Source Fully Homomorphic Encryption Library. Cryptology ePrint Archive, Paper 2022/915. (2022). https://eprint.iacr.org/2022/915 https://eprint.iacr.org/2022/915.

[4] Sonia Belaïd, Nicolas Bon, Aymen Boudguiga, Renaud Sirdey, Daphné Trama, and Nicolas Ye. 2025. Further Improvements in AES Execution over TFHE: Towards Breaking the 1 sec Barrier. *Cryptology ePrint Archive* (2025).

[5] Fabian Boemer, Yixing Lao, Rosario Cammarota, and Casimir Wierzynski. 2019. nGraph-HE: a graph compiler for deep learning on homomorphically encrypted data. In *Proceedings of the 16th ACM international conference on computing frontiers*. 3–13.

[6] Nicolas Bon, David Pointcheval, and Matthieu Rivain. 2024. Optimized homomorphic evaluation of boolean functions. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2024, 3 (2024), 302–341.

[7] Joan Boyar and René Peralta. 2012. A Small Depth-16 Circuit for the AES S-Box. In *Information Security and Privacy Research*, Dimitris Gritzalis, Steven Furnell, and Marianthi Theoharidou (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 287–298.

[8] Zvika Brakerski. 2012. Fully homomorphic encryption without modulus switching from classical GapSVP. In *Annual cryptology conference*. Springer, 868–886.

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

[10] Sergiu Carpov, Paul Dubrulle, and Renaud Sirdey. 2015. Armadillo: a compilation chain for privacy preserving applications. In *Proceedings of the 3rd International Workshop on Security in Cloud Computing*. 13–19.

[11] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2017. Homomorphic Encryption for Arithmetic of Approximate Numbers. In *Advances in Cryptology – ASIACRYPT 2017*, Tsuyoshi Takagi and Thomas Peyrin (Eds.). Springer International Publishing, Cham, 409–437.

[12] Eduardo Chielle, Oleg Mazonka, Homer Gamil, and Michail Maniatakos. 2022. Accelerating fully homomorphic encryption by bridging modular and bit-level arithmetic. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*. 1–9.

[13] Eduardo Chielle, Oleg Mazonka, Homer Gamil, and Michail Maniatakos. 2022. Accelerating fully homomorphic encryption by bridging modular and bit-level arithmetic. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*. 1–9.

[14] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2019. TFHE: Fast Fully Homomorphic Encryption Over the Torus. *Journal of Cryptology* 33 (2019), 34 – 91. https://api.semanticscholar.org/CorpusID:44099955

[15] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. August 2016. TFHE: Fast Fully Homomorphic Encryption Library. (August 2016). https://tfhe.github.io/tfhe/.

[16] Meghan Cowan, Deeksha Dangwal, Armin Alaghi, Caroline Trippel, Vincent T Lee, and Brandon Reagen. 2021. Porcupine: A synthesizing compiler for vectorized homomorphic encryption. In *Proceedings of the 42nd ACM SIGPLAN*

International Conference on Programming Language Design and Implementation. 375–389.

[17] Roshan Dathathri, Blagovesta Kostova, Olli Saarikivi, Wei Dai, Kim Laine, and Madan Musuvathi. 2020. EVA: An encrypted vector arithmetic language and compiler for efficient homomorphic computation. In *Proceedings of the 41st ACM SIGPLAN conference on programming language design and implementation*. 546–561.

[18] DESILO. 2023. Liberate.FHE: A New FHE Library for Bridging the Gap between Theory and Practice with a Focus on Performance and Accuracy. (2023). https://github.com/Desilo/liberate-fhe.

[19] Léo Ducas and Daniele Micciancio. 2015. FHEW: Bootstrapping Homomorphic Encryption in Less Than a Second. In *Advances in Cryptology – EUROCRYPT 2015*, Elisabeth Oswald and Marc Fischlin (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 617–640.

[20] Junfeng Fan and Frederik Vercauteren. 2012. Somewhat practical fully homomorphic encryption. *Cryptology ePrint Archive* (2012).

[21] Craig Gentry. 2009. Fully homomorphic encryption using ideal lattices. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*. 169–178.

[22] Craig Gentry, Shai Halevi, and Nigel P. Smart. 2012. Homomorphic Evaluation of the AES Circuit. In *Advances in Cryptology – CRYPTO 2012*, Reihaneh Safavi-Naini and Ran Canetti (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 850–867.

[23] Craig Gentry, Amit Sahai, and Brent Waters. 2013. Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based. *IACR Cryptol. ePrint Arch.* 2013 (2013), 340. https://api.semanticscholar.org/CorpusID:15191005

[24] 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. 2021. A General Purpose Transpiler for Fully Homomorphic Encryption. Cryptology ePrint Archive, Paper 2021/811. (2021). https://eprint.iacr.org/2021/811

[25] Sreekanth Kannepalli, Kim Laine, and Radames Cruz Moreno. 2021. Password monitor: Safeguarding passwords in microsoft edge. *Microsoft Research Blog* (2021). https://www.microsoft.com/en-us/research/blog/password-monitor-safeguarding-passwords-in-microsoft-edge/

[26] Kamil Kluczniak and Leonard Schild. 2024. FDFB^2: Functional Bootstrapping via Sparse Polynomial Multiplication. Cryptology ePrint Archive, Paper 2024/1376. (2024). https://eprint.iacr.org/2024/1376

[27] Aleksandar Krastev, Nikola Samardzic, Simon Langowski, Srinivas Devadas, and Daniel Sanchez. 2024. A Tensor Compiler with Automatic Data Packing for Simple and Efficient Fully Homomorphic Encryption. *Proc. ACM Program. Lang.* 8, PLDI, Article 152 (June 2024), 25 pages. https://doi.org/10.1145/3656382

[28] Long Li, Jianxin Lai, Peng Yuan, Tianxiang Sui, Yan Liu, Qing Zhu, Xiaojing Zhang, Linjie Xiao, Wenguang Chen, and Jingling Xue. 2025. ANT-ACE: An FHE Compiler Framework for Automating Neural Network Inference. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*. 193–208.

[29] Silvia Mella and Ruggero Susella. 2013. On the Homomorphic Computation of Symmetric Cryptographic Primitives. In *Cryptography and Coding*, Martijn Stam (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 28–44.

[30] Johannes Mono, Kamil Kluczniak, and Tim Güneysu. 2024. Improved Circuit Synthesis with Multi-Value Bootstrapping for FHEW-like Schemes. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2024, 4 (2024), 633–656.

[31] Roy Stracovsky, Rasoul Akhavan Mahdavi, and Florian Kerschbaum. 2022. Faster evaluation of AES using TFHE. *Poster Session, FHE. Org* (2022).

[32] Kevin Streit, Clemens Hammacher, Andreas Zeller, and Sebastian Hack. 2012. Sambamba: A runtime system for online adaptive parallelization. In *Compiler Construction: 21st International Conference, CC 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24–April 1, 2012. Proceedings 21*. Springer, 240–243.

[33] Daphné Trama, Pierre-Emmanuel Clet, Aymen Boudguiga, and Renaud Sirdey. 2023. A Homomorphic AES Evaluation in Less than 30 Seconds by Means of TFHE. In *Proceedings of the 11th Workshop on Encrypted Computing & Applied Homomorphic Cryptography (WAHC '23)*. Association for Computing Machinery, New York, NY, USA, 79–90. https://doi.org/10.1145/3605759.3625260

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

[35] Alexander Viand, Patrick Jattke, and Anwar Hithnawi. 2021. SoK: Fully Homomorphic Encryption Compilers. In *2021 IEEE Symposium on Security and Privacy (SP)*. 1092–1108. https://doi.org/10.1109/SP40001.2021.00068

[36] Alexander Viand and Hossein Shafagh. 2018. Marble: Making fully homomorphic encryption accessible to all. In *Proceedings of the 6th workshop on encrypted computing & applied homomorphic cryptography*. 49–60.

[37] Jelle Vos, Mauro Conti, and Zekeriya Erkin. 2024. Oraqle: A Depth-Aware Secure Computation Compiler. In *Proceedings of the 12th Workshop on Encrypted Computing & Applied Homomorphic Cryptography (WAHC '24)*. Association for Computing Machinery, New York, NY, USA, 43–50. https://doi.org/10.1145/3689945.3694808

[38] Benqiang Wei, Xianhui Lu, Ruida Wang, Kun Liu, Zhihao Li, and Kunpeng Wang. 2024. Thunderbird: Efficient Homomorphic Evaluation of Symmetric Ciphers in 3GPP by combining two modes of TFHE. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2024, 3 (2024), 530–573.

[39] Benqiang Wei, Ruida Wang, Zhihao Li, Qinju Liu, and Xianhui Lu. 2023. Fregata: Faster Homomorphic Evaluation of AES via TFHE. In *Information Security: 26th International Conference, ISC 2023, Groningen, The Netherlands, November 15–17, 2023, Proceedings*. Springer-Verlag, Berlin, Heidelberg, 392–412. https://doi.org/10.1007/978-3-031-49187-0_20

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